

# **Retro**

The Language and Its Implementation

By Charles Childers

**Released into the public domain.  
Share Freely.**

# **Table of Contents**

Introduction **4**

The Virtual Machine **5**

Device I/O **6**

The Forth Language **8**

Implementation Details **22**

Cross Compiler **33**

Ngaro Instruction Set **44**

List of Abbreviations **50**

## **A Word On Copyright**

I have no problem with copyrights. I believe that the author of a work should be able to enjoy exclusive rights for a limited period of time.

That said, I also believe that the copyright situation in the world today is sad. The concept of a limited period of protection is being lost. In the world today, it's not even guaranteed that an author can voluntarily relinquish his/her rights and gift a work to the public domain.

Through the years, most implementations of Retro have been released as public domain. Keeping with that spirit of freedom, I am releasing the code for the current Retro, and all the corresponding documentation I write into the public domain.

Should my right to release my copyright on Retro ever be shown to be invalid, I will release Retro under the most liberal open source license I can find. I will never restrict the use of Retro or derivatives thereof.

Charles Childers

# Introduction

Welcome to Retro!

This is a small implementation of the Forth language running on a very portable virtual machine. It is minimalist in nature, yet has a well balanced set of core functionality.

While this implementation is new, there is a lot of history behind the Retro name. It was originally developed in 1998 as a 16-bit bootable Forth, evolved into a 32-bit protected mode Forth that served as a prototype testbed for the Tunes project, and eventually picked up some ideas from colorForth before being rewritten to run on traditional operating systems. Along the way it influenced the development of many other Forth systems.

Now, ten years after it was first conceived, the Retro language can be used on most modern desktops, servers, and a growing number of mobile devices.

# The Virtual Machine

Retro runs on a virtual machine called Ngaro. This allows it to support any system capable of running an implementation of the Ngaro.

At present, there are implementations in C, JavaScript, and Java. Retro has been tested and confirmed to work on Linux, Mac OS X, Windows, and BeOS, and on x86 and Alpha hardware. In addition, it can be run in most modern web browsers including Internet Explorer 7, FireFox, Safari, and Opera.

The Ngaro VM emulates a MISC/NOSC architecture. This allows a small set of instructions with trivial encodings, simplifying the Forth implementation and development tools.

Along with a simple processor, the Ngaro VM also emulates a variety of basic hardware devices. Most implementations provide either a text console or graphical framebuffer for output, and a keyboard for input. Some also allow for saving the memory contents to disk. Other devices may be emulated, but are considered non-standard at this time.

# Device I/O

Ngaro allows communication with devices through a system of I/O ports.

## Port 0 - Wait for Hardware Event

This is used to determine if Ngaro should enter a *wait for hardware event* loop. It should be set to 0, then use the wait instruction. In Retro, this is all done by the **wait** word.

## Port 1 - Read from the Keyboard

To read a value from the keyboard, set port 0, wait, then read the key value from port 1.

## Port 2 - Character Generator

Ngaro provides a hardware character generator. This takes data off the stack, so you should make sure the proper values are on the stack before using it. For text consoles, you need to leave the character code on the stack. For framebuffer use, leave the x, y coords and the character code on the stack.

## **Port 3 - Force Video Update**

Ngaro implementations may cache output and only update the display periodically. If they do, then they should respond to this by immediately updating the display. You do not need to wait after setting this port.

## **Port 4 - Save Image**

If your Ngaro implementation allows saving images, you can use this port to do so. To save, set port 4 to 1 and wait.

## **Port 5 - Hardware Capability Detection**

To use this port, pass it a value from the list below, wait, then read back from it to obtain the results of your query. The following queries are currently supported on the stable Ngaro implementations:

- 1 Amount of memory being provided
- 2 Address of framebuffer (0 if none)
- 3 Width of framebuffer
- 4 Height of framebuffer
- 5 Depth of data stack
- 6 Depth of address (return) stack

# The Forth Language

Forth is a rich family of languages sharing common ideals, and in some cases ancestry. Despite this, no two dialects are quite the same, and some (like Retro) are significantly different than others.

The dialect provided by Retro has evolved over the last ten years through constant use, experiments, and pruning. Over time the words and their meanings change, but the list below describes the core language as it stands in the current generation of Retro.

The format given for each word is:

wordname	data stk		address stk	
description				

here	--	a		--
------	----	---	--	----

Return the address of the top of the heap

,		n	--		--
---	--	---	----	--	----

Write a cell to the top of the heap;  
increase heap pointer.



] -- || --

Turn compiler on

create "name" -- || --

Create a new dictionary entry with a  
class of data.

: "name" -- || --

Create a new dictionary entry with a  
class of word and call ].

macro "name" -- || --

Create a new dictionary entry with a  
class of macro and call ].

cr -- || --

Move the text cursor to the start of the  
next line

emit n -- || --

Display a character

type \$ptr -- || --

Display a string

words -- || --

Display a list of all named words

save            --            || --  
Save the image

clear           --            || --  
Clear the display

key             -- n           || --  
Read a value from the keyboard

accept          delim --      || --  
Read a string delimited by delim into the  
tib.

dup             n -- n n      || --  
Duplicates TOS

1+              x --y        || --  
Increment TOS

1-              x -- y        || --  
Decrement TOS

swap            x y -- y x    || --  
Exchange TOS and NOS

drop            x y -- x      || --  
Drop TOS

and                    x y -- z       || --  
Bitwise AND

or                    x y -- z       || --  
Bitwise OR

xor                   x y -- z       || --  
Bitwise XOR

@                    a -- n       || --  
Fetch a value from address

!                    n a --       || --  
Store a value to address

+                    x y -- z       || --  
Add two numbers

-                    x y -- z       || --  
Subtract two numbers

\*                    x y -- z       || --  
Multiply two numbers

/mod                   x y -- z q || --  
Divide and get remainder

<<            x y -- z       || --  
Bitwise left shift

>>            x y -- z       || --  
Bitwise right shift

nip            x y -- y       || --  
Drop NOS

over           x y -- x y x       || --  
Put a copy of NOS on top of the  
stack  
2drop           x y --       || --  
Drop TOS and NOS

not            x -- y       || --  
Same as -1 xor.

rot    x y z -- y z x       || --  
Rotate top three items on the stack

-rot   x y z -- z x y       || --  
Rotate top three items on stack  
twice

tuck   x y -- y x y       || --  
Put a copy of TOS under NOS

2dup x y -- x y x y || --  
Duplicate both TOS and NOS

on a -- || --  
Set an address to -1

off a -- || --  
Set an address to 0

/ x y -- z || --  
Divide two numbers

mod x y -- z || --  
Divide and get remainder only

neg x -- y || --  
Invert the sign of a number

execute a -- || -- a  
Call a function at address

. n -- || --  
Display a number

" "string" -- a ||--  
Parse for a string

compare        \$1 \$2 -- f    || --  
Compare two strings

in              p -- n        || --  
Read a value from a port

out             n p --        || --  
Write a value to a port

wait            --            || --  
Wait for a hardware event  
'        "name" -- a        || --  
Get the address of a named word

@+        a -- a+1 n        || --  
Fetch a value from address a, return the  
value n and increment the address

!+        n a -- a+1        || --  
Store n into address a, increment the  
address

+!        n a --            || --  
Add n to the value at address a, storing  
the results in a.

-!      n a --                    || --

Subtract n from the value at address a,  
storing the results in a.

:is      a xt --                    || --

Assign vector at xt to address a

:devector      xt --                    || --

Remove the vector from xt

is      a "name" --                    || --

Set vector name to address a.

devector      "name" --                    || --

Restore a vector to its default state.

compile      a --                    || --

Compile a call to address

literal,      n --                    || --

Compile a value as a literal

redraw      --                    || --

Update the display

getLength      \$ptr -- n                    || --

Get the length of a string

```
tempString $1 -- $2    || --  
Mark a string as temporary
```

```
keepString $1 -- $2    || --  
Mark a string as permanent
```

```
bye            --          || --  
Shutdown Ngaro VM, exit Retro
```

```
(remap-keys)      n -- n      || --  
Vector which allows replacing one key  
value with another.
```

```
.word          a --          || --  
Class handler for normal words.  
(Class id = 1)
```

```
.macro         a --          || --  
Class handler for macros.  
(Class id = 2)
```

```
.data          a -- a        || --  
Class handler for data structures.  
(Class id = 3)
```



.inline        a --            || --  
Class handler for inline words  
(Class id = 4)

with-class    a n --           || --  
Class dispatcher. N is the class id.

boot                --            || --  
Hook for custom startup code for turnkey  
applications.

depth            -- n            || --  
Return the current depth of the data  
stack

reset            ... --           || --  
Remove all items on the data stack

notfound        --            || --  
Handler called when a word is not found

s"        "string" -- a           || --  
Compile a string into a definition

[                --            || --  
Turn the compiler off

;                    --                    || --  
End a definition

::                    --                    || --  
Compile an exit point into a definition

=if                    x y --                    || --  
Compare two number for equality. Jump to  
then if condition not met.

>if                    x y --                    || --  
Compare two number for greater than. Jump  
to then if condition not met.

<if                    x y --                    || --  
Compare two number for less than. Jump to  
then if condition not met.

!if                    x y --                    || --  
Compare two number for inequality. Jump  
to then if condition not met.

then                    --                    || --  
End a conditional

repeat            --            || --  
Begin an unconditional loop

again            --            || --  
Close an unconditional loop. Branch back  
to previous repeat.

0;                n --            || --  
                  n -- n        || --  
Exit a word if TOS is 0. If 0, drops TOS.  
Otherwise leaves TOS alone.

(        "... )" --            || --  
Start a comment.

push            n --            || -- n  
Push TOS to TORS.

pop              -- n            || n --  
Pop TORS back to TOS

for              n --            || -- a  
Start a counted loop

```
next          -- n          || a --  
              --           || a --
```

End a counted loop. Decreases counter by 1. If counter is 0, exit the loop. Otherwise jump back to for. Drops the counter upon exiting the loop.

```
[']  "name" -- a          || --  
Get the address of a word.
```

```
d->class      d -- a       || --  
Dictionary: Get class slot
```

```
d->xt         d -- a       || --  
Dictionary: Get XT slot
```

```
d->name       d -- a       || --  
Dictionary: Get name slot
```

```
tx           -- a         || --  
Text X coordinate (graphics mode)
```

```
ty           -- a         || --  
Text Y coordinate (graphics mode)
```

```
last         -- a         || --  
Pointer to the most recent dictionary  
entry
```

compiler       -- a           || --  
Holds compiler state

tib            -- a           || --  
Pointer to the text input buffer

update        -- a           || --  
Variable telling Retro to force a screen  
update.

fb             -- a           || --  
Address of framebuffer

fw             -- a           || --  
Width of framebuffer

fh             -- a           || --  
Height of framebuffer

#mem           -- a           || --  
Amount of memory

# Implementation Details

It is possible to write programs in Retro without worrying the implementation choices that were made, but knowing how and why things were done will be a help if you ever need to work on the Retro core.

## Threading Model

Retro uses subroutine threading with inline machine code for some words. This was chosen primarily due to its simplicity, but also for historical reasons. (All Retro implementations since 2001 have been primarily subroutine threaded).

The subroutine threading model compiles code to native machine code, primarily as calls to other routines.

As an example:

```
: foo 1 2 + . ;
```

This will compile to:

```
lit 1  
lit 2  
call +  
call .  
;
```

There are performance issues with this (calls can be expensive), and it's not the most compact, but it does allow a lot of opportunity for optimization. Recent releases of Retro support inline machine code generation for primitives, so the above example can compile to:

```
lit 1  
lit 2  
+  
call .  
;
```

This saves a call/return operation, allowing a small, but measurable gain in performance. The compiled code is also smaller overall.

## Word Classes

Retro's interpreter makes use of an implementation technique known as *word classes*. This approach was created by Helmar Wodtke and allows for a very clean interpreter and compiler. It makes use of special words, called *class handlers*, to process execution tokens. Each word in the dictionary has a class handler associated with it. When being executed, the address of the word is pushed to the stack and the class handler is invoked. The handler then does something with the address based on various bits of state.

The standard Retro language comes with four classes.

```
.word          a --      || --  
If interpreting, call the word. If  
compiling, compile a call to the word.
```

```
.macro          a --      || --  
Always call the word. Normally used for  
words that lay down custom code at  
compile time or which have different  
compile and interpret time behaviors.
```



`.inline`                    `a --`                    `|| --`  
If interpreting, call the word. If  
compiling, copy the first opcode of the  
word into the target definition. Only  
intended for use with words that map  
directly to processor opcodes.

`.data`                    `a --`                    `|| --`  
If interpreting, leave the address on the  
stack. If compiling, compile the address  
into the target definition as a literal.

Each dictionary entry has a field that points to the  
address of its corresponding class handler.

It is possible to define custom classes. The easiest  
way to show how to add a new class is with an  
example. For this, we'll create a class for strings with  
the following behavior:

- \* If interpreting, display the string
- \* If compiling, compile the code  
needed to display the string

Retro has a convention of using a `.` as the first  
character of a class name. We'll call our new  
class `.string`

On entry to a class handler, the address (*xt*) of the word or data element is on the stack. The compiler state (often important to class handlers) is stored in a variable named `compiler`. With this in mind, we'll define our class:

```
: .string ( xt -- )  
  compiler @ 0 =if type ;; then  
  1 , , [''] type compile ;
```

The compile-time portion is a bit trickier than the interpret time since it has to lay down the proper code in the target definition. In this case a LIT instruction (opcode 1) is laid down, followed by the *xt* of the string. This is followed by code to lay down a call to `type`.

We now need a creator word to attach this class to a value. For this example we'll define a word named `displayString`: to take a form like the one shown below.

```
string-addr displayString: name
```

New dictionary entries are made using `create`, so we'll use that and change the class to our new `.string` handler.

```
: displayString: ( string-addr -- )  
  create ['] .string last @ d->class !  
  keepString last @ d->xt ! ;
```

This uses create to make a new word, then sets the class to .string and the xt of the word to the string. It also makes the string permanent using keepString. last is a variable pointing to the most recently created dictionary entry. The two words d->class and d->xt are as dictionary field accessors and are used to provide portable access to fields in the dictionary.

We can now test the new class:

```
" hello, world!" displayString: hello  
hello  
: foo hello cr ; foo
```

You can use this approach to define as many classes as you want.

## Vectors

Vectors are another important concept in Retro. Most Forth systems provide a way to define a word which can have its meaning altered later. Retro goes a step further by allowing all words defined using `:` or `macro:` to be redefined. Words which can be redefined are called *vectors*.

Vectors can be replaced by using `is`, or returned to their original definition with `devector`. For instance:

```
: foo 23 . ;  
foo  
: bar 99 . ; ' bar is foo  
foo  
devector foo  
foo
```

There are also variations of `is` and `devector` which take the addresses of the words rather than parsing for the word name. These are `:is` and `:devector`.

Becoming familiar with manipulating vectors will allow you a much greater degree of control over the Retro system.

## Dictionary Structure

The dictionary is a simple linked list. The structure of each node in this list is shown in the table below.

Location	Description
0	Holds the address of the previous entry. Set to 0 if there are no previous entries.
1	Holds the class identifier of the word.
2	Holds the address that the definition starts at.
3 .... n	String holding the name of the word. One character per cell, ending with a numeric value of 0.

New entries are made by create. The words : and macro: use create and then modify the class field as necessary.

## Parser

Retro's parser is designed to be very simple. It reads keys, adding them to the input buffer (*tib*), until a given delimiter is reached. It then returns control to the calling word. The parser is made available via the accept word.

The immediate nature of the parser means that Retro operates on input directly. Unlike most Forth implementations, interpretation and execution is not delayed until a full line is entered. Words are executed when the spacebar is pressed.

If you need to support other forms of whitespace, you will need to revector (`remap-keys`) with a new definition that replaces those forms with the space character.

## Strings

Strings in Retro are zero terminated. This is an unconventional approach for a Forth dialect, but simplifies some operations and allows a consistent approach to dealing with strings.

## The Interpretation Process

Retro has a simple interpreter. Technically speaking, there is no actual compiler. Instead, each class is assumed to be able to handle either execution or compilation of associated words or data. This shifts some bits of complexity out to the classes and allows a very streamlined interpreter loop.

The interpreter calls `accept`, passing it the `ascii` value 32 (for space) as a delimiter. Input is accepted and added to the `tib` until the delimiter is encountered.

At this point, the interpreter cycles through the dictionary, comparing the input in `tib` to the name of each entry. This loop goes from the newest to the oldest entry, and exits when a match is found.

If a match was found, the `xt` of the word is pushed to the stack and the class handler attached to its dictionary entry is called. This handler is responsible for handling the `xt` and carrying out the proper behavior for the word.

If no match was found, the interpreter attempts to convert the value to an integer. If successful, the integer value is pushed to the stack and the `.data` class handler is called.

If a conversion to integer failed, and no match was found, the interpreter calls `not found` to report the error.

This process is then repeated until Retro is shut down.



# Cross Compiler

Retro is written in a language that resembles Forth. This is actually a machine forth dialect provided by the cross compiler. The cross compiler is a piece of code providing an assembler for the Ngaro instruction set, some helper functions allowing creation of the initial dictionary and a more Forth-like approach to coding, and basic bug checking.

There is a standard form for applications written using the cross compiler:

```
begin imagename
  ... support code + data ...
main:
  ... main entry point ...
end
```

When compiling, the resulting image will be saved as *imagename*, and the code will begin with a jump to *main*:

It should be noted here that code must build on previous code; there is no support for forward references in the cross compiler.

The cross compiler has some additional words that make development of Retro easier. These setup the initial dictionary, words to call upon startup, and perform some sanity checks to help catch bugs. When using these, the form changes to:

```
begin imagename
  mark-dictionary variable last
  mark-init-chain variable initchain

  ... support code + data ...

  address word: retroname
  address macro: retroname
  address data: retroname
  address inline: retroname

  address +init

  patch-init-chain
  patch-dictionary

main:
  ... main entry point ...
cross-summary
end
```

In this, note `mark-dictionary`, `word:`, `macro:`, `data:`, `inline:`, and `patch-dictionary`. These are used to build the initial dictionary. The `word:`, `macro:`, `data:`, and `inline:` each take an address from the stack and parse for a name to add to the dictionary.

The use of `mark-init-chain`, `+init`, and `patch-init-chain` is used by Retro to support creation of a list of addresses to execute upon startup. This allows for modules to replace core functionality when the Retro image is loaded.

`cross-summary` displays some statistics and also performs a check to ensure that the stack is balanced to 0 entries. The cross compiler assumes that the stack will start and end empty; anything else implies bugs in the code being compiled and will be reported as such.

The words provided by the cross compiler follow. Many of them are similar to the core words in Retro, but be aware that some with the same name may have significant differences.

heap            -- a  
Pointer into the target memory

origin         -- a  
Address of the target memory

fid            -- n  
File ID

log            -- n  
Memory Map / Build Log File ID

lastop         -- n  
Last compiled opcode (for optimizer)

\$log           "string" --  
Append "string" to the map/log file

\n             --  
Append a newline to the map/log file

,              n --  
Compile a value into the target memory

vm:            n "name" --  
Create a new word that compiles its value  
into target memory

here                   -- a  
Return the value of the current location  
in the target memory.

nop,	--	Ngaro Instruction
lit,	--	Ngaro Instruction
dup,	--	Ngaro Instruction
drop,	--	Ngaro Instruction
swap,	--	Ngaro Instruction
push,	--	Ngaro Instruction
pop,	--	Ngaro Instruction
call,	--	Ngaro Instruction
jump,	--	Ngaro Instruction
;	--	Ngaro Instruction
>jump,	--	Ngaro Instruction
<jump,	--	Ngaro Instruction
!jump,	--	Ngaro Instruction
=jump,	--	Ngaro Instruction
@,	--	Ngaro Instruction
!,	--	Ngaro Instruction
+,	--	Ngaro Instruction
-,	--	Ngaro Instruction
*,	--	Ngaro Instruction
/mod,	--	Ngaro Instruction
and,	--	Ngaro Instruction
or,	--	Ngaro Instruction
xor,	--	Ngaro Instruction

```

<<,          --      Ngaro Instruction
>>,          --      Ngaro Instruction
0;           --      Ngaro Instruction
1+,          --      Ngaro Instruction
1-,          --      Ngaro Instruction
in,          --      Ngaro Instruction
out,         --      Ngaro Instruction
wait,        --      Ngaro Instruction

halt,        --
Bogus instruction which causes Ngaro to
cease execution of the image.

begin        "name" --
Begin compilation of target image

end          --
End compilation of target image

main:        --
Main entry point in target

label:       "label" --
Create a label

```

#                    n --  
Compile a number into the image (lit,  
followed by a value)

\$,                    string-ptr --  
Compile a string into the image

conditional            -- a  
Common code for conditionals

=if                    --        Cond. (equality)  
<if                    --        Cond. (less than)  
>if                    --        Cond. (greater than)  
!if                    --        Cond. (inequality)

then                    a --  
Close the previous conditional

:                    "name" --  
Begin a new word in the target image

'                    "name" --  
Get the address of a word in the target  
image. Compiles the address into the  
target image.

link            -- n  
Link to the latest entry in the target  
dictionary

#entries        -- n  
Number of entries in the target  
dictionary

word:           a "name" --  
Create a new word in the target  
dictionary using the word class

macro:          a "name" --  
Create a new word in the target  
dictionary using the macro class

data:           a "name" --  
Create a new word in the target  
dictionary using the data class

inline:         a "name" --  
Create a new word in the target  
dictionary using the inline class

repeat          -- a  
Begin an unconditional loop



again            a --  
End an unconditional loop

variable:    n "name" --  
Create a variable with an initial value  
of n. This is recommended for use inside  
modules.

variable       "name" --  
Create a variable with an initial value  
of 0. Not safe to use outside the core.

patch-dictionary --  
Adjust the marked memory location to  
point to the most recent dictionary  
entry. This should come after all of the  
initial entries are created.

mark-dictionary --  
Mark the current location in the target  
image as the pointer to the most recent  
dictionary entry.

init-link            -- n  
Link to the latest entry in the  
initialization chain

mark-init-chain --

Mark the current location in the target image as the pointer to the most recent item in the initialization chain.

patch-init-chain --

Adjust the marked memory location to point to the most recent entry in the initialization chain. This should come after all items have been added to the chain.

+init a --

Add the word at address a to the initialization chain

TCE -- n

Number of tail calls eliminated

+TCE --

Increment the number of tail calls eliminated

;

Extend the ; instruction to eliminate tail calls.

cross-summary       --

Display a summary of the cross-compilation results. This does some very basic sanity checks (stack depth, etc)

MODULE:       "name" --

Specify the name of a module

AUTHOR:       "name" --

Specify the author of a module

NOTES:        "text" --

Provide notes about a module

# Ngaro Instruction Set

The Ngaro processor has 31 instructions. The instructions are listed below, using the following format. All opcodes are listed in decimal.

Opcode	Name
Data Stack	Address Stack
Description	
0	NOP
--	--
Does nothing. Used for padding	
1  value	LIT
-- n	--
Push a literal to the stack	
2	DUP
n -- n n	--
Duplicate the top value on the stack	
3	DROP
n --	--
Drop the top value off the stack	



10 |address| >JUMP

--

--

Conditional branch, if NOS is greater  
than TOS

11 |address| <JUMP

--

--

Conditional branch, if NOS is less than  
TOS

12 |address| !JUMP

--

--

Conditional branch, if NOS is not equal  
to TOS

13 |address| =JUMP

--

--

Conditional branch, if NOS is equal to  
TOS

14 @

a -- n

--

Fetch the value at the memory address in  
TOS

15                                   !  
n a --                               --

Store the value on the second stack  
location to the address in T0S

16                                   +  
x y -- z                           --

Add the top two values on the stack

17                                   -  
x y -- z                           --

Subtract the top two values on the stack

18                                   \*  
x y -- z                           --

Multiply the top two values on the  
stack

19                                   /MOD  
x y -- z q                       --

Divide and get the remainder of the top  
two values on the stack

20                                   AND  
x y -- z                           --

Bitwise AND operation

21 OR

x y -- z --

Bitwise OR operation

22 XOR

x y -- z --

Bitwise XOR operation

23 <<

x y -- z --

Shift bits left

24 >>

x y -- z --

Shift bits right

25 0;

n -- n --

n -- --

Exit a subroutine and drop TOS if TOS is 0. If TOS is not 0, do nothing.

26 1+

x -- y --

Increase the value on the stack by 1



```

27          1-
x -- y      --
Decrease the value on the stack by 1

```

```
28          IN
p -- n      --
Read a value from a port
```

```
29          OUT
n p --      --
Send a value to a port
```

```
30          WAIT
--          --
Wait for the hardware to process an event
```

# List of Abbreviations

MISC	Minimal Instruction Set Computer
NOS	Next on Stack
NORS	Next on Return Stack
NOSC	No Operand Stack Computer
TIB	Text Input Buffer
TOS	Top Of Stack
TORS	Top of Return Stack
VM	Virtual Machine
XT	Execution Token, a word's address