
# Camel99 Forth

for the TI-99 Computer

## Glossary of Forth Words

Brian Fox
brian.fox@brianfox.ca
Manual Revision 2.0

# Camel99 Forth Glossary

This glossary is for Camel99 Forth for the TI-99 Home Computer. The glossary shows the words in the Forth dictionary when Camel99 starts and has loaded the DSK1.SYSTEM file. For questions or comments contact: @theBF on atariage.com

# Terms Used in the Glossary

### Stack Diagrams

The glossary shows the contents of the data stack before and after the execution of the word. You can think of these as inputs and outputs of the Forth word described.

Example: THEWORD ( <inputs> -- <outputs> )

The double hyphen divides what is on the DATA stack before execution and the results after execution.

Stack diagram terms

- addr address in TI-99 RAM
- caddr address of a byte counted string. (1st byte is the length)
- Vaddr address of Video RAM connected to TMS9918 VDP chip
- u unsigned number (0 .. 65536)
- n signed number ( 16 bits, -32767 .. 32768)
- d double number ( 32bits)
- Char a single byte or ASCII character
- Flag False (0) or True (-1)
- X1 x2 any item on the Forth data stack
- R: denotes contents of Forth return stack
- name shows that this word parses the input text to get name
- caddr len "stack string", an address/length pair on the data stack
- PAB Peripheral Access Block. Video RAM structure used for file I/O
- ior Input/output response ie: an error code
- fam file access mode (input, output, update etc.)
- VDP Video display peripheral. The TMS9918 chip.
- wid Wordlist identifier. Typically in the variable CONTEXT.

Items in square brackets are optional: [ optional stuff ]

# Terminal-IO

### Character Input

```
KEY?     ( -- Flag|char ) Checks if a key is waiting. Returns False or Key code
KEY      ( -- Char ) Waits for and fetches the pressed key
?TERMINAL ( -- Flag) Test for Break key (FNCT 4) (waits for key-release)

ACCEPT   ( caddr maxlength -- length )
         Read input into the string caddr for up to maxlength chars. Return length
```

### Character Output

```
(EMIT)   ( Char -- ) Emits a character raw, detects end of line
EMIT     ( Char -- ) Emits a character, interprets Newline and backspace
'EMIT    ( -- addr ) Hook for redirecting emit
```

```
(TYPE)    ( caddr length -- ) primitive version. Used to build TYPE
TYPE      ( caddr length -- ) Print a stack string to current cursor position

(CR)      ( -- flag) Primitive: Resets VCOL, Increments VROW, returns TRUE if VROW>L/SCR
<CR>      ( -- ) Primitive: Performs (CR) and scrolls screen if VROW > L/SCR
CR        ( -- ) Emits a line feed character (>0A)

BL        ( -- #32 ) Constant. Returns the ASCII code for a space
SPACE     ( -- ) Emits space
SPACES    ( n -- ) Emits n spaces if n is positive

." Hi!"   ( <text> ) Compiles a string and prints it when executed.

.(        "Talking comment" Prints <text> up to a closing bracket. ')'
```

### Multi-tasking

```
PAUSE     ( -- ) Task switcher, Does nothing by default.
```

### Stack Jugglers

Single-Jugglers:

```
DEPTH     ( -- +n ) Returns the number of single-cell items on the DATA stack
NIP       ( x1 x2 -- x2 )
DROP      ( x -- )
ROT       ( x1 x2 x3 -- x2 x3 x1 )
-ROT      ( x1 x2 x3 -- x3 x1 x2 )
SWAP      ( x1 x2 -- x2 x1 )
TUCK      ( x1 x2 -- x2 x1 x2 )
OVER      ( x1 x2 -- x1 x2 x1 )
?DUP      ( x -- 0 | x x )
DUP       ( x -- x x )

PICK      ( ... xi+1 xi ... x1 x0 i -- ... x1 x0 xi )
        Picks one element from the data stack and brings it to the top
        (See: DSK1.3RD4TH for faster options)
```

Return Stack Control

```
>R        ( x -- )   (R: -- x ) push x onto return stack (remove x from data stack)
DUP>R     ( x -- x)  (R: -- x ) Push copy of x onto return stack
R>        ( -- x )   (R: x -- ) pop x from return stack onto data stack
R@        ( -- x )   (R: x -- x) Get a copy of x from return stack to data stack
```

Double-Jugglers

```
2DROP     ( x1 x2 -- )
2SWAP     ( x1 x2 x3 x4 -- x3 x4 x1 x2 )
2OVER     ( x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2 )
2DUP      ( x1 x2 -- x1 x2 x1 x2 )
2>R       ( x1 x2 -- ) (R: -- x1 x2 )
2R>       ( -- x1 x2 ) (R: x1 x2 -- )
```

Stack pointers

```
SP@       ( -- addr )  Fetch data stack pointer
SP!       ( addr -- )  Store data stack pointer

RP@       ( -- addr )  Fetch return stack pointer
RP!       ( addr -- )  Store return stack pointer

SP0       ( -- addr )  Returns base address of data stack (from register 6)
RP0       ( -- addr )  Returns base address of return stack (from register 7)
```

## Logic

```
RSHIFT   ( x1 u -- x2 ) Logical right-shift of u bit-places
LSHIFT   ( x1 u -- x2 ) Logical  left-shift of u bit-places
INVERT   ( x1 -- x2 )   Invert all bits
XOR      ( x1 x2 -- x3 ) Bitwise Exclusive-OR
OR       ( x1 x2 -- x3 ) Bitwise OR
AND      ( x1 x2 -- x3 ) Bitwise AND
FALSE    ( --  0 ) False-Flag
TRUE     ( -- -1 ) True-Flag
SAR      ( x1 u -- x2 ) TMS9900 Specific: Arithmetic right-shift x1, u bit-places
```

## Machine Code Support

```
CODE name  ( -- )
             Create a header in the dictionary for a machine code word. Store current
             stack position in variable CSP

ENDCODE    ( -- )
             Ends a machine code word definition. Check that the DATA stack position is
             unchanged. Aborts with error "Unfinished" if not true.

;CODE       ( -- )
             Used with CREATE to make a defining word. Children of the new word will
             automatically run the machine code that follows ;CODE .
```

Example

```
             : CARRAY ( n -- )
                   CREATE  ALLOT        \ compile time allocates n bytes

             ;CODE ( n -- addr)         \ RUN time for all children of CARRAY
                   A108 ,   \ W TOS ADD,
                   NEXT,
                   ENDCODE

             100 CARRAY []X
             100 CARRAY []Y
```

### Integer Math (ANS with some 9900 extensions)

```
/MOD     ( n1 n2 -- n3 n4 ) n1 /MOD n2 = dividend: n4, remainder: n3
MOD      ( n1 n2 -- n3 ) n1 MOD n2 = remainder n3
/        ( n1 n2 -- n3 ) n1 / n2 = n3
*        ( u1|n1 u2|n2 -- u3|n3 ) Single Multiplication
*/       ( n1 n2 n3 -- n4 ) n1 * n2 / n3 = n4 (32bit intermediate math)
*/MOD    ( n1 n2 n3 -- n4 n5 )  n1 * n2 / n3 = n5 remainder n4 (32bit intermediate math)

MIN      ( n1 n2 -- n1|n2 ) Keeps smaller of top two items
MAX      ( n1 n2 -- n1|n2 ) Keeps greater of top two items
UMIN     ( u1 u2 -- u1|u2 ) Keeps unsigned smaller of top two items
UMAX     ( u1 u2 -- u1|u2 ) Keeps unsigned greater of top two items

2-       ( u1|n1 -- u2|n2 ) Subtracts two, optimized as one instruction
1-       ( u1|n1 -- u2|n2 ) Subtracts one, optimized as one instruction
2+       ( u1|n1 -- u2|n2 ) Adds two, optimized as one instruction
1+       ( u1|n1 -- u2|n2 ) Adds one, optimized as one instruction

2*       ( n1 -- n2 ) Arithmetic  left-shift by 1
4*       ( n1 -- n2 ) Arithmetic  left-shift by 2
8*       ( n1 -- n2 ) Arithmetic  left-shift by 3

2/       ( n1 -- n2 ) Arithmetic right-shift by 1

ABS      ( n -- u ) Return Absolute value of n
NEGATE   ( n1 -- n2 ) Negate
-        ( u1|n1 u2|n2 -- u3|n3 ) Subtraction
+        ( u1|n1 u2|n2 -- u3|n3 ) Addition

1+!      ( addr -- ) Increment value in memory addr (use with variables)
1-!      ( addr -- ) decrement value in memory addr (use with variables)
```

### Mixed Math Operators

```
UM*      ( u1 u2 -- ud )         u1 * u2 = ud
M*       ( n1 n2 -- d )          n1 * n2 = d
UM/MOD   ( ud u1 -- u2 u3 )      ud / u1 = u3 remainder u2
UD/MOD   ( ud1 ud2 -- ud3 ud4 ) ud1/ud2 = ud4 rem ud3
M/MOD    ( d  n1 -- n2 n3 )      d  / n1 = n3 remainder r2. Used for all signed division
S>D      ( n -- d ) Convert a signed single number a double number
```

### Single Comparisons

Unsigned comparisons

```
U>       ( u1 u2 -- flag )
U<       ( u1 u2 -- flag )
```

Signed comparisons

```
>                ( n1 n2 -- flag )
<                ( n1 n2 -- flag )
0<               ( n -- flag ) N is Negative ?
0>               ( n -- flag ) N is Positive ?
0=               ( x -- flag )
<>               ( x1 x2 -- flag )
=                ( x1 x2 -- flag )
WITHIN   ( x low hi -- flag) flag is true if (x >= low) and (x < hi)
```

### Number Base

```
DECIMAL          ( -- ) Sets base to 10
HEX              ( -- ) Sets base to 16
BASE             ( -- addr ) Base variable address
```

### CPU Memory Access

```
@       ( addr -- u|n ) Fetches single number from memory to data stack
!         ( u|n addr -- ) Stores single number from data stack to memory address
C@        ( caddr -- char ) Fetch byte from memory to data stack
C!        ( char caddr ) Stores byte on data stack to memory address

+!        ( u|n addr -- ) Add u or n to contents of addr (use with variables)

C+!       ( c addr -- ) Add c to byte memory location
2@        ( addr -- ud|d ) Fetches double number from memory
2!        ( ud|d addr -- ) Stores double number in memory

CMOVE     ( caddr1 caddr2 u -- ) Moves u bytes in memory
CMOVE>    ( caddr1 caddr2 u -- ) Moves u bytes safely if addr1 overlaps addr2
FILL      ( addr u c ) Fill u Bytes of Memory with value c

CONSTANT  name   ( u|n -- )  Makes a single constant. Return n when invoked
VARIABLE  name   ( n|n -- )  Makes a single variable n, initialized to zero

CELLS     ( u -- u' ) Returns u address units. TMS9900 cells are 2 bytes.
                  For TI-99 this is just multiply by two.(ie:4 CELLS returns 8)

CELL+     ( addr -- addr') Add one address unit to address. (ie: add 2)

CHARS     ( u – u) characters are 1 byte on TI-99. CHARS does nothing.
                  CHARS is used so your code will compile on machines with different
                  character sets

CHAR+     ( u – u+1)
              On TI-99 add 1 to u. Use for code compatibility with computers that use
              unicode.
```

### Video Memory

```
VC@     ( Vaddr -- )          fetch a byte from Video RAM
VC!     ( char Vaddr --)      store a byte to Video RAM
V@      ( Vaddr -- )          fetch an integer from Video RAM (2 bytes)
V!      ( n Vaddr --)         store an integer into Video RAM

VREAD   ( Vaddr addr n --)    read n bytes from Video RAM to CPU RAM (VMBR)
VWRITE  ( addr Vaddr n --)    write n bytes from CPU RAM to Video RAM (VMBW)
VFILL   ( Vaddr n char--)     fill Video RAM at Vaddr with n bytes of char

VWTR    ( c n --) "VDP Write to Register". Write c to video chip register n

TEXT    ( -- ) Set the video mode to 40 columns, black on green text and clear screen

VPOS    ( -- Vaddr) Video address of cursor on the screen
>VPOS   ( col row -- Vaddr) Compute  screen address from stack arguments
VPUT    ( c -- )  Write character c to the cursor position
TOPLN   ( -- Vaddr) Video address of the 1st address of the screen (VPG+VTOP)
LASTLN  ( -- Vaddr) Video address of the last valid address of the screen
SCROLL  ( -- ) Scroll the screen up one line

C/L@    ( -- n ) Fast fetch of variable C/L
C/L!    ( n -- ) Store n in C/L variable and compute screen size and store in C/SCR.
```

### String Routines

```
S" Hello"  Compiles a string of characters up to double quote character.
```

```
(S")      Runtime action of S". For internal use

Runtime: ( -- caddr length ) Returns address and length of the stack string.

Interpreting:
        S" has a non-standard interpreting action which allows you to create a
        string at the Forth command line for testing your code.


/STRING  ( caddr len n -- caddr' len')
       Cut n characters off the front of stack string caddr,len.
       Return the new stack string.

COUNT    ( caddr -- caddr len ) Convert counted string to stack string

C+!      ( c addr -- ) Add c to byte memory location (addr)

PLACE    ( caddr len addr -- ) Copy stack string to CPU memory address

VPLACE   ( caddr len Vaddr -- ) Copy stack string to VDP memory address

SKIP     ( caddr u char -- caddr' u') Skip leading char in stack string.
       Return the new stack string.

SCAN     ( caddr len char -- caddr' len' ) Find char in stack string.
        Return the new stack string.
```

### Sound

```
SND!   ( c -- ) Write byte c to the TMS9919 sound chip
BEEP   ( -- ) TI-99 beep sound. 1390 Hz for 170 mS on sound channel 1
HONK   ( -- ) TI-99 honk sound. 218 Hz for 170 mS on sound channel 1
```

### Number Conversion

```
>NUMBER   ( ud adr u -- ud' adr' u' ) Low level double number convertor

NUMBER?   ( caddr len -- n flag) Replaces ?NUMBER from original Camel Forth.
       Convert stack string caddr,len to a single precision signed integer using the
       current radix. (BASE)  flag=0 is a successful conversion.
```

### Printing Numbers

```
U.       ( u -- ) Print unsigned single number
.        ( n -- ) Print single number
UD.      ( ud --) Print unsigned double number
D.       ( d -- ) Print double number
```

### Pictured Numerical Output

```
>DIGIT   ( char -- u true | false ) Converts a char to a digit

[CHAR] *         Compiles code of following ASCII char. (for compiling only)
                 Returns char when executed.

CHAR *   ( -- char ) Returns code of following ASCII char (interpreting only)


HOLD     ( char -- )
         Adds character to pictured number output buffer from the front

SIGN     ( n -- )
         Add a minus sign to pictured number output buffer if n is negative

#S       ( ud1|d1 -- 0 0 )
         Add all remaining digits from the double length number to output buffer

#        ( ud1|d1 -- ud2|d2 )
         Add one digit from the double length number to output buffer

#>       ( ud|d -- caddr len )
         Drops double-length number. Returns pictured numeric output as a stack string

<#       ( -- ) Prepare pictured number output buffer (Sets HP to the buffer address)
```

### User Input and Interpretation

```
TIB      ( -- addr )      Terminal Input buffer (>FF00)
'SOURCE  ( -- addr )      Double USER Variable which contains source
SOURCE   ( -- caddr len ) Current source to interpret
>IN      ( -- addr )      Variable with current offset into source

PARSE    ( char -- caddr len ) Cuts anything delimited by char out of the input buffer

PARSE-WORD ( char – caddr len) Like PARSE but skips all leading spaces

WORD     ( char – caddr)
         Like PARSE-WORD, but copies text to HERE, adds a space to end of the text

EVALUATE ( addr len -- )  Saves SOURCE, Interpret stack string, restore SOURCE

INTERPRET ( addr len -- ) Make addr,len buffer the SOURCE and Interpret SOURCE
```

## Dictionary Management

```
ALIGN    ( -- )           Aligns dictionary pointer
ALIGNED  ( caddr -- addr ) Advances to next aligned address
CELL+    ( x -- x+4 )     Add size of one cell
CELLS    ( n -- 4*n )     Calculate size of n cells
ALLOT    ( n -- )         Move Dictionary Pointer by n bytes (can be negative)
HERE     ( -- addr|caddr ) Return current Dictionary position
,        ( u|n -- )       Append a single number to dictionary
COMPILE, ( u|n -- )       Same as comma at this time
C,       ( c -- )         Append a byte or character to dictionary
```

### Dictionary Flags and Inventory

```
HIDE     ( -- ) Makes current definition NOT visible to FIND
REVEAL   ( -- ) Makes current definition visible to FIND

INLINE   ( -- ) Makes current definition inline-able. (FUTURE)

HEADER,  ( caddr len --) Compile stack string as a dictionary entry. Does nothing
HEADER <name>  ( -- ) Parse <name> and make a new dictionary name. Does nothing

(FIND)  ( Caddr wid -- xt flag )
       Look for Caddr in wordlist wid. Returns xt and flag. Default action of FIND.
       Flag meanings:
          • -1  found a regular word
          •  0  no word found
          •  1  found an Immediate word


FIND     ( caddr -- addr flag )
       Searches for a String in Dictionary by executing the execution token contained in
       variable 'FIND.

'FIND   ( -- addr) Holds the execution token (XT) that executed by FIND See: FIND
```

## Compiler

```
EXECUTE        ( XT -- ) runs the execution token on the data stack
PERFORM        ( addr --) runs the XT in addr (faster than X @ EXECUTE)
RECURSE        ( -- ) Call the current colon definition

' name         ( -- xt ) find name in dictionary. Return executable address

['] name       ( -- )  Compiles the executable address of name as literal number

POSTPONE name  ( -- ) Compiles immediate and non-immediate words into a definition

CREATE name    ( -- ) Compile time: Create a definition with default action DOVAR
                      Runtime:  Returns the data address of the created word.

DOES>          ( -- )
               Runtime: ( -- addr ) Returns DATA address of word made by CREATE

STATE          ( -- addr ) Address of the compiler state variable

]              ( -- ) Switch to compile state. STATE variable set to TRUE
[              ( -- ) Switch to execute state. STATE variable set to FALSE

:NONAME        ( -- XT )
              Compile a Forth word definition with no HEADER but return the execution
              token.
```

```
: name          ( -- ) Start a new word definition and enable the compiler

;               ( -- ) Finishes new word definition and disable the compiler

<INTERP>        ( -- ) Default interpreter/compiler loop used after system starts

'IV             ( -- addr)
                Variable that contains the execution token (XT) run by INTERPRET

INTERPRET       ( Caddr len -- )
               Make the stack string Caddr,len the current SOURCE. Parse the source and
               execute each word in the string.
```

### User Variables

User variables are unique variables for each task. They are located immediately after register 15 in the workspace.

```
TFLAG           ( -- addr) Multi-tasking flag. AWAKE = TRUE   ASLEEP = FALSE
JOB             ( -- addr) Holds Execution token of the currently running word in a task
DP              ( -- addr) Hold dictionary pointer
HP              ( -- addr) "HOLD" pointer points to buffer used for number conversion
CSP             ( -- addr) Used to remember the current stack position for error checks
BASE            ( -- addr) Holds number BASE for a task
>IN             ( -- addr) "to-in" points to character in the source input buffer
C/L             ( -- addr) "chars-per-line" No of characters on one line in video mode
OUT             ( -- addr) User variable to count output characters
VROW            ( -- addr) Video display row
VCOL            ( -- addr) Video display column
'KEY            ( -- addr) Holds the XT for KEY action. Allows vectored char input
'EMIT           ( -- addr) XT executed by EMIT. For vectored char output
LP              ( -- addr) Leave stack pointer used by DO LOOP with LEAVE
SOURCE-ID       ( -- addr) 0 for main keyboard. File handle during INCLUDE
'SOURCE         ( -- addr) double variable holds address and length of SOURCE

TPAD            ( -- addr) Holds offset between HERE and the start of PAD.
                           Can be changed to give each task a different PAD if required.

VPG             ( -- addr) Video Page address of a task. Default is page zero. (0)
```

### Cold Start Variables

```
ORGDP    ( -- addr) Dictionary pointer used when system starts
ORGLAST  ( -- addr) Value used to set LATEST when system starts

BOOT     ( -- addr) Address of Forth word that runs first when system starts.
                    Default: COLD
```

### System Variables

```
VMODE    ( -- addr) Hold number of current video Mode. Default=2 (text)
L0       ( -- addr) 4 CELL array used a stack by LEAVE in DO LOOPs
^PAB     ( -- addr) Contains VDP address of current Peripheral Access Block (PAB)
LINES    ( -- addr) counts lines during file compilation
C/SCR    ( -- addr) hold the "chars-per-screen" for the current video mode
'IV      ( -- addr) "Interpret-vector". Holds address of <INTERPRET>
H        ( -- addr) Heap memory address. Defaults to >2000
VP       ( -- addr) Video memory pointer. Defaults to VDP >1000
CURS     ( -- addr) Holds cursor char and a blank.
                    The bytes are swapped to flash cursor.

VTOP     ( -- addr) Holds Video address of the top line of the display.
                    Used to make top lines of the screen permanently visible.

FLOOR    ( -- addr) Flag: floored or symmetric division. Default is TRUE (floored)

LATEST   ( -- addr) Holds the name field address of the last word defined

CONTEXT ( -- addr)
       Array of 9 cells. The first cell has the wordlist identifier (wid) of the
       wordlist that is searched first by FIND. See: FIND
       *Camel99 Forth has only one wordlist at start-up* SEE: DSK1.WORDLISTS

CURRENT ( -- addr) Holds the wid of the compilation wordlist.
                    (where new words will be compiled)
```

### System Constants

```
TIB      ( -- addr) Terminal input buffer >FF00
SP0      ( -- addr) Base address of the DATA stack (>FF00)
RP0      ( -- addr) Base address of the return stack (>FF80)
```

## Utility constants

```
FALSE    ( -- #0)  ANS/ISO Forth requirement
TRUE     ( -- #-1) ANS/ISO Forth requirement
0        ( -- #0)  Commonly used number. Saves space to use a constant
1        ( -- #1)  Commonly used number. Saves space to use a constant
BL       ( -- #32) Ascii value of a space character
L/SCR    ( -- #24) "Lines-per-screen"  Default value is 24.
```

### Error Handling

```
ABORT    ( -- )  Reset Data stack and Return stack and restart interpreter
?ABORT   ( flag caddr u --) Common factor in error reporting
?ERR     ( flag -- )       Generic error when word is not found
?EXEC    ( -- ) Aborts with message if STATE=TRUE
?COMP    ( -- ) Aborts with message if STATE=FALSE

!CSP     ( -- ) Set CSP variable to the current DATA stack position
?CSP     ( -- ) Aborts with message if stack position <> to address in CSP
```

# Control structures

Internally the Forth virtual machine has only BRANCH (jump forward/back) and ?BRANCH. (jump if top of data stack is 0)  The following words perform compiler magic that compile BRANCH or ?BRANCH as required and also compute the size of the jump for you.

## Branching Structures

```
IF      ( flag -- )      Jump to THEN if flag is true
ELSE    ( -- ) { flag IF ... [ELSE ...]   THEN } Path when flag is false
THEN    ( -- )           End of an IF block. (It is an endif)
```

## Indefinite Loops

```
BEGIN ... AGAIN                  Loop forever
BEGIN ... flag UNTIL             Loop until flag is true
BEGIN ... flag WHILE ... REPEAT  Loop while flag is true

REPEAT   ( -- )       Finish of a BEGIN WHILE loop.
WHILE    ( flag -- ) Check a flag in the middle of a loop
UNTIL    ( flag -- ) begin ... flag until loops as long flag is true

AGAIN    ( -- )  BEGIN ... AGAIN is an endless loop

BEGIN    ( -- addr)  Mark the start of a loop structure. Leaves address on data stack
```

## Definite Loops

### DO LOOP Structures

```
limit index  DO ... [one or more leave(s)] ... LOOP
?DO ... [one or more leave(s)] ... LOOP
DO  ... [one or more leave(s)] ... n +LOOP
?DO ... [one or more leave(s)] ... n +LOOP


?DO    ( Limit Index -- )-> (R: -- limit index )
         Enters the loop if limit and index are not equal

DO     ( Limit Index -- )(R: -- limit index )
         Pushes limit and index onto return stack and begins a loop


UNLOOP   (R: limit index -- )
         Drops innermost loop structure from the return stack

EXIT     ( -- ) Exits from current definition. Like a GOTO to the semi-colon

LEAVE    ( -- ) (R: limit index -- ) Leaves current innermost loop

J        ( -- u|n ) Returns second loop index
I        ( -- u|n ) Returns innermost loop index

+LOOP    ( u|n -- )R: limit index+n -- )
       Adds u/n to current loop index on return stack and checks whether to continue
       looping.

LOOP     ( -- ) (R: -- limit index )
       Increments loop index on return stack, by one and checks whether to continue
       looping.
```

### Internal System

```
WARM   ( -- )
       "warm start".  Resets system memory pointers, video device and dictionary to
       startup state. Does not load DSK1.SYSTEM.

COLD   ( -- )
       Performs a full reset. Runs WARM, loads DSK1.SYSTEM file and runs ABORT.

QUIT   ( -- )
       This is the Forth interpreter. Resets return stack and leave stack. Accepts up to
       128 characters into terminal input buffer (TIB) or until a carriage return.

BYE    ( -- ) Reset TI-99 and return to splash screen
```

### Comments

```
\      After a space delimiter, ignore all text on the line following '\'
(      After a space delimiter, ignore all text on the line until ')' is found
```

### File Management

```
DSRLNK  ( Vaddr -- ior) Vaddr must be the filename field of the active PAB

^PAB    ( -- Vaddr) Variable. Contains VDP address of active PAB
[PAB    ( -- Vaddr) Returns the VDP address of active PAB
```

## Pab Field selectors

```
These words are used with [PAB to select a field of data in the active PAB.
Fetch and store the data with VC@/VC!, V@/V!,  VREAD/VWRITE, VFILL or VPLACE as needed.
Examples:  [PAB FLG] VC@     S" DSK1.FILENAME" [PAB FNAME] VPLACE

FLG]    ( vaddr -- vaddr') byte
FBUFF]  ( vaddr -- vaddr') cell
RECLEN] ( vaddr -- vaddr') byte
CHARS]  ( vaddr -- vaddr') byte
REC#]   ( vaddr -- vaddr') byte
STAT]   ( vaddr -- vaddr') cell
FNAME]  ( vaddr -- vaddr') counted string in VDP RAM


PSZ      ( -- n)    Constant "pab-size". The size of PAB+BUFFER. 300 bytes. (>12C)
                    45 bytes for PAB, 256 bytes of buffer space for any file record.

?FILERR  ( ior -- ) generic file error handler. Reports ior and ABORTs.

FILEOP  ( opcode -- err)
       Call the file system operation specified by opcode. Return error code
       See: FSTAT
          • 1  CLOSE file
          • 2  READ record
          • 3  WRITE record
          • 4  REWIND file
          • 5  LOAD program image
          • 6  SAVE program image
          • 7  DELETE file
          • 8  Scratch record
          • 9  Get file status
```

FOPEN ( $addr len rec-size fam -- err)
Primitive file open word. Sets up PAB with given arguments
fam= file-access-mode

?FILE ( n -- ) Test if n > 0. Crude test for a filename argument

FGET ( buffer -- len ) Copy PAB buffer in VDP RAM to buffer and return length

FSTAT ( -- c) Perform file operation 9 on the active PAB. Return status byte

Status byte bit meaning

- All bits set (-1) bad device name
- 1 Device is write protected
- 2 Bad file access mode (wrong file type,record length etc)
- 3 Illegal operation
- 4 Out of buffer space
- 5 Read past end of file. File closes automatically
- 6 Hard device error of some kind
- 7 General File error. Normally the file does not exist.

INCLUDED ( addr u --) Compile the file defined by the stack string addr,u
This allows you to embed a file to include in a colon definition.
Example: : LOADTOOLS S" DSK1.TOOLS" INCLUDED ;

INCLUDE name ( -- ) Parse the input stream for name.
Compile the file path defined by 'name'. For interpreting only.
Usage: INCLUDE DSK1.TOOLS

NEEDS name ( -- flag) *NON-STANDARD*
Search dictionary for name, return TRUE if found

FROM name ( flag -- ) *NON-STANDARD*
INCLUDE file path 'name' if flag is FALSE. Used with NEEDS
Example: NEEDS DUMP FROM DSK1.TOOLS
Used to prevent loading library files that are already in the dictionary.