CamelForth for 8052 (internal RAM)

Threading Model

CamelForth/8052 is subroutine threaded. When a Forth word is encountered in a colon definition, an 8051 subroutine call (LCALL) is compiled to that Forth word. E.g.,

```
: FOO   DUP * ;
```

is compiled as

```
LCALL DUP
LCALL *
RET
```

Note that the Forth EXIT is compiled as an 8051 subroutine return (RET) instruction. Note also that there is no Code Field Address. In subroutine-threaded code, t*here is no distinction between high-level words and CODE words.* Both are compiled as machine code. As a side benefit, you can call a high-level Forth word from within a CODE word.

Branching

Since all words are compiled to 8051 machine code, the 8051 conditional branch (JZ) and unconditional branch (SJMP) are used for flow-of-control. For unconditional branch this presents no problem. For conditional branches (IF, UNTIL, WHILE, LOOP) an 8051 "branch sense" subroutine is called which returns the true-or-false branch condition in A. For example, IF is compiled as

```
LCALL zerosense
JZ ...
```

The subroutine `zerosense` consumes the top stack item, and returns a zero in the accumulator if that stack item was zero. This implements the Forth branches IF, UNTIL, and WHILE. Similar subroutines `loopsense` and `pluslpsense` implement LOOP and +LOOP.

Since the 8051 instructions SJMP and JZ use a single-byte offset, all branches must be no longer than +127 or -128 bytes. **This limits the maximum size of an IF...THEN statement or a DO...LOOP.** It may be necessary to "factor" large Forth words into several smaller definitions.

Optimizations

In the CamelForth kernel, many LCALLs have been replaced with shorter ACALL instructions. Also, the sequence

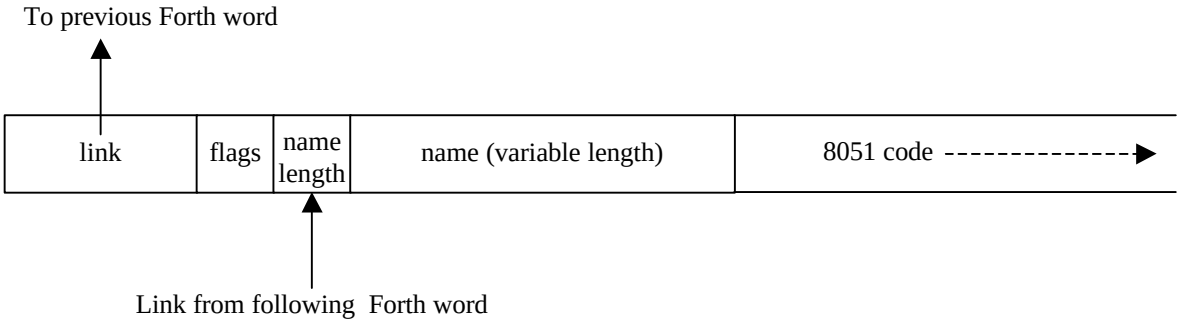
```
LCALL xxx
RET
```

is usually replaced with the shorter equivalent `LJMP xxx`. (An important exception is any word that manipulates the return stack. E.g., do *not* ever replace `LCALL RFROM` with `LJMP RFROM`!)

Note that these optimizations were performed "by hand" in the kernel source code. The CamelForth compiler does not attempt to make these optimizations when compiling Forth source code.

Header Format

CamelForth words are linked in a single linked list, with the newest defined word at the head and the oldest words last in the list. The header contains a link, a flags byte, and a counted string with the word name. It is followed immediately by 8051 machine code.



In CamelForth/8052 the "flags" byte contains only one flag. The L.S. bit is '0' to indicate an IMMEDIATE word, and '1' to indicate a normal word. The default value of the flags byte is FF hex. (This is to support Flash ROM.)

Unlike FIG-Forth, F83, and several other Forths, *no flags are stored in the name-length byte.* This allows the name to be manipulated as a standard Forth "counted string" (one byte length followed by variable-length string). Since this string is stored in Code space, the words ICOUNT and ITYPE should be used (these are equivalent to the ANSI standard COUNT and TYPE which act in Data space).

Defined Words

Since there is no Code Field, "defined words" which share a common action must be compiled with a subroutine call. The general format is

```
<Forth header>
LCALL action-routine
<any instance data in ROM>
```

There may also be RAM data associated with each instance of a defined word, as the following examples will illustrate.

Constants

Constants are stored in ROM. The action of a CONSTANT word is to fetch this value from ROM and return it on the data stack. A constant is compiled as follows:

```
<Forth header>
LCALL docon
DRW constant-value
```

NOTE when adding constants in assembler code, you must use the DRW directive instead of DW. DRW lays down a cell (16-bit) value in the little-endian order expected by CamelForth, *not* the big-endian order of the 8051.

Variables

Variables are stored in RAM. The action of a VARIABLE word is to push the address of the RAM variable on the stack. In a Harvard-model or split RAM/ROM Forth, this RAM area does not immediately follow the Forth word, so a pointer must be stored:

```
<Forth header>
LCALL dovar
DRW ram-address
```

Again note that DRW must be used.

CREATE

The ANSI Forth CREATE is used to define a table in RAM. Like a VARIABLE, the action of a CREATEd word is to push this RAM address on the stack. The implementation is similar:

```
<Forth header>
LCALL docreate
DRW ram-address
```

NOTE in the Flash ROM (Cygnal) implementation, you *cannot* use CREATE with DOES>. You must use <BUILDS instead. See below.

DOES>

A user-defined action is specified with the Forth word DOES>. This compiles a "headerless" Forth code fragment which can be used as the "code action" of the defined word. The compiled code appears as follows:

```
fragment: LCALL DODOES
      <high-level Forth code>
      ...
      ...
      ...
      <Forth header>
      LCALL fragment
      DRW ram-address
```

The "defined" word has a normal Forth header, an LCALL to the DOES> fragment, and then a pointer to an associated RAM area (akin to variables and CREATEd tables). The run-time action of the defined word is to put this RAM address on the stack, and then invoke the high-level Forth fragment.

NOTE in the Flash ROM (Cygnal) implementation, you *cannot* use CREATE with DOES>. You must use <BUILDS instead....and <BUILDS does *not* lay down the RAM address in the defined word. See below.

<BUILDS IDOES> (8052 only)

A user-defined action *with ROM data* is specified with the Forth word IDOES>. This compiles a "headerless" Forth code fragment which can be used as the "code action" of the defined word. The compiled code appears as follows:

```
fragment: LCALL DOIDOES
      <high-level Forth code>
      ...
      ...
      ...
      <Forth header>
      LCALL fragment
      <any instance data in ROM>
```

The "defined" word has a normal Forth header, an LCALL to the IDOES> fragment, and then any ROM data added by the user (normally with IC, and I,). The run-time action of the defined word is to put the **ROM** address of this instance data on the stack, and then invoke the high-level Forth fragment.

If you wish to use DOES> to get a RAM address, you must explicitly compile that address to ROM. You can do this with the phrase

```
<BUILDS HERE I,
```

and then proceed to allocate the RAM area.

Register Usage

CamelForth/8052 uses the 8051 registers as follows:

PC	Forth "Interpreter Pointer" (actually Instruction Pointer, since it is not interpreted)
SP	Forth return stack pointer
R0	Forth parameter (data) stack pointer
DPTR	TOS, top parameter stack item.
R1-R5	scratch registers
A,B	scratch registers
R6,R7	loop index

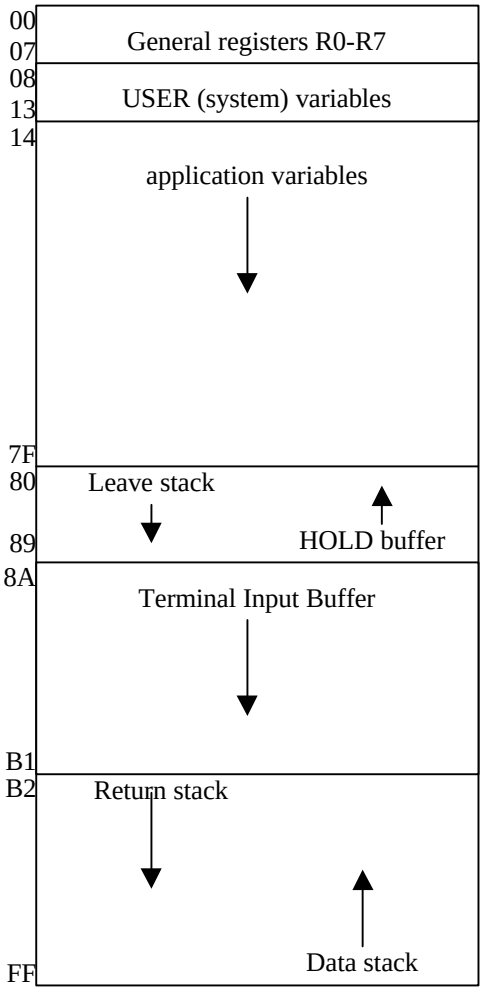
Note: Much CamelForth code assumes that registers R0-R7 can be addressed as direct registers 00-07. Do not attempt to use a different "register bank" for R0-R7.

Unlike many older Forths, CamelForth keeps the topmost data stack item in a register pair (DPTR). This speeds up many Forth operations while adding almost no complexity to the kernel.

CamelForth also keeps the loop index in a register pair (R6,R7), instead of on the top of the return stack. This speeds up loop operations while adding very little complexity. *Note* that it is the topmost loop index, and *not* the topmost return stack item, which is kept in registers. Also, the loop *limit* is still kept on the return stack. So you still must be careful not to push items on the return stack while inside a DO...LOOP.

Memory Map

Internal RAM



External Flash ROM