# v-FORTH 1.52

ZX Spectrum Next version

1990-2022 Matteo Vitturi

...oOo...

Introduction
&
Technical Information

...oOo...

**Build 20220827**

# 1 Foreword

This document introduces a Forth implementation suitable to run on **Sinclair ZX Spectrum Next**.

This is in essence a FIG-Forth ported to the new **Sinclair ZX Spectrum Next** based on my previous work **v-Forth 1.413** available at https://sites.google.com/view/vforth/vforth1413 and at https://github.com/mattsteeldue/vforth.

This version **v-Forth 1.52** is available at https://sites.google.com/view/vforth/vforth15-next and also on the GitHub repository at https://github.com/mattsteeldue/vforth-next. The main difference from the previous version is that it uses a dedicated file on SD instead of on ZX Microdrive cartridges to provide a Block/Screen facility. Though this is a working and functional piece of software, the porting is still "work-in-progress", as there are many things to do.

Starting with this version **v-Forth 1.52**, the behavior of VARIABLE is "standard" and doesn't need an initial value.

Since sub-version **v-Forth 1.51**, this Forth comes with two flavors: Direct-Threaded or Indirect-Threaded code. Direct-Threaded offers some 25% of more speed at the cost of more memory allocation for each colon-definition. See § 5 for some technical detail.

## Disclaimer

**Copying, modification and distribution of this software is allowed provided this copyright notice is kept.**
**This work is available "as-is" with no warranty whatsoever.**
**Changes may occur at any time without notification.**

I, the author am not a native English speaker and you, very likely, will find grammatical errors. In this case, it would be kindly appreciated if you could drop me a line with any suggestion and/or correction at *matteo* -underscore-*vitturi@yahoo.com.* I am not able to write a longer disclaimer than the above.

## Acknowledgment

Special thanks goes to Roland Herrera that helped to edit this whole documentation.

## 1.1 Document structure

**Chapter 2** describes how to install, activate and get acquainted with the Forth environment.
**Chapter 3** lists some important utilitiy and library you should be aware while using Forth, most of them can be imported in your session using `NEEDS`.
**Chapter 3.1:** Since a quick way of coding in Forth the old fashion Screen/Block facility, a Full Screen Editor `EDIT` is available, but as a "work-in-progress" to edit any text file, there's the Large file EDitor aka `LED` (**Chapter 3.3**) that is able to manage files as large as 17.568 rows, 85 characters per row;
**Chapter 3.2** presents `GRAPHICS` Modes and Layers library along with Color and Attributes management definitions;
**Chapter 3.4** introduces the Interrupt Service Routine library: see the demo/color-picker.f demo a an example of interrupt-driven mouse cursor.
**Chapter 3.5** continues the old fashion Block oriented Search and Locate Utility;
**Chapter 3.6** explains the inner parts of Forth introducing the `SEE` Debugger Utility.
**Chapter 3.7** shows how to exploit the Standard-ROM floating point calculator.
**Chapter 3.8** keeps track of the obsolete Line Oriented Editor on which is based the aforementioned Full Screen Editor.
**Chapter 4** gives some deeper insight and technical information.
**Chapter 5** is a straight list of error messages.
**Chapter 6** provides detailed information of Forth Dictionary where each definition is explained in a formal way. There is
**Chapter 6.1** the "core" dictionary, where almost all definitions are available at `COLD` start, then
**Chapter 6.2** introduces the optional set of definitions that provides the Case-Of structure.
**Chapter 6.3** introduces some experimental Heap Memory Facility to access the huge quantity of memory.
**Chapter 6.4** introduces the Testing Suite, to show that vForth *wants to comply* to modern Standard.

## 1.2 Legend

**Courier New** font is used whenever a Forth definition is referenced or to indicate some typed-in source code.

**Calibri** font is used elsewhere

All definitions explained in the dictionary pages are introduced in the form:

**A-WORD n1 n2 ... --- n3 n4 ...**

where "**n1 n2 ...**" represents the Stack status before **A-WORD** is executed, and "**n3 n4 ...**" represent the Stack status after **A-WORD** is executed. Special behaviour, such as IMMEDIATE definitions are explained properly.

| | | |
|---|---|---|
| **a** | memory address | 16 bits |
| **b** | byte, small unsigned integer | 8 bits |
| **c** | character | 8 bits, but often only lower 7 are significant. |
| **d** | signed double integer | 32 bits |
| **fh** | file-handle | 8 bits |
| **fp** | floating point number | 32 bits |
| **ha** | heap-pointer address (see >FAR) | 16 bits. |
| **n** | signed integer | 16 bits |
| **u** | unsigned integer | 16 bits |
| **ud** | unsigned double integer | 32 bits |
| | | |
| **f** | flag: a number evaluated as a boolean | 16 bits |
| **ff** | false flag: zero | 16 bits |
| **tf** | true flag: non-zero | 16 bits |
| | | |
| **nfa** | name field address | 16 bits |
| **lfa** | link field address | 16 bits |
| **cfa** | code field address | 16 bits |
| **pfa** | parameters field address | 16 bits |
| **xt** | execution token – same as cfa | 16 bits |

| | |
|---|---|
| **cccc** | character string or word name available in the vocabulary |
| **...** | a list of words |
| TOS | top of calculator stack |

.

# 2 Getting started

## 2.1 Installation

The most recent version of this software can be downloaded from GitHub repository as .zip file at

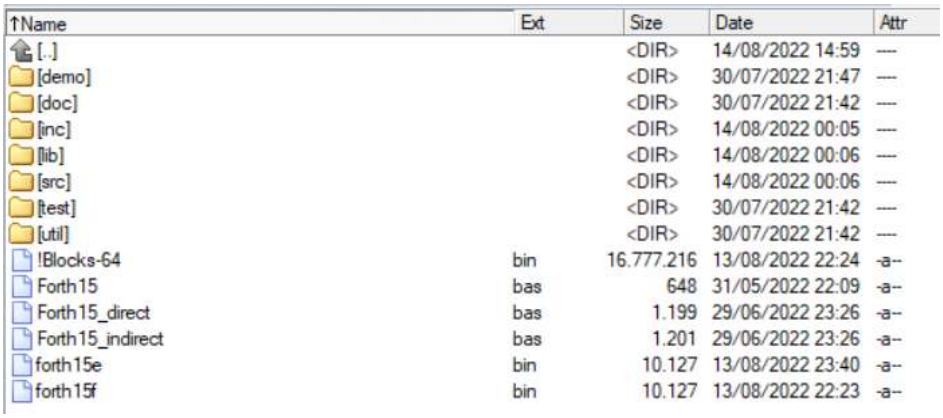
https://github.com/mattsteeldue/vforth-next/tree/master/download

The same executable programs are available in the same repository:

https://github.com/mattsteeldue/vforth-next/tree/master/SD/tools/vforth

Unzip or copy the software to "C:/tools/vForth" directory inside your Next's SD card so it appears in the following directory hierarchy:

| | |
|---|---|
| doc/ | where I keep some text image-versions of !Blocks-64.bin |
| inc/ | contains text-file of single word definitions available after you type `NEEDS CCCC`. |
| lib/ | same as inc/ but these text-file are a collection of several words that forms a "library utility", e.g. `SEE`. |
| src/ | among others, the source file of this Forth System. You can even recompile new builds. |
| test/ | contains an adaptation of John Hayes' Test Suite that tries to make this Forth more *standard*. |
| util/ | with some Perl script to manage with !Blocks-64.bin file I collect over the time |
| demo/ | some useful demo |



| ↑Name | Ext | Size | Date | Attr |
|---|---|---|---|---|
| [..] | | <DIR> | 14/08/2022 14:59 | ---- |
| [demo] | | <DIR> | 30/07/2022 21:47 | ---- |
| [doc] | | <DIR> | 30/07/2022 21:42 | ---- |
| [inc] | | <DIR> | 14/08/2022 00:05 | ---- |
| [lib] | | <DIR> | 14/08/2022 00:06 | ---- |
| [src] | | <DIR> | 14/08/2022 00:06 | ---- |
| [test] | | <DIR> | 30/07/2022 21:42 | ---- |
| [util] | | <DIR> | 30/07/2022 21:42 | ---- |
| !Blocks-64 | bin | 16.777.216 | 13/08/2022 22:24 | -a-- |
| Forth15 | bas | 648 | 31/05/2022 22:09 | -a-- |
| Forth15_direct | bas | 1.199 | 29/06/2022 23:26 | -a-- |
| Forth15_indirect | bas | 1.201 | 29/06/2022 23:26 | -a-- |
| forth15e | bin | 10.127 | 13/08/2022 23:40 | -a-- |
| forth15f | bin | 10.127 | 13/08/2022 22:23 | -a-- |

If you wish to use a different directory instead of C:/tools/vforth, you need to modify the paths in the two above Basic programs, **Forth15_direct.bas** or **Forth15_indirect.bas** and **Forth15.bas**.

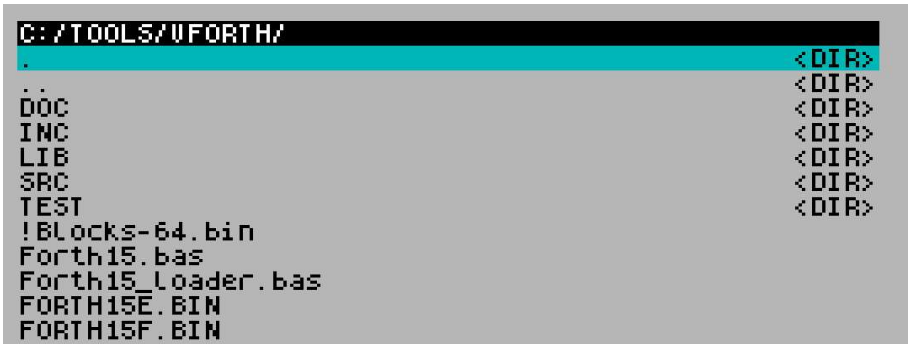
**Forth15_direct.bas** loads **Forth15f.bin** code that is the Direct-Threaded version.
**Forth15_indirect.bas** loads **Forth15e.bin** code that is the Indirect-Threaded version.

You can modify the Basic Program **Forth15_loader.bas** and set which you like better. See §5 "The inner-interpreter" for details. Indirect-Threaded version produces smaller programs, in general.

## 2.2 Activation

The Forth System is activated running a Basic program **C:/tools/vforth/forth15_loader.bas**.

This can be done using the Browser and selecting it, then clicking ENTER.




The Basic loader **forth15_loader.bas** frees upper memory setting RAMTOP to address 25345 and usually loads **forth15f.bin** (the Direct-Thread version Forth core) and then it loads a smaller Basic launcher **Forth15.bas** that you can customize for your purposes.

A Splash screen displays “Version number” and “Build date” followed by some technical system information that are obtained by executing **Screen # 11**.

Within a few seconds the system will ask if you would like to “Run Scr# 11 autoexec”: the only way to refuse is by using the **[N] key**. However, it's a good idea to allow Forth to continue to LOAD **Screen # 11** which in turn loads a few useful utilities making available, among the other words, two particular words: `EDIT` the “**Screen Editor**” and `SEE` the “**Debugger Inspector**”. This phase is executed only at *first* startup, but you can run it again using word `AUTOEXEC`

The Basic launcher **Forth15.bas** usually auto-starts the first time at `LINE 20`, so you usually won't notice, but just in case you `STOP` it or the Forth system encounters a ROM Error that forces it to suddenly return to Basic, you have two main choices:

a. type `RUN` at Basic prompt : This does a `WARM` Forth-start, preserving your previous work and buffers status.

b. type `RUN 20` at Basic prompt: This does a `COLD` start, that should reset all as if you had just loaded from SD card.

## 2.3 Case-insensitive and Case-sensitive option

By default, the Forth interpreter is *case-insensitive*, so you can type your commands using lower-case or upper-case or a mix of them with no difference. To enable or disable the case-sensitive interpretation you can use `CASEON` and `CASEOFF` definitions.

The case-insensitive option applies to the Intepreter's dictionary search only. Any new definition you are coding retains the exact case you coded them.

## 2.4 Block / Screen system and Text editor

This Forth System comes with a 16 Mbytes file named **!Blocks-64.bin** that provides a `BLOCK`-like mass-storage system to hold 32.767 `BLOCK`s (or 16,383 Screens) that can be edited using the "Full Screen Editor" utility available after you type:

```
NEEDS EDIT
```

Each `BLOCK` is 512 bytes and each Screen is 1 KByte long and can store text in **16 lines x 64 columns way** or can be used as a ***virtual-memory*** area where you can persistently store anything you like.

Accessing of the first blocks is faster than the others, so that reading the last BLOCKs available using, for example

```
DECIMAL 16383 LIST
```

it takes a noticeable amount of time. This may depend on how F_SEEK primitive is implemented.

## 2.5 Character size

In this Forth implementation I preferred **LAYER 1,2** display mode to allows 64 character per line: this is quite necessary to be able to display a whole 1024 characters in a single screen.

If you prefer **LAYER 1,1** you can add a line 61 in **Forth15.bas** wrapper as follow

```
61 LAYER 1,1: PRINT CHR$ 30; CHR$ 4;
```

to switch to **LAYER 1,1** and condensed character set. The result is quite poor in my opinion.

You can also change LAYER mode using some Layer-related definitions available after you type **NEEDS GRAPHICS** .

## 2.6 Source feeding

Before entering Forth, the Basic launcher could open text files via `OPEN#` for instance

```
OPEN# 13,"o>output.txt"
```

that can be later selected for output from Forth via `13 SELECT` to collect any output you send to this output-channel. To restore sending output to video there is an easy `VIDEO` definition that simply does `2 SELECT`.

You can modify the Basic launcher and add commands to `OPEN#` any other file *for read* so that it can be fed to Forth as a text source; for example you can add the following Basic line:

```
92 OPEN # 12,"src/Z80N-asm.f"
```

Later, this allows Forth to load such a source file using the following:

```
-12 LOAD
```

In this case, a negative number such as **-12** says `LOAD` Forth definition to start reading text from input stream #12 instead of loading from Screen # "-12", that doesn't exists. This feature, i.e., passing a negative "screen" number to `LOAD`, is not Forth standard, but an original idea of my own.

**Anyway, there is no more need to OPEN# a stream from Basic, since two new specific definitions allow you to include source from any file:** `INCLUDE` **and** `NEEDS`**.** For example,

```
INCLUDE demo/chomp-chomp.f
```

or

```
NEEDS GRAPHICS
```

**Moreover, you are allowed to edit any souce text-file using** `LED`**, the built-in editor available after you type**

```
NEEDS LED
```

See chapter "The dictionary" for more details.

## 2.7 Definitions grouped by category

Here is my own personal classification of most definitions available in this system.

### 2.7.1 Comments

| Block oriented | Line oriented | No-operation |
|---|---|---|
| ( ... ) | \ ... | NOOP |

### 2.7.2 Stack manipulation

Broadly speaking, a **Calculator-Stack** entry is a 16-bits number i.e. a CELL, while a Double-Integer value is a 32-bits number which needs two CELLS in the Calculator-Stack, the higher significant part on top of stack.
**Return-Stack** is used on entering-exiting phase of a definition *and also* to keep track of DO-LOOP index and limit.
**Floating-Point-Stack** is the standard ZX Spectrum floating-point Calculator Stack that is accessible after loading the Floating-Point-Option (§3.7). See §4 "Technical specifications" for more details.

| Single Cell | Double cells | Return Stack | Stack Inspection | Floating point Stack |
|---|---|---|---|---|
| DUP<br>OVER<br>DROP<br>SWAP<br>NIP<br>TUCK<br>ROT<br>-ROT<br>PICK<br>ROLL<br>?DUP<br>-DUP | 2DUP<br>2OVER<br>2DROP<br>2SWAP<br>2ROT | >R<br>R><br>R@<br>I'<br>DUP>R<br>R>DROP | DEPTH<br>.S | >F<br>F><br>FOP |

### 2.7.3 Comparison

Comparison involves the two top elements available on the Calculator Stack.

| Against zero | Signed | Unsigned | Double-precision |
|---|---|---|---|
| 0=<br>0<<br>0><br>NOT | =<br><<br>><br><><br>MIN<br>MAX | U<<br>DU< | D0=<br>D<<br>D= |

### 2.7.4 Output

Any output is sent to video by default, but the actual device depends on which Stream is chosen via `SELECT`.

| Single word Stack Value | Double word stack value<br>Floating point Stack | String | Other |
|---|---|---|---|
| .<br>.R<br>?<br>U. | D.<br>D.R<br>F. | ."<br>.(<br>.C<br>SPACE<br>SPACES<br>EMIT<br>EMITC<br>INVV<br>TRUV<br>MARK<br>MESSAGE<br>TYPE | SPLASH<br>CLS<br>CR<br>DEVICE<br>SELECT |

## 2.8 Integer Arithmetics

Normally all definitions act upon 16-bits (signed or unsigned) integers.
Definitions that act upon 32-bits integers have names that begin with **D** for *double*.
Mixed definitions, that involve both 16-bits and 32-bits integers begin with **M** for *mixed*.

| Arithmetics | Signed / Unsigned | Double | Constants |
|---|---|---|---|
| +<br>-<br>*<br>/<br>/MOD<br>MOD<br>2/<br>2* | +-<br>ABS<br>NEGATE<br><br>UM/MOD<br>UM* | D+<br>D-<br>DNEGATE<br>DABS<br>D+- | 0<br>1<br>2<br>3<br>-1<br><br>PI |

| Mixed | Bitwise | Increment/Decrement | Floored / Simmetric Division |
|---|---|---|---|
| M+<br>M*<br>M/<br>M/MOD<br>*/<br>*/MOD | AND<br>OR<br>XOR<br>NOT<br>RSHIFT<br>LSHIFT<br>FLIP<br>INVERT<br>SPLIT<br>UPPER | 1+<br>2+<br>CELL+<br>1-<br>2-<br>CELL- | FM/MOD<br>SM/MOD |

## 2.9 Memory

| Store & Fetch | Memory chunks | Pointers & Variables | 8K RAM Paging |
|---|---|---|---|
| !<br>@<br>2!<br>2@<br>C!<br>C@<br>+!<br>TOGGLE<br>TO (used with VALUE)<br>+TO | FILL<br>ERASE<br>BLANK<br>CMOVE<br>CMOVE><br>PAD<br>BUFFER<br>BLOCK<br>CELL<br>CELLS<br>ALIGNED<br>DUMP | RP@<br>RP!<br>SP@<br>SP!<br>S0<br>R0<br>USE<br>PREV | S"<br>HEAP<br>FAR<br>POINTER<br>H"<br>+"<br>+C<br>>FAR<br><FAR<br>HEAP-INIT<br>HEAP-DONE<br>SKIP-PAGE |

### 2.9.1 Flow control

| Counted Loop | Unconted Loop | Conditionals | System related |
|---|---|---|---|
| DO<br>?DO<br>LOOP<br>+LOOP<br>LEAVE<br>I<br>I'<br>J<br>K | BEGIN<br>WHILE<br>REPEAT<br>UNTIL or END<br>AGAIN<br><br>BACK | IF<br>THEN<br>ENDIF<br>ELSE<br>CASE<br>ENDCASE<br>OF<br>ENDOF<br>EXEC: | BYE<br>AUTOEXEC<br>COLD<br>WARM<br>ABORT<br>ERROR<br>CALL#<br>INTERPRET<br>EXECUTE<br>QUIT<br>BASIC |

### 2.9.2 Definition related

| Creators | Status & Variables | Compilation / Interpretation | Dictionary Allocation |
|---|---|---|---|
| :<br>;<br>:NONAME<br>CREATE<br>VARIABLE<br>CONSTANT<br>CODE<br>EXIT<br>!CSP<br>DOES><br><BUILDS<br>VALUE<br>USER | ?COMP<br>?CSP<br>?ERROR<br>?EXEC<br>?LOADING<br>?PAIRS<br>?STACK<br>STATE<br>CSP | COMPILE<br>[COMPILE]<br>[CHAR]<br><br>[<br>] | ALLOT<br>,<br>C,<br>."<br>.(<br>COMPILE,<br>LITERAL<br>DLITERAL |

### 2.9.3 I/O and Hardware

| I/O Ports | HW Registers | Keyboard | |
|---|---|---|---|
| P!<br>P@ | REG!<br>REG@<br>MMU7!<br>MMU7@ | ?TERMINAL<br>KEY<br>CURS | |

### 2.9.4 BLOCK / Screen related

| Block & Buffer | Input | Block-file primitives | Variables & Constants |
|---|---|---|---|
| .LINE<br>BLOCK<br>EMPTY-BUFFERS<br>FLUSH<br>INDEX<br>LIST<br>UPDATE<br>OPEN< | LOAD<br>--><br>QUERY<br>ACCEPT<br>ENCLOSE<br>CHAR<br>EXPECT<br>WHERE<br>LOCATE<br>GREP<br>BSEARCH | BLK-INIT<br>BLK-READ<br>BLK-SEEK<br>BLK-WRITE | TIB<br>FIRST<br>LIMIT<br>SOURCE-ID<br>BLK<br>>IN<br>OUT<br>SCR<br>OFFSET<br>BLK-FH<br>BLK-FNAME<br>#SEC<br>#BUFF<br>SPAN<br>B/BUF<br>B/SCR<br>C/L |

### 2.9.5 Numbers & strings

| Number to string | Base | Interpretation | Variables |
|---|---|---|---|
| <#<br>#<br>#S<br>#><br>SIGN<br>HOLD | BASE<br>HEX<br>DECIMAL<br>BINARY<br>OCTAL | NUMBER<br>(NUMBER)<br>(SGN) | NMODE<br>HLD<br>DPL<br>FLD<br>PLACE<br>EXP |

### 2.9.6 Dictionary related

| Input Stream | Vocabulary manipulation | Definition data | Variables & Constants |
|---|---|---|---|
| '<br>-FIND<br>INCLUDE<br>MARKER<br>NEEDS<br>CASEOFF<br>CASEON | FORTH<br>DEFINITIONS<br>ASSEMBLER<br>SMUDGE<br>RENAME<br>FORGET<br>ALIGN<br>ID. | CFA<br>PFA<br>NFA<br>LFA<br><NAME<br>>BODY<br>TRAVERSE<br>.WORD | WIDTH<br>WARNING<br>FENCE<br>DP<br>VOC-LINK<br>CONTEXT<br>CURRENT<br>BL |

### 2.9.7 Editor

| Screen oriented | Line oriented | | File oriented |
|---|---|---|---|
| EDIT<br><br>B<br>N<br>L<br>SAVE | H<br>D<br>RE<br>INS<br>S<br>E | .PAD<br>P<br>-MOVE<br>TEXT<br>LINE | LED<br>LED-EDIT<br>LED-SAVE<br>LED-FILE |

### 2.9.8 NextZXOS

| File hooks | Directory hooks | +3DOS hooks | |
|---|---|---|---|
| F_SEEK<br>F_CLOSE<br>F_SYNC<br>F_FGETPOS<br>F_READ<br>F_WRITE<br>F_OPEN | F_OPENDIR<br>F_READDIR | M_P3DOS | R#<br>LP<br>HANDLER |

### 2.9.9 Unsorted

R#
LP
HANDLER

## 2.10 Known bugs and improvement needed

**INTERPRET** Interpretation of long structure via LOAD cannot cope with BLOCK boundaries. This means, for example, that you cannot start an ENUMERATED structure in one BLOCK and continue it in the next BLOCK.

**INCLUDE** This word has a known bug, the `INCLUDE`d source text file must end with an empty line, otherwise the system will crash usually showing some vertical lines.

**NEEDS** has a flaw, in case of interpretation/compilation error, the file/handle remains open and you have to close it manually using something like `2 F_CLOSE .` otherwise you cannot use REMOUNT.
Since NEEDS uses INCLUDE, it has the same known bug and the source text file must end with an empty line.

**OPEN<** At the moment, this definition cannot be compiled and should be used only in interpretation phase.

**CAT"** In this version, the drive **C:** must always be specified, even for current directory.

# 3 Utilities

**WARNING: many of these definitions are still under development and specifications may change in the future. Much effort is put to keep backward compatibility.**

## 3.1 The Full Screen Editor Utility – Screen oriented

The `EDIT` definition is available after you type: `NEEDS EDIT` (or in the old way `190 LOAD` if the source it is still there and you didn't reused these Screens).

On this Forth system, as in many others, a Screen has **1.024** bytes of data spread in 16 lines, 64 bytes each.

This "Full Screen Editor Utility" is invoked using the `EDIT` definition that enters a simple page-editor that allows modifying the current Screen, i.e. the one contained in `SCR` variable. During `EDIT`, you are allowed move to the next Screen or to the previous Screen using the command explained below.

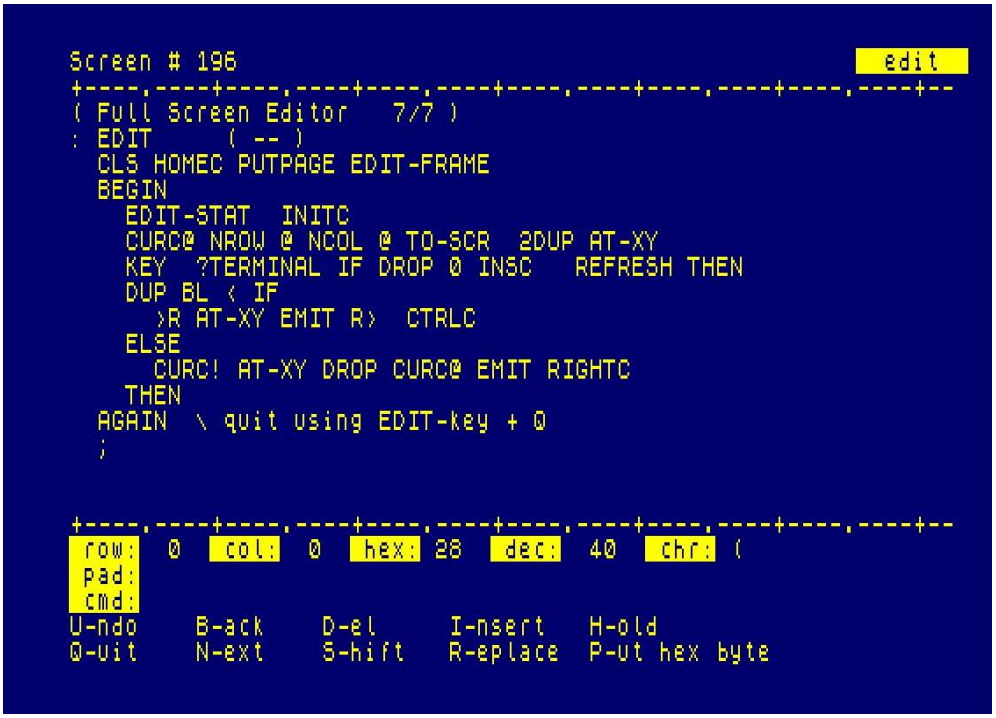
**Remember**: to *quit* `EDIT` phase, you have to use [**Edit**] key followed by [**Q**] key, in a way that mimics Unix **vi** editor.

This editor works only if the display-mode allows 64 character per line at least.

**EDIT** **---**

For example, to select, show and edit **Screen # 196** you can type:

```
DECIMAL  196  LIST        ( to set 196 the "current screen" )
EDIT                      ( to enter the editor on "current screen" )
```



The picture above shows a header reporting the Screen number and a line-ruler followed by 16 lines that make up the Screen itself.

A flashing cursor is visible at home position: The cursor has two flashing mode to distinguish **CAPS-LOCK** enabled or disabled.

The cursor keys, **[Shift]** key + **5 / 6 / 7 / 8** keys, allow the flashing cursor to be moved across the screen to point the current position inside the Screen, so text can be typed at any position in the Screen.

Current cursor positions (**row** number and **col**umn number) are shown at the bottom status bar along with current character, **dec**imal ASCII code and **hex**adecimal code of it.

**Pad** line shows the current `PAD` content. Line oriented commands handle and work with `PAD`. See the "Line Editor" chapter. If PAD contains garbage, the whole screen may become corrupted: in this case you can type **[Edit]** + **H** to copy the current line to PAD that should fix that issue.

After the **[Edit]** key **(Shift + 1 using standard PC keyboard)** the Editor recognizes the following single key-stroke commands:

> **[Edit] + Q** : **Q**uit `EDIT` Utility
> **[Edit] + U** : **U**ndo, that is re-read current screen from disk ignoring any modification done since last FLUSH. This feature is quite important, since it does for a single Screen what `EMPTY-BUFFERS` does for all of them.
> **[Edit] + H** : take (or **H**old) current line content and keep it in `PAD`
> **[Edit] + R** : **R**eplace current line with the current `PAD` content.
> **[Edit] + S** : make **S**pace at current cursor position shifting lower lines down; last line will be lost.
> **[Edit] + D** : **D**elete current line shifting up lower line, but a copy is copied to `PAD` before deletion, like **H**
> **[Edit] + I** : **I**nsert at current cursor line position the content of `PAD`: it does commands **S** and **R**.
> **[Edit] + N** : go to **N**ext screen
> **[Edit] + B** : go **B**ack to previous screen
> **[Edit] + P** : accepts **two hexadecimal digits** representing a byte and **P**ut it at cursor position. This way, non-printable characters, that is ASCII code between 0 and 31 ($00 - $1F), can be stored inside a Screen, but attention must be paid to avoid corrupting the display because most of them are **control characters**. Characters with ASCII code between $80 and $FF can be stored in a Screen, but they are emitted to video translated to the corresponding codes between $00 and $7F.

any other key has no meaning and returns the flashing cursor back to its position.

**[Delete]** (that is **Caps-Shift + ZERO**) removes a character at current cursor position, shifting left the rest of the line.

**[Break]** (that is **Caps-Shift + SPACE**) inserts a space at current cursor position, shifting right the rest of the line.

**[Caps-Lock]** (that is **Caps-Shift + 2)** accounts for a keystroke, but it is interpreted by the system to change the Caps-Lock state.

Beware, any modification you do immediately affects the underlying Buffers, so if you mess things too much so that **[Edit] + U** is not enough, there is only a way to recover it: using `EMPTY-BUFFERS` to erase all buffers without flushing to disk, before it's too late.

This "Full Screen Editor" is a work-in-progress and can be improved if needed.

## 3.2 Graphics mode and Layer facility

The following definitions are available after you type `NEEDS GRAPHICS`.
To forget this library from dictionary you can type `NO-GRAPHICS`.
This library is still work-in-progress.

The ZX Spectrum Next's machine can handle several Graphic-Modes and vForth is able to use them.

In all the following definitions, the x-coordinate is the vertical distance from the top-left corner of the grid, the y-coordinate is the horizontal distance from the top-left corner of the grid

### LAYER!  n  ---

This is a primitive definition to switch Graphic-Mode. The parameter `n` can be one of the following values and can be expressed both in DECIMAL or in HEX indifferently.

- **00** to switch to **Layer 0** - Standard Spectrum (ULA) mode, 256 w x 192 h pixels, 8 colors total (2 intensities), 32 x 24 cells, 2 colors per cell. Equivalent to Basic's LAYER 0.
- **10** to switch to **Layer 1,0** - LoRes (Enhanced ULA) mode, 128 w x 96 h pixels, 256 colors total, 1 color per pixel. Equivalent to Basic's LAYER 1,0.
- **11** to switch to **Layer 1,1** – Standard Res (Enhanced ULA) mode, 256 w x 192 h pixels, 256 colors total, 32 x 24 cells, 2 colors per cell. Equivalent to Basic's LAYER 1,1.
- **12** to switch to **Layer 1,2** – Timex HiRes (Enhanced ULA) mode, 512 w x 192 h pixels, 256 colors total, only 2 colors on whole screen. Equivalent to Basic's LAYER 1,2.
- **13** to switch to **Layer 1,3** – Timex HiColour (Enhanced ULA) mode, 256 w x 192 h pixels, 256 colors total, 32 x 192 cells, 2 colors per cell. Equivalent to Basic's LAYER 1,3.
- **20** to switch to **Layer 2** – 256 w x 192 h pixels, 256 colors total, one color per pixel. Equivalent to Basic's LAYER 2,1.

To ease of use, this word accepts `n` to be expressed both in decimal or in hexadecimal, without confusion and since there is no ambiguity, so the following two lines gives the same result

```
HEX     12 LAYER!
DECIMAL 12 LAYER!
```

This primitive definition *just* switches Graphics-Mode without any other side effect. Instead, the following definitions `LAYER0`, `LAYER10`, `LAYER11`, `LAYER12`, `LAYER13` and `LAYER20` also modify the overall behavior of the other graphics definitions.

### LAYER0  ---

Set screen mode to Standard ULA, legacy ZX Spectrum mode, also set the characters to 4 pixel wide, and modify the overall behavior of all graphics definitions to work with this specific pixel resolution.

### LAYER10  ---

Set screen mode to LoRes mode, set the characters to 4 pixel wide, and modify the overall behavior of all graphics definitions to work with this specific pixel resolution.

**LAYER11** **---**

Set screen mode to Standard Res (Enhanced ULA) mode, set the characters to 4 pixel wide, and modify the overall behavior of all graphics definitions to work with this specific pixel resolution.

**LAYER12** **---**

Set screen mode to Timex HiRes (Enhanced ULA) mode, set the characters to 8 pixel wide, and modify the overall behavior of all graphics definitions to work with this specific pixel resolution, for example, the correct aspect-ratio for CIRCLE is enforced.

**LAYER13** **---**

Set screen mode to Timex HiColour (Enhanced ULA) mode, set the characters to 4 pixel wide, and modify the overall behavior of all graphics definitions to work with this specific pixel resolution.

**LAYER2** **---**

Set screen mode to Layer 2, set the characters to 4 pixel wide, and modify the overall behavior of all graphics definitions to work with this specific pixel resolution.

**ATTRIB** **---**

Variable that specifies the byte used as color-attribute in all subsequent graphics command.

**CIRCLE x y r ---**

Draw a circle with center at `x y` and radius `r` using the current `ATTRIB` color and Graphic-Mode. As stated above, x-coordinate is the vertical and y-coordinate is horizontal.

**DRAW-LINE x0 y0 x1 y1 ---**

Draw a line from `x1 y1` to `x0 y0` using the current `ATTRIB` color and Graphic-Mode. As stated above, x-coordinate is the vertical and y-coordinate is horizontal.

**PLOT x y ---**

Draw a pixel at `x y` using the current `ATTRIB` color and Graphic-Mode. As stated above, x-coordinate is the vertical and y-coordinate is horizontal.

**PAINT x y ---**

Experimental: Try to paint a well-shaped convex area, provided that `x y` is some “center” to start. As stated above, x-coordinate is the vertical and y-coordinate is horizontal.

### 3.2.1 Colors & Attributes

Here is a set of definition that invoke the Standard-ROM routines to change the screen colors. All these definitions ends with a dot (.) to spacify that it works via `EMIT` and to avoid confusion with other definitions.

**.BORDER** **b** **---**
Immediately set the current BORDER color. It uses ROM routine $2297 via `CALL#`.

**.BRIGHT** **b** **---**
Depending on on current Graphics Mode, set the current BRIGHT attributefor any subsequent output operations.

**.FLASH** **b** **---**
Depending on on current Graphics Mode, set the current BRIGHT attribute for any subsequent output operations.

**.INK** **b** **---**
Depending on on current Graphics Mode, set the current INK color for any subsequent output operations.

**.INVERSE** **b** **---**
Depending on on current Graphics Mode, set the current INVERSE attribute for any subsequent output operations.

**.OVER** **b** **---**
Depending on on current Graphics Mode, set the current OVER attribute for any subsequent output operations.

**.PAPER** **b** **---**
Depending on on current Graphics Mode, set the current PAPER color for any subsequent output operations.

## 3.3 LED – the Large file EDitor

Source text-files can be edited directly within vForth environment using `LED` – the Large file EDitor – that handles text files up to 17.568 rows, 85 characters each. The `LED` definition is available after you type: `NEEDS LED` .

Along with `LED` you often need `CAT` .

After you type `LED` you enter a simple full-screen editor that can modify current file one screen at a time. While within `LED`, you are allowed move to next page or to previous page using the command explained below.

**Remember**: to *quit* `LED` editor, you have to use [**Edit**] key followed by [**Q**] key.

**Remember**: to *save* the file, you have to use [**Edit**] key followed by [**W**] key.

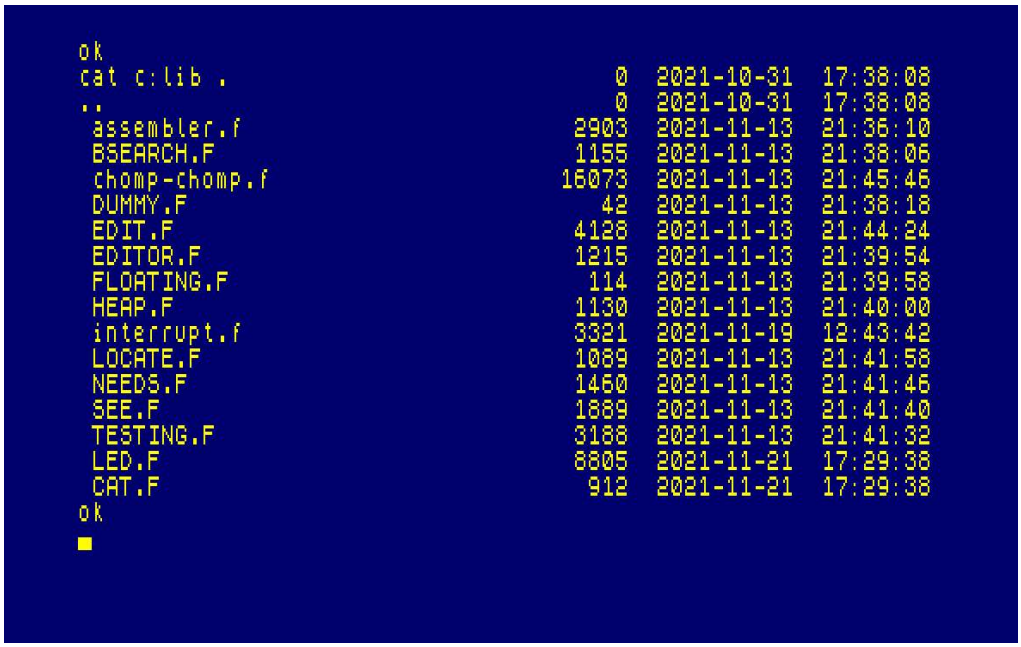
This editor works only while the display-mode is **LAYER 1,2**.

**CAT"** **---**

Available after `NEEDS CAT`. Used in the form

**CAT" c:xxx"**

it displays the content of directory **xxx** for example: **CAT" c:lib"**




**LED** **---** **cccc**

Available after `NEEDS LED`. Used in the form

**LED cccc**

opens specified file `cccc` and enters `LED` editor. For example `LED lib/cat.f`

This editor inherits most of its commands and behavior from the previously described `EDIT` editor except that is has 85 characters per line instead of 64. See previous paragraph for details..

Along with `LED` command, some more sub-commands are available to better handle a text-file.

### LED-EDIT ---

Used in the form

**LED-EDIT**

re-enters the LED editor after you quit it to continue editing the same file you previously opened that should still be in upper 8k RAM pages, provided you haven't corrupted its content in some way.

### LED-SAVE ---

Used in the form

**LED-SAVE**

saves back the file you previously open in LED editor, using the current filename you already specified using **LED** or **LED-FILE**.

### LED-FILE --- cccc

Used in the form

**LED-FILE cccc**

modify the filename that LED-SAVE will write to. This allow to save to a different filename.

## 3.4 Interrupt Service Routine

After you type `NEEDS INTERRUPTS` a new Vocabulary will be loaded in memory along with some low-level words that allow setting-up an Interrupt-Driven word: The ISR must be a single word suitably defined. This is a standard IM 2 interrupt routine implementation. In the future, I hope to be able to exploit the new Next's IM 2 interrupt vector mode. Programming an Interrupt Service Routine using Forth itself is tricky and if not correctly coded, it can impair the system or cause a system-crash. As said, this library still does not exploit the new ZX Spectrum Next interrupt vector, this will be soon implemented: this means that the all these definitions listed here below will go under a deep overhaul when I'll code it.

A Z80's maskable interrupt occurs every 20 ms (50 times per second), and when it occurs, a CALL to a specific routine is performed. In vForth we use IM 2 interrupt mode by preparing a 257 bytes vector-table at $6200, filled with $63, so that the interrupt service routine is located at $6363 to jump to the suitable code – i.e. the word `ISR-SUB` – that makes possible to a Forth word to be executed as interrupt.

First, `ISR-SUB` performs a RST $38 to fulfill the legacy ISR, then it must save the whole *Forth machine status* by pushing to stack the value of CPU registers and then saving Forth's Return-Stack-Pointer and Calculator-Stack-Pointer.
Second, it prepares Forth virtual registers (Calculator-Stack Pointer, Return-Stack-Pointer and Instruction-Pointer) to execute the xt contained in `ISR-W` variable, then a jump to the Inner-interpreter via JP (IX) is performed. Interrupts stay disabled during the execution of such a xt.

After the xt contained in `ISR-W` is executed, the `ISR-RET` word is executed restoring back the machine status by retrieving Calculator-Stack Pointer, Return-Stack-Pointer and Instruction-Pointer and then popping all CPU registers before returning from the interrupt routine and re-enabling interrupts.

It's worth to be noticed that, since an interrupt may occur in the middle of the execution of any part of Forth system, not everything can be performed within an interrupt service routine, and care must be put to avoid critical interference with the main program, such as trying to write the same VARIABLE or invoking peculiar definitions that are known to modify the code they're going to execute, such as `CASEON`, `CASEOFF`, or memory areas such as most Floating-point operations.

**INTERRUPTS** **---**
This is an `IMMEDIATE` definition that selects the `INTERRUPTS` vocabulary to make “available” the definitions described in this section.

**ISR-OFF** **---**
Disable Interrupt Utility by restoring IM 1 and I register to its default $3F value.

**ISR-XT** **xt** **---**
Variable that contains the xt of the word that will be executed in background at each Interrupt. It is always followed by the execution of `ISR-RET` so that `ISR-XT` can be viewed as the pointer to an anonymous word that contains two definitions: the *interrupt-service-routine* definition and the *return-from-interrupt* definition.

**ISR-ON** **---**
Enable Interrupt Service Utility: This word prepares “IM 2 Vector Table” at address $6200-$6300 filling it with all $63 and set Interrupt Mode 2, so that when an Interrupt is issued a CALL to address $6363 is performed.

At address $6363 is a jump to address of `ISR-SUB` body i.e. `[ ' ISR-SUB >BODY ]` and it's used in the form

```
ISR-OFF
' ISR-WORD  ISR-XT  !
ISR-ON
```

Then, *ISR-WORD* is executed in background at each Interrupt.

During an Interrupt, Forth uses a separate Calculator Stack (4 bytes below current SP) and a separate Return Stack located at $6330. Care must be paid to avoid any critical interference with the normal *foreground* Forth execution.

Typical usage is to control some Sprite movement or poll **mouse** and **joystick**, some demos are available.

The following example keeps the display filled with evenly spaced dots in Layer 1,1 or Layer 1,2 modes.

```
: ISR-WORD

      [ HEX ] 80 57FF C!
      5701 5700 FF CMOVE>
      5700 4700 100 CMOVE
      4700 4F00 100 CMOVE
;
ISR-OFF
' ISR-WORD ISR-XT !
ISR-ON
```

**ISR-EI** **---**

Low-level “enable interrupt”. It actually executes an EI opcode.

**ISR-DI** **---**

Low-level “disable interrupt”. It actually executes a DI opcode.

**ISR-IM1** **---**

Low-level “interrupt mode 1”. It actually executes an IM 1 opcode. This is the default *mode* for any ZX Spectrum.

**INT-IM2** **---**

Low-level “interrupt mode 2”. It actually executes an IM 2 opcode. It relies on a “vector table” located

**ISR-SYNC** **---**

Low-level “halt”. It actually executes an HALT opcode to force the machine wait until the next interrupt.

**SETIREG** **b** **---**

Low level Z80 register I setting. It actually executes an LD I, A opcode.

**ISR-RET** **---**

Low-level “return from interrupt” definition. It restores all registers and returns control to Forth foreground execution.

**ISR-SUB** **---**

Low-level “interrupt service routine” definition. It saves all registers and gives control to `INT-XT` background word execution. Interrupt SP is initialized at 4 bytes below current SP. Interrupt RP is initialized at $6330 and allows room for 14 cells.

## 3.5 Block Search and Locate Utility

This group of definitions allow you to look for text within the Screens / Blocks and are available after you type alternatively:

```
NEEDS LOCATE  or
NEEDS GREP  or
NEEDS BSEARCH  or
NEEDS COMPARE
```

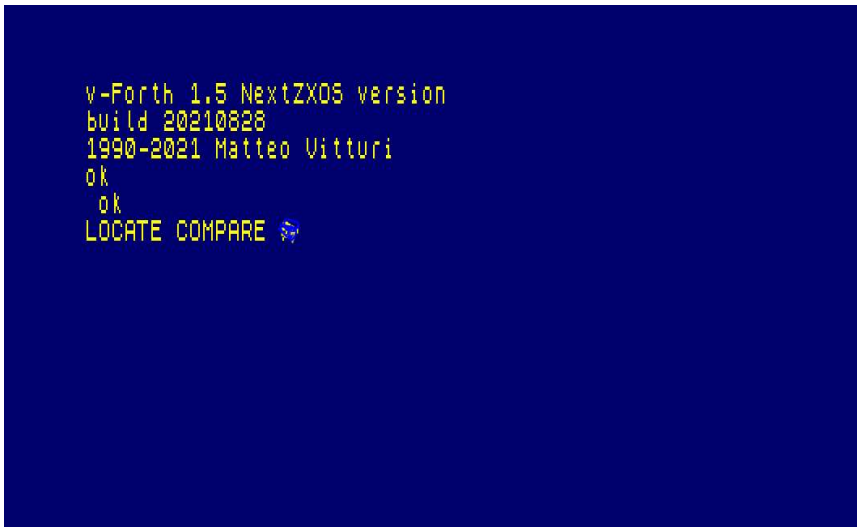
**LOCATE** ---

Used in the form

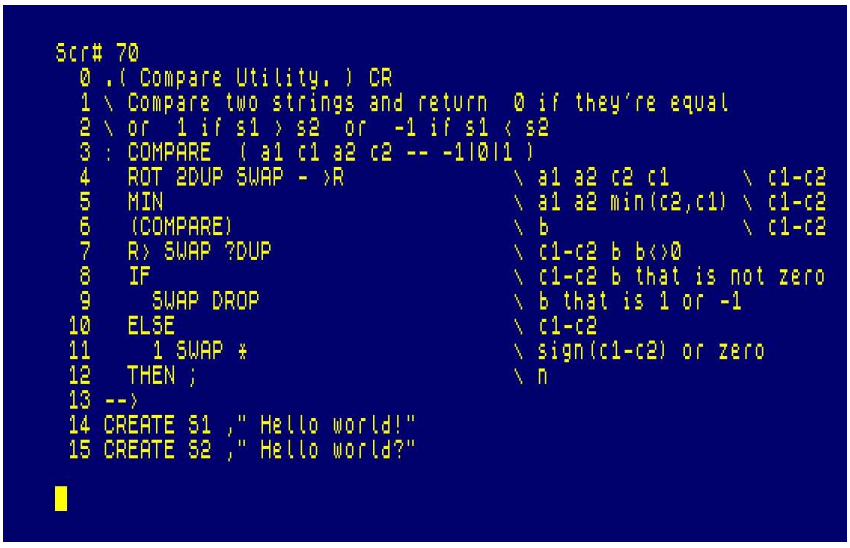
```
LOCATE cccc
```

this word examines all Screens between 1 and 2000 looking for the definition of **cccc** and shows the Screen where it found the first occurrence, and makes it the "current screen", just like `LIST` for example:

```
LOCATE COMPARE
```




takes a few seconds to search in which Screen `COMPARE` is defined, and if found it shows the Screen using `LIST`.



**GREP** ---

Used in the form

```
GREP cccc
```

this word examines all Screens between 1 and 2000 looking for any occurrence of word **cccc** showing them in a table form, for example

```
GREP COMPARE
```

will take some more time to complete and gives something like the following

**BSEARCH n1 n2** ---

Used in the form

```
n1 n2 BSEARCH cccc
```

this word examines all Screens between n1 and n2 looking for any occurrence of word **cccc** showing them in a table form. This definition is used by `GREP` that in fact is defined as `1 2000 BSEARCH .`

**COMPARE a1 b1 a2 b2** ---

Given two string descriptors, that is address and length, (a1, b1) and (a2, b2), this definition compares the two strings and returns:

| | |
|---|---|
| 0 | if they're equal |
| 1 | if String1 > String2 |
| -1 | if String1 < String2 |

For example:

```
CREATE S1 ," Hello world!"
CREATE S2 ," Hello world?"
S1 COUNT S2 COUNT COMPARE .
```

will print -1 since the two strings differs only for the last character and the ASCII code of `!` comes before the code of `?` , so the string comparison S1 < S2 is true. Compare the result of the following two rows:

```
S2 COUNT S1 COUNT COMPARE .
S1 COUNT S1 COUNT COMPARE .
```

## 3.6 Debugger Utility

The following definitions are available after you type `NEEDS SEE` or usually after a regular `AUTOEXEC`.

Also, this section exposes in detail how definitions are stored in dictionary memory.

In the **Indirect-Threaded** version, low-level definitions CFA contains the address of PFA that in turn contains the machine code of the definition; in a colon definition CFA points to the address of the routine that handles that kind of definition.

In **Direct-Threaded** version, a Low-Level definition takes two bytes less, since CFA contains the actual machine code of the definition; a Colon-definition needs one additional byte in CFA to allow room for a "CALL" op-code to the address that handles that kind of definition. This allows some 25% of more speed at the cost of using a little more memory.

### SEE ---

Used in the form

**SEE cccc**

it will print how word `cccc` is defined along with its `NFA`, `CFA`, `PFA` information.

If `cccc` is a regular colon-definition the result will show something close to the original source the word was defined from.

For example, the word **TYPE** is a colon-definition that emits to video a counted-string stored at address a, and is defined as follow:

```
: TYPE    ( a n -- )
    BOUNDS ?DO
        I C@ EMIT
    LOOP
;
```

If you type

```
SEE TYPE
```

the system will emit something like the following, depending on which build you're running:

Direct-Thread build **20220827** :

```
Nfa: 7238 84
Lfa: 723D LEAVE
Cfa: 7248 6BD8
BOUNDS (?DO) 12 I C@ EMIT (LOOP) -8 EXIT ok
```

Indirect-Thread build **20220827**:

```
Nfa: 7297 84
Lfa: 729C LEAVE
Cfa: 729E 6C86
BOUNDS (?DO) 12 I C@ EMIT (LOOP) -8 EXIT ok
```

The first line shows **TYPE** Name Field Address (**$7238 or $7297**) followed by **$84** that is the counter byte of a 4-bytes length name. The counter byte always has the most significant bit set, that is **$80** added to **$04** giving **$84**.

The second line is the Link Field Address (**$723D or $729C**) which holds a pointer to **LEAVE**'s NFA that in this case

happens to be the previous definition in the dictionary.

The third line is the Code Field Address (**$7248 or $729E**) that, in Direct Thread version, contains the actual machine code to be run which in this case is a "CALL" to the ENTER routine of every colon-definition, located at **$6BD8**. In Indirect-Thread version contains the address of the ENTER routine located at **$6C86**.

The fourth line represents the Parameter Field Address and, in this case, is in some way a definition "decompilation" but literals and offsets are shown in "inverse video" mode. For example the number **-8** after **(LOOP)** is the "offset" to where the Instruction Pointer has to jump to go back to next iteration. In this example **(?DO)** and **(LOOP)** are the *compiled counterpart* of ?DO and LOOP that in fact normally won't be *compiled,* instead they control the compilation of some other words.

In the "indirect-thread" version, the CFA holds a pointer to the machine-code part of a regular colon-definition so the "inner-interpreter" can jump to it.

Another example, the word **NIP** that removes the second element of Stack, isn't a colon-definition, but a low-level definition coded directly in machine-code as follow:

```
CODE NIP  ( n1 n2 -- n2 )
        POP       HL|     \  pop  hl
        EX(SP)HL          \  ex   (sp), hl
        Next              \  jp   (ix)
        C;
```

and if you type

```
SEE NIP
```

in the *Direct Threaded* build **20220730** version, it will emit

```
Nfa: 6A58 83
Lfa: 6A5C DROP
Cfa: 6A5E DDE3
6A5E E1  E3  DD  E9   84  54 55 43   ac]i TUC
6A66 CB  58  6A  E1   D1  E5 D5 E5   KXjaQeUe
```

In this case, since **NIP** is a low-level definition, the PFA part is shown as a hexadecimal **DUMP** that is it has no PFA part, but it's the real machine-code routine.

in the *Indirect Threaded* build **20220730** version, it will emit

```
Nfa: 6AD8 83
Lfa: 6ADC DROP
Cfa: 6ADE 6AE0
6ADE E0  6A  E1  E3   DD  E9  84  54   £jac]i T
6AE6 55  43  CB  D8   6A  ED  6A  E1   UCKXjmja
```

Again, the first line shows **NIP**'s NFA (**$6A58** or **$6AD8** in this case) and **$83**, the counter byte, that indicates a 3-bytes length word name.

The second line is **NIP**'s LFA (**$6A5C** or **$6ADC**) that contains a pointer to DROP's NFA, that is the previous definition in dictionary.

The third line is **NIP**'s CFA (**$6A5E** or **$6ADE**) which content depends on which version (direct or indirect-thread) you're using. In the indirect-thread version, this cell is a pointer to the next cell address (**$6AE0**) where the piece of machine-code lies. In the direct-thread version this address contains the machine-code itself.

In both versions, examining the subsequent DUMP you should be able to locate **E1** for `POP HL`, **E3** for `EX(SP),HL` and **DD E9** for `JP(IX)` to the inner interpreter address that is compiled by **Next** Assembler definition.

The bytes that follows in both versions – 84 54 55 43 CB – are the beginning of the subsequent definition in dictionary (**TUCK** in this case).

This utility is not perfect, but is a good way to debug and understand a Forth definition.

### 3.6.1 The Inner-interpreter

Here is a comparison between indirect-thread versus direct-thread inner-interpreter routine:

| Indirect-threaded | Direct-threaded |
|---|---|
| **NEXT:**<br>`ld a, (bc)`<br>`inc bc`<br>`ld l, a`<br>`ld a, (bc)`<br>`inc bc`<br>`ld h, a`<br>`ld e, (hl) ; 7 T`<br>`inc hl ; 6 T`<br>`ld d, (hl) ; 7 T`<br>`ex de, hl ; 4 T`<br>`jp (hl)` | **NEXT:**<br>`ld a, (bc)`<br>`inc bc`<br>`ld l, a`<br>`ld a, (bc)`<br>`inc bc`<br>`ld h, a`<br>`jp (hl)` |
| **CFA:**<br>`db ENTER` | **CFA:**<br>`call ENTER ; 17 T` |
| **ENTER:**<br>`ld hl, (RP)`<br>`ld (hl), b`<br>`inc hl`<br>`ld (hl), c`<br>`inc hl`<br>`ld (RP), hl`<br>`inc de ; 6 T`<br>`ld c, e ; 4 T`<br>`ld b, d ; 4 T`<br>`jp (ix) ; to NEXT` | **ENTER:**<br>`ld hl, (RP)`<br>`ld (hl), b`<br>`inc hl`<br>`ld (hl), c`<br>`inc hl`<br>`ld (RP), hl`<br>`pop bc ; 10 T`<br>`jp (ix) ; to NEXT` |

The Direct Threaded version simply omits the pointer de-referencing marked in yellow:
Omitting such part reduces the length of low-level definitions by two bytes, but on the other hand increases all non-low-level definitions by one byte. So, a colon-definition must have a "`call ENTER`" that takes some time to pass PFA address around. Since a Forth program mixes low-level and colon-definition, the overall speed is increased about 20-25%.

**Forth15f.bin** is the Direct-Threaded version, which is the default option.

**Forth15e.bin** is the Indirect-Threaded version: if you wish to run this version you have to modify line 120 of "Forth15_Loader.bas" Basic program

**120 LET f$="forth15e.bin"**

**DUMP** **a u ---**

Performs a "dump" of a memory area from address `a` for `u` bytes or until **[Break]** is pressed. The value of `u` is always rounded to the nearest greater multiple of 8.

Visualization is always in hexadecimal, current base is maintained. For example:

```
DECIMAL 448 60 DUMP
```

will print the Standard ROM content starting from address 448 ($01C0) for 64 bytes, i.e. the nearest greater multiple of 8 and keeps `DECIMAL` as the current `BASE`.

```
01C0  4C 49 53 D4  4C 45 D4 50  LISTLETP
01C8  41 55 53 C5  4E 45 58 D4  AUSENEXT
01D0  50 4F 4B C5  50 52 49 4E  POKEPRIN
01D8  D4 50 4C 4F  D4 52 55 CE  TPLOTRUN
01E0  53 41 56 C5  52 41 4E 44  SAVERAND
01E8  4F 4D 49 5A  C5 49 C6 43  OMIZEIFC
01F0  4C D3 44 52  41 D7 43 4C  LSDRAWCL
01F8  45 41 D2 52  45 54 55 52  EARRETUR
```

**.WORD** **a ---**

Given a `CFA`, this word prints the `ID.` It is used by `SEE` to perform some word "decompilation".

**.S** **---**

Prints the current content of Calculator Stack without destroying its content.
For example, supposing to start with an empty stack,

```
0 1 2 3 .S
```

will print

```
0 1 2 3 ok
```

**DEPTH** **--- n**

It leaves the depth of the Calculator Stack before it was executed. For example, supposing to start with an empty stack,

```
0 1 2 DEPTH .
```

will print

```
3 ok
```

## 3.7 Floating-Point Option

This is an experimental Floating-Point Option Library that exploits the native standard ZX Spectrum Floating-Point capabilities, with some differences.

To load this Floating-point Option Library you have to type `NEEDS FLOATING`.

To perform any floating-point operations you first need to push one or two numbers onto **Spectrum's calculator stack** using `>W` definition. then you need to call the floating-point calculator using `FOP` definition (that calls RST $28 service routine). Finally, you have to pop the result from Spectrum's calculator stack using `W>` definition.

For example, to define a word that returns the value of **pi** you can code something like this:

```
: PI
  [ 1.0 >W 36 FOP  \ atan(1)
    4.0 >W 04 FOP  \ *4
    W>  ]  DLITERAL
;
```

A floating point in Spectrum's calculator stack takes 5 bytes, instead in Forth Calculator Stack it takes 4 bytes only i.e. the same as a “double-integer”. This means there is a little **precision loss**: Maybe in the future we'll be able to fix this fact.

Thinking the floating-double-number stored in CPU registers HLDE, the sign is the msb of H, so you can check for sign in the integer-way. The exponent+128 is stored in the following 8 bits of HL and the significand/mantissa is stored the remaining bits of HL and 16 bits of DE. The fifth byte of a standard floating-point number is then defaulted to a fixed value.

If the floating-number is an integer between 0 and 65535, then it is kept on stack the same as a double-integer. To verify this fact you can type.

```
FLOATING  DECIMAL  65535.0  65537.0  .S
```

that displays

```
      65535  0   128  18560
```

where the two single precision integer 65535 and 0 are the representation of **65535.0** while the two integers **128** and **18560** are the internal bit-representation of **65537.0**

The integer on TOS always keeps the sign information of the floating-double-number.

Most of the definitions described below are created using `<BUILDS` and `DOES>` method.

### 3.7.1 Floating-point option activation and number conversion

To import the floating point library option you must type `NEEDS FLOATING` and then, you can use `FLOATING` to enable the floating-number interpretation and `INTEGER` to disable it and remain within the integers.

**`INTEGER`** **`---`**
Deactivate floating-point numbers mode. `NMODE` user variable is set to 0.

**FLOATING** **---**

Activate floating-point numbers mode. NMODE user variable is set to 1.

**D>F** **d** **---** **fp**

Convert a double-integer into a floating-double-number. See F>D.

**F>D** **fp** **---** **d**

Convert a floating-double-number into a double-integer truncating to the lower integer. It's the opposite of D>F.

**FLOAT** **n** **---** **fp**

Convert a single-precision-integer into a floating-double-number. See FIX.

**FIX** **fp** **---** **n**

Convert a floating-double-number a into single-precision-integer. It's the opposite of FLOAT.

### 3.7.2 Representation and constants

**F>PAD** **fp** **---** **u**

The representation of floating-double-number fp is stored in PAD. The number u is the length of the string.

**F.R** **fp** **u** **---**

Prints fp on a field of u characters to video or current SELECTed stream.

**F.** **fp** **---**

Prints fp to video or current SELECTed stream.

**1/2** **---** **fp**

Put on TOS the value 0.5.

**PI** **---** **fp**

Put on TOS the value of pi.

### 3.7.3 Arithmetics

**F-** **fp1 fp2** **---** **fp3**

Floating point difference: fp3 := fp1 – fp2

**F+** **fp1 fp2** **---** **fp3**

Floating point addition: fp3 := fp1 + fp2

**F*  fp1 fp2 --- fp3**
Floating point product: fp3 := fp1 * fp2

**F/  fp1 fp2 --- fp3**
Floating point division: fp3 := fp1 / fp2

**FNEGATE  fp1 --- fp2**
Floating point negate, i.e. : fp1 := - fp2

**FSGN  fp1 --- fp2**
Floating point sign. Fp2 is the sign of fp1.

**FABS  fp1 --- fp2**
Floating point absolute value

**F/MOD  fp1 fp2 --- fp3 fp4**
Floating point division and reminder: fp4 is the quotient of fp1 / fp2 and fp3 is the reminder.

**F**  fp1 fp2 --- fp3**
Floating point power: fp3 := fp1 ^ fp2

**FMOD  fp1 fp2 --- fp3**
Floating point module: fp3 := fp1 mod fp2

**F*/  fp1 fp2 fp3 --- fp4**
Floating point scale operation: fp4 := fp1 * fp2 / fp3 using an intermediate precision of native 5 bytes instead of 4.

**F<  fp1 fp2 --- f**
Floating point comparison: f is TRUE if fp1 < fp2, FALSE otherwise.

**F>  fp1 fp2 --- fp3**
Floating point comparison: f is TRUE if fp1 > fp2, FALSE otherwise.

**F0<  fp1 --- f**
Floating point comparison: f is TRUE if fp1 < 0, FALSE otherwise.

**F0>  fp1 --- f**
Floating point comparison: f is TRUE if fp1 > 0, FALSE otherwise.

### 3.7.4 Log, Exp, Trig

**FLN** **fp1 --- fp2**
Floating point Natural Logarithm. fp2 := ln(fp1)

**FEXP** **fp1 --- fp2**
Floating point Exponentation: fp2 := exp(fp1)

**FINT** **fp1 --- fp2**
Integer truncation. If the floating-double-number is an integer between 0 and 65535, then it is kept on stack the same as a double-integer. 1.4 FINT gives 1.0 but -1.4 FINT gives -2.0

**FSQRT** **fp1 --- fp2**
Square root.

**FSIN** **fp1 --- fp2**
Sine in radians.

**FCOS** **fp1 --- fp2**
Cosine in radians.

**FTAN** **fp1 --- fp2**
Tangent in radians

**FASIN** **fp1 --- fp2**
Arc-sine in radians

**FACOS** **fp1 --- fp2**
Arc-cosine in radians

**FATAN** **fp1 --- fp2**
Arc-tangent in radians.

**RAD>DEG** **fp1 --- fp2**
Convert radians to degrees.

**DEG>RAD** **fp1 --- fp2**
Convert degrees to radians.

### 3.7.5 Low-level definitions.

**FOP** **n** **---**
Low-level definition that invokes Floating-Point-Operation `n` .

**>W** **fp** **---**
Takes a floating-point number `d` from Calculator Stack and put to Floating-Pointer Stack.

**W>** **fp** **---**
Takes a floating-point number `d` from Calculator Stack and put to Floating-Pointer Stack.

## 3.8 Line Editor

The following definitions are available after you type `90 LOAD` or after you include the `EDITOR` vocabulary via `NEEDS EDITOR`. Most of the logic shown in this section is used by LED "The Large file Editor".

The Line Editor has a dozen words that can operate on a single line of a given Screen and helps inspect things around.

An edit session normally starts with a `LIST` on the desired Screen, this sets `SCR` user variable to the passed Screen number. `LIST` is a word already available in the "core" dictionary. To clear a Screen I foreseen a `BCLEAR` word, but I left it commented somewehere for now, deeming it too dangerous for my tastes; instead I usually use `BCOPY` from an actually empty Screen. You may type `NEEDS BCOPY`.

The word `FLUSH` flushes to disk any modification you've done on any Screen. Beware, a Screen is re-written to disk as soon as the `BUFFERS` containing it are modified. To save space, this implementation has only 7 `BUFFERS`.

`EMPTY-BUFFERS` is another vital word: it empties all buffers. It is very useful if you mistakenly overwrite or spoil a Screen during an edit operation, with it, you have the chance to "rollback" the things before anything is written to disk.

To write a line from scratch or to overwrite line, you can use `P` to "put" the following text to the given line on current screen. For example:

```
1000 LIST
0 P \ One thousand screens
L
```

This sequence selects Screen# 1,000 and put a text "\ One thousand screens" on the first line of it. The word `L` repeats the `LIST` of current screen.

To move or copy a line around, you can use `H` to "hold in `PAD`" a given line on current screen, you can change Screen if you wish, then you can complete this **copy-and-paste** operation with `INS` to "insert" or `RE` to "replace" the line you copied in advance with H. None of above words, but `H`, modify PAD content, so you can repeat the operation. There is also a way to **cut-and-paste** a line using `D` to "delete and copy to `PAD`" instead of `H`.

See also `BLOCK`, `BUFFER`, `INDEX`, `L/SCR`, `LIST`, `LOAD`, `MESSAGE`, `PAD`, `SCR`, `STRM.`, `TIB`.

This is a quick reference of involved memory areas and words that work on them.

| Text Input Buffer (keyboard)<br><br>TIB | Parsing Operation | <br><br>PAD | Edit Operations | One<br><br>BLOCK<br>BUFFER | Blanking Operations |
|---|---|---|---|---|---|
| | TEXT → | | ← H RE → | | ← E |
| | | | ← D INS → | | ← S |
| | | | P → | | |

**-MOVE a n ---**

"Line move". It moves a line, C/L bytes length, from address a to the line n of current screen, then it does an UPDATE. Current Screen is the one kept by SCR.

**.PAD ---**

"Show PAD". It prints the current PAD content.

**B ---**

"Back" one Screen. This word set to previous Screen by decreasing SCR and prints it using LIST.

**D n ---**

"Delete" a row. It deletes line n of current Screen (the one indicated by SCR), the following lines are moved up and the last one will be blanked. D executes H so that it can be followed by an INS to perform a line move.

**BCOPY n1 n2 ---**

"Block-Copy" utility that copies Screen n1 to Screen n2. SCR will contain n2.

**E n ---**

"Erase" a row. This word fills line n with spaces. It does UPDATE.

**H n ---**

"Hold" a row in PAD. This word put line n of current Screen to PAD without altering the block on disk. Current Screen is the one kept in SCR.

**INS n ---**

"Insert" from PAD. This word inserts line n using text in PAD. The original line n and the following ones are moved down and the last is lost.

**L ---**

"List" current Screen. This word does SCR @ LIST.

**LINE n --- a**

Leaves the address a of line n of current screen, the one kept in SCR. Such a screen is currently held in a buffer.

**N ---**

"Next" Screen. This word sets to next Screen by increasing SCR and prints it using LIST.

**P n ---**

"Put" a line. This word accepts the following text (delimited by a tilde character ~) as the text of line n of current Screen. Text is taken from TIB and sent to the current Screen

**RE** **n** **---**

“Replace”. This word takes text currently in `PAD` and put it to line `n`.

**S** **n** **---**

“Space” one row. This word frees line `n` moving the following lines down by one. The last line is lost

**SAVE** **---**

Does `UPDATE` and `FLUSH` saving this Screen and all previously modified Screens back to disk.

**ROOM** **---**

This word shows the room available in the dictionary, that is the difference between `SP@` and `PAD` addresses.

**TEXT** **c** **---**

This word accepts the following text and stores it to `PAD`. `c` is a text delimiter. `TEXT` does not go beyond a 0x00 [null] ASCII.

**UNUSED** **---** **n**

It returns the number of byte available in dictionary.

**WHERE** **n1** **n2** **---**

Usually executed after an error has been reported during a `LOAD` session. Maybe, this word should be included in “core” dictionary. `n1` is the value of `IN` and `n2` the value of `BLK` as were left by `ERROR`.
`WHERE` shows on screen the block number, the line number, the very same line highlighting in “inverse video” the word that caused the error.
If it is invoked after an error during the loading via `NEEDS` or `INCLUDE`, then the result is a bit poor, because it always reports the row #8 of block #1 due to the way these two definitions are coded.

# 4 Technical specifications

## 4.1 CPU Registers

Registers are used in the in the following way:

AF – Available for normal operations.
BC – **Forth Instruction Pointer**: should be preserved on Enter-and-Exit a definition and during ROM/OS calls.
DE – Available, often the Low part when used for 32-bit manipulations
HL – Available, **Work Register** often the High part when used for 32-bit manipulations
AF'– Available, sometime used for backup purpose
BC'– Available, used in I/O operations or complex definitions
DE'– Available, used in complex definitions
HL'– Available, used in complex definitions (saved at startup from Basic)
SP – **Calculator Stack Pointer**
IX – **Used to point to the Forth "inner-interpreter"** (this saves 2 T-States compared to a normal Jump). See `(NEXT)`.
IY – **Used by ZX System**, must be preserved to let keyboard to be served during Interrupts.

## 4.2 Single Cell 16 bits Integer Number Encoding

A 16 bits *integer* represents an integer number between –32768 and +32767 inclusive. The sign is kept in the most significant bit using the usual two-complement notation. Alternatively, the it represents an *unsigned integer* between 0 and +65535.

```
                    H                L
16 bit:      HL:  | sbbb bbbb | bbbb bbbb |
```

In the CPU registers, an *integer* is kept in H and L where H is the most significant part.
In memory, an *integer* is stored in two contiguous bytes in the "little-endian" way, that is the lower address has the least significant part, in the L register. The byte at higher address has the most significant part, in the H register, as usual for the Zilog Z80 processor.

## 4.3 Double cell 32 bits Integer Number Encoding

The second integer format requires two *integers* to form a 32 bits number, referred to as *double* or *long*, that allows integers between –2,147,483,648 and +2,147,483,647 where the sign is kept on the most significant bit of the first *integer*.

Imagine a *double integer* stored in CPU register in this way:

```
                H                L                D                E
32 bits:  | sbbb bbbb | bbbb bbbb | bbbb bbbb | bbbb bbbb |
```

using register H, L, D and E, with the most significant part in H, and the least in E.

Then, on the Calculator Stack the double integer requires four contiguous bytes split into two integers that it forms with the most significant integer (HL) on top of Calculator Stack (i.e., in the lower addresses), and the least significant integer (DE) the second element from top in the higher address, that is the second element from the top. So, it appears as L H E D,

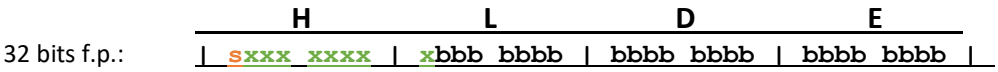
| CPU | Calculator Stack |
|---|---|
| D | SP + 3 |
| E | SP + 2 |
| H | SP + 1 |
| L | SP + 0 (Top Of Stack) |

To adhere to the Standard, in RAM it is kept as L H E D. See how `2VARIABLE` is defined to understand this fact.

| CPU | 2VARIABLE |
|---|---|
| D | Address + 3 |
| E | Address + 2 |
| H | Address + 1 |
| L | Address + 0 |

## 4.4 Double Cell Floating-Point Number Encoding

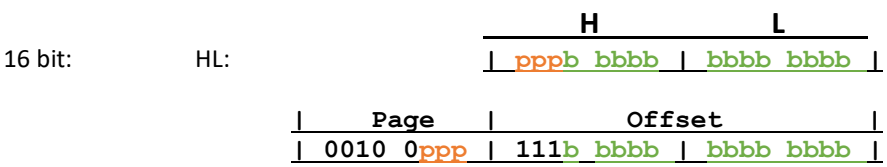
There is another optional format that use 32 bits as a *double integer*, but all bits are used in a different way to allows reopresentation a *floating point number* approximately between ±0.3E-38 and ±1.7E+38 with 6-7 precision digits.
The sign is kept in the most significant bit, the same way as a *double integer*; then eight bits follow as the exponential part, then 23 bits of mantissa or significand. The sign in this position allows (IMO) using most of the same semantics of *double integers* as per the sign of the number.




See Floating-Point Option section for more details.

## 4.5 Single Cell 16 bits Heap Pointer Address Encoding

This is Spectrum Next's peculiar 16 bits Heap Pointer Address Encoding that leverages on MMU7 i.e. Z80 memory space addresses between 0E000h and 0FFFFh. The three most significant bits represent an 8kibyte-page between 32 and 39, lower bits are taken as offset from 0E000h. A specific definition `>FAR` takes care of converting an heap-pointer address to an E000 offset and paging to MMU7 the correct 8kibyte of physical RAM. Any NextZXOS call and most of I/O operations restore page 1 at MMU7, so in most cases data stored in Heap has to be moved to PAD before being used elsewhere.



# 5 Error messages

Error messages strings are stored at Screens from # 4 to # 7 that are therefore reserved.

| Code | Message |
|---|---|
| #0 | is undefined. |
| #1 | Stack is empty. |
| #2 | Dictionary full. |
| #3 | No such a line. |
| #4 | has already been defined. |
| #5 | Invalid stream. |
| #6 | No such a block. |
| #7 | Stack full. |
| #8 | Old dictionary is full. |
| #9 | Tape error. |
| #10 | Wrong array index. |
| #11 | Invalid floating point. |
| #12 | Heap full. |
| #13 | Division by zero. |
| #14 | Patching the wrong word. |
| #17 | Can't be executed. |
| #18 | Can't be compiled. |
| #19 | Syntax error. |
| #20 | Bad definition end. |
| #21 | is a protected word. |
| #22 | Aren't loading now. |
| #23 | Forget across vocabularies. |
| #24 | RS loading error. |
| #25 | Cannot open stream. |
| #26 | Error at postit time. |
| #27 | Inconsistent fixup. |
| #28 | Unexpected fixup/commaer. |
| #29 | Commaer data error. |
| #30 | Commaer wrong order. |
| #31 | Programming error. |
| #33 | Programming error. |
| #34 | Checksum error. |
| #38 | Not a BMP file. |
| #39 | NextZXOS Opendir error. |
| #40 | NextZXOS Out of memory. |
| #41 | NextZXOS Open error. |
| #42 | NextZXOS Close error. |
| #43 | File not found. |
| #44 | NexZXOS doscall error. |
| #45 | NextZXOS pos error. |
| #46 | NextZXOS read error. |
| #47 | NextZXOS write error. |
| #50 | Incorrect result. |
| #51 | Wrong number of result. |
| #52 | Cell number before '->' does not match ...}T spec. |
| #53 | Cell number before and after '->' does not match. |
| #54 | Cell number after '->' below …}T does not match. |

# 6 The Dictionary

## 6.1 The “core” dictionary

**'*null*'  ---  (immediate)**

This is a “ghost” word executed by INTERPRET to go back to the caller once the text to be interpreted ends. This word allows you to use a **0x00** (NULL ASCII) as the end-of-text indicator in the input text stream.

**!  n  a  ---**

stores an integer n in the memory cell at address a and a + 1. Pronounced “store”.
Zilog Z80 microprocessor is a little-endian CPU that holds lower byte at lower address and higher byte in the higher address.

**!CSP  ---**

saves the value of SP register in CSP user variable. It is used by : and ; for syntax checking. ~~Also, CASE use it for the same purpose.~~

**#  d1  ---  d2**

From a double number d1 it produces the next ASCII character to be put in an output string using HOLD. The number d2 is d1 divided by BASE and is kept for subsequent elaborations. This word is used between <# and #> . See also #S.

**#>  d  ---  a  b**

terminates a numeric conversion started by <# . This word removes d and leaves the values suitable for TYPE.

**#BUFF  ---  n**

Constant, the number of available buffers. This build has 7 buffers located at address between FIRST @ and LIMIT @.

**#S  d1  ---  d2**

This word is equivalent of a series of # that is repeated until d2 becomes zero. It is used between <# and #>.

**#SEC  ---  n**

This is a constant that gives the number of available Screens/blocks.

**'  ---  cfa**

Pronounced “tick”. Used in the form

**' cccc**

this definitions leaves the **cfa** of word cccc, that is its xt or value to be compiled or passed to EXECUTE. If the word cccc is not found after the CURRENT and CONTEXT search phases, then an error #0 is raised, that is the message “cccc is undefined”. In a previous version of this Forth, this word returned **pfa**: we changed this previous standard to return **cfa**.

**(** **---** **(immediate)**

Enclose a comment. Used in the form

**( cccc )**

ignores what is between brackets. The space after ( is not considered in cccc. The comment must be delimited in the same row with a closing ) followed by a space or the end of line.

**(+LOOP)** **n** **---**

This is the primitive definition compiled by +LOOP.

**(.")** **---**

This is the primitive definition compiled by **."** and **.(**. It executes TYPE.

**(;CODE)** **---**

This is the primitive definition compiled by **;**CODE. It rewrites the **cfa** of LATEST word to make it point to the machine code starting from the following address.

**(?DO)** **---**

This is the primitive definition compiled by ?DO.
At compile-time it compiles the **cfa** of (?DO) followed by an offset as for BRANCH used to jump after the whole ?DO ... LOOP structure in case the limit equals the initial index, otherwise it is equivalent to (DO).

**(?EMIT)** **c1** **---** **c2**

Decodes the character c1 using the following table. It is used internally by EMIT.

HEX 06 → print-comma
HEX 07 → bell rings
HEX 08 → back-space
HEX 09 → tabulator
HEX 0D → carriage return
HEX 0A → new line (emitted as a 0D on the fly)

For not listed character, c2 is equal to c1.

**(ABORT)** **---**

Definition executed in case of error issued by ERROR when WARNING contains a negative number. This word usually executes ABORT but can be patched with some user defined word at the **pfa** of (ABORT).

**(COMPARE)** **a1 a2 n** **--** **b**

This word performs a lexicographic compare of n bytes of text at address a1 with n bytes of text address a2. The compare is case-sensitive or case-insensitive based on the last execution of CASEON and CASEOFF.
When executed, this word returns a numeric value

0 : if strings are equal
+1 : if string at a1 greater than string at a2
-1 : if string at a1 less than string at a2

See also CASEON and CASEOFF.

**(DO)** **---**

This is the primitive compiled by DO.

**(FIND)** **a1 a2** **--- cfa b tf**
**--- ff**

Searches in the dictionary starting from address a2 a word which text name is kept at address a1; returns a cfa, the first byte b of nfa and a tf on a successful search; elsewhere a ff only.
The search is case-sensitive or case-insensitive based on the last execution of CASEON and CASEOFF.
Address a2 must be the nfa of the first word involved in the search in the vocabulary.
In previous versions of this Forth, it returned a pfa, we change our mind.
Byte b keeps the length of the found word in the least significant 5 bits, bit 6 is the IMMEDIATE flag. Bit 5 is the SMUDGE bit. Bit 7 is always set to mark the beginning or end of the nfa.
See also CASEON and CASEOFF.

**(LEAVE)** **---**

Direct procedure compiled by LEAVE that discards the current DO-LOOP frame and executes an unconditional jump. The memory cell following (LEAVE) contains the offset to be relatively added to the Instruction Pointer to jump after the corresponding (LOOP) or (+LOOP).

**(LINE)** **n1 n2** **--- a b**

Retrieves Line n1 of Screen n2 and send it to buffer. It returns the address a within the buffer and a counter b that is C/L (=64) meaning a whole line.

**(LOOP)** **---**

This is the primitive compiled by LOOP. See also DO and +LOOP.

**(MAP)** **a2 a1 n c1 --- c2**

Translate character c1 using mapping string a2 and a2. If c1 is present within string a1 then the corresponding position within string a2 is taken as translation. If c1 is not present within string a1, then it is not translated, and c2 remanis equal to c1. n is the length of strings a1 and a2.

For example, the following definitions are used to fix the illegal characters in a filename string:

```
create ndom hex    (   : ? / * | \ < > "   )
  char : c,  char ? c,  char / c,  char * c,
  char | c,  char \ c,  char < c,  char > c,  char " c,

create ncdm hex    (   % ^ % & $ _ { } ~   )
  char _ c,  char ^ c,  char % c,  char & c,
  char $ c,  char _ c,  char { c,  char } c,  char ~ c,

: needs-ch ( a -- ) \ Replace illegal characters in filename string a
  count bounds
  Do
      ncdm ndom 9 i c@ (map) i c!
  Loop
;
```

**(NEXT)** **---** **a**

Constant. It is the address of "next" entry point for the **Inner Interpreter**. When creating word using machine code, the last op-code should be an unconditional jump to this address. If the newly created word wants to return an *integer* value on TOS, it has to do it beforehand ~~it should jump to the previous address; and if it wants to return a *double integer* value, it should jump to the next previous one~~.

This Forth implementation *always* keeps `(NEXT)` value in **IX register.** For example, to create two definitions that disables and enables interrupts, without an `ASSEMBLER`, you could use the following snippet:

```
CODE   ISR-DI   HEX
  F3 C,          \  di
  DD C, E9 C,    \  jp (ix)
  SMUDGE         \  now a dictionary search will find this word

CODE   ISR-EI   HEX
  FB C,          \  ei
  DD C, E9 C,    \  jp (ix)
  SMUDGE         \  now a dictionary search will find this word
```

**(NUMBER)** **d a** **---** **d2 a2**

Converts the ASCII text at address `a + 1` in a double integer using the current `BASE`. Number `d2` is left on top of stack for any subsequent elaborations, `a2` is the address of the first non-converted character.
In the CPU registers a double integer is kept as HLDE, on the stack is treated as two distinct integers where HL is on TOS and DE is the second from top, so that in memory it appears as LHED. Instead, in a variable declared with 2VARIABLE is stored as EDHL.
Used by NUMBER.

**(SGN)** **a** **---** **a2 f**

Determines if the character at address `a` is a sign (+ o -) and if found increments `a`. The flag f indicates the sign: `ff` for a positive sign + or no sign at all, `tf` for a negative sign - . If a is incremented then variable `DPL` is incremented as well.
Used by da `NUMBER` and `(EXP)` in the floating-point option.

***** **n1 n2** **---** **n3**

Computes the product of two integers.

***/** **n1 n2 n3** **---** **n4**

Compute (`n1` · `n2`) / `n3` using a double integer for the intermediate value to avoid precision loss.

***/MOD** **n1 n2 n3** **---** **n4 n5**

Leaves the quotient `n5` and the reminder `n4` of the operation (`n1`· `n2`) / `n3` using a double integer for the intermediate to avoid precision loss.

**+** **n1 n2** **---** **n3**

Leaves the sum of two integer.

**+!** **n a** **---**

Adds to the cell at address `a` the number `n`. It is the same as the sequence `a @ n + a !`

**+-** **n1 n2** **--- n3**

Computes `n3` as `n1` with the sign of `n2`. If `n2` is zero, it means positive.

**+BUF** **a1** **--- a2 f**

Advances the address of the buffer from `a1` to `a2`, that is the next buffer. The flag `f` is false if `a2` is the buffer pointed by `PREV`.

**+LOOP** **n1** **--- (run time)**
**a n2** **--- (compile time)**

Used in colon definition in the form

**DO ... n1 +LOOP**

At run-time `+LOOP` checks the return to the corresponding DO, `n1` is added to the index and if the index did not cross the boundary between the loop limit minus one and the loop limit, the execution jumps back to the beginning of the loop. Otherwise the execution leaves the loop. On leaving the loop, the parameters are discarded and the execution continues with the following word.
At compile-time `+LOOP` compiles `(+LOOP)` and a jump is calculated from `HERE` to `a` which is the address left on the stack by `DO`. The value `n2` is used internally for syntax checking.

**+ORIGIN** **n** **--- a**

Returns the address n bytes after the “origin”. In this build the origin is 6400h. Used rarely to modify the boot-up parameters in the origin area.

**+TO** **n** **--- cccc**

Used in the form

**n +TO cccc**

If not compiling, add the value `n` to `cccc` . At compile-time it compiles a literal pointer to `cccc`'s PFA followed by a plus-store-command `(+!)` so that later, at run-time the literal is used by the `!` word to alter `cccc` value.
Word `cccc` was created via `VALUE`. See also `TO`. This definition is available after `NEEDS +TO`.

**,** **n** **---**

It puts `n` in the following cell of the dictionary and increments `DP` (dictionary pointer) of two locations.

**,"** **---**

Compile a “Counted-ZString”. It calls `WORD` to read characters from the current input stream up to a delimiter `"` and stores such a string at `HERE`. In a “Counted-ZString” the length of the string is stored as the first byte and the string itself ends with a NUL character (0x00). For example

```
," Hello"
```

compiles: **05 48 65 6C 6C 6F 00**

where 05 is the length of “Hello” string which is followed by a 00 ‘nul’ character.

**-  n1  n2  ---  n3**

Computes `n3` = `n1` – `n2` as the difference from the penultimate and the last number on the stack.

**-->  ---**

Continues the interpretation in the next Screen during a `LOAD` .

**-1  ---  n**

This is the constant value `-1` that in this implementation is 0FFFFh. Compiling a constant result in a faster execution than a literal.

**-DUP  n  ---  n  n (non zero)**
**n  ---  n  (zero)**

Duplicates `n` if it is non zero. This word is available only for backward compatibility. See also `?DUP`.

**-FIND  ---  cfa  b  tf  (ok)**
**---  ff  (ko)**

Used in the form `-FIND cccc`.
It accepts a word (delimited by spaces) from the current input stream, storing it at address `HERE`. Then, it run a search in the `CONTEXT` vocabulary first, then in the `CURRENT` vocabulary. If the word is found, it leaves the `cfa` of the word, its length-byte `b` and a `tf`. Otherwise only a `ff`.

**-TRAILING  a1  n1  ---  a2  n2**

This definition assumes that a string `n1` characters long is already stored at address `a1` containing a space right-delimited word. It determines `n2` as the position of the first delimiter after the word.

**.  n  ---**

Emits the integer `n` followed by a space.

**."  ---  (immediate)**

Used in the form

```
."  cccc"
```

At compile-time, within a colon-definition, it accepts text from the input sources until a quote character (") is encountered, then it compiles the primitive to output the text followed by the string `cccc` . The text `cccc` is prepended by a length-counter that `TYPE` will use at run-time.
When interpreted, i.e. outside a colon-definition, immediately emits the text to output.

**.(  ---  (immediate)**

Used in the form

```
.( cccc)
```

acting as `."  cccc  "` but the string is delimited by a closed-parenthesis.

**.C** **c** **---** **(immediate)**

Used in the form

```
c .C xxxx C
```

acting as `." xxxx"` but the string is delimited by character `c`. It is a more generic form of `.(` and `."` that, in fact, use this word as their primitive.

**.LINE** **n1** **n2** **---**

Sends line `n1` of block `n2` to the current peripheral ignoring the trailing spaces.

**.R** **n1** **n2** **---**

Prints a number `n1` right aligned in a field `n2` character long, with no following spaces. If the number needs more than `n2` characters, the excess protrudes to the right.

**/** **n1** **n2** **---** **n3**

Computes `n3` = `n1`/ `n2`, the quotient of the integer division. This system uses floored division via `M/MOD` and implements `UM/MOD` in machine-code, `FM/REM` and `SM/MOD` as derived definitions. See `M/MOD` for details.

**/MOD** **n1** **n2** **---** **n3** **n4**

Computes the quotient `n4` and the reminder `n3` of the integer division `n1`/ `n2`. The reminder has the sign of `n1`. This system uses floored division via `M/MOD` and implements `UM/MOD` in machine-code, `FM/REM` and `SM/MOD` as derived definitions. See `M/MOD` for details.

**0** **---** **n**

This is a constant value zero. Compiling a constant results in a faster execution than a literal.

**0<** **n** **---** **f**

Leaves a `tf` if `n` is less than zero, `ff` otherwise.

**0=** **n** **---** **f**

Leaves a `tf` if `n` is not zero, `ff` otherwise. It is like a NOT `n`.

**0>** **n** **---** **f**

Leaves a `tf` if `n` is greater than zero, `ff` otherwise.

**0BRANCH** **f** **---**

Direct procedure that executes a conditional jump. If `f` is zero the offset in the cell following `0BRANCH` is added to the Instruction Pointer to jump forward of backward.
It is compiled by `IF`, `UNTIL` and `WHILE`.

**1** **---** **n**

Constant value 1. Compiling a constant results in a faster execution than a literal.

**1+** **n1** **---** **n2**

Increments by one the number on TOS.

**1-** **n1** **---** **n2**

Decrements by one the number on TOS.

**2** **---** **n**

Constant value 2. Compiling a constant results in a faster execution than a literal.

**2!** **d a** **---**
**n-lo n-hi a** **---**

Stores the double integer held on TOS to address a.

**2*** **n1** **---** **n2**

Doubles the number on TOS.

**2+** **n1** **---** **n2**

Increments by two the number on TOS.

**2-** **n1** **---** **n2**

Decrements by two the number on TOS.

**2/** **n1** **---** **n2**

Halves the number on TOS.

**2@** **a** **---** **d**
**a** **---** **n-lo n-hi**

Fetches the double integer at address a. to TOS.

**2CONSTANT** **d** **---** **(immediate)** **(compile time)**
**---** **d** **(run time)**

Defining word that creates a double constant. Used in the form

**d 2CONSTANT cccc**

it creates the word `cccc` and `pfa` holds the number d. When `cccc` is later executed it put `d` on TOS.
This definition is not available at startup, it has to be loaded via `NEEDS 2CONSTANT`.

**2DROP** **d** **---**
**n1 n2** **---**

Discards a double integer from the TOS, i.e. discards the top two integer.

**2DUP** **d** **--- d d**
**n1 n2** **--- n1 n2 n1 n2**

Duplicates the double integer on TOS, i.e. duplicates in order the two top integer.

**2OVER** **d1 d2** **--- d1 d2 d1**
**n1 n2 n3 n4 --- n1 n2 n3 n4 n1 n2**

Copies to TOS the second double integer from top.
This word isn't available at startup and must be included via `NEEDS 2OVER`.

**2ROT** **d1 d2 d3** **--- d2 d3 d1**
**n1 n2 n3 n4 n5 n6** **--- n3 n4 n5 n6 n1 n2**

Rotates the three top double integers, taking the third and putting it on top. The other two double integers are pushed down from top by one place. This word isn't available at startup and must be included via `NEEDS 2ROT`.

**2SWAP** **d1 d2** **--- d2 d1**

Swaps the two double integers on TOS.

**2VARIABLE** **d** **--- (immediate)** **(compile time)**
**--- a** **(run time)**

Defining word used in the form:

```
d 2VARIABLE cccc
```

creates the word `cccc` with the pfa containing the initial value `d`. When `cccc` is executed, it puts on TOS the pfa of `cccc` that is the address that holds the value `d`.
When used in the form

```
cccc @
```

the content of the double-variable `cccc` is left on TOS.
When used in the form

```
d cccc !
```

the double-value on TOS is stored to the double-variable `cccc`.
This definition is not available at startup, it must be loaded via `NEEDS 2CONSTANT`.

**3** **--- n**

Constant value 3. Compiling a constant results in a faster execution than a literal.

**3DUP** **n1 n2 n3** **--- n1 n2 n3 n1 n2 n3**

Duplicates the three top integer on Stack. This definition is available after `NEEDS 3DUP`.

**:** **---** **(immediate)**

This is a defining word that creates and begins a colon-definition. Used in the form

```
: cccc ... ;
```

creates in the dictionary a new word `cccc` so that it executes the sequence of already existing words '`...`'.
The `CONTEXT` vocabulary is set to be the `CURRENT` and compilation continues while `STATE` is not zero. Words having the bit 6 of its length-byte set are immediately executed instead of being compilated.

**:NONAME** **---** **xt**

This is a defining word that creates and begins a name-less colon-definition. It returns the xt of the word being defined. Such `xt` should be kept in some way, for example as a `CONSTANT`. Used in the form

```
:NONAME  ...  ;
CONSTANT cccc
```

This definition is available after `NEEDS :NONAME` but the file that contains this definition is named %noname.f

**;** **---** **(immediate)**

Ends a colon definition and stops compilation.. It compiles `EXIT` and executes `SMUDGE` to make the word accessible.

**;CODE** **---** **(immediate)**

Used in the form

```
: cccc ... ;CODE
```

terminates a colon definition stopping compilation of word `cccc` and compiling `(;CODE)`.
Usually `;CODE` is followed by a suitable machine-code sequence.

**;S** **---**

This is usually the last word compiled in a colon definition by `;` it does the action of returning to the calling word. It is used to force the immediate end of a loading session started by `LOAD`.
**Obsolete**, prefer `EXIT`. This word isn't available at startup and must be included via `NEEDS ;S` sequence.

**<** **n1 n2** **---** **f**

Leaves a `tf` if `n1` is less than `n2`, `ff` otherwise.

**<#** **---**

Sets `HLD` to the value of `PAD`. It is used to format numbers using `#`, `#S`, `SIGN` and `#>`. The conversion is performed using a double integer, and the formatted text is kept in `PAD`.

**<>** **n1 n2** **---** **f**

Leaves a `tf` if `n1` isn't equal to `n2`, `ff` otherwise. This word isn't available at startup and must be included via `NEEDS <>` sequence and the file loaded is {}.f

**<BUILDS** **---**

Used in a colon definition in the form

```
: cccc ... <BUILDS ... DOES> ... ;
```

Subsequent execution of `cccc` in the form

```
cccc nnnn
```

creates a new word `nnnn` with an high-level procedure that at run-time calls the `DOES>` part of `cccc`. When `nnnn` is executed, the pfa of `nnnn` is put on TOS and the executed the following `DOES>`.
`<BUILD` and `DOES>` allow writing high-level procedures instead of using machine code as `;CODE` would require.

The "Floating Point Option Library" available via `NEEDS FLOATING` provides a good example of use of this structure.

**<NAME** **cfa** **---** **nfa**

Converts a `cfa` in its `nfa`. It is the same as `>BODY NFA` sequence.
See also: `CFA`, `LFA`, `NFA`, `PFA`, `>BODY`.

**=** **n1 n2** **---** **f**

Leaves a `tf` if `n1` equals to `n2`, `ff` otherwise.

**>** **n1 n2** **---** **f**

Leaves a `tf` if `n1` is greater than `n2`, `ff` otherwise.

**>BODY** **cfa** **---** **pfa**

Converts a `cfa` in its `pfa`.
See also: `CFA`, `LFA`, `NFA`, `PFA`, `<NAME`.

**>IN** **---** **a**

User variable that keeps track of text position within an input buffer. `WORD` uses and modifies the value of `IN` that is incremented when consuming input buffer.

**>R** **n** **---**

Takes an integer from TOS and puts it on top of the Return Stack. It should be used only within a colon-definition *and* the use of `>R` should be balanced with a corresponding `R>` within the same colon-definition.

**?** **a** **---**

Prints the content of cell at address `a`. It is the same as the sequence: `a @`.

**?COMP** **---**

Raises an error message #17 if the current `STATE` is not compiling state.

**?CSP** **---**

Raises an error message #20 if the value of CSP is different from the current value of SP register. It is used to check the compilation in a colon definition.

**?DO** **n1 n2** **---** **(immediate)** **(run time)**
**---** **a n** **(compile time)**

Used in a colon definition in the form

```
?DO ... LOOP
?DO ... n3 +LOOP
```

It is used as DO to put in place a loop structure, but at run-time it first checks if n1 = n2 and in that case the loop is skipped. At run-time ?DO starts a sequence of words that will be repeated under control of an initial-index n2 and a limit n1. ?DO consumes these two value from stack and the corresponding LOOP increments the index. If the index did not cross the boundary between the loop limit minus one and the loop limit, then the executions returns to the corresponding ?DO, otherwise the two parameters are discarded and the execution continues after the LOOP.
The limit n1 and the initial value n2 are determined during the execution and can be the result of other previous operations. Inside a loop the word I copies to TOS the current value of the index.
Se also: I, DO, LOOP, +LOOP, LEAVE. In particular LEAVE allows leaving the loop at the first opportunity.
At compile-time ?DO compiles (?DO) followed by an offset like BRANCH and leaves the address of the following location and the number n to syntax-check

**?DO-** **[a1 n1] a n** **---**

This is a peculiar definition equivalent to BACK but fitted for ?DO. It computes and compiles a relative offset from a to HERE and, in the case, it completes the BRANCH part previously compiled by ?DO that left a1 and n1. It is used by LOOP, +LOOP. If the loop begins with DO then a1 and n1 won't be there and no BRANCH will be compiled.

**?DUP** **n** **---** **n n (non zero)**
**n** **---** **n (zero)**

Duplicates the value on TOS if it is not qual to zero. This is the same as –DUP.

**?ERROR** **f n** **---**

Raises an error message #n if f is true.

**?EXEC** **---**

Raises an error message #18 if we aren't compiling.

**?LOADING** **---**

Raises an error message #22 if we aren't loading. It show the illegal use of --> .

**?PAIRS** **n1 n2** **---**

Raises an error message #19 if n1 is different from n2. It is used for syntax checking by the words that completes the construction of structures DO, BEGIN, IF, CASE.

**?STACK** **---**

Raises an error message #1 if the stack is empty and we tried to consume an element from the calculator stack.
On the other hand, an error message #7 if the stack is full.

**?TERMINAL** **---** **f**

Tests the keyboard for a [BREAK] key-press. Leaves a `tf` if the [BREAK] key is pressed, `ff` otherwise. Useful to stop an indefinite loop, for example:

```
BEGIN ... ?TERMINAL UNTIL
```

**@** **a** **---** **n**

Reads cell at address `a` and put an integer on TOS.

**ABORT** **---**

Clears the stack and pass to prompt command, ~~prints the copyright message~~ and returns the control to the human operator executing `QUIT`.

**ABS** **n** **---** **u**

Leaves the absolute value of `n`.

**ACCEPT** **a n1** **---** **n2**

Transfers characters from the input terminal to the address `a` for `n1` location or until receiving a 0x13 "CR" character. A 0x00 "null" character is added. It leaves on TOS `n2` as the actual length of the received string. More, `n2` is also copied in `SPAN` user variable. See also `QUERY`.

**ACCEPT-** **a n1** **---** **n2**

As for `ACCEPT`, but it reads at most `n1` characters text from current channel/stream via `INKEY` one character at a time, It stores the text at address `a`. Not so efficient, but it allows to compile an external souce-file attached to a Basic's OPEN# stream. It does not modify `SPAN`.

**AGAIN** **---** **(immediate)** **(run time)**
**a n** **---** **(compile time)**

Used in colon definition in the form

```
BEGIN ... AGAIN
```

At run-time `AGAIN` forces the jump to the corresponding `BEGIN` and has no effect on the calculator stack. The execution cannot leave the loop (at least until a `R>` is executed at a lower level).
At compile-time `AGAIN` compiles `BRANCH` with an offset from `HERE` to `a`. The number `n` is used for syntax-check.

**ALLOT** **n** **---**

This definition is used to reserve some space in the dictionary or to free memory. It adds the signed integer `n` to `DP` (Dictionary Pointer) user variable.

**ALIGN** **---**

force HERE to an even address. This definition is available after NEEDS ALIGN.

**ALIGNED** **a1** **---** **a2**

force a1 to an even address. This definition is available after NEEDS ALIGNED.

**AND** **n1 n2** **---** **n3**

It executes an bitwise AND operation between the two integers. The operation is performed bit by bit.

**AUTOEXEC** **---**

This word is executed the first time the Forth system boots and **loads Screen# 11**. Once called, it patches ABORT definition to prevent any further executions at startup. Anyway, you can still invoke it directly.
This allows you to perform some automatic action at startup.

**B/BUF** **---** **n**

Constant. Number of bytes per buffer. In this implementation is 512.

**B/SCR** **---** **n**

Constant that indicates the number of Blocks per Screen. In Next version is 2, that means a Screen is 1024 byte long. In Microdrive version it was 1…

**BACK** **a** **---**

Computes and compiles a relative offset from a to HERE. Used by AGAIN, UNTIL, LOOP, +LOOP.

**BASE** **---** **a**

User variable that indicates the current numbering base used in input/output conversions. It is changed by DECIMAL that put ten, HEX that put sixteen, and with some extensions BINARY that put two and OCTAL that put eight.

**BASIC** **u** **---**

Quits Forth and returns to Basic returning to the caller USR the unsigned integer on TOS.

**BEGIN** **---** **(immediate)** **(run time)**
**---** **a n** **(compile time)**

Used in colon definition in one of the following forms

```
BEGIN ...   AGAIN              or
BEGIN ... f UNTIL              or
BEGIN ... f WHILE ... REPEAT   or
BEGIN ... f END
```

At compile-time, it starts one of these structures.
At run-time BEGIN marks the beginning of a words sequence to be repeatedly executed and indicates the jump point for the corresponding AGAIN, REPEAT, UNTIL or END.
With UNTIL, the jump to the corresponding BEGIN happens if on TOS there is a ff, otherwise it quits the loop.

With AGAIN and REPEAT, the jump to the corresponding BEGIN always happens.
The WHILE part is executed if and only if on TOS there is a tf, otherwise it quits the loop.

**BINARY** **---**
Sets BASE to 2, that is the binary base. This definition is available after NEEDS BINARY.

**BL** **---** **c**
Constant for “Blank”. This implementation uses ASCII and BL is 32.

**BLANK** **a** **n** **---**
Fills with “Blank” n location starting from address a.

**BLK** **---** **a**
User variable that indicates the current block to be interpreted. If zero then the input is taken from the terminal buffer TIB.

**BLK-FH** **---** **a**
Variable containing file-handle to Block's file !Blocks-64.bin.

**BLK-FNAME** **---** **a**
Variable containing the counted-zstring “!Block-64.bin”. as produced by ,” definition.
See also ,” definition.

**BLK-INIT** **---**
Initialize BLOCK system. It opens for update (read/write) file “!Block-64.bin” .

**BLK-READ** **a** **n** **---**
Read block n to address a. See also F_READ.

**BLK-SEEK** **n** **---**
Seek block n within blocks!.bin file. See also F_SEEK.

**BLK-WRITE** **a** **n** **---**
Take text content at address a to disk block n. See also F_WRITE.

**BLOCK** **n** **---** **a**
Leaves the address of the buffer that contains the block n. If the block isn’t already there, it is fetched from disk. If in the buffer there was another buffer and it was modified, then it is re-written to disk before reading the block n.
See also BUFFER, R/W, UPDATE, FLUSH.

**BOUNDS** **a n** **---** **a+n a**

Given an address and a length ( a n ) calculate the bound addresses suitable for `DO ... LOOP`.
It is used by `TYPE`.

**BRANCH** **---**

Direct procedure that executes an unconditional jump. The memory cell following `BRANCH` has the offset to be relatively added to the Instruction Pointer to jump forward or backward. It is compiled by `AGAIN`, `ELSE`, `REPEAT`.

**BUFFER** **n** **---** **a**

Makes the next buffer available assigning it the block number `n`. If the buffer was marked as modified (by `UPDATE`), such buffer is re-written to disk. The block is not read from disk. The address point to the first character of the buffer.

**BYE** **---**

Executes `FLUSH` and `EMPTY-BUFFERS`, then quits Forth and returns to Basic returning to the caller USR the value of `0 +ORIGIN`. See also `BASIC`.

**C!** **b a** **---**

Stores a byte `b` to address `a`.

**C,** **b** **---**

Puts a byte `b` in the next location available in the dictionary and increments `DP` (dictionary pointer) by 1.

**C/L** **---** **c**

Constant that indicate the number of characters per screen line. In this implementation it is 32.

**C@** **a** **---** **b**

Puts on TOS the byte at address `a`.

**CALL#** **n1 a** **---** **n2**

Performs a CALL to the routine at address `a`. First argument `n1` is passed via bc register *and* a register. The routine can return bc register which is pushed on TOS. This definition is useful to call normal ZX Spectrum ROM routines.
This definition is available after `NEEDS CALL#`.

**CASEOFF** **---**

Sets case-sensitive search OFF. changes the system behavior so that `(FIND)` can search the dictionary ignoring case, and `(COMPARE)` compares two strings ignoring case.

**CASEON** **---**

Sets case-sensitive search ON. It changes the system behavior so that (FIND) will search the dictionary case sensitive, and (COMPARE) will compare the two strings case sensitive.

**CELL** **---** **2**

In this implementation a cell is two bytes. This definition is available after `NEEDS CELL`.

**CELL+** **n1** **---** **n2**

Increments `n1` by 1 “cell”, that is two units. In this implementation a cell is two bytes.

**CELL-** **n1** **---** **n2**

Decrements `n1` by 1 “cell”, that is two units. In this implementation a cell is two bytes.

**CELLS** **n1** **---** **n2**

Doubles the number `n1` on TOS giving the number of bytes equialent to `n1` “cells”. In this implementation a cell is two bytes.

**CFA** **pfa** **---** **cfa**

Converts a `pfa` in its `cfa`. See also `LFA`, `NFA`, `PFA`, `>BODY`, `<NAME`.

**CHAR** **---** **c**

Used in the form

```
CHAR c
```

determines the first character of the next word in the input stream.

**CLS** **---**

Clears the screen using the ZX Spectrum ROM routine 0DAFh.

**CMOVE** **a1** **a2** **n** **---**

Copies the content of memory starting at address `a1` for `n` bytes, storing them from address `a2`. The content of address `a1` is moved first. See also `CMOVE>`.

**CMOVE>** **a1** **a2** **n** **---**

The same as `CMOVE` but the copy process starts from location `a1 + n - 1` proceding backward to the location `a1`.

**CODE** **---**

Defining word used in the form

```
CODE cccc
```

it creates a new dictionary entry for the definition `cccc` with the cfa ~~of such a definition pointing to its pfa~~ that is empty for the moment, `HERE` points that location; then some machine-code instruction should be added using `C,` that will be compiled from `HERE` onwards. The new word is created in the `CURRENT` vocabulary but won’t be found by `(FIND)` because it has the `SMUDGE` bit set. Once the word construction is complete, it is programmer's responsibility to execute `SMUDGE` to make visible.
This word is redefined / overridden by `ASSEMBLER` vocabulary available after `LOAD`ing Screens 100-165, this allows the programmer to use a pseudo-standard Z80 notation to create a new low-level definition using assembler directly.

Here is an example that creates a definition SYNC-FRAME to wait for the next maskable interrupt:

```
CODE SYNC-FRAME HEX
      76 C,        \ halt       ;  wait for interrupt or reset
      DD C, E9 C, \ jp (ix)     ;  jump to the inner interpreter
      SMUDGE
```

**COLD** **---**

This word executes the Cold Start procedure that restore the system at its startup state.
It sets DP to the minimum standard and executes ABORT.

**COMPILE** **---**

Used in the form

```
COMPILE cccc
```

At compile-time, it determines the xt of the word that follows COMPILE and compile it in the next dictionary cell.

**COMPILE,** **xt** **---**

Used within a colon-definition, it puts xt in the following cell of the dictionary and increments DP (dictionary pointer) of two locations.

**CONSTANT** **n** **---** **(immediate)** **(compile time)**
**---** **n** **(run time)**

Defining word that creates a constant. Used in the form

```
n CONSTANT cccc
```

it creates the word cccc and pfa holds the number n. When cccc is later executed it put n on TOS.

**CONTEXT** **---** **a**

User variable that points to the vocabulary address where a word search begins.

**COUNT** **a1** **---** **a2** **b**

Leaves the address of text a2 and a length b. It expects that the byte at address a1 to be the length-counter and the text begins to the next location.

**CR** **---**

Transmits a 0x0D to the current output peripheral.

**CREATE** **---** **(compile time)**
**---** **a** **(run time)**

Defining word used in the form

```
CREATE cccc
```

it creates a new dictionary entry for the definition cccc with the **pfa** still empty.

When cccc is executed, it puts on TOS the **pfa** of cccc
Often used with ALLOT to reserve space in the dictionary to be later used, for instance as an array.
See also VARIABLE.

**CSP** **---** **a**

User variable that temporarily holds the value of SP register during a compilation syntax error check.

**CURRENT** **---** **a**

User variable that points to the address in the Forth vocabulary where a search continues after a failing search executed in the CONTEXT vocabulary. See also LATEST.

**CURS** **---**

Shows a (flashing) cursor on current video position and wait for a keypress.
Depending on CAPS-LOCK state, the faces of flashing cursor are different depending on the content of a few bytes in ORIGIN area:

```
HEX
026 +ORIGIN C@ . → 8F      Full square graphic character
027 +ORIGIN C@ . → 8C      Lower-half square graphic character
028 +ORIGIN C@ . → 5F      Underscore character
```

When CAPS-LOCK is On the cursor switches between (8F) and _ (5F)
When CAPS-LOCK is Off the cursor switches between (8F) and (8C)
You can modify this behavior putting some suitable values on these three bytes. For example you can make disappear the flashing cursor using the following patch:

```
HEX
BL 026 +ORIGIN C!
BL 027 +ORIGIN C!
BL 028 +ORIGIN C!
```

**D+** **d1 d2** **---** **d3**

Computes d3 as the sum of d1 and d2. This is a 32 bits sum.

**D+-** **d1 n** **---** **d2**

Forces the double integer d1 to have the the sign of n.
It is used by DABS .

**D-** **d1 d2** **---** **d3**

Subtract d2 from d1. This is a 32 bits subtraction. Available after NEEDS D-.

**D.** **d** **---**
**n-lo n-hi** **---**

Emits a double integer followed by a space. The double integer is kept on stack in the format n-lo n-hi and the integer on TOS is the most significant.

**D.R** **d n ---**

Emits a double integer rigth aligned in a field n character wide. No space follows. If the field is not large enough, then the

excess protrudes to the right.

**D0=** **d** **---** **f**

True if d1 = 0. This is a 32 bits comaprison. Available after NEEDS D0=.

**D<** **d1** **d2** **---** **f**

True if d1 < d2 . This is a 32 bits comaprison. Available after NEEDS D<.

**D=** **d1** **d2** **---** **f**

True if d1 equals d2 . This is a 32 bits comaprison. Available after NEEDS D=.

**DABS** **d** **---** **ud**

Leaves the absolute value of a double integer.

**DECIMAL** **---**

Sets BASE to 10, that is the decimal base.

**DEFINITIONS** **---**

To be used in the form

**cccc DEFINITIONS**

it sets the CURRENT vocabulary to be the CONTEXT vocabulary and this allows adding new definitions to cccc vocabulary.
For example: FORTH DEFINITIONS or ASSEMBLER DEFINITIONS.
In this implementation a Forth oriented ASSEMBLER vocabulary is available as an extra-option that can be LOADed from Screens 100 -160.

**DEVICE** **---** **a**

Variable that holds the number of current channel: 2 for video, 3 for printer, and any number between 4 and 15 to refer to a Basic OPEN# channel.

**DIGIT** **c** **n** **---** **u** **tf** **(ok)**
**c** **n** **---** **ff** **(ko)**

Converts the ASCII character c in the equivalent number using the base n, followed by a a tf. If the conversion fails it leaves a ff only.

**DLITERAL** **d** **---** **d** **(immediate)** **(run time)**
**d** **---** **(compile time)**

Same as LITERAL but a 32 bits number is compiled. DLITERAL is an immediate word that is executed and not compiled.

**DMAX** **d1 d2 --- d3**
Leaves the maximum between d1 and d2. Availabler after NEEDS DMAX.

**DMIN** **d1 d2 --- d3**
Leaves the minimum between d1 and d2. Availabler after NEEDS DMIN.

**DNEGATE** **d1 --- d2**
Computes the opposite double number.

**DO** **n1 n2 --- (immediate) (run time)**
**--- a n (compile time)**
Used in colon definition in the form

**DO ... LOOP** or
**DO ... n +LOOP**

It is used to put in place a loop structure: The execution of DO starts a sequence of words that will be repeated, under control of an initial-index n2 and a limit n1. DO drops these two value from stack and the corresponding LOOP increments the index. If the index did not cross the boundary between the loop limit minus one and the loop limit, then the executions returns to the corresponding DO, otherwise the two parameters are discarded and the execution continues after the LOOP.
The limit n1 and the initial value n2 are determined during the execution and can be the result of other previous operations. Inside a loop the word I copies to TOS the current value of the index.
See also: I, DO, LOOP, +LOOP, LEAVE. In particular LEAVE allows leaving the loop immediately ~~at the first opportunity~~.
At compile-time DO compiles (DO) and leaves the address of the following location and the number n to syntax-check.

**DOES>** **---**
Word that defines the execution action of a high-level defining word. DOES> changes the pfa of the word being defined to point the words sequence compiled after DOES>. It is used in conjunction with <BUILDS. When the machine-code part of DOES> is executed, it leaves on TOS the pfa of the new word, this allows the interpreter to use this area. Obvious uses are new vocabularies (Assembler), multidimensional array and other compiling operations.

The "Floating Point Option Library" available via NEEDS FLOATING provides a good example of use of this structure.

**DP** **--- a**
User variable (Dictionary Pointer) that holds the address of next available memory location in the dictionary. It is read by HERE and modified by ALLOT.

**DPL** **--- a**
User variable that holds the number of digits after the decimal point during the interpretation of double integer. It can be used to keep track of the column of the decimal point during a number format output. For 16 bit integer it defaults to –1. It takes into account the exponential part and its sign for floating point numbers.

**DROP** **n ---**
Drops the value on TOS. See also OVER, NIP, TUCK, SWAP, DUP, ROT.

**DUP** **n** **--- n n**

Duplicates the value on TOS. See also OVER, DROP, NIP, TUCK, SWAP, ROT.

**DUP>R** **n** **--- n**

Copy TOS to the Return Stack. See also DUP, >R, R@.

**DU<** **ud1 ud2** **--- f**

True if ud1 < ud2. This is a 32 bits comaprison. Available after NEEDS D<.

**ELSE** **a1 n1** **--- a2 n2** **(immediate) (compile time)**
**--- (run time)**

Used in colon definition in the form

**IF ... ELSE ... ENDIF**
**IF ... ELSE ... THEN**

At run-time ELSE forces the execution of the false part of an IF-ELSE-ENDIF structure. It has no effects on the stack.
At compile-time ELSE compiles BRANCH and prepares the following cell for the relative offset, stores at a1 the previous offset from HERE; then it leaves a2 and n2 for syntax checking.

**EMIT** **c** **---**

Sends a printable ASCII character to the current output peripheal. OUT is incremented. 7 EMIT activates an acoustic signal. The 'null' 0x00 ASCII character is not transmitted.

**EMITC** **b** **---**

Sends a byte b character to the current output peripheal selected with SELECT. See also DEVICE.

**EMPTY-BUFFERS** **---**

Erases all buffers. Any data stored to buffers after the previous FLUSH is lost.

**ENCLOSE** **a c** **--- a n1 n2 n3**

Starting from address a, and using a delimiter character c, it determines the offset n1 of the first non-delimiter character, n2 of the first delimiter after the text, n3 of first character non enclosed.
This word doesn't go beyond a 'null' ASCII that represent a unconditional delimiter. For example:

```
1:    c  c  x  x  x  c  x      →    2  5  6
2:    c  c  x  x  x  'null'   →    2  5  5
3:    c  c  'null'            →    2  3  2
```

**END** **a n** **--- (immediate) (compile time)**
**f** **--- (run time)**

Synonym of UNTIL.

**ENDIF a n --- (immediate) (compile time)**

At run-time, ENDIF indicates the destination of the forward jump from IF or ELSE. It marks the end of a conditional structure. THEN is a synonym of ENDIF.
At compile-time ENDIF calculates the forward jump offset from a to HERE and store it at a. The number n is used for syntax checking.

**ERASE a n ---**

Erases n memory location starting from a, filling them with 'null' characters (0x00).

**ERROR b --- n1 n2**
**--- ff**

Notifies an error b and resets the system to command prompt. First of all, the user variable WARNING is examined.
If WARNING is 0 then the offending word is printed followed by a "?" character and a short message "MSG#n".
If WARNING is 1, instead of the short message, the text available on line b of block 4 (of drive 0) is displayed. Such a number can be positive ~~or negative~~ and lay beyond block 4.
If WARNING is –1 then ABORT is executed, which resets the system to command prompt. The user can (with care) modify this behavior of that by altering (ABORT).
If BLK is non zero, then ERROR leaves on the stack n1 that is the value of IN and n2 that is the value of BLK at the error moment. These numbers can then be used by WHERE to determine and show the exact error position.
If BLK is zero, then only a ff is left on TOS.
In all cases, the final action is QUIT.

**EXEC: n ---**

Vectorised fast case structure.
Used in colon definition in the form

```
: MY_ACTION_LIST ( n -- )
  EXEC:
      word0  \ executed when n = 0
      word1  \ executed when n = 1
      word2  \ executed when n = 2
   ...
;
```

to execute the word indexed by n.
**Warning**: there is no run-time checking on n and if n is out of range, then a crash is likely to happen.

**EXECUTE cfa ---**

Executes the word which cfa is held on TOS.

**EXIT ---**

This is (usually) the last word compiled in a colon definition by ; doing the action of returning to the calling word. It is used to force the immediate end of a loading session started by LOAD.

**EXP --- a**

User variable that holds the exponent in a floating-point conversion. It is not used until the **Floating Point Option** is enabled via NEEDS FLOATING.

**EXPECT a n ---**

Transfers characters from the input terminal to the address `a` for `n` location or until receiving a 0x13 “CR” character. A 0x00 “null” character is added in the following location. The actual length of the received string is kept in `SPAN` user variable. See also `ACCEPT`. This word isn't available at startup, it must be loaded using `NEEDS EXPECT`.

**FENCE --- a**

User variable that holds the (minimum) address to where `FORGET` can act.

**FILL a n b ---**

Fills `n` memory location starting from address `a` with the value of `b`.

**FIRST --- a**

User variable that holds the address of the first buffer. See also `LIMIT`.

**FLD --- a**

User variable that holds the width of output field.

**FLIP n1 --- n2**

Exchange high and lower byte of n1. Available after `NEEDS FLIP`.

**FLUSH ---**

Executes `SAVE-BUFFERS`. It saves to disk the buffers marked “modified” by `UPDATE`.

**FM/MOD d n1 --- n2 n3**

Floored Division. Leaves the quotient `n3` and the reminder `n2` of the integer division of `d` /`n1`.
This system has only UM/MOD coded in machine-code.

| Dividend | Divisor | Remainder | Quotient |
|---|---|---|---|
| 10 | 7 | 3 | 1 |
| -10 | 7 | 4 | -2 |
| 10 | -7 | -4 | -2 |
| -10 | -7 | -3 | 1 |

**FORGET ---**

Used in the form

```
FORGET cccc
```

removes from the dictionary the word `cccc` and all the preceding definitions. Care must be put when more than one `VOCABULARY` is involved. Use `MARKER` instead.
See also `DP`.

**FORTH --- (immediate)**

This is the name of the first vocabulary. Executing FORTH sets this to be the CONTEXT vocabulary. As soon as no new vocabulary is defined, all new colon definitions became part of FORTH vocabulary. FORTH is immediate, so it is executed during the creation of a colon definition to select the needed vocabulary. See also ASSEMBLER (optional vocabulary).

**F_CLOSE n --- f**

NextZXOS primitive: it closes a file handle n previously opened with F_OPEN. Flag f is 0 for OK. It uses an RST 8 call followed by $9B service number.

**F_FGETPOS n --- d f**

NextZXOS primitive: given an open file handle n returns the position d. Flag f is 0 for OK.

**F_GETLINE a n1 fh --- n2**

Given a filehandle read at most n1 characters as the next line (terminated with $0D or $0A) and stores it at address a and returns n2 as the number of bytes read, i.e. the length of line just read.

**F_INCLUDE n ---**

Given an open file-handle n, this definition includes the source from file. This definition is used by INCLUDE and NEEDS.

**F_OPEN a1 a2 n1 --- n2 f**

NextZXOS primitive: it opens a file using filespec given at address a1 and returns filehandle number n, n1 is "mode" as specified in "NextZXOS and esxDOS APIs" standard documentation. Filespec is a NUL-termianted string. Flag f is 0 for OK. It uses an RST 8 call followed by $9A service number. See F_CLOSE.

**F_READ a n1 n2 --- n3 f**

NextZXOS primitive: it reads at most n1 bytes from file handle n2 and stores them at address a. Returns n3 as the actual bytes read. Flag f is 0 for OK. It uses RST 8 call followed by $9D service number.

**F_SEEK d n ---**

NextZXOS primitive: it seeks position d at open file given by filehandle n. It uses an RST 8 call followed by $9F service number. Flag f is 0 for OK.

**F_SYNC n --- f**

NextZXOS primitive: it syncs to disk open file given by filehandle n. It uses an RST 8 call followed by $9C service number. Flag f is 0 for OK.

**F_WRITE a n1 n2 --- n3 f**

NextZXOS primitive: it takes n1 bytes at address a and writes them to filehandle n2. It uses an RST 8 call followed by $9F service number. Returns n3 as the actual bytes written. Flag f is 0 for OK.

**HANDLER --- a**

User variable that holds the current error-handler. See CATCH and THROW.

**HERE** **---** **a**

Leaves the address of next location available on the dictionary.

**HEX** **---**

Changes the base to hexadecimal, setting BASE to 16.

**HLD** **---** **a**

User variable that holds the address of last character used in a numeric conversion output.

**HOLD** **c** **---**

Used between <# and #> to put a ASCII character during a numeric format.

**HP** **---** **a**

User variable that holds the heap-address of the first free byte on Heap. See HEAP section.

**I** **---** **n**

Used between DO and LOOP (or DO and +LOOP, ?DO and LOOP, ?DO and +LOOP) to put on TOS the current value of the loop index.

**I'** **---** **n**

Used between DO and LOOP (or DO and +LOOP, ?DO and LOOP, ?DO and +LOOP). It puts on TOS the *limit* of the loop. This word isn't available at startup and must be included via NEEDS I'.

**ID.** **nfa** **---**

It prints the definition name whose nfa is on TOS.

**IF** **f** **---** **(immediate)** **(run time)**
**---** **a** **n** **(compile time)**

Used in colon definition in the form

```
IF ...  ENDIF
IF ... ELSE ... ENDIF
```

At run-time IF selects which words sequence to execute based on the flag on TOS:
If f is true, the execution continues with the instruction that follows IF ("true" part).
If f is false, the execution continues after the ELSE ("false" part).
At the end of the two parts, the executions always continues after ENDIF.
ELSE and its "false" part are optional and if omitted no "false part" will be executed and execution continues after ENDIF.
At compile time IF compiles 0BRANCH reserving a cell for an offset to the point after the corresponding ELSE or ENDIF .
The integer n is used for syntax checking.

**IMMEDIATE** **---**

Marks the latest defined word such that at compile-time it is always executed instead of being compiled. The bit 6 of the

length byte of the definition is set. This allows such definitions to handle complex compilation situation instead of burdening the main compiler.
The user can force the compilation of an immediate definition prepending a [COMPILE] to it.

### INCLUDE ---

It is used in in the form:

```
INCLUDE cccc
```

starts interpretation of text read from file cccc.
This word has a known bug, the INCLUDEd source text file must end with an empty line.
See also LOAD

### INDEX n1 n2 ---

Prints the first line of all screens between n1 and n2. Useful to quick check the content of a series of screens.

### INKEY --- b

Reads the next character available from current stream and previously selected with SELECT leving it on TOS. It is the opposite of EMITC.

### INTERPRET ---

This is the text interpreter. It executes or compiles, depending on the value of STATE, text from input buffer a word at a time. It first searches on CONTEXT and CURRENT vocabularies; if they fail, the text is interpreted as a numeric value, converted using the current BASE, and put on TOS. If that numeric conversion fails too, an error is notified with the symbol "?" followed by the word that caused the error. INTERPRET executes NUMBER and the presence of a decimal point "." indicates that the number is assumed as double integer instead of a simple integer.
After execution of the word found, the control is given back to the caller procedure.

### INVERT n1 --- n2

Inverts all bits. This definition is available after NEEDS INVERT.

### INVV ---

"Inverse video". It enables Inverse-Video attribute mode. See also TRUV.
This word isn't available at startup and must be included via NEEDS INVV.

### J --- n

Used inside a DO-LOOP gives the index of the *first* outer loop. See also I.
This word isn't available at startup and must be included via NEEDS J.
E.g.

```
DO .. DO .. J .. LOOP .. LOOP
```

In this case **J** is used to get the index of the outer DO-LOOP while **I** gives the index of the inner DO-LOOP.

### K --- n

Used inside a DO-LOOP gives the index of the *second* outer loop. See also **I**.
This word isn't available at startup and must be included via NEEDS K.
E.g.

```
DO .. DO .. DO .. K .. LOOP .. LOOP .. LOOP
```

Anyway, in Forth, it isn't a good programming technique nesting loop, better split the definition.

**KEY** **---** **b**

Waits for a key-press, without showing a flashing cursor. It leaves the ASCII code b of the character read from keyboard without printing it to video. In this implementation some SYMBOL-SHIFT key combinations are decoded as follow:

| | | | | |
|---|---|---|---|---|
| E2 | STOP | → | 7E | ~ |
| C3 | NOT | → | 7C | \| |
| CD | STEP | → | 5C | \ |
| CC | TO | → | 7B | { |
| CB | THEN | → | 7D | } |
| C6 | AND | → | 5B | [ |
| C5 | OR | → | 5D | ] |
| AC | AT | → | 7F | © |
| C7 | <= | → | 20 | same as SHIFT-1 [EDIT] |
| C9 | <> | → | 06 | same as CAPS-SHIFT + 2 and toggles CAPS-LOCK On and Off |
| C8 | >= | → | 20 | same as SHIFT-0 [BACKSPACE] |

**L/SCR** **---** **n**

Constant that indicates the number of lines per Screen. In this implementation is 16.

**LATEST** **---** **nfa**

Leaves the nfa of the latest word defined in CURRENT vocabulary.

**LEAVE** **---**

Forces the conclusion of a DO ... LOOP by compiling (LEAVE) followed by an offset to the first instruction after the corresponding LOOP or +LOOP.

**LFA** **pfa** **---** **lfa**

Converts a pfa in its lfa. See also CFA, NFA, PFA, >BODY, <NAME.

**LIMIT** **---** **a**

User variable that points to the first location above the last buffer. Normally it is the top of RAM, but not always. In this implementation, it set at $E000 to allow MMU7 as a general purpose 8K RAM bank. See also: FIRST.

**LIST** **n** **---**

Prints screen number n and sets SCR to n.

**LIT** **---** **n**

Puts on TOS the value hold in the following location. It is automatically compiled a before each literal number.

**LITERAL** **n** **---** **n** **(immediate)** **(run time)**
**n** **---** **(compile time)**

Compile-time, LITERAL compiles LIT followed by the value n in the following cell. This is an immediate word and, a colon definition, it will be executed.
It is used in the form

```
: cccc ... [ calculations ] LITERAL ... ;
```

the compilation is suspended during the calculations and, when compilation resumes, LITERAL compiles the value put on TOS during the previous calculations.

**LOAD** **n** **---**

Start interpretation of Screen n. The loading phase ends at the end of the screen or at the first occurrence of EXIT.
If n is negative, instead of loading from Screen# n, it loads text directly from stream n as previously OPEN# from Basic.
See also --->

**LOAD+** **n** **---**

Start interpretation of screen n. The loading phase ends at the end of the screen or at the first occurrence of EXIT.
See also ---> and LOAD.

**LOAD-** **n** **---**

Start interpretation of text read directly from stream n as from Basic's OPEN# n . It uses ACCEPT-.
See also ---> and LOAD.

**LOOP** **a n ---** **(immediate)** **(run time)**
**n ---** **(compile time)**

Used in colon definition in the form

```
DO ... LOOP
?DO ... LOOP
```

At run-time LOOP checks the jump to the corresponding DO. The index is incremented and the total compared with the limit; the jump back happens if the index did not cross the boundary between the loop limit minus one and the loop limit. Otherwise the execution leaves the loop. On loop leaving, the parameters are discarded and the execution continues with the following word.
At compile-time LOOP compiles (LOOP) and the jump is calculated from HERE to a which is the address left by DO on the stack. The value n2 is used internally for syntax checking.

**LP** **---** **a**

User variable for printer purposed. In this Forth implementation it is used during compilation phase by CASE.

**LSHIFT** **n1 u** **---** **n2**

Shifts left an integer n1 by u bit.

**M*** **n1 n2** **---** **d**

Mixed operation. It leaves the product of n1 and n2 ad a double integer.

**M*/** **d1 n1 n2** **---** **d2**

Mixed operation. Compute (d · n1) / n2 using a "triple precision integer" as the intermediate value to avoid precision loss.
This word isn't available at startup and must be included via NEEDS M*/. The source file is M&%.f

**M+ d u --- d2**

Mixed operation. It leaves the sum of d and unsigned u as a double integer d2.
This definition is available after NEEDS M+

**M/ d n1 --- n2**

Mixed operation. It leaves the quotient n2 of the integer division of a double integer d by the divisor n1.

**M/MOD d1 n1 --- n2 n3**

Mixed operation. It leaves the remainder n2 and the quotient n3 of the integer division of a double integer d by the divisor n1. The sign of the reminder is the same as d. This system uses floored division via M/MOD and implements UM/MOD in machine-code, FM/REM and SM/MOD as derived definitions.

**MARK a n ---**

TYPE in inverse video. This word is not available at startup, it has to be loaded via NEEDS MARK.

**MARKER --- (immediate) (run time)**

Used outside a colon defintion in the form

**MARKER cccc**

this creates a new definition cccc that once executed restores the dictionary to the status before cccc was created. This removes cccc and all subsequent definitions. This word allows forgetting across vocabularies since it keeps track of VOC-LINK, CURRENT, CONTEXT values.

**MAX n1 n2 --- n3**

Leaves the maximum between n1 and n2.

**MESSAGE n ---**

Prints to the current device the error message identified by n. If WARNING is zero, a short MSG#n is printed. If WARNING is non zero 1, line n of screen 4 (of drive 0) is displayed. Such a number can be positive or negative and lay beyond block 4. See also ERROR.

**MIN n1 n2 --- n3**

Leaves the minimum between n1 and n2.

**MMU7! n ---**

This word accepts n between 0 and 223 and map the corresponding 8K-page at E000-FFFh addresses. It is coded in Assembler and uses NEXTREG A,n Next's peculiar op-code (ED 92). See MMU7@.

**MMU7@ --- n**

This word returns a number n between 0 and 223 by asking the hardware which 8K-page is currently fitted in MMU7. See MMU7!

**MOD n1 n2 --- n3**

Divides n1 by n2 and leaves the remainder n3. The sign is the same as n1.

**MS u ---**

Waits of u milliseconds.
At 28 MHz, u must be < 8192.
At 14 MHz, u must be < 16384.
At 7 MHz, u must be < 32768.
At 3.5 MHz, u must be < 65536.
This word isn't available at startup and must be included via NEEDS MS.
N.B. Interrupts aren't disabled during execution.

**M_P3DOS n1 n2 n3 n4 a --- n4 n5 n6 n7 f**

This is the NEXTZXOS call wrapper. Parameters passed on stack are used as follow:

n1 = input parameter value for **hl** registers pair
n2 = input parameter value for **de** registers pair
n3 = input parameter value for **bc** registers pair
n4 = **a** register input parameter value
a = service routine address
n5 = **hl** returned value
n6 = **de** returned value
n7 = **bc** returned value
n8 = **a** register
f = 0 for OK, non zero for KO.

This word calls uses RST 08 followed by $94 to call the specified routine.
Value returned on register IX is also stored at HEX 2A +ORIGIN before IX is restored to its fixed value.

Some NEXTZXOS primitives are coded by their own as distinct definitions (e.g F_OPEN, F_OPENDIR, etc), but most of them are not. For example all LAYER definistions use IDE_MODE! which in turn uses M_P3DOS.

**NEEDS ---**

Used in in the form:

```
NEEDS cccc
```

if the definition cccc is not present in dictionary, then it starts interpretation of text read from file **inc/*cccc*.f** and, if this is not found, gives a second chance from file **lib/*cccc*.f**

Some characters are illegal for filename: noticeably the "double-quote" character (") is among them. In such case, these characters are converted into "tilde" (~) and *that* file is then searched for.
For example:

NEEDS S" searches the file S~.f instead of an illegal filename S".f

Here is the complete map:

```
: ? / * | \ < > "
_ ^ % & $ _ { } ~
```

At the moment we are writing, this NEEDS definitions has a flaw: in case of interpretation/compilation error, the

file/handle remains open and you have to close it manually using something like `2 F_CLOSE .`

This definition differs from `INCLUDE` because `NEEDS cccc` refers to a dictionary entry whilst `INCLUDE cccc` refers to a full-path filename with explicit extension.

This definition is defined as follow: Since any given Screen # n occupies BLOCKs n and n+1, `NEEDS` exploits BLOCK number 0 – which normally isn't reachable – and uses it as a temporary buffer for each line read from file, this way a text source line cannot exceed 511 bytes.

**NEGATE n --- -n**
Changes the sing of `n1`

**NFA pfa --- nfa**
Converts a word's pfa into its nfa. See also `CFA`, `LFA`, `PFA`, `>BODY`, `<NAME`.

**NIP n1 n2 --- n2**
Removes the second element from TOS. See also: `OVER`, `DROP`, `TUCK`, `SWAP`, `DUP`, `ROT`.

**NMODE --- a**
User variable that indicates how double numbers are interpreted. During the input, numbers can be read as double integer numbers or floating-point numbers. This variable is modified by the optional words `INTEGER` that sets it to 0 and `FLOATING` that sets it to 1.

**NOOP ---**
This token does nothing. Useful as a placeholder or to prevent crashes in `INTERPRET`.

**NOT ---**
Equivalent to `0=`

**NUMBER a --- d**
**a --- fp (compile time)**
Converts a counted string at address a with a in a double number. If `NMODE` is 0, the string is converted to double integer. Position of the last decimal point encountered is kept in `DPL`.
If `NMODE` is 1, a floating-point number conversion is tried instead of an simple double integer conversion.
If no conversion can be done, and error #0 is raised.

**OCTAL ---**
Changes the base to octal, setting `BASE` to 8. To use this word you have to type `NEEDS OCTAL`.

**OFFSET --- a**
User variable that states the beginning of "blocks area". The content of `OFFSET` is added by `BLOCK` to the number on TOS to determine the right offset to read from file open to "!Blocks.bin". Messages issued by `MESSAGE` are independent from `OFFSET`. This variable is an heritage of previous version I really could dismiss.

**OPEN<** **---** **fh**

Used in the form

```
OPEN< cccc
```

this definition invokes F_OPEN NextZXOS and opens a file `cccc`. It returns file-handle number `fh`. This definition is used by `INCLUDE`. At the moment, this definition cannot be compiled and should be used only in interpretation phase.

**OR** **n1 n2** **---** **n3**

Executes a bitwise OR operation between the two integers. The operation is performed bit by bit.

**OUT** **---** **a**

User variable incremented by `EMIT`. The user can examine and alter `OUT` to control the video formatting.

**OVER** **n1 n2** **---** **n1 n2 n1**

Copies the second number from TOS and put it on the top. See also `DROP`, `NIP`, `TUCK`, `SWAP`, `DUP`, `ROT`.

**P!** **b u** **---**

Sends to port `u` a byte `b`. Note: `u` is a 16 bit port address and an OUT (C) op-code is internally executed.

**P@** **u** **---** **b**

Accepts the byte `b` from port `u`. Note: `u` is a 16 bit port address and an IN(C) op-code is internally executed.

**PAD** **---**

Leaves on TOS the address of text output buffer. It is at a fixed distance of 68 byte over `HERE`.

**PFA** **nfa** **---** **pfa**

Converts a word's `nfa` to its `pfa`. See also `CFA`, `LFA`, `NFA`, `>BODY`, `<NAME`.

**PICK** **n** **---** **pfa**

Picks the **n-th** element from TOS. This means:

`0 PICK` is the same as `DUP`
`1 PICK` is the same as `OVER`

**PLACE** **---** **a**

User variable that holds the number of places after the decimal point to be shown during a numeric output conversion. See also `PLACES`.

**PREV** **---** **a**

User variable that points to the last referred buffer. `UPDATE` marks that buffer so that it is later written to disk.

**QUERY** **---**

Awaits from terminal up to 80 characters or until a `CR` is received. The text is stored in `TIB`. User variable `IN` is set to zero.

**QUIT** **---**

Clears the Return-Stack, stops any compilations and return the control to the operator terminal. No message is issued.

**R@** **---** **n**

Copies to TOS the value on top of Return Stack without alter it.

**R#** **---** **a**

User variable that holds the position of the editing cursor or other function relative to files.

**R/W** **a** **n** **f** **---**

Standard FIG-FORTH read-write facility. Address `a` specifies the buffer used as source or destination; `n` is the sequential number of the block; f is a flag, 0 to Write, 1 to Read. `R/W` determines the location on mass storage, performs the transfer and error checking.

**R0** **---** **a**

User variable that holds the initial value of the Return Stack Pointer. See also `RP!` and `RP@`.

**R>** **---** **n**

Removes the top value from Return Stack and put it on TOS. See also `>R`, `R@` and `RP!`.

**R>DROP** **---**

Removes the top value from Return Stack. See also `>R`, `R@` and `DROP`.

**RECURSE** **---**

Used only at compile-time inside a colon-definitions, it compiles the definition being created to put in place a recursion call. This word is available after a `NEEDS RECURSE`.

**REG!** **b** **n** **---**

Writes value `b` to Next REGister `n`.

**REG@** **n** **---** **b**

Reads Next REGister `n` giving byte `b`.

**REMOUNT** **---**

This definition is available only after `NEEDS REMOUNT`.
Enter the unmount/mount routine. Interactively the user is asked for a Y key-stroke, and the system waits for that key-stroke allowing the manipulation of the SD.

**RENAME** **---**

Used in the form:

**RENAME cccc xxxx**

it searches the word `cccc` in the `CONTEXT` vocabulary and changes its name to `xxxx`. The two word names `cccc` and `xxxx` **must have the same length**. This definition is available after `NEEDS RENAME`.

**REPEAT** **a1 n1 a2 n2** **---** **(immediate)** **(compile time)**
**---** **(run time)**

Used in colon defintion in the form:

**BEGIN ... WHILE ... REPEAT**

At run-time `REPEAT` does an inconditional jumt to the corresponding BEGIN.
At compile-time `REPEAT` compiles `BRANCH` and the offset from `HERE` to `a1` and resolves the offset from `a1` to the location after the loop; `n1` and `n1` are used for sysntax check.

**ROT** **n1 n2 n3** **---** **n2 n3 n1**

Rotates the three top elements, taking the third and putting it on top. The other two elements are pushed down from top by one place. See also `OVER`, `DROP`, `NIP`, `TUCK`, `SWAP`, `DUP`.

**ROLL** **n1 ... k** **---** **n2 ... n1**

Rotates the `k` top elements, taking the `k`-th and putting it on top. The other k – 1 elements are pushed down from top by one place. The index `k` is zero based, so that `0 ROLL` does nothing, `1 ROLL` is `SWAP` and `2 ROLL` is `ROT`.
See also `ROT`. This definitions isn't available at startup, it needs to be imported via `NEEDS ROLL`.

**RP!** **a** **---**

System procedure to initialize the Return Stack Pointer to the value passed on TOS that should be the address held in `R0` user variable.

**RP@** **---** **a**

Leaves the current value of Return Stack Pointer.

**RSHIFT** **n1 u** **---** **n2**

Shifts right an integer `n1` by `u` bit.

**S0** **---** **a**

User variable that holds the initial value of che SP register. See also: `SP!` and `SP@`.

**S>D** **n** **---** **d**

Converts a 16 bit integer into a 32 bit double integer, sign is preserved. An obsolete version `S->D` is still available via `NEEDS`.

**SCR** **---** **a**

User variable that hold the number of the last screen retrieved with `LIST`.

**SELECT** **n** **---**

Selects the current channel. As usual for ZX Spectrum, `n` is 0 and 1 for lower part of screen, 2 for the upper part, 3 for printer, 4 for "!Blocks.bin" stream. Note: `KEY` always select chanle 2 to display the (flashing) cursor.

**SIGN** **n d** **---** **d**

If `n` is negative, it puts an ASCII "-" at the beginning of the numeric string converted in the text buffer. Then, `n` is discarded while d is kept unchanged. Used between `<#` and `#>`.

**SM/REM** **d n1** **---** **n2 n3**

Symmetric Division. Leaves the quotient `n3` and the reminder `n2` of the integer division of `d` / `n1`.
This system has only UM/MOD coded in machine-code.

| Dividend | Divisor | Remainder | Quotient |
|---|---|---|---|
| 10 | 7 | 3 | 1 |
| -10 | 7 | -3 | -1 |
| 10 | -7 | 3 | -1 |
| -10 | -7 | -3 | 1 |

**SMUDGE** **---**

Used by the creation word **:** during the definition of a new word; it toggles the smudge-bit of the first byte in the nfa of the `LATEST` defined word. When a word's smudge-bit is set, it prevents the compiler to find it. This is typical for uncomplete or not correctly defined words.
It is also used to remove malformed incomplete words via

**SMUDGE FORGET cccc**

**SOURCE-ID** **---** **a**

User variable that keeps the file-handle used during `INCLUDE` or `NEEDS`.

**SP!** **a** **---**

System procedure to initialize the SP register to the address a that should be the address hold in `S0` user variable.

**SP@** **---** **a**

Returns the content of SP register before `SP@` was executed.

**SPACE** **---**

Emits a space to the current output peripheal, usually the video. See also `SELECT`.

**SPACES** **n** **---**

Emits `n` spaces.

**SPAN** **---** **a**

User variable that holds the number of characters got from the last EXPECT.

**SPLASH** **---**

Shows splash screen build date-number.

**SPLIT** **n1** **---** **n2 n3**

Split the two bytes of n1 into two separate numbers, n2 low byte, n3 high byte. Available after NEEDS SPLIT.

**STATE** **---** **a**

User variable that holds the compilator status. A non-zero value indicates a compilation in progress.

**SWAP** **n1 n2** **---** **n2 n1**

Swaps the two top element at the TOS. See also OVER, DROP, NIP, TUCK, DUP, ROT.

**THEN** **a n** **---** **(immediate)**
**---** **(compile time)**

Synonym of ENDIF.

**TIB** **---** **a**

User variable that holds the address of the Terminal Input Buffer.

**TO** **n** **---**

Used in the form:

**TO cccc**

It assigns the value n to the variable cccc previously defined via VALUE. This definition available after NEEDS TO.

**TOGGLE** **a b ---**

The byte at location address a is XOR-ed with the model b.

**TRAVERSE** **a1 n** **---** **a2**

Spans through the name-field of a definition depending on the value of n.
If n = 1, then a1 must be the beginning of the name-field, i.e. nfa itself; a2 is the address of the last byte of the name field.
If n = –1, then a1 must be the last byte of name-field and a2 will be the nfa.
Used by da NFA and PFA.

**TRUV** **---**

"True video". It disables Inverse-Video attribute mode. See also INVV.
This word isn't available at startup and must be included via NEEDS INVV.

**TUCK** **n1 n2 --- n2 n1 n2**

Takes the top element of calculator stack and copies after the second. See also OVER, DROP, NIP, SWAP, DUP, ROT.

**TYPE** **a n ---**

Sends to the current output peripheal n characters starting from address a.

**U.** **u ---**

Emits an unsigned integer followed by a space.

**U<** **u1 u2 --- f**

Leaves a tf if u1 is less than u2, a ff otherwise.

**UM*** **u1 u2 --- ud**

Unsigned product of the two integers u1 and u2. The result is a double integer.

**UM/MOD** **ud u1 --- u2 u3**

Leaves the quotient u3 and the reminder u2 of the integer division of ud / u1.

**UNTIL** **a n --- (immediate) (compile time)**
**f --- (run time)**

Used in colon definition in the forms

```
BEGIN ... UNTIL
```

At run-time UNTIL controls a conditional jump to the corresponding BEGIN when f is false; the exit from the loop happens if f is true.
At compile-time UNTIL compiles 0BRANCH and an offset from HERE to a; n is used for syntax checking.

**UPDATE** **---**

Marks as modified the most recent used buffer, the one pointed by PREV. The block contained in the buffer will be transferred to disk when that buffer is requested for another block.

**UPPER** **c1 --- c2**

This word converts a character to upper-case. If c1 is not between “a” and “z”, then c1 is left unchanged.

**USE** **--- a**

User variable that holds the buffer address of the block to be read from disk or that has just been written to.

**USER** **n ---**

Defining word used in the form

```
n USER cccc
```

creates an user variable 'cccc'. The first byte of pfa of cccc is a fixed offset for the User Pointer, that is the pointer for the user area. In this implementation there is only one User Area and a fixed User Pointer.
When cccc is later executed, it put on TOS the sum of offset and User Pointer, sum to be used ad the address for that specific user variable. The user variable are: TIB, WIDTH, WARNING, FENCE, DP, VOC-LINK, FIRST, LIMIT, EXP, NMODE, BLK, >IN, OUT, SCR, OFFSET, CONTEXT, CURRENT, STATE, BASE, DPL, FLD, CSP, R#, HLD, USE, PREV, LP, PLACE, SOURCE-ID, SPAN, HANDLER, HP.

## VALUE n ---

Defining word used in the form:

```
n VALUE cccc
```

Creates the word cccc that acts as a variable. To store a value in such a variable you have to use TO.
When cccc is later executed it directly returns the value of the variable without the need to access its address using @.
This definition is available after NEEDS VALUE.

## VARIABLE ---

Defining word used in the form:

```
VARIABLE cccc
```

creates the word cccc with the pfa containing the initial value 0. When cccc is executed, it puts on TOS the pfa of cccc that is the address that holds the value.
When used in the form

```
cccc @
```

the content of the variable cccc is left on TOS.
When used in the form

```
n cccc !
```

the value on TOS is stored to the variable cccc.

## VIDEO ---

Sets DEVICE 2 to select the video as current output peripheral. See SELECT and DEVICE.

## VOC-LINK --- a

User variable that holds the address of a field in the definition of the last vocabulary. Each vocabulary is part of a linked-list that uses that field, in each vocabulary definition, as pointer-chain.

## VOCABULARY ---

Defining word used in the form

```
VOCABULARY cccc
```

creates the word cccc that gives the name of a new vocabulary.
Later execution of

```
cccc
```

makes such vocabulary the CONTEXT vocabulary, so that it is possible to search for words defined in this vocabulary first and execute them.
Used in the form

```
cccc DEFINITIONS
```

makes such vocabulary the CURRENT vocabulary, so that it is possible to insert new definitions in it.

**WARM** **---**

Executes a warm system restart. It closes and reopen Block/Screen file then does ABORT.
It does not EMPTY-BUFFERS.

**WARNING** **--- a**

User variable that determines the way an error message is reported. If zero, only a short "MSG#n" is reported. If non zero, a long message is reported. See also ERROR.

**WHILE** **f --- (immediate) (run time)**
**a n --- a1 n1 a2 n2 (compile time)**

Used in colon defintion in the form:

```
BEGIN ... WHILE ... REPEAT
```

At run-time WHILE does a conditional execution based on f. If f is true, the execution continues to a REPEAT which will jump to the corresponding BEGIN. If f is false, the execution continues after the REPEAT quitting the loop.
At compile-time WHILE compiles 0BRANCH leaving a2 for the offset; a2 will be comsumed by a REPEAT. The address a1 and the number n1 was left by a BEGIN.

**WIDTH** **--- a**

User variable that indicates the maximum number of significant characters of the words during compilation of a definition. It must be between 1 and 31.

**WITHIN** **n1 n2 n3 --- f**

Return a true-flag if n2 <= n1 <= n3, a false-flag otherwise. This definition is available only after NEEDS WITHIN.

**WORD** **c --- a**

Reads one or more characters from the current input stream up to a delimiter c and stores such string at HERE that is left on TOS. WORD leaves at HERE the length of the string as the first byte and ends everything with at least two spaces. Further occurrences of c will be ignored.
If BLK is zero, the text is taken from the terminal input buffer TIB. Otherwise the text is taken from the disk block held in BLK. User variable >IN is added with the number of character read, the number ENCLOSE returns.

**WORDS** **---**

Shows a list of words of CONTEXT vocabulary. [Break] stops.

**XOR** **n1 n2 --- n3**

Executes a bitwise XOR operation between the two integers. The operation is performed bit by bit.

**[** **---** **(immediate)**

Used in colon defintion in the form:

```
: cccc [ ... ] ... ;
```

it suspends compilation. The words that follows [ will be executed instead of being compiled. This allows to perform some calculations or start other compilers before resuming the original compilation with ]. See also LITERAL.

**[CHAR]** **---** **(immediate)** **(compile time)**

It is the same as the sequence `[ CHAR c ] LITERAL`.
It is used in colon defintion in the form:

```
: cccc ... [CHAR] c  ... ;
```

At compile time, `[CHAR]` compiles `LIT` and the numeric value of ASCII character `c` in the following cell.

**[COMPILE]** **---** **(immediate)**

Used in colon defintion in the form:

```
: cccc ... [COMPILE] wwww ... ;
```

`[COMPILE]` forces the compilation of a definition `wwww` that is immediate. Normally immediate words aren't compiled but executed and to compile an immediate word it is not possible to use the sequence `COMPILE wwww` but it is necessary using the sequence `[COMPILE] wwww`.
For example, to create an alias `ENDIF` for `THEN` you can type:

```
: ENDIF [COMPILE] THEN ;
```

**** **---**

Used in the from:

```
\ ...
```

Any character that follows `\` until the end of line are treated as a comment.

**]** **---**

Resumes the compilation suspended by `[` so it is possible to continue the definition.

## 6.2 Case -Of structure

The following definitions are available after you type NEEDS CASE

**CASE** **n0** **---** **(immediate)** **(run time)**
**---** **a n** **(compile time)**

Used in colon definition in the form

```
n0 CASE
    n1  OF  ... ENDOF
       ...
    nz  OF  ... ENDOF
    ... ( else )
ENDCASE
```

The CASE definition marks the beginning of Case-Of structure i.e. a set of branches where only one is performed based on the value of n0. If none of the "OF clause" values matches, the ELSE part is performed, if any.
At compile time CASE leaves the previous CSP address a and a number n for syntax checking.
CASE has to be balanced by a corresponding ENDCASE.

**OF** **n0 nk** **---** **(immediate)** **(run time)**
**n1** **---** **a n2** **(compile time)**

This definition is used in colon-definition as parto of a Case-Of structure.
At run-time it compares the matching value nk with the matching value n0 that was on TOS before the beginning of the Case-Of structure.
At compile-time, it compiles (OF) that does a 0BRANCH . The numbers n1 and n2 are used for syntax checking and an address a is left and used by ENDCASE to resolve the branch.
See also CASE.

**ENDOF** **---** **(immediate)** **(run time)**
**a1 n1** **---** **a n2** **(compile time)**

This definition ends an "Of-EndOf" clause started with OF.
At compile-time it acts like a THEN, first compiling a BRANCH to be later resolved by ENDCASE to skip any subsequent "Of-End-Of" clauses and resolving here the 0BRANCH compiled by the corresponding previous OF to continue the Case-Of structure.
See also CASE.

**ENDCASE** **---** **(immediate)** **(run time)**
**a a1 … az** **---** **(compile time)**

This definition ends a Case-Of structure started with CASE.
At compile-time it compiles a DROP to discard the value n0 put on TOS before CASE and resolves all OF-ENDOF clauses to jump after the ENDCASE. Finally, it restores previous content of CSP.
See also CASE.

**(OF)** **n0 nk** **---** **(run time)**

This definition is the run-time semantic compiled by OF definition. At run-time, it compares the value now on TOS nk with the value n0 that was on TOS just before the beginning of the Case-Of structure and leave a flag to be used by the following 0BRANCH (that was compiled by OF ). When n0 equals nk, the definitions between OF and ENDOF will be executed, otherwise a jump to the word after ENDOF is performed.

## 6.3 Heap Memory Facility

The definitions that handle the Heap are available after loading via `NEEDS HEAP`.

Among ZX Spectrum Next new features is the huge amount of RAM. Strings are dictionary expensive, so it would be useful storing them in heap as constant-strings and fetch them at need. The question is how to leverage all that memory in Forth. More, 8K of room is a good place to store an array of strings, or even numeric array and implement some matrices algebra.

Considering how Forth's system areas are sorted out comparing previous and current versions, the first challenge is to move them down to free the top 8K CPU's addressable memory between 0E000h and 0FFFFh allowing MMU7 to map to any physical 8K RAM page.

There are some peculiar addresses that identify the following Forth system areas:

0F840h : Calculator Stack (SP) grows downward, Text Input Buffer (TIB) upward.,
0F8E0h : Return Stack (RP) grows downward, User Variables Area upward.
0F94Ch : the FIRST disk buffer starts here and buffers area ends just before LIMIT 0FF58h.

I coded this "move" in a few words (available in Screens #220-#223) summarized in the definition DOWN that moves these pointers "down" as follow:

0FF58h → 0E000h : LIMIT
0F9C4h → 0D9F4h : FIRST
0F8E0h → 0D9A0h : Return Stack and User Variables Area
0F840h → 0D900h : Stack Pointer and TIB

### 6.3.1 Heap Pointer encoding and decoding

A big issue arises when we need a way to encode both **page number** and **address offset** in a usual Z80 16-bits pointer variable.

Two definitions are made available to perform these coding and decoding operations: `>FAR` and `<FAR`.

Given a page number `n` and an address `a` (to be intended as an offset of addresses between E000h and FFFFh) the definition `>FAR` encodes the page number in the most significant bits of `ha` and an offset in the remaining less significant bits.

The inverse function is performed by `<FAR`. Splitting a 16-bit "heap-pointer-number" into the page part and the offset part again.

In the following paragraphs a couple of possible implementations are described in detail.

### 6.3.2 Heap Pointer description for 64 kiBytes space

The following solution allows 64K of physical RAM Heap: Since an 8K offset requires 13 bits, the remaining 3 bits can be used to encode, say, from page 32 ($20) to page 39 ($27). For example:

```
                    16 bit pointer
                 | pppb bbbb | bbbb bbbb |
                 |           |           |
  |    Page     |         Offset          |
  |  0010 0ppp  | 111b bbbb | bbbb bbbb |
```

The encoding/decoding definitions would be something like the following:

```
CODE   >FAR ( ha --- a n )
       pop     de
       ld      a,d
       and     $E0
       rlca
       rlca
       rlca
       add     $20     ; this is peculiar to this example
       ld      l,a
       ld      h,0     ; hl = page number between 32 and 39
       ld      a,d
       or      $E0
       ld      d,a     ; de = offset at $E000
       push    de
       push    hl
       jp      (ix)
```

```
CODE   <FAR ( a n --- ha )
       pop     hl      ; hl = page number between 32 and 39
       pop     de      ; de = offset at $E000
       ld      a,l
       and     $07
       rrca
       rrca
       rrca
       ld      h,a
       ld      a,d
       and     $1F
       or      h
       ld      d,a
       push    de      ; de = heap-pointer
       jp      (ix)
```

### 6.3.3 Heap structure

The Heap can be seen as a "linked-list" starting at 8K page $20 offset $0002. The User variable `HP` keeps the "heap-pointer" to the next available location on Heap. So, at startup, `HP` is $0002 that correspond to page $20 offset $0002.

A Heap memory allocation reserves the requested number of bytes and advances HP to point to the next available location on Heap. The previous value of `HP` is also stored at the location that was available *before* the memory allocation was requested to put in place a "linked-list".

In other words:
1. HP is advanced of one cell (2 bytes) to make room for the linked-list pointer.
2. Current HP value returned by the memory allocation (memory is not initialized and its content is undefined)
3. HP is advanced to the number of bytes requested.

Here is a real case example:

At startup, `HP` is $**0002** and the Heap memory looks as follows (Location is expressed in the form "$page:$offset")

```
Location              Content

 $20:$0000    $0000 ( this zero marks the "lower-end" of Heap )
 $20:$0002    first free memory byte pointed by HP
```

If we need 5 bytes of Heap we can type `5 HEAP` that will return $**0004** as the "Pointer" the Heap area just allocated, while `HP` User variable will be advanced to $0009. After the execution the memory will look like this:

```
Location              Content

 $20:$0000    $0000  ( this marks the "end" of Heap )
 $20:$0002    $0009  ( final value of HP )              ┐
 $20:$0004    00 00 00 00 00 ( 5 bytes just allocated ) │
 $20:$0009    free memory pointed by HP                 ◄┘
```

Then, we want to reserve another 8 bytes chunk and we type `8 HEAP` that will return $000B as "Pointer" to that new area of memory and `HP` will be advanced to $0013. After the execution the memory will look like this:

```
Location              Content

 $20:$0000    $0000  ( this marks the "end" of Heap )
 $20:$0002    $0009  ( pointer to next chunk            ┐
 $20:$0004    00 00 00 00 00 ( 5 bytes just allocated ) │
 $20:$0009    $0013                                     ◄┘      ┐
 $20:$000B    00 00 00 00 00 00 00 00 ( 8 bytes just allocated ) │
 $20:$0013    free memory pointed by HP                          ◄┘
```

Now, you should be able to see the Linked-List starting at $0002 that points to $0009 that points to $0013.
You can follow all these Pointers using the following procedure:

```
2    .S \  Stack is 0002 as Heap-Pointer, that is $20:$0002, the beginning of Heap Memory.
FAR .S \  Stack is E002 as real Address (and page $20 is fitted in MMU7)
@    .S \  Stack is 0009 as Heap-Pointer
FAR .S \  Stack is E009 as real Address (and page $20 is fitted in MMU7)
@    .S \  Stack is 0013 as real Address
```

Some low-level definitions are available to allow store and retrieve "to and from" Heap and how to avoid that a string isn't "paged away" in the middle of processing i.e. how to guarantee a page to stay in place across Standard-ROM calls or I/O disk operations that use page-bank C000-FFFF for their purposes:

`MMU7!` is used to fit a given 8K page number at E000h (i.e. MMU7).
`>FAR` is used to decode a "16 bit pointer" splitting it into "page & offset" as shown above.

The User Variable `HP` has been introduced to keep track of room in Heap: it'ss "the pointer" to the next available space on Heap.

The following definitions are available after loading via `80 LOAD` or via `NEEDS HEAP` (or something).

### +"  ha --- ha

Assuming `ha` is a Heap-Address Pointer to a "counted string" and this is the last chunk of memory of Heap, this definition accepts some text from the current input-source, parse it looking for a `"` that is the common "string terminator", and appends to the previous string on Heap. It returns the same Heap-Address Pointer to a "counted string" but the "count-byte" is incremented correctly.

### +C  ha c --- ha

Consume a character `c` from the current input source and append the string being created in Heap at `ha`. The heap pointer `ha` is returned unchanged.

### >FAR  ha --- a p

Given a heap-encoded pointer `ha` this definition decodes the top bits as one of the 8K-page available page `p` and the lower bits as the offset from E000h `a`. It does not modify what MMU7 page is.
This definition is available after `NEEDS >FAR` (that loads the file "`./inc/}far.f`" source file).
You are allowed to modify (with care!) the source file to obtain the desired range of pages.
See `<FAR`, `MMU7!`

### <FAR  a p --- ha

Given an offset-address `a` (to be intended as a physical address between E000h and FFFFh) and a page number `p` for an 8K-page this definition encodes the page number in the most significant bits of `ha` and an offset in the remaining bits. It does not modify MMU7 page.
This definition is available after `NEEDS <FAR` (that references the file "`./inc/{far.f`").
You are allowed to modify (with care!) the source file to obtain the desired range of pages.
See `>FAR`, `MMU7!`

### FAR  ha --- a

This definition converts a heap-pointer `ha` into an offset `a` (at E000h) and perform the correct 8K paging on MMU7.
It simply calls `>FAR` and `MMU7!`

**H"** **---** **ha**

Accepts a text from the current input-source and stores it to Heap. It returns a “heap-address-pointer” to a counted string.

**HEAP** **n** **---** **ha**

This definition reserves `n` bytes on Heap and returns the “heap-address-pointer”. This `ha` can be turned into a constant name using `POINTER`.

**POINTER** **ha** **---** **a**

It works like `CONSTANT` but it returns a "FAR-resolved" offset-pointer from E000h .
A possible use is : `S" ccc" POINTER P1`

**SKIP-PAGE** **n** **---**

Check if `n` bytes are available at the top of Heap on current 8K-page, otherwise advance HP to skip to the beginning of next 8K-page. It raises an “Heap Full” error if there is no more room in Heap.

**S"** **---** **a n**

Accept text from the current input-source and store it to Heap as a counted string.
At compile time it compiles `(S")` followed by an Heap-pointer just after it, which at run-time returns a real-address (at MMU7) and a counter representing the “counted-string” that can be used
If `STATE` is 0, i.e. we aren’t compiling, the c

**(S")** **---** **a n**

This is the run-time counterpart of `S"` that uses the Heap-Pointer in the following cell to fit the right 8K-Page in MMU7 using `FAR` definition and leave the real-address `a` and the length of the string `n`.

**HEAP-INIT** **---**

Ask NEXTZXOS to use pages $20-$27 for Heap. From this point Heap command can be used safely.

**HEAP-DONE** **---**

Release to NEXTZXOS pages $20-$27. Heap commands should not be used after that.

### 6.3.4 Heap Pointer description for 128 kiBytes space

As a mental exercise, to allow **twice** the number of pages we need one more bit for the page number at the expense of the remaining offset bits; for example, 4 bits for page number allows encoding from page 32 ($20) to page 47 ($2F) , that is 128K, and leaves the remaining 12 bits for offset to addresses. We notice that **only even** addresses can be directly referenced.

```
                    16 bit pointer
                | pppp bbbb | bbbb bbbb |
                |           |           |
   |    Page    |        Offset         |
   |  0010 pppp | 111b bbbb | bbbb bbb0 |
```

The encoding routine could be

```
CODE    >FAR ( ha --- a n )
        pop       hl
        ld        a,h
        and       $F0
        rlca
        rlca
        rlca
        rlca
        add       $20       ; i.e. 32 in decimal base
        ld        e,a
        ld        d,0       ; de = page number between 32 and 47
        add       hl,hl     ; shift
        ld        a,h
        or        $E0
        ld        h,a       ; hl = offset at $E000
        push      hl
        push      de
        jp        (ix)
```

### 6.3.5 Heap Pointer description for 256 kiBytes space

With 5 bits for page number and 11 bits for offset, we'll have 32 pages between 32 ($20) and 63 ($3F), that is 256K and only addresses divisible by 4 can be directly referenced.

```
                    16 bit pointer
                | pppp pbbb | bbbb bbbb |
                |           |           |
   |    Page    |        Offset         |
   |  001p pppp | 111b bbbb | bbbb bb00 |
```

Encoding routine

```
CODE    >FAR ( ha --- a n )
        pop       hl
        ld        a,h
        and       $F8
        rrca
        rrca
        rrca
        add       $20
        ld        e,a
        ld        d,0       ; de = page number between 32 and 63
        add       hl,hl     ; shift
        add       hl,hl     ; shift
        ld        a,h
        or        $E0
        ld        h,a       ; hl = offset at $E000
        push      hl
        push      de
        jp        (ix)
```

### 6.3.6 Heap Pointer description for 512 kiBytes space

With 6 bits for page number and 10 bits for offset, we'll have 64 pages between, say, 32 ($20) and 96 ($5F), that is 512 and only addresses divisible by 8 can be directly referenced.

```
                    16 bit pointer
                | pppp ppbb | bbbb bbbb |
                |           |           |
   |    Page    |        Offset         |
   | 01pp pppp | | 111b bbbb | bbbb b000 |
```

Encoding routine

```
CODE    >FAR ( ha --- a n )
        pop       hl
        ld        a,h
        and       $FC
        rrca
        rrca
        add       $20
        ld        e,a
        ld        d,0       ; de = page number between 64 and 127
        add       hl,hl     ; shift
        add       hl,hl     ; shift
        add       hl,hl     ; shift
        ld        a,h
        or        $E0
        ld        h,a       ; hl = offset at $E000
        push      hl
        push      de
        jp        (ix)
```

### 6.3.7 Heap Pointer description for 1024 kiBytes space

Pursuing this path to the limit we can use 7 bits for page number we can pick 128 distinct pages, for example from page 64 ($40) to page 191 ($BF) that leads to **1024K** of physical RAM, at the downside to be able to reference only physical addresses divisible by 16.

```
                    16 bit pointer
                 | pppp pppb | bbbb bbbb |
                 |           |           |
   |    Page     |        Offset         |
   | xppp pppp | | 111b bbbb | bbbb 0000 |
```

Coding/decoding routines are

```
CODE    >FAR ( ha --- a n )
        pop       hl
        ld        a,h
        srl       a
        add       $40       ;
        ld        e,a
        ld        d,0       ; de = page number between 64 and 191
        add       hl,hl
        add       hl,hl
        add       hl,hl
        add       hl,hl     ; shift hl 4 bits left
        ld        a,h
        or        $E0
        ld        h,a       ; de = offset at $E000
        push      hl
        push      de
        jp        (ix)
```

```
CODE    <FAR ( a n --- ha )
        pop       de        ; de = page number between 64 and 191
        pop       hl        ; hl = offset at $E000
        ld        a,e
        sub       $40
        add       hl,hl
        add       hl,hl
        add       hl,hl
        add       hl,hl     ; shift h 4 bits right
        rla                 ; A receives HL msb
        ld        l,h
        ld        h,a
        push      hl
        jp        (ix)
```

## 6.4 Testing Suite

This is an adaptation of the ANS test harness based on the work originally developed by John Hayes, see https://forth-standard.org/standard/testsuite for details.

The suite is loaded using `NEEDS TESTING` and “Core test-set” can be execute by typing

```
INCLUDE  ./test/core-tests.f
```

In general, a test is given in the form

```
T{  ...  ->  ...  }T
```

for example:

```
T{ 1 DUP -> 1 1 }T
```

**TESTING** **---**

This word is much like a comment, it displays the whole source line where it is.

**T{** **---**

Begin a test phrase that ends with `}T`. It records pre-test stack depth to be compared later.

**->** **---**

Record depth and contents of stack to be copared after `}T`.

**}T** **---**

End a test phrase begun with `T{`. It compares two stack images. Any discrepancies is shown by repeating the current test SOURCE line involved followed by one of the error

## 6.5 Other Utilities

**SHOW-PROGRESS n ---**

Useful within long-lasting definitions to display a “rolling-bar” that show that your ZX Spectrum hasn’t hanged or crashed. This word isn’t available at startup and must be included via `NEEDS SHOW-PROGRESS`.

**?VOCAB ---**

Useful to see which VOCABULARY is CURRENT, CONTEXT and the linked-list described by VOC-LINK.

## 7 The Memory Map

| Address | Name | Description |
|---|---|---|
| 0000-3FFF | | ROM of Spectrum |
| 4000-47FF | | Display file (top) |
| 4800-4FFF | | Display file (middle) |
| 5000-57FF | | Display file (bottom) |
| 5800-5AFF | | Attribute file. |
| 5B00-5BFF | | System variables 128K RAM (former Printer buffer) |
| 5C00-5CEF | | System variables |
| | *CHANS | Stream map |
| | *PROG | Basic program |
| | *VARS | Basic variables |
| | *E_LINE | Line in editing |
| | *WORKSP | Workspace |
| | *STKBOT | Floating point Stack Bottom |
| | *STKEND | Floating point end |
| | *SP | Z80 Stack Pointer register in Basic |
| 61FF | *RAMTOP | Logical RAM top (RAMTOP var is 23730) |
| 6200-6300 | | IM2 ISR vector table |
| 6301-6330 | | Return Stack during ISR (20 entries) |
| 6331-6362 | | Stack area during OS operations |
| 6363 | | ISR entry point (JP address) |
| 6366 | ORIGIN | Forth Origin |
| | | FENCE @ |
| | LATEST | CURRENT @ @ |
| | HERE | DP @ |
| | PAD | HERE 68 + (44h) |
| | ... | Dictionary grows upward |
| | ... | Free memory |
| | SP@ | Calculator Stack grows downward |
| D0E8 | | S0 @ |
| D0E8 | TIB | TIB @ |
| | RP@ | Return Stack grows downward: it can hold 80 entries |
| D188 | | R0 @ |
| D188-D1D8 | | User variables area (about 50 entries) |
| D1E4 | FIRST | First buffer: There are 7 buffers (516 * 7 = 3612 bytes) |
| E000 | LIMIT | First byte outside Forth. |
| E000-FFFF | MMU7 | 8K Page that can page any of the 224 banks of RAM |
| FFFF | P_RAMT | Phisical ram-top |

# Contents

1 Foreword....2
Disclaimer....2
Acknowledgment....2
1.1 Document structure....3
1.2 Legend....4
2 Getting started....5
2.1 Installation....5
2.2 Activation....6
2.3 Case-insensitive and Case-sensitive option....7
2.4 Block / Screen system and Text editor....7
2.5 Character size....7
2.6 Source feeding....8
2.7 Definitions grouped by category....9
2.7.1 Comments....9
2.7.2 Stack manipulation....9
2.7.3 Comparison....9
2.7.4 Output....10
2.8 Integer Arithmetics....10
2.9 Memory....11
2.9.1 Flow control....11
2.9.2 Definition related....11
2.9.3 I/O and Hardware....12
2.9.4 BLOCK / Screen related....12
2.9.5 Numbers & strings....12
2.9.6 Dictionary related....13
2.9.7 Editor....13
2.9.8 NextZXOS....13
2.9.9 Unsorted....13
2.10 Known bugs and improvement needed....14
3 Utilities....15

3.1 The Full Screen Editor Utility – Screen oriented....15
EDIT --- ....15
3.2 Graphics mode and Layer facility....17
LAYER! n ---....17
LAYER0 ---....17
LAYER10 ---....17
LAYER11 ---....18
LAYER12 ---....18
LAYER13 ---....18
LAYER2 ---....18
ATTRIB ---....18
CIRCLE x y r ---....18
DRAW-LINE x0 y0 x1 y1 ---....18
PLOT x y ---....18
PAINT x y ---....18
3.2.1 Colors & Attributes....19
.BORDER b ---....19
.BRIGHT b ---....19
.FLASH b ---....19
.INK b ---....19
.INVERSE b ---....19
.OVER b ---....19
.PAPER b ---....19
3.3 LED – the Large file EDitor....20
CAT" --- ....20
LED --- cccc ....20
LED-EDIT --- ....21
LED-SAVE --- ....21
LED-FILE --- cccc ....21
3.4 Interrupt Service Routine....22
INTERRUPTS --- ....22
ISR-OFF --- ....22

ISR-XT xt --- 22
ISR-ON --- 22
ISR-EI --- 23
ISR-DI --- 23
ISR-IM1 --- 23
INT-IM2 --- 23
ISR-SYNC --- 23
SETIREG b --- 23
ISR-RET --- 23
ISR-SUB --- 23
3.5 Block Search and Locate Utility 24
LOCATE --- 24
GREP --- 25
BSEARCH n1 n2 --- 25
COMPARE a1 b1 a2 b2 --- 25
3.6 Debugger Utility 26
SEE --- 26
3.6.1 The Inner-interpreter 28
DUMP a u --- 29
.WORD a --- 29
.S --- 29
DEPTH --- n 29
3.7 Floating-Point Option 30
3.7.1 Floating-point option activation and number conversion 30
INTEGER --- 30
FLOATING --- 31
D>F d --- fp 31
F>D fp --- d 31
FLOAT n --- fp 31
FIX fp --- n 31
3.7.2 Representation and constants 31
F>PAD fp --- u 31

F.R fp u --- ....31
F. fp --- ....31
1/2 --- fp....31
PI --- fp....31
3.7.3 Arithmetics....31
F- fp1 fp2 --- fp3....31
F+ fp1 fp2 --- fp3....31
F* fp1 fp2 --- fp3....32
F/ fp1 fp2 --- fp3....32
FNEGATE fp1 --- fp2....32
FSGN fp1 --- fp2....32
FABS fp1 --- fp2....32
F/MOD fp1 fp2 --- fp3 fp4....32
F** fp1 fp2 --- fp3....32
FMOD fp1 fp2 --- fp3....32
F*/ fp1 fp2 fp3 --- fp4....32
F< fp1 fp2 --- f....32
F> fp1 fp2 --- fp3....32
F0< fp1 --- f....32
F0> fp1 --- f....32
3.7.4 Log, Exp, Trig....33
FLN fp1 --- fp2....33
FEXP fp1 --- fp2....33
FINT fp1 --- fp2....33
FSQRT fp1 --- fp2....33
FSIN fp1 --- fp2....33
FCOS fp1 --- fp2....33
FTAN fp1 --- fp2....33
FASIN fp1 --- fp2....33
FACOS fp1 --- fp2....33
FATAN fp1 --- fp2....33
RAD>DEG fp1 --- fp2....33

DEG>RAD fp1 --- fp2 ..... 33
3.7.5 Low-level definitions ..... 34
FOP n --- ..... 34
>W fp --- ..... 34
W> fp --- ..... 34
3.8 Line Editor ..... 35
-MOVE a n --- ..... 36
.PAD --- ..... 36
B --- ..... 36
D n --- ..... 36
BCOPY n1 n2 --- ..... 36
E n --- ..... 36
H n --- ..... 36
INS n --- ..... 36
L --- ..... 36
LINE n --- a ..... 36
N --- ..... 36
P n --- ..... 36
RE n --- ..... 37
S n --- ..... 37
SAVE --- ..... 37
ROOM --- ..... 37
TEXT c --- ..... 37
UNUSED --- n ..... 37
WHERE n1 n2 --- ..... 37
4 Technical specifications ..... 38
4.1 CPU Registers ..... 38
4.2 Single Cell 16 bits Integer Number Encoding ..... 38
4.3 Double cell 32 bits Integer Number Encoding ..... 38
4.4 Double Cell Floating-Point Number Encoding ..... 39
4.5 Single Cell 16 bits Heap Pointer Address Encoding ..... 39
5 Error messages ..... 40

6 The Dictionary.....41
6.1 The “core” dictionary.....41
'null' --- (immediate).....41
! n a ---.....41
!CSP ---.....41
# d1 --- d2.....41
#> d --- a b.....41
#BUFF --- n.....41
#S d1 --- d2 .....41
#SEC --- n.....41
' --- cfa.....41
( --- (immediate).....42
(+LOOP) n ---.....42
(.") ---.....42
(;CODE) ---.....42
(?DO) ---.....42
(?EMIT) c1 --- c2.....42
(ABORT) ---.....42
(COMPARE) a1 a2 n -- b.....42
(DO) ---.....43
(FIND) a1 a2 --- cfa b tf.....43
(LEAVE) --- .....43
(LINE) n1 n2 --- a b.....43
(LOOP) ---.....43
(MAP) a2 a1 n c1 --- c2.....43
(NEXT) --- a.....44
(NUMBER) d a --- d2 a2.....44
(SGN) a --- a2 f.....44
* n1 n2 --- n3.....44
*/ n1 n2 n3 --- n4.....44
*/MOD n1 n2 n3 --- n4 n5.....44
+ n1 n2 --- n3.....44

+! n a --- ....45
+- n1 n2 --- n3....45
+BUF a1 --- a2 f....45
+LOOP n1 --- (run time)....45
+ORIGIN n --- a....45
+TO n --- cccc....45
, n --- ....45
," --- ....45
- n1 n2 --- n3....46
--> --- ....46
-1 --- n....46
-DUP n --- n n (non zero)....46
-FIND --- cfa b tf (ok)....46
-TRAILING a1 n1 --- a2 n2....46
. n ---....46
." --- (immediate)....46
.( --- (immediate)....46
.C c --- (immediate)....47
.LINE n1 n2 ---....47
.R n1 n2 ---....47
/ n1 n2 --- n3....47
/MOD n1 n2 --- n3 n4....47
0 --- n....47
0< n --- f....47
0= n --- f....47
0> n --- f....47
0BRANCH f ---....47
1 --- n....48
1+ n1 --- n2....48
1- n1 --- n2....48
2 --- n....48
2! d a --- ....48

2* n1 --- n2....48
2+ n1 --- n2....48
2- n1 --- n2....48
2/ n1 --- n2....48
2@ a --- d....48
2CONSTANT d --- (immediate) (compile time)....48
2DROP d --- ....49
2DUP d --- d d....49
2OVER d1 d2 --- d1 d2 d1....49
2ROT d1 d2 d3 --- d2 d3 d1....49
2SWAP d1 d2 --- d2 d1....49
2VARIABLE d --- (immediate) (compile time)....49
3 --- n....49
3DUP n1 n2 n3 --- n1 n2 n3 n1 n2 n3....49
: --- (immediate)....50
:NONAME --- xt....50
; --- (immediate)....50
;CODE --- (immediate)....50
;S --- ....50
< n1 n2 --- f....50
<# ---....50
<> n1 n2 --- f....50
<BUILDS ---....51
<NAME cfa --- nfa....51
= n1 n2 --- f....51
> n1 n2 --- f....51
>BODY cfa --- pfa....51
>IN --- a ....51
>R n ---....51
? a --- ....51
?COMP --- ....51
?CSP --- ....52

?DO n1 n2 --- (immediate) (run time).....52
?DO- [a1 n1] a n ---.....52
?DUP n --- n n (non zero).....52
?ERROR f n ---.....52
?EXEC ---.....52
?LOADING ---.....52
?PAIRS n1 n2 ---.....52
?STACK ---.....53
?TERMINAL --- f.....53
@ a --- n.....53
ABORT ---.....53
ABS n --- u.....53
ACCEPT a n1 --- n2.....53
ACCEPT- a n1 --- n2.....53
AGAIN --- (immediate) (run time).....53
ALLOT n ---.....53
ALIGN ---.....54
ALIGNED a1 --- a2.....54
AND n1 n2 --- n3.....54
AUTOEXEC ---.....54
B/BUF --- n.....54
B/SCR --- n.....54
BACK a ---.....54
BASE --- a.....54
BASIC u ---.....54
BEGIN --- (immediate) (run time).....54
BINARY ---.....55
BL --- c.....55
BLANK a n ---.....55
BLK --- a.....55
BLK-FH --- a.....55
BLK-FNAME --- a.....55

BLK-INIT --- ....55
BLK-READ a n --- ....55
BLK-SEEK n --- ....55
BLK-WRITE a n --- ....55
BLOCK n --- a....55
BOUNDS a n --- a+n a....56
BRANCH --- ....56
BUFFER n --- a....56
BYE --- ....56
C! b a ---....56
C, b --- ....56
C/L --- c....56
C@ a --- b....56
CALL# n1 a --- n2....56
CASEOFF --- ....56
CASEON --- ....56
CELL --- 2....57
CELL+ n1 --- n2....57
CELL- n1 --- n2....57
CELLS n1 --- n2....57
CFA pfa --- cfa....57
CHAR --- c....57
CLS --- ....57
CMOVE a1 a2 n --- ....57
CMOVE> a1 a2 n ---....57
CODE --- ....57
COLD ---....58
COMPILE ---....58
COMPILE, xt ---....58
CONSTANT n --- (immediate) (compile time)....58
CONTEXT --- a....58
COUNT a1 --- a2 b....58

CR --- ....58
CREATE --- (compile time)....58
CSP --- a....59
CURRENT --- a....59
CURS --- ....59
D+ d1 d2 --- d3....59
D+- d1 n --- d2....59
D- d1 d2 --- d3....59
D. d --- ....59
D.R d n --- ....59
D0= d --- f....60
D< d1 d2 --- f....60
D= d1 d2 --- f....60
DABS d --- ud....60
DECIMAL --- ....60
DEFINITIONS --- ....60
DEVICE --- a....60
DIGIT c n --- u tf (ok)....60
DLITERAL d --- d (immediate) (run time)....60
DMAX d1 d2 --- d3....61
DMIN d1 d2 --- d3....61
DNEGATE d1 --- d2....61
DO n1 n2 --- (immediate) (run time)....61
DOES> --- ....61
DP --- a....61
DPL --- a....61
DROP n --- ....61
DUP n --- n n....62
DUP>R n --- n ....62
DU< ud1 ud2 --- f....62
ELSE a1 n1 --- a2 n2 (immediate) (compile time)....62
EMIT c --- ....62

EMITC b --- ....62
EMPTY-BUFFERS --- ....62
ENCLOSE a c --- a n1 n2 n3....62
END a n --- (immediate) (compile time)....62
ENDIF a n --- (immediate) (compile time)....63
ERASE a n --- ....63
ERROR b --- n1 n2....63
EXEC: n --- ....63
EXECUTE cfa --- ....63
EXIT --- ....63
EXP --- a....63
EXPECT a n --- ....64
FENCE --- a....64
FILL a n b --- ....64
FIRST --- a....64
FLD --- a....64
FLIP n1 --- n2....64
FLUSH --- ....64
FM/MOD d n1 --- n2 n3....64
FORGET --- ....64
FORTH --- (immediate)....65
F_CLOSE n --- f....65
F_FGETPOS n --- d f....65
F_GETLINE a n1 fh --- n2....65
F_INCLUDE n --- ....65
F_OPEN a1 a2 n1 --- n2 f....65
F_READ a n1 n2 --- n3 f....65
F_SEEK d n --- ....65
F_SYNC n --- f....65
F_WRITE a n1 n2 --- n3 f....65
HANDLER --- a....65
HERE --- a....66

HEX --- ....66
HLD --- a....66
HOLD c --- ....66
HP --- a....66
I --- n ....66
I' --- n ....66
ID. nfa --- ....66
IF f --- (immediate) (run time)....66
IMMEDIATE --- ....66
INCLUDE ---....67
INDEX n1 n2 ---....67
INKEY --- b ....67
INTERPRET --- ....67
INVERT n1 --- n2....67
INVV --- ....67
J --- n ....67
K --- n ....67
KEY --- b ....68
L/SCR --- n....68
LATEST --- nfa ....68
LEAVE --- ....68
LFA pfa --- lfa ....68
LIMIT --- a....68
LIST n --- ....68
LIT --- n ....68
LITERAL n --- n (immediate) (run time)....68
LOAD n --- ....69
LOAD+ n --- ....69
LOAD- n --- ....69
LOOP a n --- (immediate) (run time)....69
LP --- a....69
LSHIFT n1 u --- n2....69

M* n1 n2 --- d....69
M*/ d1 n1 n2 --- d2....69
M+ d u --- d2....70
M/ d n1 --- n2....70
M/MOD d1 n1 --- n2 n3....70
MARK a n ---....70
MARKER --- (immediate) (run time)....70
MAX n1 n2 --- n3....70
MESSAGE n ---....70
MIN n1 n2 --- n3....70
MMU7! n ---....70
MMU7@ --- n....70
MOD n1 n2 --- n3....71
MS u ---....71
M_P3DOS n1 n2 n3 n4 a --- n4 n5 n6 n7 f....71
NEEDS ---....71
NEGATE n --- -n....72
NFA pfa --- nfa....72
NIP n1 n2 --- n2....72
NMODE --- a....72
NOOP ---....72
NOT ---....72
NUMBER a --- d....72
OCTAL ---....72
OFFSET --- a....72
OPEN< --- fh....73
OR n1 n2 --- n3....73
OUT --- a....73
OVER n1 n2 --- n1 n2 n1....73
P! b u ---....73
P@ u --- b....73
PAD ---....73

PFA nfa --- pfa....73
PICK n --- pfa....73
PLACE --- a....73
PREV --- a....73
QUERY --- ....74
QUIT --- ....74
R@ --- n....74
R# --- a....74
R/W a n f --- ....74
R0 --- a....74
R> --- n....74
R>DROP --- ....74
RECURSE --- ....74
REG! b n --- ....74
REG@ n --- b....74
REMOUNT --- ....74
RENAME ---....75
REPEAT a1 n1 a2 n2 --- (immediate) (compile time)....75
ROT n1 n2 n3 --- n2 n3 n1....75
ROLL n1 ... k --- n2 ... n1....75
RP! a --- ....75
RP@ --- a....75
RSHIFT n1 u --- n2....75
S0 --- a....75
S>D n --- d....75
SCR --- a....76
SELECT n --- ....76
SIGN n d --- d....76
SM/REM d n1 --- n2 n3....76
SMUDGE --- ....76
SOURCE-ID --- a....76
SP! a --- ....76

SP@ --- a....76
SPACE --- ....76
SPACES n --- ....76
SPAN --- a....77
SPLASH --- ....77
SPLIT n1 --- n2 n3....77
STATE --- a....77
SWAP n1 n2 --- n2 n1....77
THEN a n --- (immediate)....77
TIB --- a....77
TO n --- ....77
TOGGLE a b --- ....77
TRAVERSE a1 n --- a2....77
TRUV ---....77
TUCK n1 n2 --- n2 n1 n2....78
TYPE a n --- ....78
U. u --- ....78
U< u1 u2 --- f....78
UM* u1 u2 --- ud....78
UM/MOD ud u1 --- u2 u3....78
UNTIL a n --- (immediate) (compile time)....78
UPDATE --- ....78
UPPER c1 --- c2....78
USE --- a....78
USER n --- ....78
VALUE n --- ....79
VARIABLE ---....79
VIDEO --- ....79
VOC-LINK --- a....79
VOCABULARY ---....79
WARM --- ....80
WARNING --- a....80

WHILE f --- (immediate) (run time).....80
WIDTH --- a.....80
WITHIN n1 n2 n3 --- f.....80
WORD c --- a.....80
WORDS --- .....80
XOR n1 n2 --- n3.....81
[ --- (immediate) .....81
[CHAR] --- (immediate) (compile time) .....81
[COMPILE] --- (immediate) .....81
\ --- .....81
] --- .....81
6.2 Case -Of structure.....82
CASE n0 --- (immediate) (run time).....82
OF n0 nk --- (immediate) (run time).....82
ENDOF --- (immediate) (run time).....82
ENDCASE --- (immediate) (run time).....82
(OF) n0 nk --- (run time).....82
6.3 Heap Memory Facility.....83
6.3.1 Heap Pointer encoding and decoding.....83
6.3.2 Heap Pointer description for 64 kiBytes space.....84
6.3.3 Heap structure.....85
+" ha --- ha.....86
+C ha c --- ha.....86
>FAR ha --- a p.....86
<FAR a p --- ha.....86
FAR ha --- a.....86
H" --- ha.....87
HEAP n --- ha.....87
POINTER ha --- a.....87
SKIP-PAGE n --- .....87
S" --- a n.....87
(S") --- a n.....87

HEAP-INIT --- ....87
HEAP-DONE --- ....87
6.3.4 Heap Pointer description for 128 kiBytes space....88
6.3.5 Heap Pointer description for 256 kiBytes space....89
6.3.6 Heap Pointer description for 512 kiBytes space....90
6.3.7 Heap Pointer description for 1024 kiBytes space....91
6.4 Testing Suite....92
TESTING --- ....92
T{ --- ....92
-> --- ....92
}T --- ....92
6.5 Other Utilities....93
SHOW-PROGRESS n --- ....93
?VOCAB --- ....93
7 The Memory Map....94
Contents....95