QL Brouhabouha Forth

Ewald Pfau (1994); edited by Marcos Cruz (programandala.net) in 2016

QL Brouhabouha Forth

Ewald Pfau (1994); edited by Marcos Cruz (programandala.net) in 2016
2016-01-21
Contents of this file

1. About this document

This document contains the whole documentation of the QL Broauhabouha Forth system (last version, from 1994-05).

The original documentation consisted of three text files. They have been combined into one document and converted to Asciidoctor[1] format, in order to make the manual easier to use and preserve it for the future[2].

The contents have not been modified, except:

- Fixed some typos found during the edition.
- Did some little layout changes, required by the new format.
- Changed the "kB" unit to "KiB"; and "F79" to "Forth 79".
- Added some footnotes, in order to provide updated information.
- Added some missing stack comments.
- Added automatic section numbering and table of contents.
- Renamed the subsection "Stack handling" to "Data stack handling" and moved it before "Return stack handling".

2. Terms of distribution

BBS Sysops!

Runtime information is only preserved by QDOS archivers!

Do **not** re-archive the executable on a different operating system than QDOS! The archive contains OS header information needed for execution.

Distribution is welcome, but **only** allowed with all original text files being at least part of what is distributed, and the executable file remains the original one.

3. Terms of usage

Brouhabouha Forth for Sinclair QL operating system QDOS comes along with permission granted for free use in noncommercial environments. That is, no profit is made intentionally by its use. Elsewise contact with the author at one of the below given addresses for special license terms is mandatory.

This is a forth interpreter and compiler mainly as described by ANS draft proposal dpANS-6 from June 1993. That paper should be the best description as well for this forth system[3].

No expectations may be made other than for occupying some memory and perhaps stealing some CPU time.

Comments and reports are welcome.

EMail[4]:
2:316/9.0@fidonet
ehp at ist dot tu-graz dot ac dot at

Surface Mail:
Hoenigtal 145 - 8301 Lassnitzhoehe - Austria/Europe

There also exists a similar version for IBM/DOS machines and a reduced version for 8051 micro controller. Developping tools exist for implementation of versions running on different controllers and platforms.

Thanks to QL community: To Jan Bredenbeek for helping to keep things together via EMail, to Franz Herrmann for his port of LHarc Archiver, to Jonathan Hudson for his "Ibmdisk" tool.

Thanks to Forth community: To Johannes Teich for providing news and putting questions.

4. For novice users

Only some few hints, no tutorial:

1. There is no other delimiter in forth interpreter than white space or end of line (end of block, end of file). Exceptions only if characters are scanned for, as with '"' (double-quotes) or ')' (right paren). So `(` (left paren) is a Forth word as all that other stuff and as well delimited by a blank (but it will scan for a right paren which by that action does not need a delimiter).
2. The easiest way to leave the system is to say `BYE`.
3. The easiest way to look around is to use the tools provided: `WORDS`, `.S`, `SEE <name>`, `DUMP`. While interpreting, strings may be input with `S" <string>"` â’ after that, parameters of a string, address and length, are waiting on the stack to be dealt with, as consumed by `TYPE` or by `R/W OPEN-FILE` or doubled by `2DUP` and so on; `.S` will show at any time what is on the stack.
4. Most words are built with defining word `:` (colon), followed by a new_name, a sample of actions, and the definition ended by `;` (semicolon). Words built that way, colon definitions, "secondaries", may be "seen" by `SEE <new_name>`. Other defining words are `CONSTANT`, `VARIABLE`, `VALUE`, `CREATE`. By help of `DOES>`, as well defining words may be defined, mainly to inherit the same runtime code to different data structures composed with `CREATE` and `ALLOT`. Since this is an easy way, there are no predefined arrays. The data representation of any definition is the execution token (formerly called "cfa", "code field address"), which may be found by `' <name>` ("tick" of name) and consumed by `EXECUTE` or by `CATCH`.
5. Words starting with a dot by convention are words which will show something. A dot by its own will print a number. Numbers are input by typing them in, so they are on the stack. For the interpreter, numbers containing a dot somewhere between the digits are interpreted to be double-cell numbers, the high value being top of stack. Numbers being put in with a leading '$', '&' or '%' are taken to be composed of hex digits, decimal or binary, independant of what is the momentary valid base held in variable `BASE`.
6. Leo Brodie's book *Thinking Forth* has been published again[5].

5. Some differences to Forth 79 and Forth 83

1. Only data memory may be regarded as being addressable!
2. A `DOES>`-clause only is a valid runtime for definitions which have been created by direct or indirect use of `CREATE`. From a word generated by a (child of) `CREATE` the address of associated data memory may be calculated by applying `>BODY` to the execution token of that word. The amount of needed data space should have been reserved by use of `ALLOT` or `,` ("comma") or `C,` ("C-comma"). Proper use of `ALIGN` must be taken care of, if memory shall be addressable as storage area for cells.
3. To control recovery from ambiguos situations, execution tokens may be executed by being consumed as a parameter of `CATCH`. By this, a frame is initialized for stack and return stack, which will be discarded if a `THROW` is executed by a called instance, with a parameter other than zero. This parameter will then be returned by `CATCH`. Elsewise, a zero is consumed by `THROW`, and `CATCH` returns zero, so the frame has been worked thru step by step. If `CATCH` returns a non-zero value, so an ambiguos situation is given, which may be dealt with as the special situation dictates. This non-zero parameter may be handed over then to another `THROW` following the special action taken in this case. By each of the interpreting loops, `EVALUATE`, `LOAD`, `INCLUDE-FILE`, `QUIT`, such a frame is

established, which will be discarded as a last instance for recovery, which will yield into the action of QUIT. So ambiguos situations may be "thrown" in any case, and control is given back to the interpreting loop then. If interpreting is done from operating system commandline, so control is given back to the operating system, to ensure a workable condition if interpreting is done in unattended situations. As well, if the returnstack frame is out of bounds, a BYE is executed. Throw-values in the range -4095â'¦0 are reserved for system own use.

4. Instead of vocabularies, switchable by names, now there are wordlists, switchable by parameters. WORDLIST will install a new wordlist and return a wordlist identifier ("wid"). This parameter is consumed by SET-CURRENT to switch the compiling wordlist. GET-CURRENT returns the identifier of the active compiling wordlist. GET-ORDER will return a set of parameters, the count being on top of stack. SET-ORDER will switch the active search-order, consuming a set of parameters with its count on top of stack, the identifier of the first searched wordlist being next on stack. FIND will search in the active search-order. SEARCH-WORDLIST only will search in one wordlist, given as parameter. FIND will consume the address of a counted string, and if the name contained in that string is not found, it will return that address and zero as top of stack. SEARCH-WORDLIST will consume the parameters of a string as address and length plus a wordlist identifier, and if not found, only will return zero. If the searched string could be found to be the name of a definition in the wordlists which had been scanned, so the execution token of that definition is returned, and a flag, being greater than zero, if that word is immediate, elsewise a flag being less than zero. This is the only way, the immediacy of a definition may be determined.
5. To inherit a compiling action to the runtime of a definition, instead of formerly used COMPILE <name> and [COMPILE] <name> now the "immediacy smart" POSTPONE <name> is provided, as well as COMPILE, ("compile-comma"). By POSTPONE, "name" will be compiled by itself if it is immediate, so it will be executed at runtime instead of being executed at compile time, elsewise "name" will be compiled at runtime. COMPILE, takes an execution token as parameter and appends it at runtime to the definition of which the compilation is being in progress.
6. There are four defined input streams now. This is for strings in memory, via EVALUATE (given parameters for that string), for files containing text lines â' via INCLUDE-FILE (given a file handle) or INCLUDED (given parameters for the string of a file name) â' , for block files via THRU or LOAD (given two block numbers or one; and for this implementation: LOAD-FILE given a block number and a file handle), and console, via QUIT (no parameters; state of the machine is reset to interpreting, debugging is switched off, returnstack and catch-frames are reset).
7. SAVE-INPUT will return a set of parameters with the count being on top of stack, being an internal description of the momentary position valid for the interpreter. By RESTORE-INPUT it will be tried to set the position valid for the interpreter to the position as described by the given set of parameters, which should come from an earlier execution of SAVE-INPUT; will return a true flag if this was possible, a false flag otherwise. No switching across different input streams or files should be done this way (even if this implementation is capable of it).
8. SOURCE will return the parameters of the whole string which for the moment is the input for the interpreter, consisting of one line for console or ASCII file, or one block for block files, or the string given to EVALUATE. Within the input string, the valid position being interpreted next may be set by reading the value held in variable >IN and writing it back again. An offset obtained from scanning in that string may be added or subtracted. The rest of the string which waits for being interpreted may be calculated by: SOURCE >IN @ /STRING. This will leave a length of zero if the input stream is empty for this line or block.
9. The rest of this string is discarded by REFILL â' and by that, from the input stream the next line or block is tried to be obtained. A flag is returned, zero in case the input stream had been a string in memory for EVALUATE or an end of file has been reached.
10. A counted string being returned by WORD does no more depend on being delimited by a blank (or zero in Forth 79). A delimiting blank will be appended anyway after the isolated string has been moved to the position in memory returned by HERE. The former number conversion, which needed that

delimiting blank has been changed in being built upon the behaviour of >NUMBER â’ which takes as parameters a double cell accumulation value and the parameters of a string, these values being updated and returned. If the string is empty afterwards, so its length is found as the length parameter being zero, so no delimiter is needed.

11. Formerly used immediate word ASCII, to obtain a character from the input stream, has been replaced by non-immediate CHAR and immediate [CHAR]. EXPECT â’ to obtain a string as typed in from keyboard â’ has been replaced by ACCEPT; instead of the resulting length of the input string being held in variable SPAN now the result is returned on stack. Instead of fomerly used FORGET <name> â’ after having made allocations -, now markers should be set using MARKER <name>, before making allocations. Execution of <name> then will have the same effect as now obsolete FORGET <name>.

6. Implementation characteristics and special behaviour

By use of ADJUST-SIZES and SAVE-FILE (described somewhere else in one of these texts), the sizes of RAM, which the executable claims from QDOS, may be set to individual needs. This is set to be 192 KiB in this distributed version.

For File Wordset words RENAME-FILE and RESIZE-FILE, the QDOS extension traps provided by TK2 should be workable (Trap 3, functions $4A and $4B). No other extensions are required except provision for enough memory, that is at least a 256 KiB expansion card on original QL.

Floating Point wordset is missing.

Only two internal throw codes are implemented, these are -2 for ABORT" and -1 for ABORT. -254 THROW will give the behaviour of BYE (this may change in future versions).

Environmental queries with ENVIRONMENT? only will deliver a dummy argument of zero.

The executable will claim 192 KiB from the operating system at startup. This is 112 KiB for code, 48 KiB for headers and 32 KiB for data[6]. With the block editor loaded as-is, for free use remain about 42 KiB for code, 22 KiB for headers, 19 KiB for data. After execution of marker NOEDIT, this increases to 59 KiB, 27 Kb, 23 KiB, with no editor. This may be adjusted to individual needs by use of ADJUST-SIZES and SAVE-FILE for next start-up of saved executable.

QDOS channel IDs are held in an internal table keeping a maximum of 32 values. Forth I/O-handle parameters for I/O-access are offsets into this table. Positions in this table having become free are re-used.

Memory in data area is addressable as cells only with aligned addresses.

EMIT of non-graphical characters is handled by QDOS. It will print a filled rectangular.

Editing of keyboard input via ACCEPT is done via QDOS trap IO.FLINE, so commandline history will be functionable if implemented, and deleting, cursor positioning, and inserting of characters is working. IO.FLINE is started as a secondary job from QDOS. Forth job is named "ZQF" in QDOS job table. Secondary job is named "ZQF/con_".

After the given string length for ACCEPT has been input, or the ENTER key is pressed, the call is terminated. The code for the ENTER key is the code for LF.

KEY will act upon character input in the range of character codes 0â’¦255. Keypresses giving other values are discarded if waiting or scanning for input with `KEY` or `KEY?`. The set of all keycodes may be obtained by use of `EKEY` or scanned for by use of `EKEY?`.

A character storage cell has the width of one address unit. At any time the machine may be regarded as character aligned. The only memory operator acting upon address units is `MOVE`. All other memory operators will act upon character or cell units.

No provision is made to store definition names containing characters which are not printable characters, via keyboard or ASCII file input. If such names have been defined, using input from block files or via `EVALUATE`, so the corresponding execution tokens may be found as well the same way.

If the input stream is switched to an ASCII file and interpreting is done via `INCLUDE-FILE` or `INCLUDED`, so characters with codes smaller than 32 are treated as white space, except for the codes for CR and LF, which are taken as EOL markers. End of line is reached with CR, LF, or the sequence CR LF.

While compiling, control flow stack for compiling of conditionals is the parameter stack. Non-immediate words `CS-PICK` and `CS-ROLL` will copy or exchange control flow values.

A value of greater than 36 kept in variable `BASE` will give an undefined behaviour for digit conversion. A value of up to 36 will provide conversion to digits 0â’¦9,Aâ’¦Z. `DECIMAL` will reset conversion to digits 0â’¦9.

Compilation of an `ABORT" <text>"` sequence will be compiled as a sequence starting with a conditional branch and an inline string containing <text>.

Maximum sizes are: 255 characters for counted strings, 30 for definition names and the output of `WORD`. Parsed strings may be as long as the input field where they are taken from. If 'S"' is executed while interpreting, so the parsed string is kept in a circular buffer, and at least the two last input strings are available, each with a maximum size of 255 characters.

User input and output devices are opened at start of program. The first input device is the commandline as given to QDOS as parameter with program call. This line is evaluated. After that, all further input is taken via QDOS call IO.FLINE. User output device is a console window set to no border, CSIZE 0,0, INK 7, PAPER 0 at position 0, 0, sized 512 * 256 with cursor switched on and the window cleared. First IO.FLINE and output window are set up, then the commandline is evaluated, then the banner is shown. This behaviour may change or as well be made configurable in future versions.

The accessible dictionary space is data space only, this is separate from code space and name space.

One address unit is 8 bits wide.

Numbers are stored in cells sized 4 address units, in two’s complement representation for signed values.

Single cell signed numbers may take values from -2^31 to 2^31-1. Single cell unsigned numbers may take values from 0 to 2^32-1. Positive single cell numbers may take values from 0 to 2^31-1.

Double cell signed numbers may take values from -2^63 to 2^63-1. Double cell unsigned numbers may take values from 0 to 2^64-1. Positive double cell numbers may take values from 0 to 2^63-1.

Data space of dictionnary is contiguos. Stack and return stack are kept inside this area and should be regarded as non-addressable â’ positions may change with heap allocations. Strings returned by `S"`, `C"`, `WORD`,

PARSE, SOURCE should be regarded as read-only and be copied before they are written to.

Buffer for WORD is shared with buffer for pictured number conversion and kept at the address returned by HERE. WORD will not return a string with a size greater than 31 characters.

Storage area for one cell is sized 4 address units. Storage area for one character is sized 1 address unit.

Keyboard input terminal buffer is sized 128 characters.

Storage area for pictured numeric output string is sized 64 characters, 32 of which are shared with the buffer for WORD.

The size of the scratch area, starting at the address returned by PAD, is all unused data memory. So PAD UNUSED will give a maximum temporary storage area.

Finding definition names internally is done by uppercasing the result of WORD from keyboard input and uppercasing the names searched for. Names of new definitions are stored uppercase. This case independent behaviour may be switched off by writing 0 to variable CAPS. After that, names of new definitions are stored as given and finding is done case dependant with input as given.

The system prompt, after a line from keyboard has been successfully interpreted, is "ok" followed by as many dots, but not more than 32, as are items on the stack. If state of machine is the compiling state, instead of "ok" "]" is output. Cursor is set to a new line then.

Operators , /, MOD, /MOD, /, and */MOD are not provided and have to be defined using the preferred method of rounding characteristics, given either by SM/REM, symmetrical, or FM/MOD, towards negative infinite.

Definitions by using SM/REM are:

```
: /MOD   ( n1 n2 n3 -- n4 n5)  over 0< swap sm/rem ;
: /      ( n1 n2 -- n3)        /mod swap drop ;
: MOD    ( n1 n2 -- n3)        /mod drop ;

: *      ( n1 n2 -- n3)        m* drop ;
: */MOD  ( n1 n2 n3 -- n4 n5)  >R m* R> sm/rem ;
: */     ( n1 n2 n3 -- n4)     */mod swap drop ;
: /MOD   ( n1 n2 -- n3 n4)     over 0< swap sm/rem ;
```

In compiling state, variable STATE contains -1, else 0.

Arithmetic overflow or underflow will give results modulo 2^32, the operators seen as unsigned numbers. Division by zero will give results of 0 for quotient and remainder.

While compiling a runtime part of a definition in a DOES>-clause, the name of the defining word of which the compilation is in progress, will not be found. It is revealed only after finishing the definition with ; (semicolon).

No system words will use the scratch area starting at PAD.

Return stack size is 256 cells. Parameter stack size is all of unused memory growing downwards, if not used as scratch area starting at PAD, growing upwards. By UNUSED, a stack size of 256 cells is taken into account.

7. Words and tools not covered by the ANS Forth paper

7.1. Words available in Forth wordlist

```
U<=>  ( u1 u2 -- -1/0/1)
```

Compare unsigned two values given, return 0 if both are the same, -1 if the value next on stack is smaller than the value on top of stack, 1 otherwise.

```
CASE?  ( n1 n2 -- n1 0 // -1)
```

Compare two values given, return -1 if both are the same, elsewise leave the value next on stack and zero on top of stack.

```
UMIN  ( u1 u2 -- u3)
```

Leave unsigned the smaller one from two values given.

```
UMAX  ( u1 u2 -- u3)
```

Leave unsigned the bigger one from two values given.

```
$CRC16+  ( n0 a n -- n1)
```

Apply a cyclic redundancy check calculation to the string in memory given by its parameters, address and length, starting with the accumulating value given as the next but one value on stack, returning the accumulated value.

```
CREATE-JOB  ( #code #data -- a job-id err#)
PJOB  ( priority job-id -- err#)
AJOB  ( priority 0/-1 job-id -- err#)
SUSJOB  ( #ticks job-id -- err#)
RELJOB  ( job-id -- err#)

F#@  ( id -- channel#)
```

Lookup the OS specific channel number of the handle number as it has been returned by `OPEN-FILE` or by `CREATE-FILE`.

```
IN-CHANNEL  ( -- a)
```

Variable, containing the handle number used for input from keyboard.

```
CRT-CHANNEL  ( -- a)
```

Variable, containing the handle number used for output to console.

```
CSIZE  ( n1 n2 --)
PAPER  ( n --)
INK  ( n --)
CURS-ON  ( --)

DEL  ( --)
```

Destructive backspace is output on console window if cursor is not already in first column.

```
SKIP  ( a n char -- a+ n-)
SCAN  ( a n char -- a+ n-)

CAPS  ( -- a)
```

Address of a variable in which -1 is kept if some system words shall act case independant. These are `FIND` and `SEARCH-WORDLIST` for the interpreter as well as `COMPARE` and `SEARCH` for string words. Writing 0 to `CAPS` will switch to case dependant behaviour.

```
CAPITAL  ( char -- char)
```

A given character is returned uppercased.

```
STACK|  ( ix..iy -- ix..iy // "abcd...|xyz...|")
```

Rearrange the order of values on stack, taking each letter of a first series of letters up to a first pipe sign '|', to be one item on stack in the order as is, the last letter representing the value on top of stack, and rearrange, or as well duplicate or drop, following the series of letters up to a trailing pipe sign.

```
USE  ( -- // "name")
```

To the block file system, file <name> is assigned to. This file is opened and the file handle is stored in variable `BLOCK-FID`. If a block file already has been open with it's file handle stored in `BLOCK-FID`, it is closed after the new file could be opened.

If on the same drive the medium is to be changed, and a file is open on the first medium, so this file should be closed by: `BLOCK-FID @ CLOSE-FILE THROW 0 BLOCK-FID !`.

```
LOAD-FILE  ( n hdl --)
```

A file represented by a file handle is regarded to be a block file, the value stored in `BLOCK-FID` is saved, the block number given is loaded as with `LOAD`, and `BLOCK-FID` restored afterwards.

```
SAVE-FILE  ( a n -- err#)
```

An image of the momentary running executable is saved as a file the name of which given by its parameters, address and length. A nonzero value is returned if operation failed.

```
GET-SIZES  ( -- #code #data #hdrs)
```

Three values are returned on stack reflecting the amount of memory sizes the executable would claim from the operating system at next startup. These are for maximum size of code, of data, and, as top of stack, of headers.

```
ADJUST-SIZES  ( #code #data #hdrs --)
```

Three values are taken from stack to be the sizes of memory which will be claimed from operating system more than momentarily needed, at next start-up of the momentary running executable, if it is saved after adjusting sizes by `SAVE-FILE`. The values are for additional size of code, of data and of headers area, the last one being expected as top of stack. All parameters being zero will leave an unexpandable execution image which will claim about 100 KiB if editor is loaded, about 80 KiB without. Data memory already reserved by

ALLOCATE is not included and should be added to the parameter for the amount of the size of data area. For use of directory access words OPEN-DIR, READ-DIR, CLOSE-DIR, an amount of 100 address units should be reserved for each instance this will be made use of.

Caution! Parameters for an image exceeding available RAM will leave a non-executable image!

```
.MEM  ( --)
```

Some values are displayed to reflect the current state of the system, remaining sizes of memory areas, identifiers of wordlists, tasks and heap memory areas.

```
TICKS  ( -- n)
```

A value is returned which is incremented once in a second.

```
TICKS>T&D  ( n -- ss mm hh dd mm yy)
```

Calculate the values for time and date from the value as given by TICKS.

```
.DATE  ( --)
```

The current date is output to the console as given by the OS.

```
.TIME  ( --)
```

The current time is output to the console as given by the OS.

```
AVAILABLE  ( -- u)
```

Maximum size which may be reserved on the heap by ALLOCATE is returned.

```
ERRORLEVEL  ( -- a)
```

7.2. Words available in Assembler wordlist (if assembler loaded or not)

```
>CODES  ( -- a)
```

Variable, holds address of a table, in which the execution tokens are kept for all assembler memory and dictionnary accesses. As a default, assembler code is generated in code area. This may be changed by providing a new table containing execution tokens for the actions of C@, @, C!, !, HERE, ALLOT, ALIGN, C, ("C-comma") and , ("comma").

7.3. Words available in Editor wordlist (if editor loaded or not)

```
B/BLK  ( -- 1024)
```

Return size of one block of a block file.

```
BLK-UM*  ( u -- d)
```

Multiply a given value with B/BLK, returning a double cell product.

```
BLK/MOD  ( d -- mod div)
```

Divide a double cell number by `B/BLK`, returning modulo and quotient.

```
-SKIP  ( a0 n char -- a0 n-)

-SCAN  ( a0 n char -- a0 n-)

WSCAN  ( a n1 n2 -- a+ n-)
```

In an array of cells represented by address of first cell and count of cells, a value is scanned for which is given as top of stack. If this value could be found, so the parameters of the rest of the array is returned, starting at the address at which the searched value is stored in and the numbers of cells remaining, including the found position. Elsewise zero is returned as top of stack and the first address behind the array.

```
?ESC  ( -- 1 /key:esc , -1 /cr , 0 , /space = wait)
```

Keyboard is scanned for a keypress. Enter key will return -1, ESC key will return 1, else 0. If space key has been pressed, a second press of space key is waited for. Other keypresses are skipped.

```
TAB+  ( n --)
```

Cursor is positioned to next tab stop for a field width given as parameter. If this field does not fit into the rest of the line, a new line is started.

7.4. Debugging

```
XDUMP  ( 'a n --)
```

Show a dump of code memory not addressable with data memory words.

```
'SEE  ( cfa --)
```

Apply the function of `SEE` on the execution token given instead of the name of a definition.

```
DEBUG  ( -- /"name")
```

Debugger is initialized to be called each time, the given <name> will be executed next, if it is a secondary forth definition. Exception handling and QUIT will disable the debugger. Debugger is set to single step mode, so after each step, the Enter key has to be pressed. Instead a line may be given to the interpreter while debugging in single step mode. After execution of this line, the prompt will be the forth prompt with the leading string "debug: ". Next step will be executed if Enter is pressed with an empty commandline. Debugging keys input is taken from keyboard even if input source is a file. Some words used by the debugger itself cannot be debugged, elsewise restrictions for interpreter use while debugging are minimal. With shared code of tasks, only the same instance of code will be debugged, the debugger itself is working in.

```
'DEBUG  ( cfa --)
```

Apply the function of `DEBUG` on the execution token given instead of the name of a definition.

```
FT  ( --)
```

Debugging mode is switched to "Fast Trace". This may be done at any time after the debugger has been initialized. If Enter key is pressed, so mode is switched to single step mode. If space key is pressed, so output is halted until another pressing of the space key.

```
NODEBUG  ( --)
```

While in single step debugging mode, debugger is switched off and execution continuos without debugger.

7.5. Internal Tasking

```
SELF  ( -- a)
```

Within a task, its own task identifier "tcb" is given.

```
PAUSE  ( --)
```

Essentially a nope, if tasker is switched off, elsewise a hook, which enables next active task to continue execution until this one by itself executes a PAUSE. Each system word performing I/O ends with the action of PAUSE.

```
TASKER-ON  ( --)
```

Behaviour of PAUSE is switched to be a hook into next task in the circular list of current active tasks.

```
TASKER-OFF  ( --)
```

Behaviour of PAUSE is switched to be a NOPE. May be executed within tasks instead of semaphores to protect access to open files if used in conjunction with TASKER-ON â' the interpreter is switched off also this way, if it is not the current task from where this is executed.

```
WAKE  ( tcb --)
```

A task is put into the circular list of current active tasks.

```
SLEEP  ( tcb --)
```

A task is taken out from the circular list of current active tasks. If code of a task is placed in the memory area for which a marker has been defined, so the action of SLEEP is performed for this task, if that marker is executed and the task is active. If a task is not an endless loop, so it will stop by performing the action of SLEEP, and re-initialize itself after finishing, for being restartable again by WAKE.

```
RUNS  ( tcb -- /"name")
```

To an existing and "sleeping" task, a new action is assigned to, given by <name>.

```
TASK:  ( Execution:         -- tcb)
       ( Compilation: #R #S --  /"name")
```

Start of definition of a task <name>. Parameters are to be given for sizes of task own return stack and parameter stack, in address units, should be at least 64 for each of them. HERE and PAD called from within a task will give addresses inside the task. If scratch pad area shall be used inside the task, its size should be taken into account of the size for the parameter stack.

7.6. Directory access

```
OPEN-DIR  ( a n method -- dir-id err#)
```

Given a string in data memory by its parameters, and the value for a method (which will be ignored), an access will be tried to be opened to a directory on the drive of which the name is contained in the string. An internal identification value and 0 as top of stack will be returned if the operation succeeded, elsewise a value different from 0 as top of stack and a descardable value at next position on stack.

```
NEXT-DIR  ( dir-id -- dir-a err#)
```

Given a directory identification as returned by `OPEN-DIR`, the next entry from the directory on the associated drive is read and the address of a specfic storage area is returned together with 0 if the operation succeeded, elsewise a non-zero value and a discardable value at next position on stack.

```
CLOSE-DIR  ( dir-id -- err#)
```

Given a directory identification as returned by `OPEN-DIR`, all other access to the directory on the associated drive by help of this value is discarded. Temporarilly allocated heap storage area is set free again.

```
>FATTR  ( dir-a -- a2)
```

In the specific storage area as given by `READ-DIR` the address of a character field is calculated, from which a value may be read (by `C@`) which should reflect the attribute which the OS stored as associated with the file of which the directory entry just has been read.

```
>FTIME&DATE ( dir-a -- ss mm hh dd mm yy)
```

From the specific storage area as given by `READ-DIR`, six values for time and date are calculated which are associated with the file to which the just read directory entry belongs to. The values are returned the same way as with forth facility word `TIME&DATE`, with the values for seconds, minutes, hour, day, month, year on stack, the value for the year of the date being on top of the stack.

```
>FSIZE  ( dir-a -- a2)
```

In the specific storage area as given by `READ-DIR`, the address of a double-cell field is calculated, in which the size of the file may be read (by `2@`) of which the directory entry just has been read. After an `OPEN-DIR` but before a `READ-DIR` this will give the amount of free memory available on the medium.

```
>FNAME  ( dir-a -- a2)
```

In the specific storage area as given by `READ-DIR`, the address of a counted string is calculated, which holds the name of the file of which the directory entry just has been read. A counted string may be converted to the parameters of a string with address and length by applying `COUNT` to the address of a counted string as parameter. After an `OPEN-DIR` but before a `READ-DIR` this will give the name the medium has been given when it was formatted.

```
DIR  ( -- //"name")
```

The directory on a drive is shown of which <name> is given, with names of files, sizes, and asterisks if they are executables.

```
HEADER-READ  ( id -- dir-a err#)
```

QDOS Header information of an open file is read into an area starting at "dir-a". Tools for queries from directory entries may be applied; information for file-size is mapped as a double-cell value.

```
HEADER-WRITE  ( id -- err#)
```

QDOS Header information of an open file is written back using "dir-a" as obtained by `HEADER-READ`. If header information for data allocation is to be changed, the adequate value should be written as a single cell value into offset 6 of "dir-a" with `!`. If header information for file access is to be changed, the adequate value (1 for an executable, 0 for a data file) should be written into offset 5 of "dir-a" as a character value with `C!`.

8. The block File Editor

Before editing, first a block file has to be assigned to be the default block file for the interpreter, using `USE <name>`.

If while editing more files have been opened, so the last one edited will be the one which is assigned to be the default block file after leaving the editor.

By invoking the editor, an "ID" is asked for, holding two characters; this will be put into the top line of each changed block while editing, if this line was not empty. The ID is put together with current date in the form "ddmmmyyid" right justified. If the ID query is skipped by only hitting the enter-key, so no ID’s will be written. If an ID is active, the query is skipped.

Pushed lines and find-strings are preserved if the editor is left and entered again. By entering the editor, state of the debugger and values on the stack remain untouched except for the case of read/write/open failures.

Cut&Paste between files may be done either by Push&Pop for lines, or by setting markers as "from" and "up to" for blocks, then switching to target file, then "copy marked blocks" and "delete marked blocks". There should be no restriction on size of copied areas other than dictated by the medium. Files will be resized, cut (if TK2 is workable) or enlargened as necessary.

The functions for insertion of one block or deletion of current block only work if dummy markers are set. Deletion of current block will leave the last block of the file empty. Insertion first will use an empty last block, before enlargening the file.

By using a block file with a size smaller than 1024 bytes, that is less than one full block, this block probably is filled with 0’s instead of with blanks â’ following the default behaviour of the block cache system of the interpreter. These lines should be deleted and the file should be flushed then, so the size is reset to be 1024 bytes. Insertion of one block after that only will work then, if this "block 0" is not left empty (that is: at least a backslash as first character in first line may be written). Block 0 never will be taken as input for the interpreter, it is always a comment block. Block 1 for usual is the block containing loading instructions for random access into the file.

```
ED  ( --)
```

Start editing with the block of which the number is held in variable `SCR`. By default this is 1, at start-up this is 0, elsewise the number of the last block edited.

```
EDIT  ( blk --)
```

Start editing with the block of which the number is given as a parameter.

```
ID-OFF  ( --)
```

Cancel a previously put-in ID.

```
ALL-FLUSH  ( --)
```

Flush all files open for the editor from within the interpreter.

```
ALL-CLOSE  ( --)
```

Close all files open for the editor except the one last edited, which is the default block file for the interpreter.

8.1. Keys

The following keys are assigned to: (A=ALT, S=Shift, ^=CTRL)

Handle cursor

```
                    jump:
  left   right      column
A-left A-right      start/end of line
S-left S-right      word

up     down         row
S-up   S-down       top/bottom of screen

Tab    S-Tab        +/-  8 columns
Enter               next line, 1st column
```

Handle chars

```
^left  ^right       delete
^H                  delete
^Q                  toggle mode for insert/overwrite
```

Handle lines

```
^N insert
^Y delete
^Z delete
^O clear-EOL

^S split
^M join if next line fits

F3 push (buffer may hold 32 lines)
F4 push&copy
F5 pop if line fits
```

Handle screens

```
 ^up    page back
 ^down  page ahead
A-up    page back
A-down  page ahead

S-A-up    1st screen of file
S-A-down  last screen of file
```

```
S-F5     "shadow" screen â’  toggle half of file-size back/ahead

^U       undo if updated
^D       delete current screen if marker set (ignore marker)
^T       insert one screen if marker set (ignore marker)
```

Find/Replace

```
F2  S-F2   input & find ahead / backwards
^L  ^K     find next / previous

( not implemented, but reserved keys: )
( ^R  ^E   find&replace next / previous)
```

Handle markers

```
  F1  goto
A-F1  set
A-F2  reset

A-F3  copy marked blocks
A-F4  delete marked blocks
```

Handle files

```
 ^F3   open
 ^F4   close

S-F3   swap
S-F4   roll

 ^F    flush all open files
 ^B    make a backup
```

Leave editor

```
 ESC modified & flushed
^X   unmodified if updated
```

9. Table of optimizing compiler actions

1.a

```
   exe-An
       & exe-B
     ==> exe-Cn
```

1.b

```
     A[  exe-A1  exe-A2  exe-A3 ... ]
     B=  ............................ exe-B
     C[  exe-C1  exe-C2  exe-C3 ... ]
```

1.c

```
     A[  R>    SWAP ROT     2DROP ]
     B=  ......................... DROP
     C[  RDROP NIP  ROTDROP 3DROP ]

     A[  2R>       2R>/SWAP  R>/RDROP RDROP/R>  2R@      ]
     B=  ................................................ DROP
     C[  RDROP/R>  R>/RDROP  2RDROP   2RDROP    2R@/DROP ]

     A[  NIP     DROP     2R>       2R>/SWAP  2R@ ]
     B=  ........................................ NIP
```

```
C[  NIPNIP  DROPNIP  R>/RDROP  RDROP/R>  R@  ]

A[  ROT      OVER      2DUP       R>        0       2R>/SWAP  ]
B=  ..................................................... SWAP
C[  ROTSWAP  OVERSWAP  2DUP/SWAP  R>/SWAP   0/SWAP  2R>       ]

A[  ROT   SWAP    DUP   ]
B=  ....................... ROT
C[  ROTROT  SWAPROT  DUPROT ]

A[  DROP  2DROP  2SWAP  R>  2R>  2R>/SWAP ]
B=  ............................................ 2DROP
C[  3DROP  4DROP  2NIP   R>/2DROP  2RDROP  2RDROP   ]

A[  2R>     ]
B=  ........... 2SWAP
C[  2R>/2SWAP ]

A[  2SWAP  2OVER ]
B=  ............. 2OVER
C[  2TUCK  4DUP  ]

A[  2ROT  ]
B=  ....... 2ROT
C[  -2ROT ]

A[  DUP     SWAP     >R        SWAP/>R ]
B=  ................................... >R
C[  DUP/>R  SWAP/>R  SWAP/2>R  2>R     ]

A[  >R  ]
B=  ........ R@
C[  DUP/>R ]

A[  R>     DROP    ]
B=  .................. R>
C[  2R>/SWAP  DROP/R> ]

A[  2DUP      2SWAP     ]
B=  ..................... 2>R
C[  2DUP/2>R  2SWAP/2>R ]

A[  DUP   ]
B=  ....... @
C[  DUP/@ ]

A[  2DUP    ]
B=  ......... C!
C[  2DUP/C! ]

A[  SWAP    OVER    TUCK    DUPROT   2DUP   ]
B=  ..................................... !
C[  SWAP/!  OVER/!  TUCK/!  DUPROT/! 2DUP/! ]

A[  OVER    TUCK    2DUP    CELLS    @   ]
B=  ................................... +
C[  OVER/+  TUCK/+  2DUP/+  CELLS/+  @/+ ]

A[  OVER    TUCK    2DUP    2DUP/SWAP    CELLS    @   ]
B=  ............................................... -
C[  OVER/-  TUCK/-  2DUP/-  2DUP/SWAP/-  CELLS/-  @/- ]
```

```
A[  2SWAP    ]
B=  .......... D-
C[  2SWAP/D- ]

A[  0<   0>   =   <>  <    >   U<   U>  ]
B=  ..................................... 0=
C[  0>=  0=<  <>  =   >=  =<  U>=  U=< ]

A[  2DUP/=   2DUP/<>  2DUP/- 2DUP/<   2DUP/>   2DUP/U<   2DUP/U>  ]
B=  ............................................................... 0=
C[  2DUP/<>  2DUP/=   2DUP/= 2DUP/>=  2DUP/=<  2DUP/U>=  2DUP/U=< ]

A[  D0=   D0<  D=  D<  DU<  ]
B=  ......................... 0=
C[  D0<>  D0>= D<> D>= DU>= ]

A[  1        ]
B=  .......... /STRING
C[  1/STRING ]

A[  0      BL    ]
B=  .............. FILL
C[  ERASE  BLANK ]

A[  @       ]
B=  ......... EXECUTE
C[  PERFORM ]

A[  DROP      1       0      -1      FALSE   TRUE   ]
B=  ................................................ EXIT
C[  DROP/EXIT 1&EXIT  0&EXIT -1&EXIT 0&EXIT -1&EXIT ]
```

2.a

```
exe-A
    & exe-Bn
==> exe-Cn
```

2.b

```
 A=  exe-A
 B[  exe-B1  exe-B2  exe-B3 ... ]
 C[  exe-C1  exe-C2  exe-C3 ... ]
```

2.c

```
A=  DROP
B[  SWAP       DUP       0      ]
C[  DROP/SWAP  DROP/DUP  DROP/0 ]

A=  SWAP
B[  <  >  U<  U>  -       OVER  MOVE      ]
C[  >  <  U>  U<  SWAP/-  TUCK  SWAP/MOVE ]

A=  2DUP
B[  =        <>       -       <        >      ]
C[  2DUP/=   2DUP/<>  2DUP/-  2DUP/<   2DUP/> ]

A=  2DUP
B[  U<       U>       AND       OR       XOR      ]
C[  2DUP/U<  2DUP/U>  2DUP/AND  2DUP/OR  2DUP/XOR ]

A=  2DUP/SWAP
B[  <       >        U<       U>       -            ]
```

```
    C[  2DUP/>  2DUP/<   2DUP/U>  2DUP/U<  2DUP/SWAP/-  ]
```

3.a

```
       XBRA[  IF  WHILE  UNTIL  ]
```

3.b

```
       rel-exe        rel-exe

       &  XBRA        &  0=
       ==>  rel/0BRA  &  XBRA
       ==>  rel/?BRA
```

3.c

```
 rel-exe            rel/0BRA        rel/?BRA

   PAUSE          PAUSE/0BRA      PAUSE/?BRA
      0<             0</0BRA         0</?BRA
      0>             0>/0BRA         0>/?BRA
       =              =/0BRA          =/?BRA
      <>             <>/0BRA         <>/?BRA
       <              </0BRA          </?BRA
       >              >/0BRA          >/?BRA
      U<             U</0BRA         U</?BRA
      U>             U>/0BRA         U>/?BRA
  WITHIN         WITHIN/0BRA     WITHIN/?BRA
       -             <>/0BRA         <>/?BRA
     AND            AND/0BRA        AND/?BRA
      OR             OR/0BRA         OR/?BRA
     XOR            XOR/0BRA        XOR/?BRA
    ?DUP           ?DUP/0BRA       ?DUP/?BRA
     DUP            DUP/0BRA        DUP/?BRA
    SWAP           SWAP/0BRA       SWAP/?BRA
   CASE?          CASE?/0BRA
   2DUP/=         2DUP/=/0BRA     2DUP/=/?BRA
   2DUP/<>       2DUP/<>/0BRA    2DUP/<>/?BRA
   2DUP/-        2DUP/<>/0BRA    2DUP/<>/?BRA
   2DUP/<         2DUP/</0BRA     2DUP/</?BRA
   2DUP/>         2DUP/>/0BRA     2DUP/>/?BRA
   2DUP/>=        2DUP/</?BRA     2DUP/</0BRA
   2DUP/=<        2DUP/>/?BRA     2DUP/>/0BRA
   2DUP/U<       2DUP/U</0BRA    2DUP/U</?BRA
   2DUP/U>       2DUP/U>/0BRA    2DUP/U>/?BRA
   2DUP/U>=      2DUP/U</?BRA    2DUP/U</0BRA
   2DUP/U=<      2DUP/U>/?BRA    2DUP/U>/0BRA
   2DUP/AND     2DUP/AND/0BRA   2DUP/AND/?BRA
   2DUP/OR       2DUP/OR/0BRA    2DUP/OR/?BRA
   2DUP/XOR     2DUP/XOR/0BRA   2DUP/XOR/?BRA
   D0=               D0=/0BRA        D0=/?BRA
   D0<               D0</0BRA        D0</?BRA
   D=                 D=/0BRA         D=/?BRA
   D<                 D</0BRA         D</?BRA
   DU<               DU</0BRA        DU</?BRA
```

3.d

```
       rel-exe        rel-exe

       &  XBRA        &  0=
       ==>  rel/?BRA  &  XBRA
       ==>  rel/0BRA
```

3.e

```
   rel-exe        rel/?BRA        rel/0BRA

      0>=         0</?BRA         0</0BRA
      0=<         0>/?BRA         0>/0BRA
       >=         </?BRA          </0BRA
       =<         >/?BRA          >/0BRA
      U>=         U</?BRA         U</0BRA
      U=<         U>/?BRA         U>/0BRA
     D0<>         D0=/?BRA        D0=/0BRA
     D0>=         D0</?BRA        D0</0BRA
      D<>         D=/?BRA         D=/0BRA
      D>=         D</?BRA         D</0BRA
     DU>=         DU</?BRA        DU</0BRA
```

4.a

```
        exe-An
           &  exe-Bn
          ==>  exe-Cn
```

4.b

```
     A[  exe-A1  exe-A2  exe-A3 ... ]
     B[  exe-B1  exe-B2  exe-B3 ... ]
     C[  exe-C1  exe-C2  exe-C3 ... ]
```

4.c

```
     BRA[ AHEAD  ELSE  AGAIN  REPEAT  LEAVE ]
```

4.d

```
     A[  PAUSE      PAUSE  PAUSE       0    0    ]
     B[  BRA      LOOP  +LOOP       DO   ?DO  ]
     C[  PAUSE/BRA  PAUSE/LOOP  PAUSE/+LOOP  0DO  ?0DO ]
```

10. Table of Forth words

10.1. Data stack handling

10.1.1. CORE

```
DROP       ( n --)
DUP        ( n -- n n)
SWAP       ( n1 n2 -- n2 n1)
OVER       ( n1 n2 -- n1 n2 n1)
ROT        ( n1 n2 n3 -- n2 n3 n1)
?DUP       ( 0 -- 0  // n -- n n)
2DROP      ( n1 n2 --)
2DUP       ( n1 n2 -- n1 n2 n1 n2)
2SWAP      ( n1 n2 n3 n4 -- n3 n4 n1 n2)
2OVER      ( n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2)
DEPTH      ( ix -- ix n)
```

10.1.2. EXT

```
NIP        ( n1 n2 -- n2)
TUCK       ( n1 n2 -- n2 n1 n2)
PICK       ( nx .. np .. n0 p -- nx .. np .. n0 np)
ROLL       ( nx .. np .. n0 p -- nx .. np-1 .. n0 np)
```

10.2. Return stack handling

10.2.1. CORE

```
EXIT        ( -- R: i --)
EXECUTE     ( cfa -- R: -- i)
>R          ( n -- R: -- n)
R>          ( -- n R: n --)
R@          ( -- n R: n -- n)
```

10.2.2. EXT

```
2>R         ( n1 n2 --  R: n1 n2)
2R>         ( -- n1 n2  R: n1 n2)
2R@         ( -- n1 n2  /R: n1 n2 -- n1 n2)
```

10.2.3. DOUBLE EXT

```
2ROT        ( n1 n2 n3 n4 n5 n6 -- n3 n4 n5 n6 n1 n2)
```

10.3. Arithmetic calculation

10.3.1. CORE

```
=           ( n1 n2 -- f)
U<          ( n1 n2 -- f)
<           ( n1 n2 -- f)
>           ( n1 n2 -- f)
0=          ( n -- f)
0<          ( n -- f)
AND         ( n1 n2 -- n3)
OR          ( n1 n2 -- n3)
XOR         ( n1 n2 -- n3)
INVERT      ( n1 -- n2)
LSHIFT      ( n1 n2 -- n3)
RSHIFT      ( n1 n2 -- n3)
+           ( n1 n2 -- n3)
-           ( n1 n2 -- n3)
NEGATE      ( n -- n)
ABS         ( n1 -- n2)
MIN         ( n1 n2 -- n3)
MAX         ( n1 n2 -- n3)
```

10.3.2. EXT

```
<>          ( n1 n2 -- f)
U>          ( n1 n2 -- f)
0<>         ( n -- f)
0>          ( n -- f)
WITHIN      ( n1 n2 n3 -- f)
```

10.3.3. DOUBLE

```
D=          ( d1 d2 -- f)
D+          ( d1 d2 -- d3)
DNEGATE     ( d1 -- d2)
DABS        ( d1 -- d2)
```

```
D<          ( d1 d2 -- f)
D0=         ( d1 -- f)
D0<         ( d1 -- f)
DMIN        ( d1 d2 -- d3)
DMAX        ( d1 d2 -- d3)
M+          ( d1 n -- d2)
D-          ( d1 d2 -- d3)
```

10.3.4. EXT

```
DU<         ( d1 d2 -- f)
```

10.4. Arithmetic constants

10.4.1. CORE

```
ALIGNED     ( a -- a)
1+          ( n -- n+1)
1-          ( n -- n-1)
2*          ( n -- n*2)
2/          ( n -- n/2)
CELL+       ( a -- a+)
CELLS       ( n -- n+)
CHAR+       ( a -- a+)
CHARS       ( n -- n+)
S>D         ( n -- d)
>BODY       ( cfa -- a2)
```

10.4.2. EXT

```
TRUE        ( -- -1)
FALSE       ( -- 0)
```

10.4.3. DOUBLE

```
D>S         ( d -- n)
D2*         ( d1 -- d2)
D2/         ( d1 -- d2)
```

10.4.4. STRING

```
/STRING     ( a1 u1 n1 -- a1+ u1-)
```

10.5. Memory access

10.5.1. CORE

```
C@          ( a -- b)
C!          ( b a --)
@           ( a -- n)
!           ( n a --)
2@          ( a -- d)
2!          ( d a --)
+!          ( n a -)
MOVE        ( a1 a2 n --)
FILL        ( a n b --)
COUNT       ( a$ -- a+1 n)
```

10.5.2. EXT

```
ERASE      ( a n --)
```

10.5.3. STRING

```
CMOVE      ( a1 a2 n --)
CMOVE>     ( a1 a2 n --)
BLANK      ( a n --)
-TRAILING  ( a1 n1 -- a1 n2)
COMPARE    ( a1 n1 a2 n2 --  -1/0/1)
SEARCH     ( a1 n1 a2 n2 -- a3 n3 f)
```

10.6. Enhanced arithmetics

10.6.1. CORE

```
UM/MOD     ( d n1 -- u2/rem u3/div)
FM/MOD     ( d n1 -- n2 n3)
SM/REM     ( d n1 -- n2 n3)
UM*        ( u1 u2 -- ud)
M*         ( n1 n2 -- d)

\ missing
/MOD       ( n1 n2 -- n3 n4)
/          ( n1 n2 -- n3)
MOD        ( n1 n2 -- n3)
*          ( n1 n2 -- n3)
*/MOD      ( n1 n2 n3 -- n4 n5)
*/         ( n1 n2 n3 -- n4)
```

10.6.2. DOUBLE

```
M*/        ( d1 n1 n2 -- d2)
```

10.7. System values

10.7.1. CORE

```
BL         ( -- 32)
>IN        ( -- a)
SOURCE     ( -- a n)
STATE      ( -- a)
BASE       ( -- a)
```

10.7.2. SEARCH

```
FORTH-WORDLIST   ( -- wid)
```

10.7.3. BLOCK

```
BLK        ( -- a)
```

10.7.4. BLOCK EXT

```
SCR         ( -- a)
```

10.7.5. FILE

```
BLOCK-FID   ( -- a)
SOURCE-ID   ( -- n)
```

10.8. Value access

10.8.1. CORE

```
HERE        ( -- a)
ALLOT       ( n --)
ALIGN       ( --)
C,          ( b --)
,           ( n --)
```

10.8.2. EXT

```
PAD         ( -- a)
COMPILE,    ( cfa --)
UNUSED      ( -- u)
ENVIRONMENT?      ( -- 0)
```

10.9. Number conversion

10.9.1. CORE

```
DECIMAL     ( --)
>NUMBER     ( d a n -- d a+ n-)
<#          ( --)
HOLD        ( b --)
SIGN        ( n --)
#           ( d1 -- d2)
#S          ( d1 -- d2)
#>          ( d -- a n)
```

10.9.2. EXT

```
HEX         ( --)
```

10.10. Output

10.10.1. CORE

```
TYPE        ( a n --)
EMIT        ( b --)
SPACE       ( --)
SPACES      ( n --)
CR          ( --)
.           ( n --)
U.          ( n --)
```

10.10.2. EXT

```
.(          ( --)
.R          ( n1 n2 --)
U.R         ( n1 n2 --)
```

10.10.3. DOUBLE

```
D.          ( d --)
D.R         ( d n --)
```

10.10.4. FACILITY EXT

```
EMIT?       ( -- f)
PAGE        ( --)
AT-XY       ( n1 n2 --)
```

10.11. Search order

10.11.1. CORE

```
FIND              ( $a -- a 0 , cfa -2/-1/1/2)
```

10.11.2. SEARCH

```
SEARCH-WORDLIST   ( a n wid -- 0 , cfa -2/-1/1/2)
GET-ORDER         ( -- widn..wid1 n)
SET-ORDER         ( widn..wid1 n --)
GET-CURRENT       ( -- wid)
SET-CURRENT       ( wid --)
DEFINITIONS       ( --)
```

10.11.3. EXT

```
ALSO              ( --)
PREVIOUS          ( --)
FORTH             ( --)
ONLY              ( --)
```

10.11.4. TOOLS EXT

```
ASSEMBLER   ( --)
EDITOR      ( --)
```

10.12. Input & parsing

10.12.1. CORE and FILE

```
KEY         ( -- b)
ACCEPT      ( a n1 -- n2)
WORD        ( b -- a$)
CHAR        ( -- b)
'           ( -- cfa)
(           ( --)
S"          ( -- a n)
\                ( --)
```

```
PARSE            ( b -- a n)
RESTORE-INPUT    ( nx ... n1 x -- f)
SAVE-INPUT       ( -- nx ... n1 x)
REFILL           ( -- f)
```

10.12.2. FACILITY EXT

```
KEY?       ( -- f)
EKEY?      ( -- f)
EKEY       ( -- b)
```

10.12.3. TOOLS EXT

```
[THEN]     ( n --)
[ELSE]     ( n -- n)
[IF]       ( f -- n)
```

10.13. Error handling

10.13.1. ERROR

```
CATCH      ( cfa  --  0 / #-throw)
THROW      ( # --)
```

10.14. Blocks handling

10.14.1. BLOCK

```
BUFFER           ( n -- a)
BLOCK            ( n -- a)
UPDATE           ( --)
SAVE-BUFFERS     ( --)
FLUSH            ( --)
```

10.14.2. BLOCK EXT

```
EMPTY-BUFFERS    ( --)
LIST             ( n --)
```

10.15. File handling

10.15.1. FILE

```
R/W              ( -- n)
R/O              ( -- n)
FILE-POSITION    ( hdl -- d err#)
REPOSITION-FILE  ( d hdl -- err#)
FILE-SIZE        ( hdl -- d err#)
RESIZE-FILE      ( d hdl -- err#)
CLOSE-FILE       ( hdl -- err#)
OPEN-FILE        ( a n1 n2 -- hdl err#)
CREATE-FILE      ( a n1 n2 -- hdl err#)
DELETE-FILE      ( a n - err#)
READ-FILE        ( a n1 hdl -- n2 err#)
WRITE-FILE       ( a n hdl -- err#)
READ-LINE        ( a n hdl -- n f err#)
```

```
WRITE-LINE       ( a n hdl -- err#)
```

10.15.2. EXT

```
FLUSH-FILE       ( hdl -- err#)
RENAME-FILE      ( a1 n1 a2 n2 -- err#)
FILE-STATUS      ( a1 n1 -- n2 err#)
```

10.16. Loop engine

10.16.1. CORE

```
QUIT       ( --  /R: ix -- iy)
ABORT      ( ix -- iy /R: ix -- iy)
EVALUATE   ( a n --)
```

10.16.2. BLOCK EXT

```
THRU       ( u1 u2 --)
LOAD       ( n --)
```

10.16.3. FILE

```
INCLUDE-FILE     ( hdl  --)
INCLUDED         ( a n --)
```

10.17. Code compiler

10.17.1. CORE

```
]          ( --)
[          ( --)
:          ( -- /R: -- i  /C: --  /"name")
DOES>      ( /Runtime:      -- a  /R: i -- )
           ( /Generation:   --    /R: i -- )
           / /Compile:      --    /R:  -- i)
;          ( -- /R: i --  /C: --)
RECURSE    ( --)
IMMEDIATE  ( --)
LITERAL    ( -- n  /C: n --)
[CHAR]     ( -- n  /C: -- /"name")
[']        ( -- n  /C: -- /"name")
POSTPONE   ( --  /C: --  /"name")
S"         ( /Interpret: -- a n)
           ( /Runtime:   -- a n)
           ( /Compile:   -- /"string")
."         ( --  /C: -- /"string")
ABORT"     ( --  /C: -- /"string")
```

10.17.2. EXT

```
:NONAME    ( -- /R: -- i  /C: -- cfa)
C"         ( -- $a  /C: -- /"string")
```

10.17.3. DOUBLE

```
2LITERAL   ( -- d  /C: d --)
```

10.17.4. TOOLS EXT

```
CODE       ( -- n  /"name")
```

10.17.5. LOCAL

```
(LOCAL)    ( /C: a n/0 --)
LOCAL|     ( /C: -- /"name name ... |")
TO         ( n --  /C: -- /"name")
```

10.18. Flow of control

10.18.1. CORE

```
IF         ( f --  /C: -- a)
THEN       ( --  /C: a --)
ELSE       ( --  /C: a1 -- a2)
BEGIN      ( --  /C: -- a)
UNTIL      ( f --  /C: a --)
WHILE      ( f --  /C: a1 -- a2 a1)
REPEAT     ( --  /C: a1 a2 --)
DO         ( n2 n1 -- /R: ix/nodo -- iy/do  /C: -- a)
LOOP       ( --  /R: iy/do -- [iy/do]|[ix/nodo]  /C: a --)
+LOOP      ( n --  /R: iy/do -- [iy/do]|[ix/nodo]  /C: a --)
I          ( -- n  /R: iy/do -- iy/do)
UNLOOP     ( --  /R: iy/do -- ix/nodo)
LEAVE      ( --  /R: iy/do -- ix/nodo)
```

10.18.2. EXT

```
AGAIN      ( -- /C: a --)
?DO        ( n2 n1 -- /R: ix/nodo -- [ix/nodo]|[iy/do]  /C: -- a)
J          ( -- n /R: ix/dodo -- ix/dodo)
```

10.18.3. TOOLS EXT

```
AHEAD      ( --  /C: -- a)
CS-PICK    ( ax..a1 x -- ax..a1 ax)
CS-ROLL    ( ax ax-1..a1 x -- ax-1..a1 ax)
```

10.19. Data compiler

10.19.1. CORE

```
CREATE     ( --  /"name" /Run: -- a)
CONSTANT   ( n -- /"name"  /Run: -- n)
VARIABLE   ( -- /"name"  /Run: -- a)
```

10.19.2. EXT

```
VALUE       ( n --  /"name"  /Run: -- n)
TO          ( n -- /"name"  /C: -- /"name")
MARKER      ( -- /"name")
```

10.19.3. DOUBLE

```
2CONSTANT   ( d --  /"name"  /Run: -- d)
2VARIABLE   ( --  /"name"  /Run: -- a)
```

10.19.4. SEARCH

```
WORDLIST    ( -- wid)
```

10.20. Tools

10.20.1. TOOLS EXT

```
DUMP        ( a n --)
?           ( a --)
.S          ( --)
WORDS       ( -- /"char")
SEE         ( -- /"name")
BYE         ( --)
```

10.20.2. SEARCH EXT

```
ORDER       ( --)
```

10.20.3. FACILITY EXT

```
MS          ( n --)
TIME&DATE   ( -- n1 n2 n3 n4 n5 n6)
```

10.20.4. MEMORY

```
ALLOCATE    ( n -- a err#)
FREE        ( a -- err#)
RESIZE      ( a n -- a err#)
```

---

1. See the Asciidoctor website.
2. This document is part of Brouhabouha Forth *repackaged*, which can be downloaded from http://programandala.net/en.program.brouhabouha_forth.html or http://www.dilwyn.me.uk/language/index.html.
3. See the final documentation of ANS Forth (also known as Forth 94), and the new Forth-2012 Standard.
4. The email at ist.tu-graz.ac.at does not exist anymore.
5. See the web edition of *Starting Forth* by Marcel Hendrix and the web edition of *Starting Forth* by forth.com.
6. Including the Forth data space, and the Forth heap from which `allocate` reserves memory.

Last updated 2016-01-21 18:27:17 CET