The Design and Implementation of Retro 10

Table of Contents

1. Introducing Retro

Retro is an implementation of the Forth language. It is minimalistic, but not totally minimal. In Retro you'll find influences from many sources. This is the tenth generation of Retro, and the fifth completely new implementation.

There is a long history behind Retro. Originally created in 1998 by Tom Novelli, it evolved from a 16-bit bootable Forth into a 32-bit bootable Forth that served as a prototype low level language for the Tunes project. A third incarnation, written using a custom assembler and metacompiler never fully worked, and the fourth was heavily influenced by colorForth though retaining the syntax of a classical Forth. When Tom ceased development, Charles Childers took over development and with help from many others brought the fourth generation code to many operating systems and significantly increased features. Along the way it spawned and influenced many other Forths.

The tenth generation is the first complete rewrite since Charles took over. In this incarnation, Retro is smaller and cleaner than before, yet still supports many of the features that were added in the previous generations.

The Retro language and implementation is continually evolving and changing with use. As much as possible this documentation will be kept in sync with the current code, though it may lag behind at times. If you find inaccuracies or have questions on specific areas that aren't clear here, please report them to Charles Childers (crc@retroforth.org).

2. Ngaro, a Virtual Machine

Rather than being written in x86 assembly language, Retro 10 was written to run on a custom virtual machine named Ngaro. This allows Retro to run on a wide variety of devices with minimal effort.

2.1 Processor and Memory Architecture

Ngaro emulates a MISC (*minimal instruction set computing*) processor. This provides two stacks, one for data and the other for addresses. There are instructions for reordering data on the data stack and moving values between the stacks. The MISC processor being emulated has 31 instructions (*see Appendix 2*). Encodings are trivial. Each instruction takes one memory location, with a few taking an additional value from the following memory location.

The memory is setup as simple as possible. Each memory location holds a single 32-bit value. Depending on the hardware being emulated, some memory will be used by the framebuffer.

2.2 Emulated Devices

Ngaro can emulate a number of I/O devices. Retro can detect the virtual hardware being emulated and adjusts the I/O words automatically.

A minimal Ngaro implementation will provide a console for output and keyboard for input. Some implementations provide a graphical console (as a framebuffer), but most only provide a traditional text console. The console device has a built in character generator to handle basic textual output. Additional devices may be provided, but should be treated with care as implementations are not required to provide them.

2.3 I/O Ports

The I/O devices are accessed through a set of I/O ports. The first 13 ports (0 - 12) are reserved for common functionality expected to be supported by all implementations. Ports above 12 can be claimed by implementers for custom devices.

2.3.1 Port 0

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.

2.3.2 Port 1

This is used to read a value from the keyboard. Wait, then read the key value from port 1. Example:

```
wait 1 in
```

2.3.3 Port 2

Use the hardware character generator to display a character. This will take data off the stack, so you should make sure there are enough values on the stack before using it.

Example (*for framebuffer*):

```
( x y character )
100 200 98
```

```
1 2 out
wait
```

Example (*for console*):

```
( character)
98
1 2 out
wait
```

2.3.4 Port 3

This is used to force video updates. 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.

Example:

```
0 3 out
```

2.3.5 Port 4

This port is used to save the image. Allowing saving of images is not required, but is strongly recommended if the underlying host platform will support it. To save, set port 4 to 1 and wait.

Example:

```
1 4 out wait
```

2.3.6 Port 5

This port is a fairly new addition and is used to allow images to query the capabilities of the Ngaro implementation. It is a read/write port, you pass it a given value, 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

2.4 Images

Ngaro works with memory image. An image corresponds exactly to what will be in memory while Ngaro is running. Some implementations of Ngaro allow for saving of memory images and reloading them later. The image file does not preserve any of the internal registers, so must be able to handle reloading as if it were being loaded for the first time. After loading the image, the processor jumps to address 0 and execution proceeds from there.

2.5 Implementations

There are several implementations of Ngaro. These are mostly straight ports of the original implementation, but some differ significantly in the internal approach.

2.5.1 C

There are three implementations in C. The first two are closely related and differ only in the emulated devices. These were developed by Charles Childers and use a switch based processor. Both framebuffer and text console output is supported.

The third was developed by Matthias Schirm and uses a threaded processor. It emulates a text console. This is the fastest implementation of Ngaro.

2.5.2 JavaScript

A JavaScript implementation was developed using the original C implementation as a guide. It is also switch based, and emulates a text console. Given limitations with HTML as a front end, it uses forms and innerHTML to approximate the keyboard and text output areas.

The JavaScript implementation can not save images.

2.5.3 Java *(experimental)*

The JavaScript implementation was used to develop a Java implementation. This shares the same limitations and implementation approach, including the emulated interface.

2.5.4 Java 2 Micro Edition (J2ME) *(experimental)*

An implemention of Ngaro for J2ME has been developed, but is not officially supported yet. Work on integrating it into the main codebase is ongoing.

2.5.4 Perl *(experimental, incomplete)*

A Perl implementation is being developed, but is not yet complete. It's based on the original C implementation and emulates the text console device set.

3. Retro's Implementation

3.1 Threading Model

The threading model used in Retro is subroutine threading. This has been used successfully in Retro since the fourth generation, and allows for a fairly simple compiler and potential optimizations that make the code density tradeoffs worthwhile.

Subroutine threading works by compiling code to a series of native instructions, with calls to other words. For instance, given the following:

```
: foo + . ;
```

Retro will compile:

```
label: foo
  call +
  call .
  return
```

In the future this model will allow optimizations such as eliminating tail calls and inlining of small words for performance.

3.2 Word Classes

3.2.1 Introduction to Word Classes

Retro 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.

3.2.2 Standard Classes

Class Name	Stack	Usage
.word	a --	At compile time, compile a call to the word. At interpret time, execute the word.
.macro	a --	Execute the word. The word may compile code or add data to the heap based on any form of state.
.data	a -- a	At compile time, lay down code to return the address of a data element. At interpret time, return the address of the data.

3.2.3 Adding New 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 behaviour:

- 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 **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 words **d->class** and **d->xt** are *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.

3.3 Vectors

The ninth generation of Retro introduced word vectors. This is a technique that allows for replacing existing words with new functionality. It is related to deferred words, which are common in the mainstream Forths, but is extended to allow words to have a default definition which can be restored later.

In Retro all non-data words are 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**. It is highly recommended that you become comfortable with manipulating vectors as this will allow you a much greater degree of control over the Retro system.

3.4 The Parser

In Retro, the parser operates directly on the input stream, as it is being entered. This means that words are executed immediately upon hitting the space bar. In a normal build, only space marks separation between words. This behavior can be alterted either by loading the *whitespace* module into the Retro image, or by entering a replacement for **(remap-keys)** at runtime.

3.5 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.

4. The Cross Compiler

Retro is written in a custom language that combines an assembler for the Ngaro instruction set with a tiny subset of Forth syntax. It's essentially a limited *machine forth* dialect. The cross compiler generates both an image file and an image map providing addesses and symbolic names for each word and data structure.

The cross compiler is small, but consists of many words. The full list of words is provided in Appendix 3. There is a standard form for applications written using the cross compiler:

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

```
end
```

Code is built on previous code; the cross compiler does not support forward references. When **begin** is encountered, the compiler lays down a jump instruction, which is later patched to the address of **main:**. The **end** word lays down code to cause Ngaro to cease execution of the image, and saves the image to disk, removing any unused cells that follow **end** from the image size.

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 +init

  patch-init-chain
  patch-dictionary

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

In this, note **mark-dictionary**, **word:**, **macro:**, **data:**, and **patch-dictionary**. These are used to build the initial dictionary. The **word:**, **macro:**, and **data:** 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.

5. Writing Modules

Retro, while minimalistic in nature, allows for expansion of the core functionality through the use of modules. Most modules are written to be loaded into the image during the build

process, but is is also possible to load some modules into an existing image.

When building Retro, the core loads a file called *extensions*. This file contains a list of all existing modules which can be included in the image. To use a module, simply uncomment (remove the **#!**) the line before building. Any modules you write should be placed in *rdev/retro/modules*, and be added to *extensions*.

The standard layout for compile-time modules is:

```
MODULE: name of the module
AUTHOR: name of the developer(s) of the module
NOTES:  some text describing the module and it's functionality.
NOTES:  there can be multiple NOTES: lines.

#! ------------------------------------------------------------

   ... code + data ...

#! ------------------------------------------------------------

   address +init

   address word: retroname
   address macro: retroname
   address data: retroname
```

When mapping words into the dictionary, you should start with normal words, followed by macros, and finally list the data words. You do not have to make any module words visible, nor do you have to use **+init**. You should use **+init** if your module needs custom initialization code to be run upon loading the image.

This structure is not forced, but modules should follow this fairly closely if you want them included in the standard distribution. Consistency helps ensure that they are maintainable.

Compile-time modules have full access to all internal words (See *Appendix 1*) and thus can make significant changes to Retro's functionality. The only limit on functionality and size is this: Retro has a maximum core size of 30k cells. All compile-time modules and data must fit in this.

You can load code into an existing image using a manual process. This assumes that you are running on some form of Unix. It's pretty simple:

```
./build
cd bin
cat filename | ./ngaro forth.image
```

There is a limiting factor in doing this. The standard Retro image only recognizes space as valid whitespace, so any use of tabs or newlines will cause problems. This can be fixed by building an image with the *whitespace* module included, or by having your code start with the following line. This should be one single line. Be sure to include a space at the end of the line:

```
: fix dup 10 =if drop 32 then dup 13 =if drop 32 then
      dup 9 =if drop 32 then ; ' fix is (remap-keys)
```

If you want, this can be disabled later by doing:

```
devector (remap-keys)
```

Make sure to end your file with **save**, followed by **bye**. Assuming there are no serious bugs, your changes will be present the next time you load your image.

6. The Words

Retro has many words. A large number of them are used internally; these are discussed in *Appendix 1*. All words accessible from the interpreter are shown in the table below. Stack behaviors are for *runtime*, not compile time.

Name	Data Stack	Address Stack	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
home	--	--	Position the text cursor to the top left corner of the screen
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	$ptr $ptr -- flag	--	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 by 1
!+	n a -- a+1	--	Store *n* into address *a*, increment the address by 1
+!	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	$ptr -- $ptr2	--	Mark a string as temporary
keepString	$ptr -- $ptr2	--	Mark a string as permanent
bye	--	--	Shutdown Ngaro VM
(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 --	--	Class handler for data structures. (Class id = 3)
with-class	a n --	--	Class dispatcher. N is the class id.
boot	--	--	Hook for custom startup code for turnkey applications.
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
vector	--	--	Make a word a vector. Must be the first word in the definition.
for	n --	-- a	Start a counted loop
next	-- n --	a --	End a counted loop. Decreases counter by 1. If counter is 0, exit the loop. Otherwise jump back to **for**. Drops the counter on exiting.
[']	"*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

Appendix 1: Internal Words

In addition to the visible words, there are a large number of internal words which are visible only during the build process. The table below lists all of these words, and provides some descriptions and stack usage for them. All words in this table are vectors and can be replaced with alternatives by modules. Words in bold are visible in the finished image, but may have different naming there.

Word	Stack	Description
last	-- a	Pointer to the most recent dictionary entry
init-chain	-- a	Pointer to the most recent entry in the initialization chain
dup	x -- x x	Duplicates TOS
1+	x -- y	Increment TOS
1-	x -- y	Decrement TOS
swap	x y -- y x	Exchange TOS and NOS
drop	x --	Drop TOS

and	x y -- z	Bitwise AND
or	x y -- z	Bitwise OR
xor	x y -- z	Bitwise XOR
@	a -- n	Fetch data from memory
!	n a --	Store data to memory
+	x y -- z	Add
-	x y -- z	Subtract
*****	x y -- z	Multiply
/mod	x y -- z n	Divide and get remainder
<<	x y -- z	Bitwise left shift
>>	x y -- z	Bitwise right shift
out	x y --	Send a value to a port
in	x -- n	Read a value from a port
wait	--	Wait for an input event
nip	x y -- y	Drop the NOS
over	x y -- x y x	Put a copy of NOS on the top of the stack
2drop	x y --	Drop TOS and NOS
rot	x y z -- y z x	Rotate the top three values on the stack
-rot	x y z -- z x y	Rotate the top three values on the stack twice
tuck	x y -- y x y	Put a copy of TOS under the NOS
2dup	x y -- x y x y	Duplicate TOS and NOS
on	a --	Turn a variable on (set to -1)
off	a --	Turn a variable off (set to 0)
/	x y -- z	Divide
mod	x y -- z	Get remainder only
neg	x -- y	Negate TOS
execute	a --	Execute the code at address *a*
@+	a -- a+1 n	Fetch a value from address *a*, return the value *n* and increment the address by 1
!+	n a -- a+1	Store *n* into address *a*, increment the address by 1

+!	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*.
tx	-- a	Text X coordinate (for framebuffer)
ty	-- a	Text Y coordinate (for framebuffer)
fb	-- a	Holds the address of the framebuffer
update	-- a	Enable/Disable video update (used by **redraw**)
redraw	--	Force video update (for framebuffer)
fb:cr	--	Move cursor to the next line (for framebuffer)
move	--	Move the cursor by one character (for framebuffer)
fb:emit	n --	Display a character (for framebuffer)
fb:home	--	Reset TX and TY to top-left corner of the display. This is used by fb:clear
(type)	a0 -- a1	Internal loop for **type**
type	a --	Display a string
(clear)	--	Internal loop for fb:clear (for framebuffer)
fb:clear	--	Clear the screen (for framebuffer)
tty:emit	c --	Display a character (for tty)
tty:cr	--	Move the cursor to the next line (for tty)
tty:home	--	Move the cursor to the top-left corner of the display (for tty)
tty:clear	--	Clear the screen (for tty)
emit	c --	Display a character
cr	--	Move the cursor to the next line
clear	--	Clear the display

home	--	Move the cursor to the top-left corner of the display
save	--	Save the image
bye	--	Shutdown RETRO
words	--	Display a list of all defined words
tib	-- a	Text Input Buffer
>in	-- a	Pointer into the **tib**
break-char	-- a	Holds the character used as a delimiter by **accept**.
(remap-keys)	--	Hook to remap key values before they can be processed. Called by **key** right before returning the value.
key	-- n	Read a single keypress
>tib	n --	Add a character to the **tib**
++	--	Increment **>in**
(eat-leading)	--	Eat leading keystrokes matching **break-char**. This is used by **accept** to ignore extra spaces at the start of a token.
(accept)	-- c	Internal loop used by **accept**.
accept	c --	Read a string delimited by *c*. Modifies **break-char**.
state	-- a	Compiler state (can be either on or off)
heap	-- a	Current memory location to compile code and/or data to
compiling?	--	Exit the word if the compiler **state** is true.
t-here	-- a	Return the current memory location
t-,	n --	Compile a value into the current memory location
t-]	--	Start the compiler
t-[	--	Suspend the compiler
t-;;	--	Compile a **;** instruction
t-;	--	Compile a **;** instruction and stop compiling

($,)	a0 -- a1	Internal loop used by **$**
$	a --	Compile a string into the definitions
create	"name" --	Create a new dictionary entry
(:)	--	Common code for **t-:** and **t-macro:**
t-:	"name" --	Start a new definition (*word class*)
t-macro:	"name" --	Start a new definition (*macro class*)
t-(	"comment)" --	Start a comment
t-push	n --	Compile a **push** instruction
t-pop	-- n	Compile a **pop** instruction
$flag	-- a	Flag used by various string words
n=n	x y --	Sets *$flag* to 0 if the values do not match
get-set	a0 a1 -- x y	Get values from two strings
next-set	a0 a1 -- a2 a3	Increment two string pointers
(skim)	a0 a1 -- a2 a3	Internal loop used by **compare**
compare	a1 a2 -- f	Compare two strings
(strlen)	--	Internal loop used by **getLength**
getLength	--	Get the length of a string
STRINGS	-- a	Starting address for the permanent string buffer
SAFE	-- a	Used during creation of temporary strings. On invoking **"** this points to the temporary string buffer. It is incremented for each character in the string.
LATEST	-- a	Start address of the most recent permanent string
(reset-$)	--	Reset SAFE to the start of the temporary string buffer
(next)	--	Increment SAFE
(copy)	a0 -- a1	Copy a string into the temporary string buffer.

tempString	a0 -- a1	Move a string to the temporary buffer
keepString	a0 -- a1	Move a string to the permanent string buffer
t-"	"text" -- a	Get a string (interpret)
t-s"	"text" -- a	Get a string (compiler) and compile a pointer to it into the current defintion
#value	-- a	Internal value used for numeric words
#flag	-- a	Internal value used for numeric words
num	-- a	Internal value used for numeric words
num-ok	-- a	Internal value used for numeric words
negate?	-- a	Internal value used for numeric words
isdigit?	c -- f	Returns true if the character is a number. False if not.
char>digit	c -- n	Convert an ASCII character to a number
digit>char	n -- c	Convert a number to an ASCII character
isNegative?	a -- a+1	Sets **negate?** to true if the number starts with a - character
(convert)	a0 -- a1	Internal loop used by **>number**
>number	a -- n	Convert a string to a number
(isnumber)	a0 -- a1	Internal loop used by **isnumber?**
isnumber?	a -- f	Returns true if string *a* is a valid number.
number>digits	n -- c0 ... cN	Break a number into individual characters. Used by **.**
digits>screen	c0 ... cN --	Display a sequence of characters. Used by **.**
.	n --	Display a number
found	-- a	Search result flag

which	-- a	Pointer to the found dictionary header
d->class	d -- a	Dictionary: Get Class slot
d->xt	d -- a	Dictionary: Get XT slot
d->name	d -- a	Dictionary: Get Name slot
(search)	--	Internal loop used by **search**
search	a --	Search for a word in the dictionary.
t-=if	x y --	Compile a **!jump** instruction
t->if	x y --	Compile a **<jump** instruction
t-<if	x y --	Compile a **>jump** instruction
t-!if	x y --	Compile a **=jump** instruction
t-then	--	Patch the last conditional jump
t-repeat	--	Start an unconditional loop
t-again	--	Close an unconditional loop
t-0;	n -- n || n --	Compile a **0;** instruction. Leaves *n* alone if *n* is non-zero, otherwise drops *n*.
t-'	"name" -- a	Return the address of a word. If the word is not found, display an error and return 0.
t-[']	"name" -- a	Get the address of a word and compile it into the current definition.
devector	"name" --	Remve the vector from *name*
is	a "name" --	Assign the vector named *name* to address *a*
:devector	xt --	Remove the vector from *xt*
:is	a xt --	Assign vector at *xt* to address *a*
compile	xt --	Compile a call to *xt* into the current definition
literal,	n --	Compile *n* into the current definition as a literal
cold	-- a	Variable determining whether or not to display the **copytag**.
copytag	-- a	Copyright / Bootup message. Used only by **runonce**.

boot	--	Hook for turnkey applications.
runonce	--	Runs when the image is loaded. Performs initializations, then calls **boot**.
nomatch	-- a	String to display when no match could be found. Used by **notfound**.
okmsg	-- a	String to display as the "ok" prompt
.word	a --	Word class handler
.macro	a --	Macro class handler
.data	n --	Data class handler
with-class	a --	Execute an XT with the proper class handler
notfound	--	Called when a lookup fails. Will display the **nomatch** string.
find-word	--	Search for a word in the dictionary. Used by **listen**.
find-number	--	Attempt to convert a token to a number. Used by **listen**.
ok	--	Display the *ok* prompt if not compiling. Used by **listen**.
listen	--	The main interpreter loop
initialize	--	Run each word in the initialization chain once
fw	-- a	Height of framebuffer
fh	-- a	Width of framebuffer
#mem	-- a	Amount of memory

Appendix 2: Ngaro Instruction Set

The MISC processor emulated by Ngaro has 31 instructions. Instruction encodings are trivial: one instruction per cell. A few (lit, call, and the branch forms) take an additional value from the following cell. All opcodes listed here are 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
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 -- 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.

Appendix 3: Cross Compiler Words

The cross compiler contains an assembler for the Ngaro instruction set, and some helper words that provide a simple machine forth implementation. These words are shown in the table below.

Word	Stack	Description
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	--	Conditional (equality)
<if	--	Conditional (less than)
>if	--	Conditional (greater than)
!if	--	Conditional (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*
repeat	-- a	Begin an unconditional loop
again	a --	End an unconditional loop
variable:	n "name" --	Create a variable with an initial value of *n*
variable	"name" --	Create a variable with an initial value of 0
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 elimintated
;	--	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

Appendix 4: Building Retro

A4.1 Building on a Unix-like Platform

Retro should be buildable on most Unix-like systems with minimal effort.

Prerequisites:

- Retro Source: http://retroforth.org
- GCC
- BASH or KSH shell

Optional:

- libSDL
- Java Development Kit

Preparation:

Decompress the RETRO package (rdev-YYYYMMDD.tar.gz).

```
zcat retro-YYYYMMDD.tar.gz | tar xv
```

Building:

Once the package has been decompressed, building it is easy:

```
cd retro-YYYYMMDD
./build
```

The build script will display a menu of targets that can be built. Make a selection and press **enter**.

A4.2 Building on Windows

Building Retro on Windows is possible, but significantly more involved than building on a Unix-like system. The instructions here show how to build Ngaro. Building a copy of the Retro image (*forth.image*) is not yet documented on Windows.

Prerequisites:

- Dev C++
  - http://prdownloads.sourceforge.net/dev-cpp/devcpp-4.9.9.2_setup.exe
- libSDL Runtime
  - http://libsdl.org/release/SDL-1.2.13-win32.zip
- libSDL Development Libraries (for mingw32)
  - http://libsdl.org/release/SDL-devel-1.2.13-mingw32.tar.gz
- Retro Development Kit (rdev)
  - Get the latest one from http://retroforth.org

Process:

1. Install Dev C++
2. Unzip the SDL runtime and copy *SDL.dll* to *C:\WINDOWS*
   (*This will allow the library to be used system wide*)
3. Unzip the SDL development libraries and copy the contents of
   i386-mingw32msvc\include\SDL to *C:\Dev-C++\Include*
4. Copy the libraries from the *i386-mingw32msvc\lib* to *C:\Dev-C++\Lib*
5. Create a new empty Dev C++ project and add the following files from *retro-YYYYMMDD\ngaro\c-crc\sdl* to it:
   1. **endian.c**
   2. **functions.h**
   3. **loader.c**
   4. **ngaro_windows.c**
   5. **sdl_devices.c**
   6. **vm.c**
   7. **vm.h**
8. Go to *Project -> Options* and add the following to the Compiler commands text box:
   -DUSE_SDL
9. Go to *Project -> Options* and add the following to the Linker commands:
   -lmingw32 -lSDLmain -lSDL -mwindows
10. Compile everything