

v-FORTH 1.5

ZX Spectrum Next version

1990-2020 Matteo Vitturi

Technical Info

Introduction

This document introduces some technical details about this Forth implementation and the glossary of all core words.

This is a straight FIG-Forth I ported to the new **Sinclair ZX Spectrum Next** based on my previous work “vForth 1.413” available at <https://github.com/mattsteeldue/vforth>.

This version “vForth 1.5” is available on GitHub repository too at <https://github.com/mattsteeldue/vforth-next>. The first main big difference from the previous version is that it uses a dedicated file on SD instead of on a ZX Microdrive cartridge.

Even if this is a working piece of software, the porting is still a work-in-progress, there are many things to do.

Disclaimer

**Copying, modifying and distributing this software is allowed provided this copyright notice is kept.
This work is available “as-is” with no whatsoever warranty.**

The author – me – is not a native English speaking 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. I am not able to write a longer disclaimer than the above.

Contents

Introduction.....	2
Disclaimer	2
Technical specifications	12
CPU Registers.....	12
16 bits Number Encoding	12
32 bits Number Encoding	12
Floating-Point Number Encoding	13
Dictionary	13
Legenda	13
Core dictionary	14
'null' --- (immediate)	14
! n a ---	14
!CSP ---	14
# d1 --- d2	14
#> d --- a b	14
#BUFF --- n	14
#S d1 --- d2 CORE	14
#SEC --- n	14
' --- cfa	15
(--- (immediate)	15
(+LOOP) n ---	15
(.") ---	15
(;CODE) ---	15
(?DO) ---	15
(?EMIT) c1 --- c2	15
(ABORT) ---	15
(COMPARE) a1 a2 n -- b	15
(DO) ---	16
(FIND) a1 a2 --- cfa b tf	16
(LINE) n1 n2 --- a b	16
(LOOP) ---	16
(NEXT) --- a	16
(NUMBER) d a --- d2 a2	16
(SGN) a --- a2 f	16
* n1 n2 --- n3	17
*/ n1 n2 n3 --- n4	17
*/MODn1 n2 n3 --- n4 n5	17
+ n1 n2 --- n3	17
+! n a ---	17
+- n1 n2 --- n3	17
+BUF a1 --- a2 f	17
+LOOP n1 --- (run time)	17
+ORIGIN n --- a	17
, n ---	18

- n1 n2 --- n3.....	18
--> ---	18
-1 --- n.....	18
-DUP n --- n n (non zero)	18
-FIND --- cfa b tf (ok)	18
-TRAILING a1 n1 --- a2 n2	18
. n ---.....	18
." --- (immediate)	18
.(--- (immediate)	18
.C c --- (immediate)	19
.LINE n1 n2 ---.....	19
.R n1 n2 ---.....	19
/ n1 n2 --- n3.....	19
/MOD n1 n2 --- n3 n4.....	19
0 --- n.....	19
0< n --- f.....	19
0= n --- f.....	19
0> n --- f.....	19
0BRANCH f ---.....	19
1 --- n.....	20
1+ n1 --- n2.....	20
1- n1 --- n2.....	20
2 --- n.....	20
2! d a ---.....	20
2* n1 --- n2.....	20
2+ n1 --- n2.....	20
2/ n1 --- n2.....	20
2@ a --- d.....	20
2DROP d ---	20
2DUP d --- d d.....	20
2OVER d1 d2 --- d1 d2 d1.....	20
2ROT d1 d2 d3 --- d2 d3 d1	21
2SWAP d1 d2 --- d2 d1.....	21
3 --- n.....	21
: --- (immediate)	21
; --- (immediate)	21
;CODE --- (immediate)	21
;S --- (immediate)	21
< n1 n2 --- f.....	21
<# ---.....	21
<BUILDS ---.....	22
<NAME cfa --- nfa.....	22
= n1 n2 --- f.....	22
> n1 n2 --- f.....	22
>BODY cfa --- pfa.....	22
>R n ---	22
? a ---.....	22

?COMP	---	22
?CSP	---	22
?DO n1 n2	--- (immediate) (run time)	22
?DUP n	--- n n (non zero)	23
?ERROR f n	---	23
?EXEC	---	23
?LOADING	---	23
?PAIRS n1 n2	---	23
?STACK	---	23
?TERMINAL	--- f	23
@ a	--- n	23
ABORT	---	23
ABS n	--- u	24
ACCEPT a n1	--- n2	24
ACCEPT- a n1	--- n2	24
AGAIN	--- (immediate) (run time)	24
ALLOT n	---	24
AND n1 n2	--- n3	24
AUTOEXEC	---	24
B/BUF	--- n	24
B/SCR	--- n	24
BACK a	---	25
BACK-[a1 n1] a n	---	25
BASE	--- a	25
BASIC u	---	25
BEGIN	--- (immediate) (run time)	25
BL	--- c	25
BLANKS a n	---	25
BLK	--- a	25
BLOCK n	--- a	25
BRANCH	---	26
BUFFER n	--- a	26
BYE	---	26
C! b a	---	26
C, b	---	26
C/L	--- c	26
C@ a	--- b	26
CASEOFF	---	26
CASEON	---	26
CELL+ n1	--- n2	26
CELL- n1	--- n2	26
CELLS n1	--- n2	27
CFA pfa	--- cfa	27
CHAR	--- c	27
CLS	---	27
CMOVEa1 a2 n	---	27
CMOVE>a1 a2 n	---	27

COLD	---	27
COMPILE	---	27
CONSTANT n	--- (immediate) (compile time)	27
CONTEXT	--- a	27
COUNT a1	--- a2 b	27
CR	---	28
CREATE	--- a	28
CSP	--- a	28
CURRENT	--- a	28
D+ d1 d2	--- d3	28
D+- ud n	--- d	28
D. d	---	28
D.R d n	---	28
DABS d	--- ud	28
DECIMAL	---	28
DEFINITIONS	---	29
DEVICE	--- a	29
DIGIT c n	--- u tf (ok)	29
DL	--- a	29
DLITERAL d	--- d (immediate) (run time)	29
DMINUS d1	--- d2	29
DO n1 n2	--- (immediate) (run time)	29
DOES>	---	30
DOSSCALL n1 n2 n3 a	--- n4 n5 n6 n7	30
DP	--- a	30
DPL	--- a	30
DROP n	---	30
DUP n	--- n n	30
ELSE a1 n1	--- a2 n2 (immediate) (compile time)	30
EMIT c	---	31
EMITC b	---	31
EMPTY-BUFFERS	---	31
ENCLOSE a c	--- a n1 n2 n3	31
END a n	--- (immediate) (compile time)	31
ENDIF a n	--- (immediate) (compile time)	31
ERASE a n	---	31
ERROR b	--- n1 n2	31
EXECUTE cfa	---	32
EXP	--- a	32
EXPECT a n	---	32
FENCE	--- a	32
FILL a n b	---	32
FIRST	--- a	32
FLD	--- a	32
FLUSH	---	32
FORGET	---	32
FORTH	--- (immediate)	32

HERE	---	a	32
HEX	---	a	33
HLD	---	a	33
HOLD	c ---		33
I	---	n	33
ID.	nfa ---		33
IF	f ---	(immediate) (run time)	33
IMMEDIATE	---		33
IN	---	a	33
INDEX	n1 n2 ---		33
INKEY	---	b	34
INTERPRET	---		34
L/SCR	---	n	34
KEY	---	b	34
LATEST	---	nfa	34
LEAVE	---		34
LFA	pfa ---	lfa	34
LIMIT	---	a	35
LIST	n ---		35
LIT	---	n	35
LITERAL	n ---	n (immediate) (run time)	35
LOAD	n ---		35
LOAD-	n ---		35
LOOP	a n ---	(immediate) (run time)	35
LP	---	a	35
LSHIFT	n1 u ---	n2	36
M*	n1 n2 ---	d	36
M/	d n1 ---	n2 n3	36
M/MOD	ud1 u1 ---	u2 ud3	36
MAX	n1 n2 ---	n3	36
MESSAGE	n ---		36
MIN	n1 n2 ---	n3	36
MINUS	n1 ---	n2	36
MOD	n1 n2 ---	n3	36
NFA	pfa ---	nfa	36
NIP	n1 n2 ---	n2	36
NMODE	---	a	36
NOOP	---		37
NUMBER	a ---	d	37
NXTDRV	n1 ---	n2	37
NXTRD	a n ---		37
NXTSTP	n ---		37
NXTWR	a n ---		37
OFFSET	---	a	37
OR	n1 n2 ---	n3	37
OUT	---	a	37
OVER	n1 n2 ---	n1 n2 n1	37

P! u b ---	38
P@ n --- b	38
PAD ---	38
PFA nfa --- pfa	38
PLACE --- a	38
PREV --- a	38
QUERY ---	38
QUIT ---	38
R --- n	38
R# --- a	38
R/W a n f ---	38
R0 --- a	39
R> --- n	39
RENAME ---	39
REPEATa1 n1 a2 n2 --- (immediate) (compile time)	39
ROT n1 n2 n3 --- n2 n3 n1	39
RP! a ---	39
RP@ --- a	39
RSHIFT n1 u --- n2	39
S->D n --- d	39
S0 --- a	39
SCR --- a	40
SELECT n ---	40
SIGN n d --- n	40
SMUDGE ---	40
SP! a ---	40
SP@ --- a	40
SPACE ---	40
SPACES n ---	40
SPAN --- a	40
STATE --- a	40
STRM --- a	40
SWAP n1 n2 --- n2 n1	41
THEN a n --- (immediate)	41
TIB --- a	41
TO n ---	41
TOGGLE a b ---	41
TRAVERSE a1 n --- a2	41
TUCK n1 n2 --- n2 n1 n2	41
TYPE a n ---	41
U. u ---	41
U< u1 u2 --- f	41
UM* u1 u2 --- ud	41
UM/MOD ud u1 --- u2 u3	42
UNTIL a n --- (immediate) (compile time)	42
UPDATE ---	42
UPPER c1 --- c2	42

USE	---	a	42	
USER	n	---	42	
VALUE	n	---	42	
VARIABLE	n	---	42	
VIDEO	---		43	
VOC-LINK	---	a	43	
VOCABULARY	---		43	
WARM	---		43	
WARNING	---	a	43	
WHILE	f	---	(immediate) (run time)	43	
WIDTH	---	a	43	
WORD	c	---	a	44
WORDS	---		44	
X	---		44	
XOR	n1	n2	---	n3.....	44
[---	(immediate)	44	
[CHAR]	---	(immediate)	(compile time)	44	
[COMPILE]	---	(immediate)	44	
\	---		44	
]	---		45	
Line Editor.....				46	
-MOVE	a	n	---	46
B	---		46	
D	n	---	46	
BCOPY	n1	n2	---	46
E	n	---	46	
H	n	---	46	
INS	n	---	46	
INVV	---		47	
L	---		47	
LINE	n	---	a	47
N	---		47	
P	n	---	47	
RE	n	---	47	
S	n	---	47	
ROOM	---		47	
TEXT	c	---	47	
TRUV	---		47	
WHERE	n1	n2	---	47
Case -Of structure.....				49	
CASE	n0	---	(immediate) (run time)	49	
OF	n0	nk	---	(immediate) (run time)	49
ENDOF	---	(immediate) (run time)		49	
ENDCASE	---	(immediate) (run time)		49	
(OF)	n0	nk	---	(run time)	49
Debugger facility.....				52	
.S	---		52	

DEPTH	---	n	52
DUMP	a	---	52
SEE	---	n	52
Error messages			54

Technical specifications

CPU Registers

This is a straight FIG-Forth Z80 implementation. Registers are used in the in the following way:

- AF – Used for normal operations.
- BC – **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'– Not used: available in fast Interrupt via EXX
- DE'– Not used: available in fast Interrupt via EXX
- HL'– Not used: available in fast Interrupt via EXX (saved at startup from Basic)
- SP – Calculator Stack Pointer
- IX – Not used: somewhere used for backup purpose
- IY – Used by ZX System

Much care has been taken to avoid any use of alternate registers (at least with interrupts enabled). This should allow users to create their own fast-response interrupt routine with EXX instead of pushing away all registers.

16 bits 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:	s b b b b b b b	b b b b b b b b

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.

32 bits Number Encoding

The second integer format requires two *integers* that form a 32 bits number said *double* or *long* that allows an integer between –2.147.483.648 and +2.147.483.647, and 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:	s b b b b b b b b b	b b b b b b b b b b	b b b b b b b b b b	b b b b b b b b b b

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)

More confusingly, in RAM it is kept as E D L H. see how 2VARIABLE is defined to understand this fact.

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

Floating-Point Number Encoding

There is a third 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.

	H	L	D	E
32 bits f.p.:	s x x x x x x x x x	x b b b b b b b b b	b b b b b b b b b b	b b b b b b b b b b

Dictionary

Legenda

In this list:

a	address: memory address	16 bits
b	byte: 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
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	

Core dictionary

'null' --- (immediate)

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

! n a ---

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

!CSP ---

It saves the value of SP register in `CSP` user variable. It is used by `:` and `;` for syntax checking.

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 / BASE` and is kept for subsequent elaborations. It is used between `<#` and `#>`. See also `#S`.

#> d --- a b

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

#BUFF --- n

This is a constant that gives the number of available buffers. This build has 3 buffers located at address between `FIRST @` and `LIMIT @`.

#S d1 --- d2 CORE

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

it 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 is raised.

In a previous version of this Forth, this word returned pfa: we changed this previous standard to return cfa.

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

Used in the form

(cccc)

it 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 an end-of-line.

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

This is the primitive compiled by **+LOOP**.

(. ") **---**

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

(;CODE) **---**

This is the primitive compiled by **;CODE**. It rewrites the cfa of **LATEST** word so that it points to the machine code starting from the following cell.

(?DO) **---**

This is the primitive compiled by **?DO**.

At compile-time compiles the cfa of **(?DO)** followed by an offset like **BRANCH** does that is used to jump after the whole **?DO ... LOOP** structure if the limit equals the initial index, otherwise it is equivalent to **(DO)**.

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

It 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

For not listed character, **c2** is equal to **c1**.

(ABORT) **---**

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

when it finds a positive sign + or no sign at all, `tf` for a negative sign -. If `a` is incremented then variable `DPL` is incremented aswell. Used by `da NUMBER` and `(EXP)` in the floatin-point option.

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

It leaves the product of two integers.

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

It executes $(n1 \cdot n2) / n3$ using an intermediate double integer to avoid precision loss.

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

It leaves the quotient `n5` and the reminder `n4` of the operation $(n1 \cdot n2) / n3$ using an intermediate double integer to avoid precision loss.

+ **n1 n2 --- n3**

It leaves the sum of two integer.

+! **n a ---**

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

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

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

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

It 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**

It gives 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.

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

It leaves $n_3 = n_1 - n_2$ as the difference from the penultimate and the last number on the stack.

It continues the interpretation in the next Screen during a `LOAD`.

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

It duplicates n if it is non zero.

Used in the form `-FIND cccc`.

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

It prints the integer `n` followed by a space.

Used in the form

At compile-time, within a colon-definition, `compile` 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)

it acts as **. " cccc "** but the string is delimited in a different way

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

Used in the form

c .C xxxx C

Acts 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** **---**

It sends line **n1** of block **n2** to the current peripheral ignoring the trailing spaces.

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

It 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**

It leaves **n3 = n1 / n2**, the quotient of the integer division.

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

It leaves the quotient **n4** and the remainder **n3** of the integer division **n1 / n2**. The remainder 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**

It leaves a **tf** if **n** is less than zero, **ff** otherwise.

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

It leaves a **tf** if **n** is not zero, **ff** otherwise. It is like a NOT **n**.

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

It 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 or 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
 It increments by one the number on TOS.

1- n1 --- n2
 It 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 ---
 It stores the double integer held on TOS to address a.

2* n1 --- n2
 It doubles the number on TOS.

2+ n1 --- n2
 It increments by two the number on TOS.

2/ n1 --- n2
 It halves the number on TOS.

2@ a --- d
a --- n-lo n-hi
 It fetches the double integer at address a. to TOS.

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

2DUP d --- d d
 It duplicates the double integer on TOS, i.e. duplicates in order the two top integer.

It copies to TOS the second double integer from top.

It 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.

It swaps the two double integers on TOS.

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

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

creates in the dictionary a new word `cccc` so that it executes the sequence of already existing words '...'.
 The sequence of words is executed in the order in which they are listed in the dictionary.

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 compiled.

It ends a colon definition and stops compilation.. It compiles ;S and execute SMUDGE to make the word findable.

Used in the form

it terminates a colon definition stoppin copilation of word `cccc` and compiling `(;CODE)`. Usually `;CODE` is followed by suitable machine code sequence..

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`.

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

It 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

It 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

It leaves a tf if n1 equals to n2, ff otherwise.

>

n1 n2

f

It 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.

>R

n

It 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

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

?COMP

It raises an error message #17 if the current STATE is not compile state.

?CSP

It 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 execution returns to the corresponding `?DO`, otherwise the two parameters are discarded and the execution continues after the `LOOP`.

The limit $n1$ and the initial value $n2$ are determined during the execution and can be the result of other previous operations. Inside a loop the word `I` copies to TOS the current value of the index.

See also: `I`, `DO`, `LOOP`, `+LOOP`, `LEAVE`. In particular `LEAVE` allows leaving the loop 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

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

It duplicates the value on TOS if it is not equal to zero. This is the same as `-DUP`.

```
?ERROR        f  n          ---
```

It raises an error message $\#n$ if f is true.

```
?EXEC          ---
```

It raises an error message $\#18$ if we aren't compiling.

```
?LOADING      ---
```

It raises an error message $\#22$ if we aren't loading. It shows the illegal use of `-->`.

```
?PAIRS        n1  n2      ---
```

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

```
?STACK        ---
```

It raises an error message $\#1$ if the stack is empty and we tried to consume an element from the calculator stack.

It raises an error message $\#7$ if the stack is full.

```
?TERMINAL      ---  f
```

It tests the keyboard. Leaves a tf if the `[BREAK]` key is pressed, ff otherwise.

```
@             a          ---  n
```

It puts on TOS the integer currently held in the cell at address a .

ABORT ---

It 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**

It leaves the absolute value of `n`.

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

It 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 `ACCEPT`.

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 source-file attached to the stream. It does not modify `SPAN`.

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

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

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 called the first time the Forth system boot to load Screen# 1. Once called it patches itself to prevent further runs.

B/BUF --- **n**

Constant that is the number of bytes per buffer. In this implementation is 512.

B/SCR --- **n**

Constant that indicates the number of Blocks per Screen. In this implementation is 1.

BACK a ---

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

BACK- [a1 n1] a n ---

It calculates 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 begin with `DO` then `a1` and `n1` aren't there.

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

It 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 the 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 true , otherwise it quits the loop.

BL --- C

Constant for “Blank”. This implementation uses ASCII and `BL` is 32.

BLANKS a n ---

It 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`.

BLOCK	n	---	a
-------	---	-----	---

It 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`.

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

It 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

It 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!

It stores a byte b to address a .

C,

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

C/L

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

C@

It puts on TOS the byte at address a .

CASEOFF

It sets case-sensitive search OFF. changes the system behavior so that (FIND) can search the dictionary ignoring case.

CASEON

It sets case-sensitive search ON. It changes the system behavior so that (FIND) will search the dictionary case sensitive.

CELL+

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

CELL-

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

text begins to the next location.

CR ---

It transmits a 0x0D to the current output peripheral.

CREATE --- a

Defining word used in the form

CREATE cccc

it creates a new dictionary entry for the definition cccc. The cfa of such a definition points to its pfa that is empty for the moment. HERE points this location.

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.

Used by : and CONSTANT.

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

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

D+- ud n --- d

It leaves d that is ud with the sign of n.

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

It 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 ---

It prints a double integer right 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

It leaves the absolute value of a double integer.

DECIMAL ---

It sets BASE to 10, that is the decimal base.

DEFINITIONS

To be used in the form

cccc DEFINITIONS

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 an **ASSEMBLER** vocabulary is available as an extra-option that can be **LOAD**ed from screens 100 - 160.

DEVICE

--- a

Variable that holds the number of current channel: 2 for video, 3 for printer, 4 for the file open to "!Blocks.bin", etc.

DIGIT

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

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

DL

--- a

User variable that keeps the data-stream number used in a **LOAD** from stream using a negative screen number.

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.

DMINUS

d1 --- d2

It leaves 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 limit **n1** and the initial value **n2** are determined during the execution and can be the result of other previous operations. Inside a loop the word **I** copies to TOS the current value of the index.

See also: **I**, **DO**, **LOOP**, **+LOOP**, **LEAVE**. In particular **LEAVE** allows leaving the loop 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 use are new vocabularies (Assembler), multidimensional array and other compiling operations.

DOSSCALL n1 n2 n3 a --- n4 n5 n6 n7

This is the ZXNEXTOS call wrapper. Number passed on stack are used as follow:

- n1 = hl input parameter value
- n2 = de input parameter value
- n3 = bc input parameter value
- a = service routine address
- n4 = hl returned value
- n5 = de returned value
- n6 = bc returned value
- n7 = error code or zero when everything is OK.

This word takes care of paging in and out RAM and ROM, and calling the specified routine.

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 ---

It drops the value on TOS. See also **OVER**, **NIP**, **TUCK**, **SWAP**, **DUP**, **ROT**.

DUP n --- n n

It 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 ---

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

EMITC b ---

It sends a byte `b` character to the current output peripheral selected with `SELECT`. See also `DEVICE`.

EMPTY-BUFFERS ---

It 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 ---

It erases `n` memory location starting from `a`, filling them with 0x00 'null' characters.

```

ERROR          b    ---  n1  n2
                ---  ff

```

It notifies an error `b` and resets the system to command prompt. First of all, the user variable `WARNING` is examined.

If WARNING is 0 then the offending word is printed followed by a “?” character and a short message “MSG#n”.

If **WARNING** is 1, instead of the short message, the text available on line **b** of block 4 (of drive 0) is displayed. Such a number can be positive or negative and lay beyond block 4.

If `WARNING` is `-1` then `ABORT` is executed, which resets the system to command prompt. The user can (with care) modify this behavior of that by altering `(ABORT)`.

If BLK is non zero, then ERROR leaves on the stack n1 that is the value of IN and n2 that is the value of BLK at the error moment. These numbers can then be used by WHERE to determine and show the exact error position.

In any case, the final action is QUIT .

If BLK is zero, then only a ff is left on TOS.

HEX --- a

It 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.

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 ---

It 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.

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.

INDEX n1 n2 ---

It prints the first line of screen between `n1` and `n2`. Handy to quick check the content of a series of screens.

INKEY

--- b

It reads the next character available from current stream and previously selected with `SELECT` leaving 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 these 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.

L/SCR

--- n

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

KEY

--- b

It 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 space
C8 >=	→	20 space
C9 <>	→	06 as CAPS-SHIFT + 2 and toggles CAPS-SHIFT On and Off,

LATEST

--- nfa

It leaves the `nfa` of the latest word defined in `CURRENT` vocabulary.

LEAVE

It forces the conclusion of a `DO . . . LOOP` setting the limit at the current index `I`, inducing an exit at the first occasion. The index remains unaltered and the execution continues normally up to the following `LOOP` or `+LOOP`.

LFA

pfa

--- lfa

It 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 ---

It prints screen number n. Sets **SCR** to n.

LIT --- n

It 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)

At 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 ---

It starts interpretation of screen n. The loading phase ends at the end of the screen or at the first occurrence of ; S.

If n is negative, instead of loading from Screen# n, it loads text directly from Basic's OPEN# n stream using **ACCEPT-**.

See also -->

LOAD- n ---

It starts interpretation of screen n. The loading phase ends at the end of the screen or at the first occurrence of ; S.

See also -->

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/ **d 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**.

M/MOD **ud1 u1** **---** **u2 ud3**

Mixed operation. Leaves the remainder **u2** and the quotient **ud3** of the unsigned integer division of a double integer **d** by the divisor **n1**.

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

It leaves the maximum between **n1** and **n2**.

MESSAGE **n** **---**

It 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**

It leaves the minimum between **n1** and **n2**.

MINUS **n1** **---** **n2**

It changes the sing of **n1**

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

It divides **n1** by **n2** and leaves the remainder **n3**. The sign is the same as **n1**.

NFA **pfa** **---** **nfa**

It converts a word's **pfa** into its **nfa**. See also **CFA**, **LFA**, **PFA**, **>BODY**, **<NAME**.

NIP **n1 n2** **---** **n2**

It 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

integers or as 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)

It 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.

NXTDRV n1 --- n2

Takes `STRM` to serve to `NEXTZXOS` call. See also `DOSCALL`.

NXTRD a n ---

Variable dedicated to `NEXTZXOS`. It calls `DOS_READ NEXTZXOS / +3e API`.

See also `DOSCALL`.

NXTSTP n ---

Variable dedicated to `NEXTZXOS`. It sets position on blocks-file calling `DOS_SET_POSITION NEXTZXOS / +3e API`.

See also `DOSCALL`.

NXTWR a n ---

Variable dedicated to `NEXTZXOS`. It calls `DOS_WRITE +3e API`. See also `DOSCALL`.

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`.

OR n1 n2 --- n3

It 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

It copies the second number from TOS and put it on the top. See also `DROP`, `NIP`, `TUCK`, `SWAP`, `DUP`, `ROT`.

P! **u** **b** ---

It 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@ **n** --- **b**

It 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 ---

It leaves on TOS the address of text output buffer. It is at a fixed distance of 68 byte over **HERE**.

PFA **nfa** --- **pfa**

It converts a word's **nfa** to its **pfa**. See also **CFA**, **LFA**, **NFA**, **>BODY**, **<NAME**.

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 ---

It 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 ---

It clears the Return-Stack, stops any compilations and return the control to the operator terminal. No message is issued.

R --- **n**

It 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.

User variable that holds the initial value of the Return Stack Pointer. See also `RP!` and `RP@`.

It removes the top value from Return Stack and put it on TOS. See also `>R`, `R` and `RP` !.

Used in the form:

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.

Used in colon definition in the form:

At run-time REPEAT does an unconditional jump 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 syntax check.

It rotates the three top integers, taking the third and putting it on top. The other two integers are pushed down from top by one place. See also `OVER`, `DROP`, `NIP`, `TUCK`, `SWAP`, `DUP`.

System procedure to initialize the Return Stack Pointer to the value passed on TOS that should be the address held in R0 user variable.

It leaves the current value of Return Stack Pointer.

It shifts right an integer `n1` by `u` bit.

It converts a 16 bit integer into a 32 bit double integer, sign is preserved.

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

SCR --- a

User variable that hold the number of the last screen retrieved with `LIST`.

SELECT n ---

It 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 --- n

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. 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`

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

It returns the content of SP register before `SP@` was executed.

SPACE ---

It sends a space to the current output peripheal, usually the video. See also `SELECT`.

SPACES n ---

It sends `n` spaces.

SPAN --- a

User variable that holds the number of characters got from the last `EXPECT`.

STATE --- a

User variable that holds the compiler 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`.

UM/MOD **ud u1 --- u2 u3**

It leaves the quotient **u3** and the remainder **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 **---**

It 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 as 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 **@**.

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

It sets `DEV\ICE` to 2 select the video as current output peripheral.

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 use that field 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

It executes a warm system restart. Executes `EMPTY-BUFFERS` and `ABORT`.

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 definition 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 consumed 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**

It reads characters from the current input stream up to a delimiter **c** and stores such string at **HERE** that is left on TOS. **WORD** leaves, as the first byte, the length of the string 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** return.

WORDS **---**

It lists the words of **CONTEXT** vocabulary. Pressing Break stops.

X **---**

It show the splash screen.

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

It executes a XOR binary operation between the two integers. The operation is performed bit by bit.

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

Used in colon definition 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 definition 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 definition in the form:

```
: cccc ... [COMPILE] wwwwww ... ;
```

[COMPILE] forces the compilation of a definition **wwwwww** 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 wwwwww** but it is necessary to use the sequence **[COMPILE] wwwwww**.

**** **---**

Used in the from:

```
\ ...
```

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

]

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

Line Editor

The following definitions are available after you give `10 LOAD`.

Line Editor is formed by 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.

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` repeat 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`, `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)	Parsing Operation		Edit Operations	One	Blanking Operations
TIB		PAD		BLOCK	
	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 .

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 n_1 to Screen n_2 . SCR will contain n_2 .

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.

INVV ---

“Inverse video”. It enables Inverse-Video attribute mode. See also TRUV.

L. ---

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

LINE	n	---	a
------	---	-----	---

It leave 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

ROOM ---

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

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.

TRUV

“True video”. It disables Inverse-Video attribute mode. See also `INVV`.

Questa definizione è disponibile solo dopo il caricamento del "Line Editor" tramite 10 LOAD.

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.

Case -Of structure

The following definitions are available after you gave 17 LOAD.

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 OBRANCH (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.

Debugger facility

The following definitions are available after Line Editor loading via 20 LOAD.

.S ---

Prints the current status of the Calculator Stack. For example, supposing to start with an empty stack,

0 1 2 3 .S

will print

0 1 2 3 ok

.WORD ---

Given a CFA, this word prints the ID. It is used by SEE to perform some word “de-compile”

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

DUMP ---

Performs a “dump” of a memory area from address *a* for 128 bytes or until BREAK is pressed.

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

448 DUMP

will print the Standard ROM content starting from address 448 (01C0h):

```
1C0 4C4953D4 4C45D450 LISTLETP
1C8 415553C5 4E4558D4 AUSENEXT
1D0 504F4BC5 5052494E POKEPRIN
1D8 D4504C4F D45255CE TPLOTTRUN
1E0 534156C5 52414E44 SAVERAND
1E8 4F4D495A C549C643 OMIZEIFC
1F0 4CD34452 41D7434C LSDRAWCL
1F8 4541D252 45545552 EARRETUR
```

SEE --- n

Used in the form

SEE cccc

it will print how the word *cccc* is defined along with NFA, CFA, PFA data. If *cccc* is a “colon-definition” the result will show

For example, the word **TYPE** is a “colon-definition” and is defined as follow:

```
: type ( a n -- )
  over + swap
  ?Do
    i c@ emit
```

```
Loop  
;
```

If you give

SEE TYPE

it will print

```
Nfa: 7153 84  
Lfa: 7158 COUNT  
Cfa: 715A 6BB0  
OVER + SWAP (?DO) 12 I C@ EMIT (LOOP) -8
```

The first line shows **TYPE**'s CFA (7153h in this case) followed by 84h, the counter byte, that indicates a 4-bytes length word name. The second line is **TYPE**'s LFA (7158h) that is a pointer to **COUNT**'s NFA, that is the previous word defined in the dictionary. The third line is **TYPE**'s CFA: this is a pointer to the ;CODE machine code of : that it's the entry-point of every "colon-definition". The fourth line represents the PFA and is somehow the "decompilation" of the above definition.

Another example the word **CODE** isn't a colon-definition, instead it is coded directly in machine-code using the ASSEMBLER vocabulary available on Screens 100-160, as follow:

```
CODE nip ( n1 n2 -- n2 )  
POP HL  
EX(SP)HL  
Next  
C;
```

and if you give

SEE NIP

it will print

```
Nfa: 69D8 83  
Lfa: 69DC DROP  
Cfa: 69DE 69E0  
69E0 E1E3C334 64845455 acC4d TU  
69E8 43CBD869 EE69E1D1 CKXiniaQ  
...
```

In this case, since **NIP** is not a colon-definition, the PFA part is just a **DUMP** call you can **BREAK** at any time.

Again, the first line shows **NIP**'s CFA (69D8h in this case) followed by 83h, the counter byte, that indicates a 3-bytes length word name. The second line is **NIP**'s LFA (69DCh) that is a pointer to **DROP**'s NFA, that is the previous word defined in the dictionary. The third line is **NIP**'s CFA that points to the following cell address 69E0h, that is **NIP**'s PFA where the small piece of machine-code lies. We should be able to see **E1** for POP HL, **E3** for EX (SP),HL and **C3** for JP to "Next" inner interpreter address **6434h**.

The following bytes 84 45 55 are the beginning of the next word in dictionary (**TUCK** in this case).

Error messages.

Code	Message
#0	?
#1	Stack is 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.
#45	NextZXOS pos error.
#46	NextZXOS read error.
#47	NextZXOS write error.