MOVING FORTH

Part 6: the Z80 high-level kernel

by Brad Rodriguez

This article first appeared in The Computer Journal #69 (September/October 1994).

ERRATA

There are two goofs in the CAMEL80.AZM file I presented in TCJ#67. The minor goof is that the name length specified in the HEAD macro for the Forth word > was incorrectly typed as 2 instead of 1.

The major goof results from a subtlety of CP/M console I/O. KEY must not echo the typed character, and so used BDOS function 6. KEY? used BDOS function 11 to test non-destructively for the presence of a keypress. Unfortunately, BDOS function 6 does not "clear" the keypress detected by function 11! I have now rewritten KEY? to use BDOS function 6 (see Listing 1). Since this is a "destructive" test, I had to add logic to save the "consumed" keypress and return it when KEY is next used. This new logic can be used whenever your hardware (or operating system) provides only a destructive test-for-keypress.

HIGH LEVEL DEFINITIONS

In the last installment I did not expound greatly on the source code. Each Forth "primitive" performs a miniscule, sharply-defined function. It was almost all Z80 assembler code, and if it wasn't obvious *why* a particular word was included, I hope it was clear *what* each word did.

In this installment I have no such luxury: I will present the high level definitions which embody the elegant (and tortuous) logic of the Forth language. Entire books have been written [1,2,3] describing Forth kernels, and if you want complete mastery I highly recommend you buy one of them. For TCJ I'll limit myself to some of the key words of the compiler and interpreter, given in Listing 2.

TEXT INTERPRETER OPERATION

The text or "outer" interpreter is the Forth code which accepts input from the keyboard and performs the desired Forth operations. (This is distinct from the address or "inner" interpreter, NEXT, which executes compiled threaded code.) The best way to understand it is to work through the startup of the Forth system.

1. The CP/M entry point (see listing in previous installment) determines the top of available memory, set the stack pointers (PSP,RSP) and user pointer (UP), establishing the memory map shown in Figure 1. It then sets the "inner" interpreter pointer (IP) to execute the Forth word **COLD**.

2. **COLD** initializes the user variables from a startup table, and then does **ABORT**. (**COLD** will also attempt to execute a Forth command from the CP/M command line.)

3. **ABORT** resets the parameter stack pointer and does **QUIT**.

4. **QUIT** resets the return stack pointer, loop stack pointer, and interpret state, and then begins to interpret Forth commands. (The name is apt because **QUIT** can be used to abort an application and get back to the "top level" of Forth. Unlike **ABORT**, **QUIT** will leave the parameter stack contents alone.) **QUIT** is an infinite loop which will **ACCEPT** a line from the keyboard, and then **INTERPRET** it as Forth commands. When not compiling, **QUIT** will prompt "ok" after each line.

5. **INTERPRET** is an almost verbatim translation of the algorithm given in section 3.4 of the ANS Forth document. It parses one space-delimited string from the input, and tries to **FIND** the Forth word of that name. If the word is found, it will be either executed (if it is an IMMEDIATE word, or if in the "interpret" state, STATE=0) or compiled into the dictionary (if in the "compile" state, STATE<>0). If not found, Forth attempts to convert the string as a number. If successful, **LITERAL** will either place it on the parameter stack (if in "interpret" state) or compile it as an in-line literal value (if in "compile" state). If not a Forth word and not a valid number, the string is typed, an error message is displayed, and the interpreter **ABORT**s. This process is repeated, string by string, until the end of the input line is reached.

THE FORTH DICTIONARY

Whoa! How does the interpreter "find" a Forth word by name? Answer: Forth keeps a "dictionary" of the names of all Forth words. Each name is connected in some fashion with the executable code for the corresponding word.

There are many ways to store a set of strings for searching: a simple array, a linked list, a multiple linked list, hash table, etc. Almost all are valid here -- all Forth asks is that, if you reuse a name, the *latest* definition is found when you search the dictionary.

It's also possible to have several sets of names ("vocabularies", or "wordlists" in the new ANSI jargon). This lets you reuse a name *without* losing its previous meaning. For example, you could have an integer +, a floating-point +, even a + for strings...one way to achieve the "operator overloading" so beloved by the object-oriented community.

Each string may be connected with its executable code by being physically adjacent in memory -- i.e., the name appears in memory just before the executable code, thus being called the "head" or "header" of the Forth word. Or the strings may be located in a totally different part of memory, and connected with pointers to executable code ("separate heads").

You can even have unnamed ("headless") fragments of Forth code, if you *know* you'll never need to compile or interpret them. ANSI only requires that the ANS Forth words be findable.

The design decisions could fill another article. Suffice it to say that CamelForth uses the simplest scheme: a single linked list, with the header located just before the executable code. No vocabularies... although I may add them in a future issue of TCJ.

HEADER STRUCTURE (FIGURE 2)

Still more design decisions: what data should be present in the header, and how should it be stored?

The minimum data is the name, precedence bit, and pointer (explicit or implicit) to executable code. For simplicity, CamelForth stores the name as a "counted string" (one byte of length, followed by N characters). Early Forth Inc. products stored a length but only the first three characters, for faster comparisons (the actual improvement gained is another hot debate). Fig-Forth compromised, flagging the last character with MSB high in order to allow either full-length or truncated names. Other Forths have used packed strings [4], and I suspect even C-style null-terminated strings have been used.

The "precedence bit" is a flag which indicates if this word has IMMEDIATE status. IMMEDIATE words are executed *even during compilation*, which is how Forth implements compiler directives and control structres. There are other ways to distinguish compiler directives -- Pygmy Forth [5], for example, puts them in a separate vocabulary. But ANS Forth essentially mandates the use of a precedence bit [6]. Many Forths store this bit in the "length" byte. I have chosen to put it in a separate byte, in order to use the "normal" string operators on word names (e.g. **S=** within **FIND**, and **TYPE** within **WORDS**).

If the names are kept in a linked list, there must be a link. Usually the latest word is at the head of the linked list, and the link points to a previous word. This enforces the ANSI (and traditional) requirement for redefined words. Charles Curley [7] has studied the placement of the link field, and found that the compiler can be made significantly faster if the link field comes *before* the name (rather than after, as was done in Fig-Forth).

Figure 2 shows the structure of the CamelForth word header, and the Fig-Forth, F83, and Pygmy Forth headers for comparison. The "view" vield of F83 and Pygmy is an example of other useful information which can be stored in the Forth word header.

Remember: it's important to distinguish the header from the "body" (executable part) of the word. They need not be stored together. The header is only used during compilation and interpretation, and a "purely executable" Forth application could dispense with headers entirely. However, headers must be present -- at least for the ANSI word set -- for it to be a legal ANS Forth System.

When "compiling" a Forth system from assembler source code, you can define macros to build this header (see HEAD and IMMED in CAMEL80.AZM). In the Forth environment the header, *and the Code Field*, is constructed by the word **CREATE**.

COMPILER OPERATION

We now know enough to understand the Forth compiler. The word **:** starts a new high-level definition, by creating a header for the word (**CREATE**), changing its Code Field to "docolon" (**!COLON**), and switching to compile state (**]**). Recall that, in compile state, every word encountered by the text interpreter is compiled into the dictionary instead of being executed. This will

continue until the word **;** is encountered. Being an IMMEDIATE word, **;** will execute, compiling an **EXIT** to end the definition, and then switching back to interpret state (**[**).

Also, **:** will **HIDE** the new word, and **;** will **REVEAL** it (by setting and clearing the "smudge" bit in the name). This is to allow a Forth word to be redefined in terms of its "prior self". To force a recursive call to the word being defined, use **RECURSE**.

Thus we see that there is no distinct Forth "compiler", in the same sense that we would speak of a C or Pascal compiler. The Forth compiler is embodied in the actions of various Forth words. This makes it easy for you to change or extend the compiler, but makes it difficult to create a Forth application *without* a built-in compiler!

THE DEPENDENCY WORD SET

Most of the remaining high-level words are either a) necessary to implement the compiler and interpreter, or b) provided solely for your programming pleasure. But there is one set which deserves special mention: the words I have separated into the file CAMEL80D.AZM (Listing 3).

One of the goals of the ANSI Forth Standard was to hide CPU and model dependencies (Direct or Indirect Threaded? 16 or 32 bit?) from the application programmer. Several words were added to the Standard for this purpose. I have taken this one step further, attempting to encapsulate these dependencies *even within the kernel*. Ideally, the high-level Forth code in the file CAMEL80H.AZM should be the same for all CamelForth targets (although different assemblers will have different syntax).

Differences in cell size and word alignment are managed by the ANS Forth words **ALIGN ALIGNED CELL+ CELLS CHAR+ CHARS** and my own addition, **CELL** (equivalent to **1 CELLS**, but smaller when compiled).

The words **COMPILE, !CF ,CF !COLON** and **,EXIT** hide peculiarities of the threading model, such as a) how are the threads represented, and b) how is the Code Field implemented? The value of these words becomes evident when you look at the differences between the direct-threaded Z80 and the subroutine-threaded 8051:

word	compiles on Z80	compiles on 8051
COMPILE,	address	LCALL address
!CF	CALL address	LCALL address
,CF	!CF & allot	3 bytes !CF & allot 3 bytes
!COLON	CALL docolon	nothing!
,EXIT	address of EXIT	RET

(**!CF** and **,CF** are different for indirect-threaded Forths.)

In similar fashion, the words **,BRANCH ,DEST** and **!DEST** hide the implementation of high-level branch and loop operators. I have tried to invent -- without borrowing from existing Forths! -- the minimal set of operators which can factor out all the implementation differences. Only time, expert criticism, and many CamelForths will tell how successful I've been.

So far I have *not* been successful factoring the differences in header structure into a similar set of words. The words **FIND** and **CREATE** are so intimately involved with the header contents that I haven't yet found suitable subfactors. I have made a start, with the words **NFA>LFA NFA>CFA IMMED? HIDE REVEAL** and the ANS Forth words **>BODY IMMEDIATE**. I'll continue to work on this. Fortunately, it is practical for the time being to use the identical header structure on all CamelForth implementations (since they're all byte-addressed 16-bit Forths).

NEXT TIME...

I will probably present the 8051 kernel, and talk about how the Forth compiler and interpreter are modified for Harvard architectures (computers that have logically distinct memories for Code and Data, like the 8051). For the 8051 I will print the files CAMEL51 and CAMEL51D, but probably only excerpts from CAMEL51H, since (except for formatting of the assembler file) the high-level code shouldn't be different from what I've presented this issue...and Bill needs the space for other articles! Don't worry, the full code will be uploaded to GEnie.

However, I may succumb to demands of Scroungemaster II builders, and publish the 6809 CamelForth configured for the Scroungemaster II board. Whichever I do next, I'll do the other just one installment later.

REFERENCES

1. Derick, Mitch and Baker, Linda, Forth Encyclopedia, Mountain View Press, Route 2 Box 429, La Honda, CA 94020 USA (1982). Word-by-word description of Fig-Forth.

2. Ting, C. H., Systems Guide to fig-Forth, Offete Enterprises, 1306 South B Street, San Mateo, CA 94402 USA (1981).

3. Ting, C. H., Inside F83, Offete Enterprises (1986).

4. Ewing, Martin S., The Caltech Forth Manual, a Technical Report of the Owens Valley Radio Observatory (1978). This PDP-11 Forth stored a length, four characters, and a link in two 16-bit words.

5. Sergeant, Frank, Pygmy Forth for the IBM PC, version 1.4 (1992). Distributed by the author, available from the Forth Interest Group (P.O. Box 2154, Oakland CA 94621 USA) or on GEnie.

6. J. E. Thomas examined this issue thoroughly when converting Pygmy Forth to an ANSI Forth. No matter what tricks you play with relinking words, strict ANSI compliance is violated. A regrettable decision on the part of the ANS Forth team.

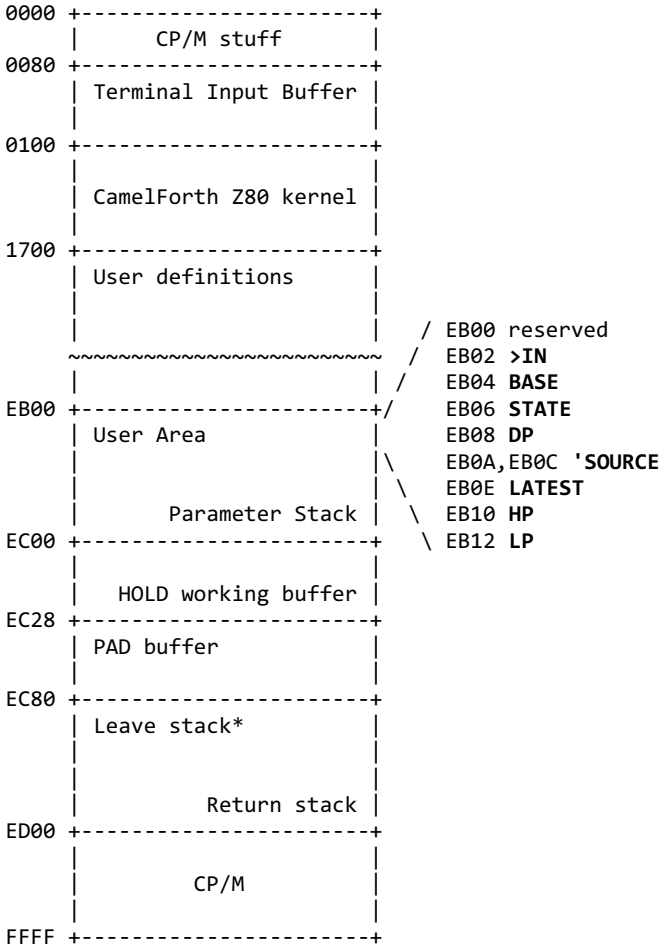
7. In private communication.

The source code for Z80 CamelForth is *now* available on GEnie as CAMEL80.ARC in the CP/M and Forth Roundtables. Really. I just uploaded it. (Apologies to those who have been waiting.)

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

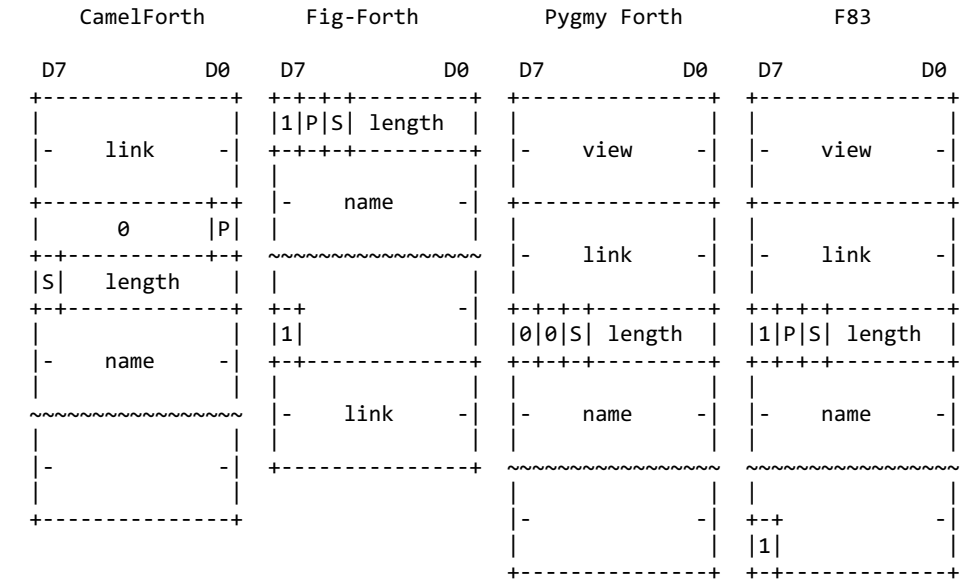
FIGURE 1. Z80 CP/M CAMELFORTH MEMORY MAP

assuming CP/M BDOS starts at ED00 hex.



* used during compilation of DO..LOOPs.

FIGURE 2. HEADER STRUCTURES



Link - in CamelForth and Fig-Forth, points to the previous word's Length byte. In Pygmy Forth and F83, points to the previous word's Link.

P - Precedence bit, equals 1 for an IMMEDIATE word (not used in Pygmy).

S - Smudge bit, used to prevent FIND from finding this word.

1 - in Fig-Forth and F83, the length byte and the last character of the name are flagged with a 1 in the most significant bit (bit 7).

View - in Pygmy Forth and F83, contains the block number of the source code for this word.

Continue with Part 7 | Back to publications page