v-FORTH 1.51

ZX Spectrum Next version

1990-2021 Matteo Vitturi

Introduction
&
Technical Info

Build 2021126

1. Forewords

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 1.5** is available at https://sites.google.com/view/vforth/vforth15-next and on GitHub repository too 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. Even if this is a "working" piece of software, the porting is still a work-in-progress, there are many things to do.

This new sub-version **v-Forth 1.51** comes with the following improvement: now, each definition is formed by a direct address thread, while in v-Forth 1.5 is formed by an indirect address thread. See §5. "Debugger Utility" for details on how the various core-routine are implemented.

From this version v-Forth is a "direct-thread" instead of an "indirect-thread" Forth system. Any Low-Level definition saves two bytes, but any non Low-Level definitions needs one additional byte because CFA does not contain an address anymore, instead CFA contains the real machine-code to be executed. For non Low-level definitions CFA is a three byte instruction "CALL aaaa" to the ;CODE part routine that handles that kind of definition.

Disclaimer

The author – me – is not a native English speaker and, for certain, you will find grammatical errors. In case, it would be very 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.

Legenda

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
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

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

In alternative, 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 as follow:

This Forth System comes with a 16 Mbytes file named **!Blocks-64.bin** that provides a BLOCK-like mass-storage system to hold 16.383 BLOCKS (or Screens) that can be edited using the "Full Screen Editor" utility available after you give

```
NEEDS EDIT
```

Each BLOCK / Screen is 1 kiByte long and can store text in **16 lines x 64 columns way** or can be viewed as a *virtual-memory* area where you can persistently store anything you like.

Here is the directory hierarchy:

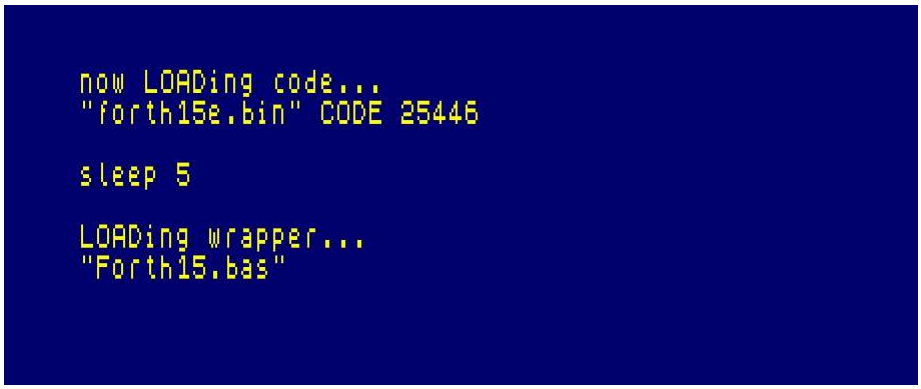
doc/	where I am used to keep some text image-versions of !Blocks-64.bin
inc/	contains text-file of single word definitions available after you give NEEDS *word.*
lib/	same as inc/ but text-file are a collection of several words that forms a "library utility", e.g. SEE.
src/	among the others, the source 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

Name	Ext	↓Size	Date	Attr
[..]		<DIR>	16/09/2021 21:43	----
[doc]		<DIR>	16/09/2021 21:43	----
[inc]		<DIR>	16/09/2021 22:16	----
[lib]		<DIR>	16/09/2021 22:16	----
[src]		<DIR>	16/09/2021 21:43	----
[test]		<DIR>	16/09/2021 21:43	----
[util]		<DIR>	16/09/2021 21:43	----
!Blocks-64	bin	16.776.704	05/09/2021 19:17	-a--
forth15e	bin	10.127	05/09/2021 19:17	-a--
Forth15_loader	bas	1.159	05/09/2021 19:17	-a--
Forth15	bas	652	05/09/2021 19:17	-a--

If you wish to use a different directory instead of C:/forth, you need to modify the paths in the two Basic programs.

The Forth System is activated by running a Basic program **C:/forth/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; it loads **forth15e.bin** (the Forth core) and then it loads a smaller Basic launcher **Forth15.bas** 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 using **[N] key**. It is anyway a good idea allowing Forth to continue and LOAD **Screen # 11** that in turn loads a few useful Screens which make available, among the others, two pecuiliar 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 `AUTOEXEC` word.

The Basic launcher **Forth15.bas** usually auto-starts the first time at `LINE 20`, so you usually won't care, 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. give `RUN` : This does a `WARM` start, preserving your previous work and buffer status.

b. give `RUN 20` : This does a `COLD` start, restoring all as you just loaded from SD card.

Before entering Forth, the Basic launcher does (or could do) an `OPEN# 13,"o>output.txt"` that can be later chosen 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 idea of mine.

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`. See chapter "5. The dictionary" for more details.

In this Forth implementation I preferred LAYER 1,2 display mode to allow 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 give `NEEDS LAYERS` .

3. The Full Screen Editor Utility – BLOCK oriented

The `EDIT` definition is available after you give: `NEEDS EDIT` or in the old way `190 LOAD`.

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 `EDIT` definition. This enters a simple full-screen editor that can modify current Screen, one screen at a time. While using `EDIT`, you are allowed move to next Screen or to previous Screen using the command explained below.

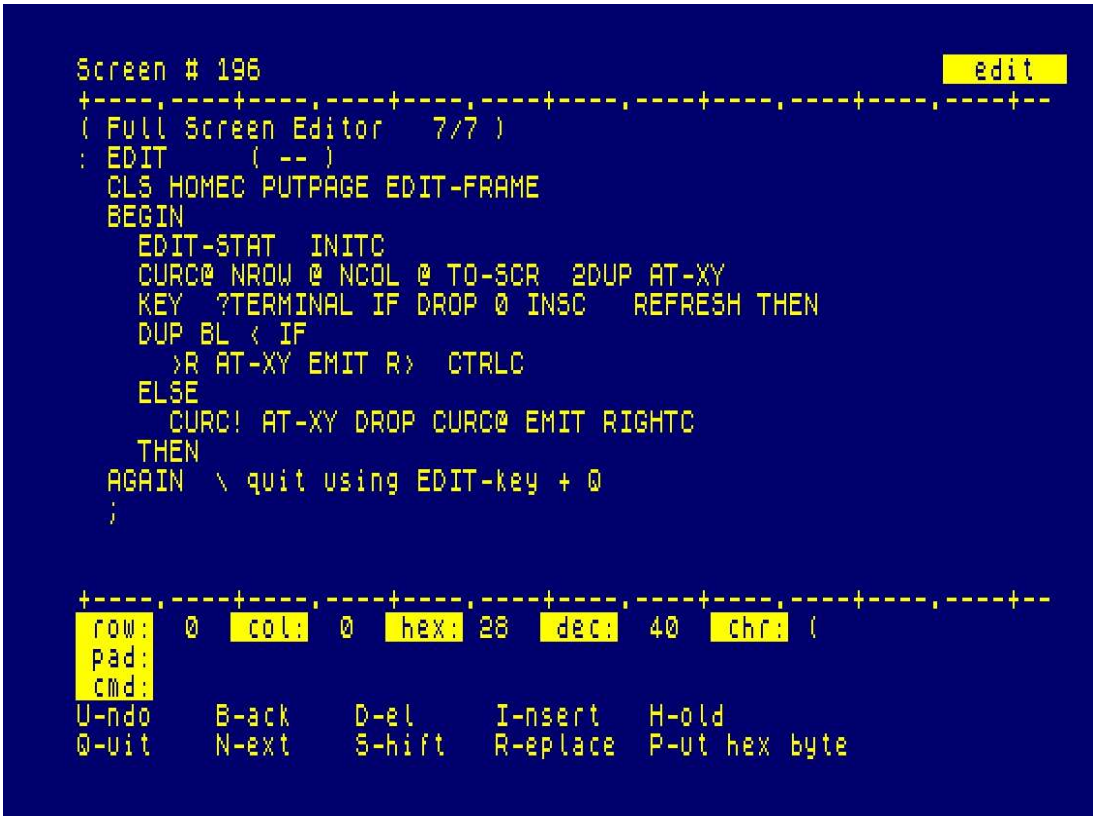
Remember: to *quit* `EDIT` editor, you have to use [**Edit**] key followed by [**Q**] key, in a way that mimics Unix **vi** editor.

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

EDIT **---**

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

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



It shows a header reporting the Screen number and a line-ruler followed by 16 lines that compose 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, i.e. [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) is shown at 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.

After **[Edit] key (i.e. 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 the display to become corrupted 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.

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

4. Search and Locate Utility

The following definition are available after you give alternatively:

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

Using the old way `70 LOAD` will compile them all, using source from Screens# 70-75.

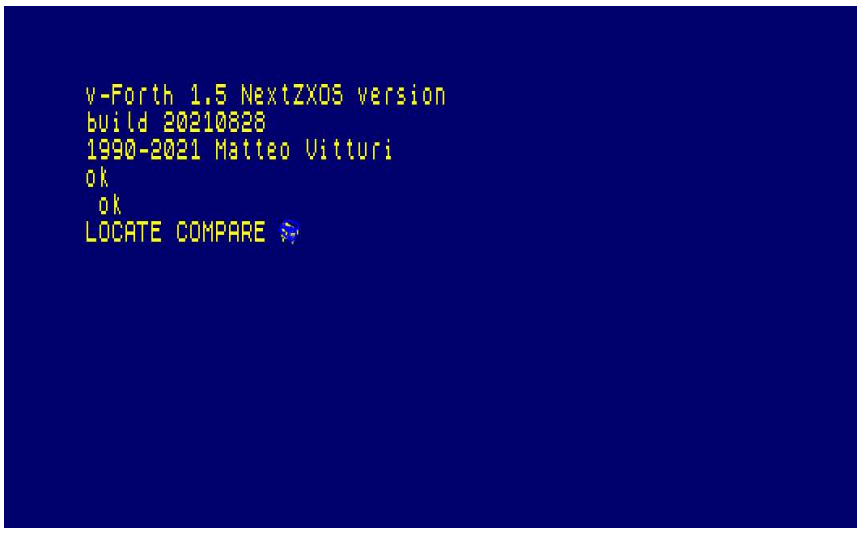
LOCATE ---

Used in the form

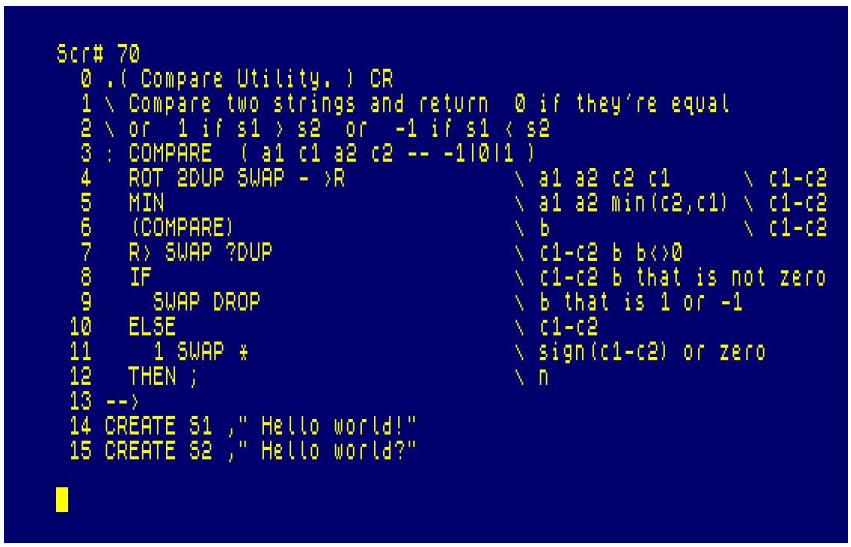
```
LOCATE cccc
```

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 0 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 .
```

5. Debugger Utility

The following definitions are available after loading if you give `NEEDS SEE` or via `20 LOAD` or after a regular `AUTOEXEC`.

SEE ---

Used in the form

SEE cccc

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

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, defined as follow:

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

If you give

```
SEE TYPE
```

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

```
Nfa: 71BB 84
Lfa: 71C0 BOUNDS
Cfa: 71C2 6B87
BOUNDS (?DO) 12 I C@ EMIT (LOOP) -8 EXIT ok
```

The first line shows **TYPE** Name Field Address (**$71E1** in this case) 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 (**$71C0**) which holds a pointer to **BOUNDS**'s NFA that in this case happens to be the previous definition in the dictionary.

The third line is the Code Field Address (**$71C2**) that – in this new "direct-thread" 1.51 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 **$6B87**.

In the previous "indirect-thread" 1.5 version, the CFA holds a pointer to the machine-code part of a regular colon-definition so the "inner-interpreter" can jump to it.

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.

Another example, the word **NIP** that removes the second element of Stack, isn't a colon-definition, instead it's 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 give

```
SEE NIP
```

under the previous *indirect thread* 1.5 version it will emit

```
Nfa: 6A87 83
Lfa: 6A8B DROP
Cfa: 6A8D 6A8F
6A8D 8F 6A E1 E3 DD E9  84 54   ac]i T
6A95 55 43 CB 9D 6A B2  6A E1  UCK j ja
```

In this case, since **NIP** is a low-level definition, the PFA part is shown as a hexadecimal **DUMP**.

Instead, using the newer *direct thread* 1.51 version it will emit

```
Nfa: 6A07 83
Lfa: 6A0B DROP
Cfa: 6A0D DDE3
6A0D E1 E3 DD E9  84 54 55 43  ac]i TUC
6A15 CB 07 6A E1  D1 E5 D5 E5  K jaQeUe
```

In this case, since **NIP** is not a colon-definition, its has no PFA part.

Again, the first line shows **NIP**'s CFA (**$6A87** or **$6A07** in this case) and **$83**, the counter byte, that indicates a 3-bytes length word name.

The second line is **NIP**'s LFA (**$6A8B** or **$6A0B**) that contains a pointer to DROP's NFA, that is the previous definition in dictionary.

The third line is **NIP**'s CFA (**$6A8D** or **$6A0D**) 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 (**6A8F**) where the small piece of machine-code lies. In the direct-thread version this address contains the routine and, in this case, it shows part of the sub-sequent code (**DDE3**).

In both versions, 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 – 84 54 55 43 – 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 Forth.

The Inner-interpreter

Here is the indirect-threaded inner-interpreter routine

```
ld      a, (bc)    // bc is the "instruction pointer"
inc     bc
ld      l, a
ld      a, (bc)
inc     bc
ld      h, a       // hl now contains xt (CFA) of word being executed
ld      e, (hl)    // that contains the address of machine code
inc     hl
ld      d, (hl)
ex      de, hl
jp      (hl)       // and jump to it
```

The direct-threaded one simply omits the pointer de-reference marked in yellow.

```
ld      a, (bc)    // bc is the "instruction pointer"
inc     bc
ld      l, a
ld      a, (bc)
inc     bc
ld      h, a       // hl now contains xt of word being executed
                   // but there's no need to dereference it
jp      (hl)       // so directly jump to it
                   // where the real code is
                   // or a CALL to a code part is
```

Omitting such de-reference part increases the overall speed about 20-25% and reduces the length of low-level definitions by two bytes, but on the other hand increments all non-low-level definitions by one byte.

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.
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
```

6. Technical specifications

CPU Registers

Registers are used in the in the following way:

AF – Used for normal operations.
BC – **Forth Instruction Pointer**: should be preserved on enter-exit a definition and during ROM/OS calls.
DE – Free (Low part when used for 32-bit manipulations)
HL – **Work Register** (High part when used for 32-bit manipulations)
AF'– Not used, somewhere used for backup purpose
BC'– Used in I/O operations or in complex definitions
DE'– Used in I/O operations or in complex definitions
HL'– Used in I/O operations or 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) word.
IY – Used by ZX System, must be preserved to let keyboard to be served during Interrupt.

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. 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 "little-endian" way, that is the lower address has the least significant part, the in L register. The byte at higher address has the most significant part, the one in H register, as usual for Zilog Z80.

Double cell 32 bits Integer Number Encoding

The second integer format requires two *integers* to form a 32 bits number said *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* kept in CPU register in the 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 Calculator Stack the *double integer* requires four contiguous bytes split in the two *integers* that forms it 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 is in the higher address, that is the second element from top. so it appears as L H E D,

CPU	Calculator Stack
D	SP + 3

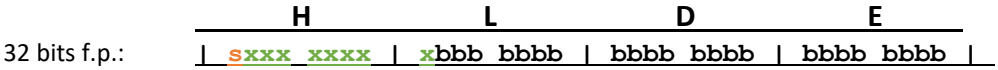
E	SP + 2
H	SP + 1
L	SP + 0 (Top Of Stack)

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

Double Cell Floating-Point Number Encoding

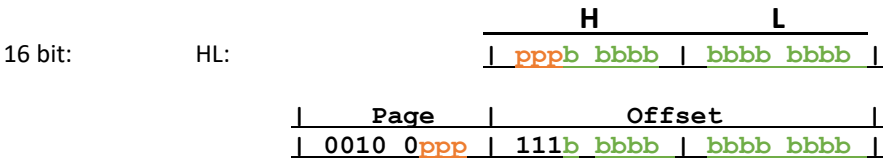
There is another optional format that use 32 bits as a *double integer*, but all bits are used in a different way to allows to represent a *floating point number* approximately between –1.7 * 10^38 and +1.7 * 10^38 with 6-7 precision digits.
The sign is kept in the most significant bit, the same way as a *double integer*; then eight bits follow as the exponential part, then 23 bits of mantissa. 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.

Single Cell 16 bits Heap Pointer Address Encoding

There 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 64 and 71, 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.



7. Error messages.

Error messages strings are stored at Screens from 4 to 6 that are therefore reserved.

Code	Message
#0	is undefined.
#1	Stack empty.
#2	Dictionary full.
#3	No such line.
#4	has already been defined.
#5	Invalid stream.
#6	No such block.
#7	Stack is full!
#8	Old dictionary is full.
#9	Tape error.
#10	Wrong array index.
#11	Invalid floating point.
#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.
#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.

8. The 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 3 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 that discards the current DO-LOOP frame and executes an unconditional jump. The memory cell following (LEAVE) has the offset to be relatively added to the Instruction Pointer to jump after the corresponding (LOOP) or (+LOOP). It is compiled by LEAVE.

(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 not present within string a1, then it is not translated and c2 = c2 if it is not translated. n is the length of both a1 and a2.
For example:

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

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

: needs-ch ( a -- ) \ Replace illegal character in filename string a
  count bounds
  Do
      ncdm ndom 9 Literal 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 created word wants to return an *intger* value on TOS, 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. For example, to create two definitions that disable and enable interrupts, without an ASSEMBLER, you could use the following snippet:

```
CODE   INT-DI   HEX
  F3 C,          \  di
  C3 C, (NEXT) , \  jp (NEXT)
  SMUDGE         \  now a dictionary search will find this word

CODE   INT-EI   HEX
  FB C,          \  ei
  C3 C, (NEXT) , \  jp (NEXT)
  SMUDGE         \  now a dictionary search will find this word
```

This Forth implementation *always* keeps (NEXT) value in **IX register**, so that the previous snipped should be written as:

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

CODE   INT-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 the total compared with the limit. The jump back happens :

a) while index < limit if `n1` > 0;
b) while index > limit if `n1` < 0.

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.

, 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.

-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** **---**

Prints the integer n followed by a space.

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

Used in the form

." cccc "

At compile-time, within a colon-definition, compiles the primitive to output the text followed by the string cccc (delimited by "). 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 sends the text to output.

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

Used in the form

.(cccc)

acting as ." cccc " but the string is delimited in a different way

.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.

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

Computes the quotioent n4 and the reminder n3 of the integer division n1/ n2. The reminder has the sign of n1.

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**

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`.

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`.

2DROP **d** **---**
n1 n2 **---**
Discards a double integer from the TOS, i.e. discards the top two integer.

2DUP **d** **--- d d**
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.

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

: **--- (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.

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

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

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

Used in the form

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

terminates a colon definition stoppin copilation of word cccc and compiling (;CODE). Usually ;CODE is followed by 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.

< **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.

<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.

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

Converts a cfa in its nfa. It is the same as the sequence >BODY NFA.
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>`.

? 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 is less than the limit, the executions returns to the corresponding `?DO`, otherwise the two parameters are discarded and the execution continues after the `LOOP`.
The limt `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 BACK equivalent definition fitted for `?DO`. It computes and compiles a relative offset from `a` to `HERE` and in 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.

?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. Leaves a `tf` if the [BREAK] key is pressed, `ff` oherwise.

@ **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.

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

It executes an AND binary 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.

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.

BL --- c
Constant for "Blank". This implementation uses ASCII and `BL` is 32.

BLANKS a n ---
Fills with "Blanks" `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`.

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+ **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 at `HERE`. 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 a programmer responsibility to execute `SMUDGE`. This word is 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.

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

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

D+- **ud n** **---** **d**

Computes d that is ud with the sign of n.

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

Prints 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** **---**

Prints 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.

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 an 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.

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 is less than the limit, the executions returns to the corresponding DO, otherwise the two parameters are discarded and the execution continues after the LOOP.
The limt 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 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.

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.

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 0x00 'null' characters.

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 followd 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.

EXECUTE **cfa** **---**

Executes the word which cfa is held on TOS.

EXIT **---**

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.

EXP **---** **a**

User variable that holds the exponent in a floating-point conversion.

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.

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.

FLUSH **---**

Executes SAVE-BUFFERS. It saves to disk the buffers marked "modified" by UPDATE.

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. See `MARKER`.

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 option: 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 option: 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 option: 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 option: 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 option: 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 option: 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 option: 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.

HERE --- a

Leaves the address of next location available on the dictionary.

HEX --- a

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.

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.
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.

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**

Shows a (flashing) cursor on current video position and waits for a keypress. 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]
C8	>=	→	20	same as SHIFT-0 [BACKSPACE]
C9	<>	→	06	same as CAPS-SHIFT + 2 and toggles CAPS-LOCK On and Off

Depending on CAPS-LOCK state, the faces of flashing cursor are different. They depend 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!
```

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 can be set at E000h to allow MMU7 as a general purpose 8K RAM bank. See also: `FIRST`.

LIST **n** **---**

Prints screen number n. 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 defintion 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 while the index is less than the 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. Not used.

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*/.

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 NEESS 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.

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.

M_P3DOS n1 n2 n3 n4 a --- n4 n5 n6 n7 f
This is the NZXOS 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.

NEEDS ---
It is 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 not found tryes from file **lib/cccc.f**

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.

Since any given Screen # n occupies BLOCKs n and n+1, NEEDS exploits BLOCK number 0 that normally isn't reachable and use it as a temporary buffer for each line read from file, this way a text source line cannot exceed 511 bytes.

Some characters are illegal for filename: noticeably the "double-quote" character (") is among them. In such case, these characters are converted into "tilde" (~) and this file is then searched for.
For example:

NEEDS S" searches for the file inc/S~.f instead of an illegal filename inc/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 `3 F_CLOSE .`
[...to be fixed…].

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.

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.
If no conversion can be done, and error #0 is raised.

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.

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.

OR **n1 n2 --- n3**

Executes an OR binary 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!`.

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`.

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 1 ROLL is SWAP, 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.

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.

S0 --- a
User variable that holds the initial value of che SP register. See also: SP! and SP@.

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 #>.

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 --- a
Shows splash screen build date-number.

STATE --- a
User variable that holds the compilator status. A non-zero value indicates a compilation in progress.

STRM --- a
Variable containing the stream number used by the Screens/Blocks facility. Used by NextZXOS calls.
See also NXTDRV, NXTSTP, NXTRD, NXTWR.

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** **---**

Prints 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, DL.

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 **n** **---**

Defining word used in the form:

```
n VARIABLE cccc
```

creates the word cccc with the pfa containing the initial value n. When cccc is executed, it puts on TOS the pfa of cccc that is the address that holds the value n.

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.

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. Pressing Break stops.

XOR n1 n2 --- n3

Executes a XOR binary 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.

\ ---

Used in the from:

```
\ ...
```

Any character that follow \ until the end of line are treated as a comment.

] ---

Resumes the compilation suspended by [so it is possible to complete the definition.

9. Line Editor

The following definitions are available after you give `90 LOAD` or after you include the `EDITOR` vocabulary via `NEEDS EDITOR`.

There is no single definition that needs all the following words, but `EDIT` is the best candidate: so instead of using the old fashion `90 LOAD` you may use `NEEDS EDIT`.

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 in Screen# 13 for now, deeming it too dangerous for my tastes; instead I usually use `BCOPY` from an actually empty Screen. You may give `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 three `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 the 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#1000 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) TIB	Parsing Operation	 PAD	Edit Operations	One BLOCK 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.

10. Case -Of structure

The following definitions are available after you give `17 LOAD` or `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 word `CASE` 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.
At compile time `CASE` leaves 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 word is used in colon-definition within a Case-Of structure.
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.
At compile-time, it compiles `(OF)` and `0BRANCH` using `n1` and `n2` for syntax checking and leaving `a` to be used by `ENDCASE` to resolve `0BRANCH`.
See also `CASE`.

ENDOF **---** **(immediate)** **(run time)**
a1 n1 **---** **a n2** **(compile time)**

This word ends an "Of-EndOf" clause started with `OF`.
At compile-time it acts like a `THEN`, first compiling a BRANCH that will be resolved by `ENDCASE` to skip any subsequent "Of-End-Of" clauses and resolving 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 word 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 word represents the run-time semantic compiled by `OF` word. 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.

11. Interrupt Service Routine

Giving `NEEDS INTERRUPT` a new Vocabulary will be loaded in memory along with some back-end words that allows set-up an Interrupt-Driven word: The ISR must be a single word suitably defined

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.

INTERRUPT **---**
This is an `IMMEDIATE` word that selects the `INTERRUPT` vocabulary to make visible the words described in this section.

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

INT-W **---**
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 `INT-RET` so that `INT-W` can be viewed as an anonymous word that contains two words: the *interrupt* word and the *return-from-interrupt* word.

INT-ON **---**
Enable Interrupt 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 required a CALL to address $6363 is performed.

At address $6363 is set a jump to address of `INT-SUB` body i.e. `[ ' INT-SUB >BODY ]`
It is used in the form

```
INT-OFF
' ISR-WORD  INT-W  !
INT-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. Attention must be paid to avoid any unwanted interference with the normal *foreground* Forth execution.

Typical usage is to control some Sprite movement.

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

```
HEX
: ISR-WORD
      80 57FF !
      5701 5700 FF CMOVE>
      5700 4700 100 CMOVEs
      4700 4F00 100 CMOVE
;
INT-OFF
' ISR-WORD INT-W !
INT-ON
```

INT-EI **---**

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

INT-DI **---**

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

INT-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

INT-SYNC **---**

Low-level “halt”. It actually executes an HALT opcode.

SETIREG **---**

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

INT-RET **---**

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

INT-SUB **---**

Low-level “interreupt service routine” definition. It saves all registers and gives control to `INT-W` 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.

12. Heap Memory Facility

The definitions that handle the Heap are available after loading via `80 LOAD` or `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

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.

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)
```

Heap structure

The Heap can be envised as a “linked-list” starting at 8 kiB 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 unpredictable)
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 give `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 give `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.

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)
```

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)
```

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)
```

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)
```

13. 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 giving

```
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

14. 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 give `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 vForth 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 extend and 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 mantissa is stored the remaining bits of HL and 16 bits of DE. B is defaulted to a fixed value.

If the floating-double-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 give.

```
FLOATING  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 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.

Activation and conversion

To import the library option you must give NEEDS FLOATING then,

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`.

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 `SELECT`ed stream.

F. **fp ---**
Prints fp to video or current `SELECT`ed stream.

1/2 **--- fp**
Put on TOS the value 0.5.

PI **--- fp**
Put on TOS the value of pi.

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.

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.

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.

15. 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`.

16. 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