



The Retro Language

Charles Childers

Published: 2008

Tag(s): Reference Manual Handbook

Part 1

Introduction

Chapter 1

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.

Chapter 2

Setting Up

There are two ways to get started with Retro. For most people, the easiest option is to download the binary package, which includes executables for most common operating systems.

If you choose to go with the binary package, copy the **retro** binary and the **retroImage** to a directory. Depending on your OS, you may need to check for library dependencies (the binaries generally require *ncurses* to be installed), but it should run on most systems without problems.

For those who want or need to build things themselves, follow the appropriate set of instructions and you shouldn't have any problems.

1. Building Retro (Linux, BSD, Mac OS X, BeOS)

If you're running a Unix-like OS, building is pretty easy. Make sure you have the prerequisites listed below, then run the following command:

make

Prerequisites:

- GCC
- Make
- ncurses library

After running *make*, you should have several new files in your top-level directory. Two of these are needed to run Retro:

- *retro*
- *retroImage*

Create a new working directory, and copy these files to it. Now you're all set!

2. Building Retro (Windows)

Unfortunately Retro is significantly more difficult to build on a Windows system. The following steps work for building the **retro.exe** executable.

- Install Dev C++ (<http://prdownloads.sourceforge.net/dev-cpp/devcpp-4.9.9.2-setup.exe>)
- Install the Curses Devpack (<http://devpaks.org/details.php?devpak=5>)
- Create an empty project and add the following files from *retro-dev-kit* to it:
 functions.h
 disassemble.c
 endian.c
 loader.c
 ngaro.c
 devices curses.c
 vm.c
 vm.h
- Go to Project -> Options and add the following to the Linker commands:
 -lcurses
- Compile everything

There is no way to build a **retroImage** on Windows at this time. You can get one from <http://build.retroforth.org>

Copy your newly created **retro.exe** and **retroImage** to a directory and you're set.

Chapter 3

Using Retro

1. Starting Retro

To start Retro, run the **retro** executable.

Retro has some differences from traditional Forths. The first is that input is processed as it is typed, with space being the only recognized whitespace character. So to get started, let's try getting a list of words provided by Retro. Type **words** and hit space. You should see a list of word names. These are covered in Part 2.

Tip: If you mess up, type some random characters and hit the space bar. Retro will report a word-not-found condition, and continue to handle input without problems.

Tip: All word names in the standard Retro image are lowercase. Word names are case sensitive.

2. Saving Your Image

Since Retro is divided into a virtual machine and an image file, it is possible to save your current image. The save process does not preserve the stack or internal registers, but all code and data that is stored in memory will be present when you next start Retro. To save the image, just type **save** and hit space.

3. Leaving Retro

To exit Retro, enter **bye** and hit space.

Certain errors may also cause Retro to exit.

4. Loading Words from Files

Retro allows loading of files through a command line parameter.

./retro —with filename

You can specify several files, but each one needs to be preceded by **—with**.

Tip: If you specify multiple files, the last one you specify will be the first one processed by Retro.

Part 2

The Language

Chapter 1

The Words

1. Reading the List

The documentation for the words is pretty sparse, but basic details are provided for each one. Each word takes on the following format:

- wordname
data stack effect
address stack effect
Description of the word

Stack effects are given in a form like this: before — after

2. Words in the Core

The following words are provided by the standard retroImage.

- 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
 $x\ y \rightarrow y\ x$
—
Exchange TOS and NOS
- drop
 $x\ y \rightarrow x$
—
Drop TOS
- and
 $x\ y \rightarrow z$
—
Bitwise AND
- or
 $x\ y \rightarrow z$
—
Bitwise OR
- xor
 $x\ y \rightarrow z$
—
Bitwise XOR
- @
 $a \rightarrow n$
—
Fetch a value from address
- !
 $n\ a \rightarrow$
—
Store a value to address
- +
 $x\ y \rightarrow z$
—
Add two numbers
- -
 $x\ y \rightarrow z$
—
Subtract two numbers
- *
 $x\ y \rightarrow z$
—
Multiply two numbers
- /mod
 $x\ y \rightarrow z\ q$
—
Divide and get remainder
- <<
 $x\ y \rightarrow z$
—
Bitwise left shift

- >>
 $x\ y \text{ --- } z$
—
Bitwise right shift
- nip
 $x\ y \text{ --- } y$
—
Drop NOS
- over
 $x\ y \text{ --- } x\ y\ x$
—
Put a copy of NOS on top of the stack
- 2drop
 $x\ y \text{ ---}$
—
Drop TOS and NOS
- not
 $x \text{ --- } y$
—
Same as -1 xor.
- rot
 $x\ y\ z \text{ --- } y\ z\ x$
—
Rotate top three items on the stack
- -rot
 $x\ y\ z \text{ --- } z\ x\ y$
—
Rotate top three items on stack twice
- tuck
 $x\ y \text{ --- } y\ x\ y$
—
Put a copy of TOS under NOS
- 2dup
 $x\ y \text{ --- } x\ y\ x\ y$
—
Duplicate both TOS and NOS
- on
 $a \text{ ---}$
—
Set an address to -1
- off
 $a \text{ ---}$
—
Set an address to 0
- /
 $x\ y \text{ --- } z$
—
Divide two numbers

- mod
 $x\ y \text{ --- } z$
—
Divide and get remainder only
- neg
 $x \text{ --- } y$
—
Invert the sign of a number
- execute
 $a \text{ ---}$
— a
Call a function at address
- .
 $n \text{ ---}$
—
Display a number
- "
"string" — a
—
Parse for a string
- compare
 $1\ 2 \text{ --- } f$
—
Compare two strings
- in
 $p \text{ --- } n$
—
Read a value from a port
- out
 $n\ p \text{ ---}$
—
Write a value to a port
- wait
—
—
Wait for a hardware event
- '
"name" — a
—
Get the address of a named word
- @+
 $a \text{ --- } a+1\ n$
—
Fetch a value from address a , return the value n and increment the address
- !+
 $n\ a \text{ --- } a+1$
—
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.
- 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

Part 3

Implementation

Chapter 1

The Virtual Machine

1. Overview

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.

Ngaro 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, Ngaro 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. Most also allow for saving the memory contents to disk. Other devices may be emulated, but are considered non-standard at this time.

2. Image Files

The VM works with image files, which are binary snapshots of the emulated memory. This allows for some nice things, including saving the current memory image, and reloading it later.

The current implementation of the image file does not save any of the internal registers or stack contents, so each time an image is loaded, the image contents need to be able to start fresh.

Image files are used with all implementations of the VM, but saving images does not work in the JavaScript implementation, and saving images under Windows is problematic.

3. I/O Devices

Ngaro allows communication with the emulated 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.

Displaying a negative character will clear the screen.

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

4. Instruction Set

The MISC processor has 31 instructions. These are listed below, using the following format.

Tip: All opcodes are listed in decimal.

- Opcode # / Name
Data Stack
Address Stack
Description

-
- **0 NOP**
—
—
Does nothing. Mostly 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
- **4 SWAP**
x y — y x
—
Exchange the top and second item on the stack
- **5 PUSH**
n —
— n
Push the top item on the data stack to the address stack
- **6 POP**
— n
n —
Pop the top item on the address stack to the data stack
- **7 |address| CALL**
—
— a
Call a subroutine
- **8 |address| JUMP**
—
—
Branch unconditionally to a memory location
- **9**
—
a —
Return from a subroutine
- **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 TOS
- **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;**
 $n \leftarrow n \text{ OR } \text{---} n$

Exit a subroutine and drop TOS if TOS is 0. If TOS is not 0, do nothing.
- **26 1+**
 $x \leftarrow y$

Increase the value on the stack by 1
- **27 1-**
 $x \leftarrow y$

Decrease the value on the stack by 1
- **28 IN**
 $p \leftarrow n$

Read a value from a port
- **29 OUT**
 $n \leftarrow p$

Send a value to a port
- **30 WAIT**

Wait for the hardware to process an event

Chapter 2

The Internals

1. The Interpreter

Retro has a simple interpreter. 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 a match is not found, the interpreter tries to convert the token 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 **notfound** to report the error.

This process is then repeated until Retro is shut down.

Tip: Retro has no separate compiler loop. Instead, each class handler is responsible for compiling the code for words associated with it. This allows a very straightforward interpreter loop.

2. 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 has four classes defined.

- **.forth**
a —
—
If interpreting, call the word. If compiling, compile a call to the word.
- **.macro**
a —

-
- Always call the word. This is normally used for words that lay down custom code at compile time, or which need to have different behaviors during compilation.
 - `.inline`
a —
—
If interpreting, call the word. If compiling, copy the first opcode of the word into the target definition. This is only useful for use with words that map directly to processor opcodes.
 - `.data`
a — a
—
If interpreting, leave the address on the stack. If compiling, compile the address into the target definition as a literal.

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, lay down the code needed to display the string

Retro has a convention of using a `.` as the first character of a class name. In continuing this tradition, we'll call our new class **`.string`**

*Tip: On entry to a class, the address of the word or data structure is on the stack. The compiler state (which most classes will need to check) is in a variable named **`compiler`**.*

A first step is to lay down a simple skeleton. Since we need to lay down custom code at compile time, the class handler will have two parts.

```
: .string ( a — ) compiler @ 0 =if ( interpret time ) ; then ( compile time )
```

We'll start with the interpret time action. We can replace this with **`type`**, since the whole point of this class is to display a string object.

```
: .string ( a — ) compiler @ 0 =if type ; then ( compile time )
```

The compile time action is more complex. We need to lay down the machine code to leave the address of the string on the stack when the word is run, and then compile a call to **`type`**. If you look at the instruction set listing, you'll see that *opcode 1* is the instruction for putting values on the stack. This opcode takes a value from the following memory location and puts it on the stack. So the first part of the compile time action is:

```
: .string ( a — ) compiler @ 0 =if type ; then 1 , , ;
```

Tip: Use `,` to place values directly into memory. This is the cornerstone of the entire compiler.

One more thing remains. We still have to compile a call to **type**. We can do this by passing the address of **type** to **compile**.

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

And now we have a new class handler. The second part is to make this useful. We'll make a creator word called **displayString**: to take a string and make it into a new word using our **.string** class. This will take a string from the stack, make it permanent, and give it a name.

*Tip: New dictionary entries are made using **create**. The class can be set after creation by accessing the proper fields in the dictionary header. Words starting with **d->** are used to access fields in the dictionary headers.*

```
: displayString: ( "name" — ) 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.

3. 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 a series of calls to other routines.

As an example:

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

This will compile to:

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

The subroutine threading model allows a lot of opportunity for optimization. Recent releases of Retro support inline machine code generation for primitives, so the above example can now 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.

4. 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 re-defined. 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.



www.feedbooks.com

Food for the mind