MOVING FORTH

Part 5: The Z80 Primitives

by Brad Rodriguez

This article first appeared in The Computer Journal #67 (May/June 1994).

THE CODE I PROMISED

At long last, I am ready to present the complete source code for an (I hope) ANSI compliant Forth, CamelForth[1]. As an intellectual exercise -- and to ensure a clear copyright -- I've written this code entirely from scratch. (Do you know how hard it is to *not* look at excellent code examples?) Of course, my experience with various Forths has no doubt influenced some design decisions.

Due to space limitations, the source code will be presented in four installments (if you can't wait, complete files will be on GEnie):

1. Z80 Forth "primitives," in assembler source
2. 8051 Forth "primitives," likewise
3. Z80/8051 high-level kernel, likewise
4. complete 6809 kernel, in metacompiler source

For CamelForth I'm trying to use exclusively public-domain tools: for the Z80, the Z80MR assembler under CP/M [3]; for the 8051, the A51 cross-assembler on an IBM PC [4], and for the 6809, my own metacompiler under F83 for CP/M, IBM PC, or Atari ST.

By "kernel" I mean the set of words that comprises a basic Forth system, including compiler and interpreter. For CamelForth this is the ANS Forth Core word set, plus any non-ANSI words necessary to implement the Core word set. A Forth kernel is usually written partly in machine code (as CODE words), and partly in high-level Forth. The words which are written in machine code are called the "primitives," since, in the final analysis, the entire Forth system is defined in terms of just these words.

Exactly *which* words should be written in machine code? The selection of the optimal set of primitives is an interesting debate. A smaller set of primitives makes for easier porting, but poorer performance. I've been told that a set of 13 primitives is sufficient to define all of Forth -- a *very slow* Forth. eForth [2], designed for easy porting, had a more generous set of 31 primitives. My rules are these:

1. Fundamental arithmetic, logic, and memory operators are CODE.
2. If a word *can't* be easily or efficiently written (or written at all) in terms of other Forth words, it should be CODE (e.g., U<, RSHIFT).
3. If a simple word is used frequently, CODE may be worthwhile (e.g., NIP, TUCK).
4. If a word requires fewer bytes when written in CODE, do so (a rule I learned from Charles Curley).
5. If the processor includes instruction support for a word's function, put it in CODE (e.g. CMOVE or SCAN on a Z80 or 8086).
6. If a word juggles many parameters on the stack, but has relatively simple logic, it may be better in CODE, where the parameters can be kept in registers.
7. If the logic or control flow of a word is complex, it's probably better in high-level Forth.

For Z80 CamelForth I have a set of about 70 primitives. (See Table 1.) Having already decided on the Forth model and CPU usage (see my previous TCJ articles), I followed this development procedure:

1. Select the subset of the ANSI Core word set which will be primitives. (Subject to revision, of course.)
2. From the ANSI descriptions, write assembler definitions of these words, plus the processor initialization code.
3. Run this through the assembler, fixing source code errors.
4. Test that you can produce working machine code. I usually add a few lines of assembler code to output a character once the initialization is complete. This seemingly trivial test is crucial! It ensures that your hardware, assembler, "downloader" (EPROM emulator or whatever), and serial communications are all working!

5. (Embedded systems only.) Add another assembler code fragment to read the serial port and echo it back...thus testing *both* directions of communications.
6. Write a *high-level* Forth fragment to output a character, using *only* Forth primitives. (Usually something like "LIT,33h,EMIT,BYE".) This tests the Forth register initialization, the stacks, and the threading mechanism. Problems at this stage can usually be traced to logic errors in NEXT or in the initialization, or data stack goofs (e.g. stack in ROM).
7. Write a colon definition to output a character, and include it in the high-level fragment from step 6. (E.g., define BLIP as "LIT,34h,EMIT,EXIT" and then test the fragment "LIT,33h,EMIT, BLIP,BYE".) Problems at this stage are usually with DOCOLON or EXIT logic, or return stack goofs.
8. At this point you can write some tools to help you with debugging, such as words to display in hex a number on the stack. Listing 1 shows a simple test routine to do a never-ending memory dump (useful even if your keyboard doesn't work). This tests the primitives DUP, EMIT, EXIT, C@, ><, LIT, 1+, and BRANCH, as well as several levels of nesting. Plus, it doesn't use DO..LOOP, which are often difficult to get working. When this code works, you have some confidence that your basic Forth model is valid.
9. From here on it's just testing the remaining primitives -- DO..LOOP, UM/MOD, UM*, and DODOES are particularly tricky -- and adding high-level definitions. I like to get a rudimentary interpreter going next, so that I can test words interactively.

With this set of primitives you can begin writing Forth code. Sure, you have to use an assembler instead of a Forth compiler, but -- as Listing 1 suggests -- you can use high-level control flow and nesting to write useful code that would be more difficult to write in assembler.

READ THE CODE!

I've run out of abstractions for today. If you want to learn more about how a Forth kernel works and is written, study Listing 2. It follows the Forth convention for documentation:

```
WORD-NAME    stack in -- stack out    description
```

WORD-NAME is the name by which *Forth* knows the word. Often these names include peculiar ASCII characters, so an approximation must be used when defining assembler labels (such as ONEPLUS for the Forth word 1+).

stack in are the arguments this word expects to see on the stack, with the topmost stack item always on the right. stack out are the arguments this word will leave on the stack, likewise.

If the word has a return stack effect (other than nesting, that is), an additional return stack comment will be added after "R:"

```
stack in -- stack out    R: stack in -- stack out
```

ANSI Forth defines a number of useful abbreviations for stack arguments, such as "n" for a signed single-cell number, "u" for an unsigned single-cell number, "c" for a character, and so on. See Table 1.

REFERENCES

[1] Definition of a camel: a horse designed by committee.

[2] Ting, C. H., eForth Implementation Guide, July 1990, available from Offete Enterprises, 1306 South B Stret, San Mateo, CA 94402 USA.

[3] Z80MR, a Z80 Macro Assembler by Mike Rubenstein, is public-domain, available on the GEnie CP/M Roundtable as file Z80MR-A.LBR. Warning: do *not* use the supplied Z1.COM program, use only Z80MR and LOAD. Z1 has a problem with conditional jumps.

[4] A51, PseudoCorp's freeware Level 1 cross-assembler for the 8051, is available from the Realtime and Control Forth Board, (303) 278-0364, or on the GEnie Forth Roundtable as file A51.ZIP. PseudoCorp's commercial products are advertised here in TCJ.

Source code for Z80 CamelForth is available on this site at http://www.camelforth.com/public_ftp/cam80-12.zip.

Continue with Part 6 | Back to publications page