CamelForth

Resources

- CamelForth Z80
- On RC2014 Picasso (Minit II) ROM it is available in 8 KiB Bank 8 (JP: `0001`).
- Port of Brad Rodriguez' Z80 CamelForth to the RC2014
- Forth Tutorials
  - Forth_Primer
  - Fig FORTH glossary

Forth overview

- Some Fort implementations like `gforth` are case-insensitive, CamelForth is case-sensitive.
- There are two operation modes:
  1. Interpreted - what you type in the console is executed every time enter is pressed,
  2. Compiled - each word is compiled into binary representation (its address) that can be executed later; some language constructs that require dictionary storage to work can only be used in compiled mode when we are defining contents of a word.
- There are two main stacks consisting of cells (16bit in case of CamelForth):
  1. Value/parameter stack - where various type of data values and word addresses are stored and processed by words,
  2. Return stack - where return address is stored for each word call; it is also used to store flow control data (like iterators and conditions) and can be used by words for temporary cell storage (`>R` and `R>`).
- The interpreter treats any number as literal to be put on stack and any word as call to a word - these are space separated.
- There are special cases for words that switch into compiled mode (`:`, `CREATE`) that take a word as argument following it; also `'` to get address of the word without calling its action - word address on stack can be executed with `EXECUTE`.
- Each word consist of a link to previously defined word, name, parameter field (data storage area) and code filed (action to call on storage area address):
  - Words can be created and are registered in program memory under a given name using `CREATE` or `:`.
  - When word name is called its parameter filed address is placed on the stack and associated action code is jumped to.

    - When using `CREATE` data can be stored under address of word using `!`, a buffer of cells or bytes can be allocated using `ALLOT`. `,` can be used to append cells to last created word.
    - Action code can be appended to word using `:` or `CREATE` and appending `DOES>`.

Comments

- **[not supported in CamelForth]** `\` (*back slash*) comment to the end of line.
- `(` and `)` to comment inline.

Stack manipulation

```
1 2 3      ( put 1 2 and 3 on stack )
.          ( pip from stack and dispaly )
.S         ( show stack (gfort) )
2 DUP      ( duplicate: 2 2 )
1 2 SWAP   ( swap 2 items: 2 1 )
1 2 OVER   ( copy secondmost item to top: 1 2 1 )
1 2 DROP   ( drop top item: 1 )
1 2 3 ROT  ( move thirdmost item to top: 2 3 1 )
1 2 NIP    ( drop secondmost: 2 )
1 2 TUCK   ( dup top before secondmost: 2 1 2  )
```

Return stack

Word calls put address in return stack and `;` is compiled as `EXIT` that pops it and jump to it. It is also used by loops to keep track of condition variables and iterators and value for `I`.

- `>R` pops data stack value on return stack.
- `R>` pops return stack value to data stack.
- `R@` duplicates returns stack value to data stack.

These can be used to temporary store values on return stack but it has to be balanced and may mess with `I` and loop data if used inside loops.

Note that these only work in compiled context.

Arithmetic

```
1 2 3 4 + - * ( 3+4=7, 2-7=-5, 1*-5=-5 )
```

Note that CamelForth `.S` displays signed integers, `.` will print correct result.

Defining words

```
: DUBL 2 * ; ( define DUBL word that puts 2 on stack and multiplies two stack items )
3 DUBL       ( now can be called like any other word: 6 )
```

Printing strings

Strings are put on stack as pair of address and length.

```
: FOO S" Hello HxD"  ( record string on stack )
TYPE ;               ( print the string )
FOO
```

Note that in CamelForth `S"` can only be used in compiled context.

Introspection

```
WORDS       ( list defined words )
```

Variables

Variables are words, calling vairable puts its address on stack.

```
VARIABLE blah  ( define variable word )
42 blah !      ( store 42 in blah variable )
blah @         ( put value on stack )
blah ?         ( read and print variable )

5 CONSTANT five  ( declare constant )
```

```
five SPACES      ( print 5 spaces )
```

In Forth `ture` is `-1` and `false` is `0`. CamelForth does not have `TRUE` and `FALSE` words defined.

Control flow

`IF` checks result on top of the stack (`-1` - true, `0` - false). `then` is end of if.

```
VARIABLE var
4 var !
: test var @ 5 >
IF ." Greater" CR
ELSE ." Less or equal" CR
THEN ;
test
```

- **[not supported in CamelForth]** `?DO` takes end and start value and `LOOP` calls next iteration or is end of loop.
- `DO` same as `?DO` but iterates at least once. `I` in context of iteration puts iteration count on the stack. `2 +loop` will step by 2 while `-1 +loop` will count down. `leave` breaks the loop.

```
: test 11 1 DO I . CR LOOP ;
test
```

```
: fib 0 1
BEGIN
DUP >R ROT DUP R> >      ( condition )
WHILE
ROT ROT DUP ROT + DUP .  ( body )
REPEAT
DROP DROP DROP ;         ( after loop has executed )
20 fib
```

```
: lcd
BEGIN
SWAP OVER MOD  ( body )
DUP 0=         ( condition )
UNTIL DROP . ;
27 21 lcd      ( 3 )
```

Infinite loop.

```
: test BEGIN ." Diamonds are forever" CR AGAIN ;
```

Arrays

Access works by adding offset of size `cells` to base array address and then accessing via it like normal variable.

```
CREATE foo 16 CELLS ALLOT  ( creates array foo with 16 elements )
5 foo 2 CELLS + !          ( store 5 in index 2 )
4 foo !                    ( store 4 in index 1 )
foo 2 CELLS + @            ( put value at index 2 on stack )
foo @                      ( put value at index 1 on stack )
```

Constants are added with `,` word.

```
CREATE SIZES 18 , 21 , 24 , 27 , 30 , 255 ,
```

Strings

Have two forms:

1. Address and length - consumed by `TYPE`,
2. Counted string - `ALLOC` allocated address where first byte is string length and rest is the string itself.

- `S"` creates string constant. It can be copied to an allocated buffer with `CMOVE`.
- `COUNT` can be used to get count from counted string.
- `CHAR+` or `CHARS` adds character(s) worth of bytes to address.

```
: test S" hello bar" TYPE ;  ( puts address and length on stack and print it out )
```

```
CREATE foo 16 CHARS ALLOT  ( allocate 16 character long buffer )
: hello S" Hello World!" ; ( put "Hello World!" )
hello                      ( put address and length to "Hello World!" word )
foo !                      ( store string lenght in first byte - c-addr )
foo COUNT                  ( get the length and first byte address -c-addr c-addr u )
CMOVE                      ( copy string from contant to the buffer+1 )
```

```
foo COUNT TYPE             ( print counted string )
```

On CamelForth `foo COUNT` looks like it gives wrong address but if one prints it with `.` it shows negative number that is actually 1 more than `foo` so it is correct.

Place constant string in variable buffer.

```
: PLACE TUCK ! COUNT CMOVE ;
CREATE bar 16 CHARS ALLOT
: hello S" Hello World!" ; ( put "Hello World!" )
hello bar PLACE            ( store "Hello World!" in bar )
bar COUNT
TYPE                       ( print "Hello World!" from counted string variable bar )
```

Note that CamelForth has non-standard `>COUNTED` that does what `PLACE` does.

Pointers

```
: goodbye ." Goodbye" CR ;
: hello ." Hello" CR ;
VARIABLE a
: GREET a @ EXECUTE ;
' hello a !
GREET
' goodbye a !
GREET
```

- `'` puts address of following word on stack without executing it.
- `EXECUTE` execute word from address on stack.

Meta words

```
: displaynumber CREATE , DOES> @ . ;
11 displaynumber anumber
anumber
```

- `CREATE` takes a name (as in `CREATE foo`), in this case `CREATE anumber`, and creates an empty word for that name.
- `,` appends to the created word (storage) a cell from stack (`11`)

- When calling created word (just `anumber`) it puts its address on the stack, `@` can read cell behind that address (`11`).
- `DOES>` will attach an action to the word that executes when word is called (just `anumber`) and after its address is put on stack. In this example we read value from that words address (`11`) and print it with `.`.

CamelForth Words

- GLOSSARY OF WORDS

Low level

```
    TABLE 1.  GLOSSARY OF WORDS IN CAMEL80.AZM
    Words which are (usually) written in CODE.

NAME   stack in -- stack out          description

  Guide to stack diagrams:  R: = return stack,
  c = 8-bit character, flag = boolean (0 or -1),
  n = signed 16-bit, u = unsigned 16-bit,
  d = signed 32-bit, ud = unsigned 32-bit,
  +n = unsigned 15-bit, x = any cell value,
  i*x j*x = any number of cell values,
  a-addr = aligned adrs, c-addr = character adrs
  p-addr = I/O port adrs, sys = system-specific.
  Refer to ANS Forth document for more details.

               ANS Forth Core words
These are required words whose definitions are
specified by the ANS Forth document.

!      x a-addr --           store cell in memory
+      n1/u1 n2/u2 -- n3/u3             add n1+n2
+!     n/u a-addr --           add cell to memory
-      n1/u1 n2/u2 -- n3/u3        subtract n1-n2
<      n1 n2 -- flag           test n1<n2, signed
=      x1 x2 -- flag                   test x1=x2
```

```
>      n1 n2 -- flag           test n1>n2, signed
>R     x --   R: -- x        push to return stack
?DUP   x -- 0 | x x                DUP if nonzero
@      a-addr -- x         fetch cell from memory
0<     n -- flag             true if TOS negative
0=     n/u -- flag           return true if TOS=0
1+     n1/u1 -- n2/u2                add 1 to TOS
1-     n1/u1 -- n2/u2         subtract 1 from TOS
2*     x1 -- x2             arithmetic left shift
2/     x1 -- x2            arithmetic right shift
AND    x1 x2 -- x3                    logical AND
CONSTANT   n --           define a Forth constant
C!     c c-addr --           store char in memory
C@     c-addr -- c         fetch char from memory
DROP   x --                     drop top of stack
DUP    x -- x x            duplicate top of stack
EMIT   c --           output character to console
EXECUTE   i*x xt -- j*x   execute Forth word 'xt'
EXIT   --                 exit a colon definition
FILL   c-addr u c --        fill memory with char
I      -- n   R: sys1 sys2 -- sys1 sys2
                    get the innermost loop index
INVERT x1 -- x2                 bitwise inversion
J      -- n   R: 4*sys -- 4*sys
                       get the second loop index
KEY    -- c           get character from keyboard
LSHIFT x1 u -- x2        logical L shift u places
NEGATE x1 -- x2                  two's complement
OR     x1 x2 -- x3                     logical OR
OVER   x1 x2 -- x1 x2 x1        per stack diagram
ROT    x1 x2 x3 -- x2 x3 x1     per stack diagram
RSHIFT x1 u -- x2        logical R shift u places
R>     -- x    R: x --      pop from return stack
R@     -- x    R: x -- x       fetch from rtn stk
SWAP   x1 x2 -- x2 x1          swap top two items
UM*    u1 u2 -- ud       unsigned 16x16->32 mult.
UM/MOD ud u1 -- u2 u3     unsigned 32/16->16 div.
UNLOOP --   R: sys1 sys2 --       drop loop parms
U<     u1 u2 -- flag         test u1<n2, unsigned
```

```
VARIABLE   --             define a Forth variable
XOR    x1 x2 -- x3                    logical XOR

               ANS Forth Extensions
These are optional words whose definitions are
specified by the ANS Forth document.

<>     x1 x2 -- flag               test not equal
BYE    i*x --                      return to CP/M
CMOVE  c-addr1 c-addr2 u --      move from bottom
CMOVE> c-addr1 c-addr2 u --         move from top
KEY?   -- flag        return true if char waiting
M+     d1 n -- d2            add single to double
NIP    x1 x2 -- x2              per stack diagram
TUCK   x1 x2 -- x2 x1 x2        per stack diagram
U>     u1 u2 -- flag         test u1>u2, unsigned

               Private Extensions
These are words which are unique to CamelForth.
Many of these are necessary to implement ANS
Forth words, but are not specified by the ANS
document.  Others are functions I find useful.

(do)   n1|u1 n2|u2 --  R: -- sys1 sys2
                            run-time code for DO
(loop) R: sys1 sys2 --  | sys1 sys2
                           run-time code for LOOP
(+loop)  n --   R: sys1 sys2 --  | sys1 sys2
                          run-time code for +LOOP
><     x1 -- x2                        swap bytes
?branch  x --                  branch if TOS zero
BDOS   DE C -- A                   call CP/M BDOS
branch --                           branch always
lit    -- x         fetch inline literal to stack
PC!    c p-addr --            output char to port
PC@    p-addr -- c           input char from port
RP!    a-addr --         set return stack pointer
RP@    -- a-addr         get return stack pointer
SCAN   c-addr1 u1 c -- c-addr2 u2
```

```
                              find matching char
SKIP   c-addr1 u1 c -- c-addr2 u2
                             skip matching chars
SP!    a-addr --           set data stack pointer
SP@    -- a-addr           get data stack pointer
S=     c-addr1 c-addr2 u -- n      string compare
               n<0: s1<s2, n=0: s1=s2, n>0: s1>s2
USER   n --              define user variable 'n'
```

High level

```
     TABLE 1.  GLOSSARY OF "HIGH LEVEL" WORDS
      (files CAMEL80D.AZM and CAMEL80H.AZM)

NAME   stack in -- stack out          description

  Guide to stack diagrams:  R: = return stack,
  c = 8-bit character, flag = boolean (0 or -1),
  n = signed 16-bit, u = unsigned 16-bit,
  d = signed 32-bit, ud = unsigned 32-bit,
  +n = unsigned 15-bit, x = any cell value,
  i*x j*x = any number of cell values,
  a-addr = aligned adrs, c-addr = character adrs
  p-addr = I/O port adrs, sys = system-specific.
  Refer to ANS Forth document for more details.

               ANS Forth Core words
These are required words whose definitions are
specified by the ANS Forth document.

#      ud1 -- ud2       convert 1 digit of output
#S     ud1 -- ud2        convert remaining digits
#>     ud1 -- c-addr u      end conv., get string
'      -- xt              find word in dictionary
(      --                      skip input until )
*      n1 n2 -- n3                signed multiply
*/     n1 n2 n3 -- n4                     n1*n2/n3
*/MOD  n1 n2 n3 -- n4 n5     n1*n2/n3, rem & quot
```

```
+LOOP  adrs --   L: 0 a1 a2 .. aN --
,      x --                  append cell to dict
/      n1 n2 -- n3                 signed divide
/MOD   n1 n2 -- n3 n4   signed divide, rem & quot
:      --                begin a colon definition
;                          end a colon definition
<#     --                begin numeric conversion
>BODY  xt -- a-addr           adrs of param field
>IN    -- a-addr            holds offset into TIB
>NUMBER  ud adr u -- ud' adr' u'
                         convert string to number
2DROP  x1 x2 --                      drop 2 cells
2DUP   x1 x2 -- x1 x2 x1 x2       dup top 2 cells
2OVER  x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2  per diag
2SWAP  x1 x2 x3 x4 -- x3 x4 x1 x2     per diagram
2!     x1 x2 a-addr --              store 2 cells
2@     a-addr -- x1 x2              fetch 2 cells
ABORT  i*x --   R: j*x --      clear stack & QUIT
ABORT" i*x 0  -- i*x   R: j*x -- j*x  print msg &
       i*x x1 --       R: j*x --      abort,x1<>0
ABS    n1 -- +n2                   absolute value
ACCEPT c-addr +n -- +n'    get line from terminal
ALIGN  --                              align HERE
ALIGNED addr -- a-addr           align given addr
ALLOT  n --              allocate n bytes in dict
BASE   -- a-addr           holds conversion radix
BEGIN  -- adrs         target for backward branch
BL     -- char                     an ASCII space
C,     char --                append char to dict
CELLS  n1 -- n2                cells->adrs units
CELL+  a-addr1 -- a-addr2   add cell size to adrs
CHAR   -- char              parse ASCII character
CHARS  n1 -- n2                chars->adrs units
CHAR+  c-addr1 -- c-addr2   add char size to adrs
COUNT  c-addr1 -- c-addr2 u      counted->adr/len
CR     --                          output newline
CREATE --              create an empty definition
DECIMAL --             set number base to decimal
DEPTH  -- +n             number of items on stack
```

```
DO     -- adrs   L: -- 0        start of DO..LOOP
DOES>  --           change action of latest def'n
ELSE   adrs1 -- adrs2         branch for IF..ELSE
ENVIRONMENT?  c-addr u -- false      system query
EVALUATE  i*x c-addr u -- j*x    interpret string
FIND   c-addr -- c-addr 0     ..if name not found
                xt  1         ..if immediate
                xt -1         ..if "normal"
FM/MOD d1 n1 -- n2 n3     floored signed division
HERE   -- addr          returns dictionary pointer
HOLD   char --           add char to output string
IF     -- adrs          conditional forward branch
IMMEDIATE   --           make last def'n immediate
LEAVE  --    L: -- adrs             exit DO..LOOP
LITERAL x --      append numeric literal to dict.
LOOP   adrs --   L: 0 a1 a2 .. aN --
MAX    n1 n2 -- n3                 signed maximum
MIN    n1 n2 -- n3                 signed minimum
MOD    n1 n2 -- n3               signed remainder
MOVE   addr1 addr2 u --                smart move
M*     n1 n2 -- d       signed 16*16->32 multiply
POSTPONE  --      postpone compile action of word
QUIT   --    R: i*x --    interpret from keyboard
RECURSE --             recurse current definition
REPEAT adrs1 adrs2 --          resolve WHILE loop
SIGN   n --                 add minus sign if n<0
SM/REM d1 n1 -- n2 n3   symmetric signed division
SOURCE -- adr n              current input buffer
SPACE  --                          output a space
SPACES n --                       output n spaces
STATE  -- a-addr             holds compiler state
S"     --                  compile in-line string
."     --                 compile string to print
S>D    n -- d          single -> double precision
THEN   adrs --             resolve forward branch
TYPE   c-addr +n --         type line to terminal
UNTIL  adrs --        conditional backward branch
U.     u --                    display u unsigned
.      n --                      display n signed
```

```
WHILE  -- adrs              branch for WHILE loop
WORD   char -- c-addr n  parse word delim by char
[      --                enter interpretive state
[CHAR] --               compile character literal
[']    --          find word & compile as literal
]      --                   enter compiling state

               ANS Forth Extensions
These are optional words whose definitions are
specified by the ANS Forth document.

.S     --                    print stack contents
/STRING a u n -- a+n u-n              trim string
AGAIN  adrs --           uncond'l backward branch
COMPILE,  xt --            append execution token
DABS   d1 -- +d2        absolute value, dbl.prec.
DNEGATE d1 -- d2         negate, double precision
HEX    --                  set number base to hex
PAD    -- a-addr                  user PAD buffer
TIB    -- a-addr            Terminal Input Buffer
WITHIN n1|u1 n2|u2 n3|u3 -- f     test n2<=n1<n3?
WORDS  --                 list all words in dict.

               Private Extensions
These are words which are unique to CamelForth.
Many of these are necessary to implement ANS
Forth words, but are not specified by the ANS
document.  Others are functions I find useful.

!CF    adrs cfa --      set code action of a word
!COLON --            change code field to docolon
!DEST  dest adrs --        change a branch dest'n
#INIT  -- n         #bytes of user area init data
'SOURCE  -- a-addr           two cells: len, adrs
(DOES>)  --              run-time action of DOES>
(S")   -- c-addr u           run-time code for S"
,BRANCH xt --         append a branch instruction
,CF    adrs --               append a code field
,DEST  dest --            append a branch address
```

```
,EXIT  --             append hi-level EXIT action
>COUNTED  src n dst --        copy to counted str
>DIGIT n -- c                 convert to 0..9A..Z
>L     x --   L: -- x         move to Leave stack
?ABORT f c-addr u --            abort & print msg
?DNEGATE  d1 n -- d2      negate d1 if n negative
?NEGATE  n1 n2 -- n3     negate n1 if n2 negative
?NUMBER  c-addr -- n -1    convert string->number
                -- c-addr 0      if convert error
?SIGN  adr n -- adr' n' f       get optional sign
     advance adr/n if sign; return NZ if negative
CELL   -- n                      size of one cell
COLD   --                 cold start Forth system
COMPILE --          append inline execution token
DIGIT? c -- n -1          ..if c is a valid digit
         -- x  0          ..otherwise
DP     -- a-addr             holds dictionary ptr
ENDLOOP  adrs xt --   L: 0 a1 a2 .. aN --
HIDE   --                "hide" latest definition
HP     -- a-addr                     HOLD pointer
IMMED?  nfa -- f             fetch immediate flag
INTERPRET    i*x c-addr u -- j*x
                           interpret given buffer
L0     -- a-addr            bottom of Leave stack
LATEST -- a-addr          last word in dictionary
LP     -- a-addr              Leave-stack pointer
L>     -- x   L: x --       move from Leave stack
NFA>CFA  nfa -- cfa        name adr -> code field
NFA>LFA  nfa -- lfa        name adr -> link field
R0     -- a-addr              end of return stack
REVEAL --              "reveal" latest definition
S0     -- a-addr           end of parameter stack
TIBSIZE  -- n                         size of TIB
U0     -- a-addr           current user area adrs
UD*    ud1 d2 -- ud3           32*16->32 multiply
UD/MOD ud1 u2 -- u3 ud4          32/16->32 divide
UINIT  -- addr       initial values for user area
UMAX   u1 u2 -- u                unsigned maximum
UMIN   u1 u2 -- u                unsigned minimum
```

IO ports

It is possible to set B register (high address byte) when doing output to port:

```
88 FF54 PC!
```

Here:

- `88` is byte for data.
- `FF` is high address byte
- `54` is low address byte (port)

CamelForth implementation details

- CamelForth adds *NEXT* assembly macro invocation to each word implementation to load next word to be executed and jump to its implementation.
- `EXECUTE` pops address of word from stack and jumps to it.
- `DOES>` will then put old interpreter pointer on return stack and load new pointer and parameter field address from parameter stack and put parameter field address as top of stack, then call *NEXT*.
- After word definition is executed, *EXIT* is called that pops return address from return stack and jumps to it with *NEXT*.
- CamelForth does not use function call instructions but does uses `SP` register and `pop`/`push` to manipulate parameter stack.
- In essence all execution happens within action code of words in the dictionary and execution flow jumps from one word to another keeping return addresses on the return stack.
- Data values are stored on parameter stack and in dictionary, associated to words parameter field.
- Constants and variables are just words that don't have actions and only use parameter filed address to store values in the dictionary.

Z80 CPU register usage:

```
; Direct-Threaded Forth model for Zilog Z80
; 16 bit cell, 8 bit char, 8 bit (byte) adrs unit
;    Z80 BC = Forth TOS (top Param Stack item)
;        HL =       W    working register
;        DE =       IP   Interpreter Pointer
```

```
;        SP =       PSP  Param Stack Pointer
;        IX =       RSP  Return Stack Pointer
;        IY =       UP   User area Pointer
;    A, alternate register set = temporaries
```

- `BC` register keeps the top stack value at all times while the stack has the following items.

Revision #35
Created 2025-06-13 18:48:54 IST by hxd
Updated 2026-05-21 16:02:55 IST by hxd