Welcome to RetroForth

Hello, and welcome to the RetroForth Handbook. This book will hopefully help you learn to use RetroForth and answer most (if not all) of your questions.

What Is RetroForth?

> *There are plenty of full-featured Forth compilers, e.g. IsForth, BigForth, Win32forth, Gforth, and the commercial Forths.*
>
> *RetroForth [since version 3] is modeled after cmForth, Colorforth, Eforth and Pygmy. It uses some, but not all, of Chuck Moore's newer ideas. It's clean, elegant, and tiny - only about 20k source/10k binary for Linux. It may not have many features, and it may not be particularly useful by itself, but it's easy to grasp, easily adapted to various uses - even on other CPU architectures. I think that's our niche. --Tom Novelli*

RetroForth is an implementation of Forth for Linux, Windows, FreeBSD, BeOS, and other OSes for the x86 CPU line. It can also be used as an operating system.

The design and implementation have been worked on continually over seven years. The current version represents the result of this effort, and we feel that RetroForth provides a clean, solid language for writing truly useful software.

Getting Started

You can always get the latest version of RetroForth from http://www.retroforth.org. At the time of this writing, the latest version is 8.0.

There are different downloads for each port. You can choose between two archive formats: .tar.gz or .zip. If you're using Windows, we recommend getting the .zip form. Users of other OSes should get the .tar.gz format since it preserves file permissions properly.

Once you download a copy, decompress it, and enter the newly created ***retro8*** directory. For most ports there will be three or four subdirectories. The ***bin*** subdirectory contains the executable copy of RetroForth. On most systems it will contain one file named ***rf*** (or ***rf.exe*** for Windows users).The next directory will be named ***lib***. This contains the *code library*. The library is not included in the Native or L4 ports. In the ***source*** directory the full source code is provided. Finally you may have a ***doc*** directory containing a copy of this handbook or other documentation.

On most systems you simply need to run the ***rf*** or ***rf.exe*** file in ***bin*** to run RetroForth. On a few systems you will need to rebuild it yourself. Setup for Native and L4 users is a more involved process. We will take a look at that in the *Ports* section of this handbook.

> *Unless you're using one of the generic ports, you don't need to bother rebuilding unless you make changes to the source tree. Your binary won't be any better matched for your OS since we don't use any libraries, except in the generic and generic+ffi ports.*

Source Tree

There are many files in the source tree, but it's pretty easy to find what you need due to careful organization:

core.asm Port-independent code
core.dict Port-independent dictionary declarations
macros.fasm Macros used in the Forth Core
rf.asm Port-specific code
raw.asm For RetroForth/Native only.
blocks/base The core wordset for RetroForth.
blocks/blockio RetroForth 8-style block I/O.
blocks/classic-ui Classic Forth "OK"-style prompt.
blocks/colors Colors for use with output.
blocks/portio For Native and L4 only.
blocks/red* The block editor.

You might see a core.f file in your distribution. This is auto-generated by the build process, given the contents of the blocks/ directory as input.

Terminology

RetroForth is not a conventional forth, so some of the terminology used may be different than you're used to. Here are some of the words that we often use in discussions.

Heap *This is where compiled code and data are placed. Strings are also allocated from the heap. h0 is a variable pointing to the heap; here is a word returning the current top of the heap.*

Dictionary *This is where the headers for compiled words go. Note that the compiled code is not placed here. You can use the word last to obtain a pointer to the most recent dictionary entry.*

Block Buffer *When you load a blockfile into memory, this is where it's placed. The word offset gives you a pointer to the start of the first block.*

TIB *Text Input Buffer. When you type code at the console, it's placed here while being interpreted. The word tib returns the address of the TIB. >in is a pointer into the TIB.*

RED *The RetroEditor, it's included in every copy of RetroForth.*

FFI *Foreign Function Interface. This is used to load and invoke external libraries.*

The Glossary

RetroForth has many words that you can use in your programs. Many of them will be familiar to those who already know another Forth, but some will be quite new. This section attempts to provide a relatively up-to-date overview of all the words.

Please be aware that a current copy of the glossary is provided with all releases of RetroForth 8. Since releases occur more rapidly than updates to the Handbook, you should review the glossary.txt file for the most accurate list of words and their uses.

Each entry has a format like:

wordname

A brief description of the word

Takes:
Returns:

Some optional notes about the word

Core Words

---

forth macro

Use: Switch the active dictionary for compilation
Take: --
Return: --
Notes:

---

find mfind

Use: Find a word in either the forth (find) or macro (mfind) dictionaries.
Take: a #
Return: xt (if found)
a # (if not found)
Notes: Returns TRUE if found, FALSE if not found.

>number

Use: Convert a string to an integer on the stack
Takes: a #
Returns: n (if a valid number in the current base)
a # (if not a valid number in this base)
Notes: Returns TRUE if the conversion worked; FALSE if it failed.

---

interpret

Use: Process input from the TIB
Take: --
Return: -- (results of evaluation)
Notes: 'eval' also uses this internally

---

eval

Use: Evaluate a string
Take: a #
Return: -- (the results of the evaluation are left on the stack)
Notes:

1, 2, 3, ,

Use: Inline 1, 2, 3, or 4 bytes to HERE
Take: n
Return: --
Notes:
---
entry

Use: Create a new dictionary entry from a string
Take: a #
Return: --
Notes: This does not define a body for the entry, it just creates an entry pointed to HERE

]

Use: The compiler
Take: --
Return: --
Notes: This is where the compilation of a word actually takes place
---
compile

Use: Compile a CALL to a word
Take: a
Return: --
Notes: The CALL is relative, not absolute

[

Use: Stop a compile, drop back to the interpreter
Take: --
Return: --
Notes: Does not terminate the definition, use ; for that.

;;

Use: compile an exit to a definition
Take: --
Return: --
Notes: If the last thing compiled was a call, change to a jmp. Otherwise just compiles in a ret.

;

Use: End a definition
Take: --
Return: --
Notes: Calls ;; and then jumps to [

:

Use: Main compiler wrapper
Take: --
Return: --
Notes: Creates an entry (saving the old state of HERE and LAST), then jumps to]
---
literal

Use: Compiles a literal into a definition
Take: n
Return: --
Notes:

#

Use: Display the unsigned value of the TOS
Take: n
Return: --
Notes: . and u. are written around this. # does not leave a trailing space.
---
parse

Use: Parse input stream
Take: <char>
Return: a #
Notes: Parses TIB until it encounters <char> or the end of a line.
---
reset

Use: Reset the stack to the default starting point
Take: --
Return: --
Notes: Also replaces all values on the stack with 0's

---

create

Use: create a variable
Take: --
Return: --
Notes: parses the input stream for the name

---

does>

Use: Modify the runtime behavor of a 'create'd word
Take: --
Return: addr
Notes: changes 'call dovar' to 'call <code after does>'
Adds some code to push the address of the data part
to the stack.

---

last

Use: Returns the address of the last dictionary entry
Take: --
Return: addr
Notes:

---

there

Use: Returns the address of the block buffer
Take: --
Return: addr
Notes:

---

tib

Use: Push the address of the TIB
Take: --
Return: addr
Notes:

CORE Variables

---

h0 Pointer to HERE
base The current numeric base
>in Pointer to the current location in the input stream
word? Pointer to a word that handles errors when words
aren't found

Macros

1+   1-

Use: Increment (1+) or Decrement (1-) the TOS.
Take: n
Return: n+1
Notes: Inlines to "inc eax" or "dec eax"
---
swap

Use: swap the top two items on the stack
Take: x y
Return: y x
Notes: inlined to "xchg eax, [esi]"
---
drop

Use: drop the tos, setting nos as the new tos
Take: ... n
Return: ...
Notes: inlines to "lodsd"
---
nip

Use: drop the nos
Take: ... x y
Return: ... y
Notes: inlines to ""
---
true   false

Use: set the value of the flag
Take:
Return:
Notes: On x86 this uses the carry flag and can be used with
         ?if for conditionals.

---

f: m:

Use: Force the compiler to compile a call to a forth word (f:) or macro (m:).
Take:
Return:
Notes: This can be useful if you need to call a macro at runtime, or if you have a forth word with the same name as a macro.

These words parse the input stream
Example: : foo f: bar ; | compile a call to the forth word "bar"
: baz m: bat ; | compile a call to the macro named "bat"

[']

Use: Compile the address of a word into the definition
Take:
Return:
Notes: This only searches the forth dictionary, not macros

This is a parsing word
Example: : foo ['] words . ;

---

s"

Use: Compile a string into a definition
Take:
Return: a #
Notes: The string is placed in a special memory buffer and the address/count is stored in the definition.

This is a parsing word.
Example: : foo s" hello, world!" type cr ;

."

Use: Same as s" but also compiles in a call to 'type'
Take:
Return:
Notes: Basically this is provided to make displaing strings easier. It is a parsing word.

:

```
Use:     Define a new entry point
Take:
Return:
Notes:   Define a new entry point into a definition.

         This is a parsing word
Example: : foo 1 2 : bar + 3 * ;
Notes:   In this example, foo puts 1 and then 2 onto the stack
         Execution falls through to 'bar' which adds the tos
         and nos and then multiplies by 3. Calling 'bar' by
         itself will use the values you placed on the stack.
```

(

```
Use:     Comments in definitions
Take:
Return:
Notes:   Allow things like stack comments. Comments are
         terminiated by a ) character.

         This is a parsing word
Example: : foo ( x -- x+2 )  2 + ;
```

>r

```
Use:     Place a value on the return stack
Take:    x
Return:
Notes:   Drops the TOS
```

---

r>

```
Use:     Take the top of the return stack and place on data
         stack
Take:
Return:  x
Notes:   Drops the TORS
```

---

r

```
Use:     Place a copy of the TORS on the data stack
Take:
Return:  x
Notes:   Does not drop the TORS
```

---
repeat

Use: Start a structured loop
Take:
Return:
Notes:
---
again

Use: Repeat an unconditional loop
Take:
Return:
Notes:
Example: : foo 0 repeat dup . 1+ again ;
---
until

Use: Repeat a counted loop
Take: x
Return: x-1 (see notes)
Notes: When x=0 the loop exits

Decrements tos *before* comparing to 0. This means that if you say "0 until" an essentially infinite loop will result.

The counter is the TOS
Example: : foo 10 repeat dup . 1- until cr ;
---
for

Use: Start a counted loop
Take:
Return:
Notes: This is the same as 'repeat >r'
---
next

Use: Repeat a counted loop
Take:
Return:
Notes: The counter is on the return stack, not the data stack.

See the notes on 'until' for more details.

This is the same as 'r> until'
Example: : foo 10 for r . next cr ;

```
-------------------------------------------------------------
=if

Use:     Conditional
Take:    x y
Return:
Notes:   Continue execution if x and y are equal, otherwise
         skip to after 'then'
-------------------------------------------------------------
<>if

Use:     Conditional
Take:    x y
Return:
Notes:   Continue execution if x and y are not equal, else
         skip to after 'then'
-------------------------------------------------------------
>if

Use:     Conditional
Take:    x y
Return:
Notes:   Continue execution if x is greater than y,
         otherwise skip to after 'then'
-------------------------------------------------------------
<if

Use:     Conditional
Take:    x y
Return:
Notes:   Continue execution if x is less than y,
         otherwise skip to after 'then'
-------------------------------------------------------------
?if

Use:     Conditional
Take:
Return:
Notes:   Continue execution if the flag is true, otherwise
         skips to after 'then'
-------------------------------------------------------------
(if)

Use:     Conditional (not to be called directly!)
Take:    x
Return:
Notes:   This is used to implement other conditionals.
-------------------------------------------------------------
```

Forth Words
---
swap

Use: Swap the TOS and NOS
Take: x y
Return: y x
Notes: This is also a macro
---
drop

Use: Drop the TOS
Take: ... x
Return: ...
Notes: This is also a macro
---
nip

Use: Drop the NOS
Take: ... x y
Return: ... y
Notes: This is also a macro
---
dup

Use: Duplicate the TOS
Take: x
Return: x x
Notes:
---
and

Use: Bitwise AND
Take: x y
Return: z
Notes:
---
or

Use: Bitwise OR
Take: x y
Return: z
Notes:

---
xor

Use: Bitwise XOR
Take: x y
Return: z
Notes:
---
not

Use: Bitwise NOT
Take: x
Return: y
Notes:

@ c@

Use: Fetch a value
Take: a
Return: n
Notes: Fetch a cell (@) or byte (c@) from the address provided
---
! c!

Use: Store a value
Take: a n
Return:
Notes: Store a cell (!) or byte (c!) to the provided address
---
hex

Use: Change the base to hexadecimal
Take:
Return:
Notes: Base 16
---
decimal

Use: Change the base to decimal
Take:
Return:
Notes: Base 10

---
binary

Use: Change the base to binary
Take:
Return:
Notes: Base 2
---
octal

Use: Change the base to octal
Take:
Return:
Notes: Base 8

+

Use: Add TOS and NOS
Take: x y
Return: z
Notes: x+y

-

Use: Subtract NOS from TOS
Take: x y
Return: z
Notes: x-y
---
*

Use: Multiply TOS and NOS
Take: x y
Return: z
Notes: x*y

/mod

Use: Divide and determine remainder
Take: x y
Return: d r
Notes:

/

Use: Divide NOS by TOS
Take: x y
Return: z
Notes: x/y

---

mod

Use: Divide NOS by TOS and return the remainder
Take: x y
Return: z
Notes:

---

negate

Use: Negate the TOS
Take: x
Return: x
Notes: The same as doing -1 *

1+

Use: Add 1 to TOS
Take: x
Return: x+1
Notes: This is also a macro

1-

Use: Subtract 1 from TOS
Take: x
Return: x-1
Notes: This is also a macro

(

Use: Comments
Take:
Return:
Notes: This parses until a) is encounted and ignores the parsed code.

This is a parsing word

|

Use: Comments
Take:
Return:
Notes: This parses until the end of the line and ignores the parsed code. This type of comment does not work in definitions

This is a parsing word

```
-------------------------------------------------------------
wsparse

Use:     Parse ahead until the next whitespace
Take:
Return:  a #
Notes:   Whitespace is either a space, tab, or EOL character

         This is a parsing word
-------------------------------------------------------------
lnparse

Use:     Parse ahead until the end of the line
Take:
Return:  a #
Notes:   EOL can be a cr or lf character

         This is a parsing word
-------------------------------------------------------------
>>

Use:     Shift right
Take:    x y
Return:
Notes:
-------------------------------------------------------------
<<

Use:     Shift left
Take:
Return:
Notes:
-------------------------------------------------------------
here

Use:     Return the top of the heap
Take:
Return:  a
Notes:   h0 is a variable pointing to the top of the heap. So
         'here' is the same as doing 'h0 @'
-------------------------------------------------------------
allot

Use:     Allocate memory on the heap
Take:    x
Return:
Notes:   Allocate X bytes of memory
-------------------------------------------------------------
```

```
-------------------------------------------------------------
cells

Use:     Convert a number of cells to the size in bytes
Take:    x
Return:  x*4
Notes:   Cells are 4 bytes (a dword on x86) in size
-------------------------------------------------------------
cell+

Use:     Increase TOS by the size of a cell
Take:    x
Return:  x+4
Notes:   Cells are 4 bytes (a dword on x86) in size
-------------------------------------------------------------
later

Use:     Delay execution of a word until the caller finishes
Take:
Return:
Notes:   This is tricky to learn, but very powerful
-------------------------------------------------------------
exit,

Use:     Compile an exit instruction
Take:
Return:
Notes:   Compiles a RET ($c3) on x86
-------------------------------------------------------------
create:

Use:     Create a new dictionary entry
Take:
Return:
Notes:   Does a 'wsparse entry' to create the entry

         This is a parsing word
-------------------------------------------------------------
variable

Use:     Create a variable
Take:
Return:
Notes:   This creates a variable with an initial value of 0

         This is a parsing word
-------------------------------------------------------------
```

```
-------------------------------------------------------------
variable:

Use:     Create a variable
Take:    x
Return:
Notes:   This creates a variable with an initial value of 'x'

         This is a parsing word
-------------------------------------------------------------
rot

Use:     Rotate the stack
Take:    x y z
Return:  y z x
Notes:
-------------------------------------------------------------
-rot

Use:     Rotate the stack twice
Take:    x y z
Return:  z x y
Notes:
-------------------------------------------------------------
tuck

Use:     Tuck a copy of the TOS under the NOS
Take:    ... x y
Return:  ... y x y
Notes:
-------------------------------------------------------------
over

Use:     Place a copy of the NOS over the TOS
Take:    ... x y
Return:  ... x y x
Notes:
-------------------------------------------------------------
2drop

Use:     Drop the top two entries on the stack
Take:    ... x y
Return:  ...
Notes:
-------------------------------------------------------------
```

```
------------------------------------------------------------
2dup

Use:     Duplicate the top two entries on the stack
Take:    ... x y
Return:  ... x y x y
Notes:   The same as doing 'over over'
------------------------------------------------------------
'

Use:     Obtain the address of a word
Take:
Return:  a
Notes:   This does not recognize macros

         This is a parsing word
------------------------------------------------------------
alias

Use:     Bind an address to a name
Take:    a
Return:
Notes:   Using this with loc: and ;loc allows localized
         fatoring

         This is a parsing word
------------------------------------------------------------
execute

Use:     Execute the provided address
Take:    a
Return:
Notes:   The address is passed on the stack. No validation is
         performed, so be careful when using this
------------------------------------------------------------
literal,

Use:     Compile a literal to HERE
Take:    x
Return:
Notes:
------------------------------------------------------------
0;

Use:     Exit a word if the TOS is 0
Take:    x
Return:  x (if not zero)
           (drops tos if it is 0)
Notes:   Useful in loops
------------------------------------------------------------
```

---

list

Use: Array holding the addresses of 'last' during a loc: and ;loc pairing
Take:
Return: a
Notes: Not intended for use by words other than loc: and ;loc

---

loc:

Use: Start a locally factored definition
Take:
Return:
Notes: loc: is to be used with ;loc. You can nest loc: up to four times.

;loc

Use: End a locally factored definition
Take:
Return:
Notes: This removes all word names between the prior loc: and this. The definitions are left behind.

---

fill

Use: Fill memory with a specified byte
Take: a # b
Return:
Notes: b is the byte to fill memory with

---

move

Use: Copy a region of memory from one location to another
Take: s d #
Return:
Notes: s is source, d is dest, # is the number of bytes to move

---

pad

Use: Return the address of the PAD
Take:
Return: a
Notes: Strings are initially placed in PAD

```
-------------------------------------------------------------
>pad

Use:     Copy a string to the PAD
Take:    a #
Return:  a #
Notes:   Destroys the original contents of PAD
-------------------------------------------------------------
"

Use:     Create a temporary string
Take:
Return:  a #
Notes:   The string is placed in PAD using >PAD
-------------------------------------------------------------
."

Use:     Display a string
Take:
Return:
Notes:   The string is placed in PAD before displaying it
-------------------------------------------------------------
$,

Use:     Compile a temporary string into the current definition
Take:    addr #
Return:
Notes:   The address and count are inlined to HERE, followed
         by a jmp over the string, and a nop at the end.
-------------------------------------------------------------
zt

Use:     Make a temporary zero-terminated string
Take:    a #
Return:  a
Notes:   This replaces zt-make and zt-free from the 7.x series
-------------------------------------------------------------
cr

Use:     Next output will be on the next line
Take:
Return:
Notes:
-------------------------------------------------------------
```

---
space

Use: Display a space
Take:
Return:
Notes:

.

Use: Display a signed number
Take: n
Return:
Notes:
---
u.

Use: Display an unsigned number
Take: n
Return:
Notes:
---
words

Use: Display all currently defined words in the current dictionary
Take:
Return:
Notes:
---
ok

Use: The interpreter loop
Take:
Return:
Notes: You can customize this to provide a prompt if you want one.

Editor Words

---

blk

Use: Variable storing the current block number
Take:
Return: a
Notes: Mainly used internally

#blks

Use: Variable storing the number of blocks to load/save
Take:
Return: a
Notes: The word 'blocks' is used to set this

(block)

Use: Return the address of the current block
Take:
Return: a
Notes: The same as: blk @ block

---

block

Use: Return the address of a block
Take: n
Return: a
Notes:

(line)

Use: Return the address of a line in the current block
Take: n
Return: a
Notes: Used internally

---

p n

Use: Go to the next or previous block
Take:
Return:
Notes: p is previous block, n is next block

---
d

Use: Delete a line
Take: n
Return:
Notes:
---
x

Use: Delete the current block
Take:
Return:
Notes:
---
eb

Use: Evaluate the current block
Take:
Return:
Notes: Evaluates the entire block at once
---
el

Use: Evaluate a line in the current block
Take: n
Return:
Notes:
---
e

Use: Evaluate the current block
Take:
Return:
Notes: Evaluates each line separately
---
ia

Use: Insert a line at a specified column
Take: column# line#
Return:
Notes: This is a parsing word.
Example: 1 4 ia Hello

---
i

Use: Insert a line
Take: line#
Return:
Notes: This is a parsing word
Example: 5 i Hello again
---
s

Use: Select a new block
Take: n
Return:
Notes: The same as doing 'blk !'

\

Use: Evaluate the startup block
Take:
Return:
Notes: The first block (0) is reserved for comments. This
         does '1 s e'. It's assumed that the loaded block will
         chainload the rest.

\f

Use: Evaluate a loaded file
Take:
Return:
Notes: This will assume that you've used "load" to load a
         file to the block buffer. It'll evaluate everything
         in the loaded file.
---
blocks

Use: Set the number of blocks to load or save
Take: n
Return:
Notes: The same as '#blks !'
---
new

Use: Erase all of the blocks
Take:
Return:
Notes:

```
-------------------------------------------------------------
v

Use:     View the current block
Take:
Return:
Notes:   Also shows the 'status' information
-------------------------------------------------------------
.s

Use:     Display the top ten items on the stack
Take:
Return:
Notes:
-------------------------------------------------------------
status

Use:     Display some status information
Take:
Return:
Notes:   Displays the block number, and does '.s'
-------------------------------------------------------------
use

Use:     Select a new active file
Take:
Return:
Notes:   Not defined in the native or L4 ports

         This is a parsing word
Example: use lib/retrospect
-------------------------------------------------------------
r

Use:     Read the selected file to the block memory
Take:
Return:
Notes:
-------------------------------------------------------------
w

Use:     Write the blocks to the selected file
Take:
Return:
Notes:
-------------------------------------------------------------
```

```
-------------------------------------------------------------
load

Use:     Select a file and read it into memory
Take:
Return:
Notes:   This is a parsing word
Example: load lib/retrospect
-------------------------------------------------------------
```

Color and Console Words

---
clear

Use: Clear the screen
Take:
Return:
Notes:
---
home

Use: Go to the home (upper left hand corner) of the screen
Take:
Return:
Notes:
---
normal

Use: Reset the display attributes to the normal setting
Take:
Return:
Notes: Normal is generally white text on a black background
---
black red green yellow blue magenta cyan white

Use: Change the foreground color
Take:
Return:
Notes:
---
onBlack onRed onGreen onYellow onBlue onMagenta
onCyan onWhite

Use: Change the background color
Take:
Return:
Notes:

System Interface Words

```
-------------------------------------------------------------
syscall    (Linux)

Use:     Perform a system call operation
Take:    a0 a1 ... a# # n
Return:  n
Notes:   n is the syscall number, # is the number of arguments
         a0 through a# are the arguments to the syscall
-------------------------------------------------------------
syscall    (Dex4u)

Use:     Perform a system call operation
Take:    edi esi edx ecx ebx eax
Return:  edi esi edx ecx ebx eax
Notes:
-------------------------------------------------------------
from

Use:     Select a library to import from
Take:
Return:
Notes:   Only in the Windows and Generic+FFI ports

         This is a parsing word
Example: from kernel32.dll
         from libc.so
-------------------------------------------------------------
import

Use:     Import a "stdcall" style function
Take:    n
Return:
Notes:   Only in the Windows and Generic+FFI ports

         This is a parsing word
Example: 10 import foobar
         | This will import a function named foobar that takes
         | 10 arguments
-------------------------------------------------------------
```

```
-------------------------------------------------------------
cimport

Use:     Import a C-style function
Take:    n
Return:
Notes:   Only in the Windows and Generic+FFI ports

         This is a parsing word
Example: 10 cimport foobar
         | This will import a function named foobar that takes
         | 10 arguments
-------------------------------------------------------------
p1@  p2@

Use:     Read from a port
Take:    port
Return:  n
Notes:   p1@ is for bytes,  p2@ is for words

         Only in Native and L4
-------------------------------------------------------------
p1!  p2!

Use:     Write to a port
Take:    n port
Return:
Notes:   p1! is for bytes,  p2! is for words

         Only in Native and L4
-------------------------------------------------------------
```

The Core

We call the assembly portion of RetroForth the core. The core is an interesting place, since it provides the initial functionality needed to bootstrap RetroForth, yet it doesn't actually provide a lot of inherent functionality.

The core consists of a handful of words and variables used to implement the interpreter and compiler, but not stack manipulation, arithmetic, or even memory management. It's really a skeleton on which a dialect of Forth can be built.

The core also encompasses a few I/O words like key, emit, type, and #, but this is only to make life easier for those doing new ports.

Most people don't need to delve deeply into the inner workings of the core, but if you want to modify the compiler itself, or alter some internal functionality, you should read the rest of this section.

Dictionary Structure

We have chosen to implement a simple dictionary structure for RetroForth. Basically there are two linked lists; one for each dictionary. Each entry looks like this (in assembly):

```
dd link_to_previous_entry
dd address_of_word
db length_of_the_name
db 'the_name_itself'
```

The first entry in the dictionary has a link to the address 0 in the first field. This is used to signify the end of the dictionary. We provide a meta-variable named last which provides a pointer to the most recent entry.

What if you want to get at the first entry? Define a word that walks back through the dictionary and returns the address of the first entry. Something like this can work:

```
: first last repeat dup @ 0; nip again ;
```

Memory Layout

RetroForth has a pretty simple memory layout. It's divided into five sections:

Kernel (About 2-3 K)
TIB (1 K)
Dictionary (256 K)
Heap (1 M)
Block Buffer (128 K)

The kernel contains the barebones compiler and the initial bootstrap code. The other sections are mentioned in the terminology section. The memory in TIB and beyond is considered ok to mess with. (We recommend not directly modifying the kernel portion unless you're very familiar with RetroForth's internals)

To get at each of these areas we provide a number of words:

`tib`	Returns the address of the tib
`last`	Returns the address of the most recent dictionary entry
`h0`	Variable pointing to the top of the heap
`here`	Returns the address of top of the heap
`there`	Returns the address of the block buffer

Those are the provided words for accessing these areas

To get to a specific section, just add the offsets starting with ‘tib’

```
: dictionary [ tib 1024 + ] literal ;
: heap [ dictionary $40000 + ] literal ;
```

There are some dictionary headers outside the dictionary region, but these are defined in the assembly source. It is possible to get access to them by walking through the dictionary.

Assembler Macros

There are quite a few macros that we use in the assembly part of the source. Several of these are used for readability reasons. A few are used to build the initial dictionaries. If you intend to work on the core you need to understand these.

upsh	Push a value to the data stack
upop	Pop a value from the data stack
drop	Drop the TOS
dup	Duplicate the TOS
code	Add a new Forth word
mcode	Add a new compiler macro
next	End a definition
var	Add a new variable
inline	Inline some bytes
fdef	Creates a Forth-vocabulary dictionary for a colon entry (new as of 8.0)
mdef	Creates a Macro-vocabulary dictionary for a colon entry (new as of 8.0)
vdef	Creates a Forth-vocabulary dictionary for a variable (new as of 8.0)

upsh and upop

Use upsh and upop to place or obtain the values of memory locations, registers, and integer values on the stack. dup and drop are like the Forth primitives.

var, code, and mcode

Using code and mcode is slightly more difficult:

```
code 'forthname', assemblyname
mcode 'forthname', assemblyname
var 'forthname', assemblyname, value
```

Use next in place of RET for readability.

fdef, mdef, and vdef

As of RetroForth 8.0, word headers are no longer kept in the same code space as the definition contents. Therefore, when coding new assembly language primitives, it is not sufficient to just use var, code, or mcode macros. You must also update the core.dict file with corresponding fdef, mdef, or vdef macro invocations, to provide the proper dictionary linkage. The syntax for each is as follows:

```
fdef 'forthname', assemblyname
mdef 'forthname', assemblyname
vdef 'forthname', assemblyname
```

Bootstrapping

RetroForth's bootstrap process is a bit different than in most Forths. We define a small core kernel containing the basic compiler and a few I/O words. Everything else is built from source code upon startup. This source code is included directly into the binary. This allows the binaries to be self-contained easily.

The source code is generally placed in a file named core.f before inclusion into the core. The build process regenerates this from several smaller files in the source/blocks directory.

These smaller files are what define the RetroForth language and environment. Let's take a quick look at them:

base Contains the RetroForth language (required)
welcome Displays a nice welcome message (optional)
colors Support for foreground/background colors (optional)
red The editor (optional, needs colors)
red-view The view portion of the editor (optional, needs red)
blockio Load and save blocks (optional, needs red)
edit The editor UI (optional, needs red-view)

The only one required is base. The default system includes the others as they provide a lot of useful functionality in terms of making RetroForth a useable development environment.

Note: the source blocks need to leave the address of the custom interpreter on the stack. The default base provides this.

Ports

For the first five years of development, RetroForth was a purely Native forth. This changed during the development of the Release 6 codebase, when Tom Novelli ported it to Linux. Since then it's been ported to numerous other operating systems.

During the development of Release 7, we decided to allow some interaction with the host OS. This is provided in various ways. Under Linux, FreeBSD, and Dex4u you can use system calls, and under Windows and Generic+FFI you can use external libraries. On both the Native and L4 ports, you can directly access the hardware. Further details are provided in the Glossary section.

In addition to this, there are various dependencies when building. The next few pages in this handbook are dedicated to explaining the installation and build process for each of the ports.

Please note that you can rebuild the source for Linux, FreeBSD, and Windows by using fasm on the rf.asm file in the source directory. For the other ports, use the methods described below.

Linux

To get started, run the ***rf*** binary in the ***bin*** subdirectory. In many cases, doing the following at a shell will work:

```
./bin/rf
```

Adjust the path as needed. If you like RetroForth, we recommend keeping a copy in your *~/bin* directory or somewhere else in your path.

Building a custom version

Requirements:

- The source code (included with all copies of RetroForth)
- FASM (from http://www.flatassembler.net)
- GNU make (The BSD make might work, but this has not been tested)

To build:

- Edit the "Makefile" as neccessary
- Run "make" at a command line
- Test the newly generated binary

If you didn't modify the makefile, the binary will be in the ***bin*** subdirectory. It will be named ***rf*** by default.

FreeBSD

To get started, run the ***rf*** binary in the ***bin*** subdirectory. In many cases, doing the following at a shell will work:

```
./bin/rf
```

Adjust the path as needed. If you like RetroForth, we recommend keeping a copy in your *~/bin* directory or somewhere else in your path.

Building a custom version

Requirements:

- The source code (included with all copies of RetroForth)
- FASM (from http://www.flatassembler.net)
- GNU make (The BSD make might work, but this has not been tested)
- Linux emulation (to allow use of FASM)

To build:

- Edit the *Makefile* as neccessary
- Run *make* at a command line
- Do *brandelf -t FreeBSD bin/rf* (adjust the path as needed)
- Test the newly generated binary

If you didn't modify the makefile, the binary will be in the ***bin*** subdirectory. It will be named ***rf*** by default.

Windows

To get started, run the ***rf.exe*** binary in the ***bin*** subdirectory. One way to do this is to double click on the ***bin*** directory and then to double click on the ***rf.exe*** file that appears. If you like RetroForth, we recommend keeping a copy somewhere that you can run it easily.

Building a custom version

Requirements:

- The source code (included with all copies of RetroForth)
- FASM (from http://www.flatassembler.net)
- GNU make is recommended, but not required

To build:

- Edit the *Makefile* or build.bat as neccessary
- Run *make* or *build.bat* at a command line
- Test the newly generated binary

If you didn't modify the makefile or build.bat, the binary will be in the ***bin*** subdirectory. It will be named ***rf.exe*** by default.

Generic

To get started, run the ***rf*** binary in the ***bin*** subdirectory. If you are using Linux, doing the following at a shell should work:

```
./bin/rf
```

Adjust the path as needed. If you like RetroForth, we recommend keeping a copy in your *~/bin* directory or somewhere else in your path.

Building a custom version

Requirements:

- The source code (included with all copies of RetroForth)
- FASM (from http://www.flatassembler.net)
- GNU make (The BSD make might work, but this has not been tested)
- GCC (or another C compiler with ELF support)

To build:

- Edit the *Makefile* as neccessary
- Run *make* at a command line
- Test the newly generated binary

If you didn't modify the makefile, the binary will be in the ***bin*** subdirectory. It will be named ***rf*** by default.

Generic + FFI

To get started, run the ***rf*** binary in the ***bin*** subdirectory. If you are using Linux, doing the following at a shell should work:

```
./bin/rf
```

Adjust the path as needed. If you like RetroForth, we recommend keeping a copy in your *~/bin* directory or somewhere else in your path.

Building a custom version

Requirements:

- The source code (included with all copies of RetroForth)
- FASM (from http://www.flatassembler.net)
- GNU make (The BSD make might work, but this has not been tested)
- GCC (or another C compiler with ELF support)
- A copy of libdl.so

To build:

- Edit the *Makefile* as neccessary
- Run *make* at a command line
- Test the newly generated binary

If you didn't modify the makefile, the binary will be in the ***bin*** subdirectory. It will be named ***rf*** by default.

Dex4u

To get started, put the ***rf.dex*** binary (from the ***bin*** subdirectory) on a floppy or CD. At the Dex4u shell, do:

```
a:
run rf.dex
```

Building a custom version

Requirements:

- The source code (included with all copies of RetroForth)
- FASM (from http://www.flatassembler.net)
- GNU make is recommended

To build:

- Edit the *Makefile* as neccessary
- Run *make* at a command line
- Test the newly generated binary

If you didn't modify the makefile, the binary will be in the ***bin*** subdirectory. It will be named ***rf.dex*** by default.

Native

Installation of the Native system is fairly straightforward. In the ***bin*** directory there is a file named ***diskimage***. You will need to write this to a floppy. Under Linux or FreeBSD it can be done by:

```
dd if=bin/diskimage of=/dev/fd0
```

Under Windows, use RaWriteWin or a similar tool. Once this is done, you can boot it. The ***diskimage*** can also be used with emulators like Bochs and Qemu.

Building a custom version

Requirements:

- The source code (included with all copies of RetroForth)
- FASM (from http://www.flatassembler.net)
- GNU make is recommended

To build:

- Edit the *Makefile* as neccessary
- Run *make* at a command line
- Make a boot floppy and test

If you didn't modify the makefile, the new diskimage will be in the ***bin*** subdirectory. It will be named ***diskimage*** by default.

L4Ka::Pistachio

This port has the most complex installation process. (Once you set up a boot disk, updating it is easy though)

To build yourself the bootdisk, this is what I did. The steps you'll need to take will depend on how your L4Ka installation is set up, so I just used macro-like string substitutions. You're smart enough to figure that out, though, right?

```
# dd if=/dev/zero of=boot.disk bs=1024 count=1440
# mkfs -t ext2 boot.disk
# mkdir img
# mount -o loop boot.disk img
# mkdir img/boot
# cp $GRUB_LIBS/stage1 img/boot
# cp $GRUB_LIBS/e2fs_stage_1_5 img/boot
# cp $GRUB_LIBS/stage2 img/boot
# cp $L4HOME/ia32-kernel img/boot
# cp $L4HOME/kickstart img/boot
# cp $L4HOME/sigma0 img/boot
```

Now follow the steps required by GRUB to make the boot floppy truly bootable. Later updates simply require copying the bin/rf8.kapp

We will be providing a ready-made diskimage in the future, so this process will become somewhat simpler then.

Building a custom version

Requirements:

- The source code (included with all copies of RetroForth)
- FASM (from http://www.flatassembler.net)
- GNU make (The BSD make might work, but this has not been tested)
- GCC

To build:

- Edit the *Makefile* as neccessary
- Run *make* at a command line
- Copy the newly generated binary to your boot disk and test

If you didn't modify the makefile, the binary will be in the ***bin*** subdirectory. It will be named ***rf8.kapp*** by default.

The Editor

The editor (called *RetroEditor* or *RED*) plays a big role in RetroForth. It's the primary tool for editing blockfiles in the library, and allows a clean, easy to learn way to save and edit code within the RetroForth environment.

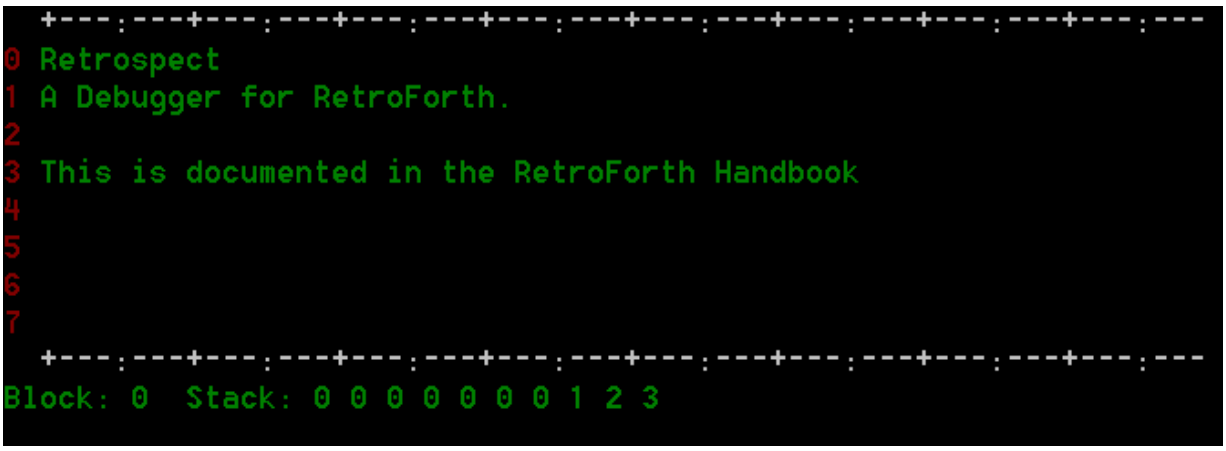
This editor is not new. It's been around since the days of RetroForth 4, but never has it been this closely integrated into RetroForth as a whole. It will take you a bit of practice to become proficient at editing since *RED* is line-oriented.

Blocks in RetroForth are 512 bytes. This gives you 8 lines with 64 characters per line. In most Forths blocks are 1024 bytes. We have some reasons for shorter blocks (for starters they seem to encourage better factoring; for another they map directly to physical sectors in the Native and L4 versions).

So let's take a look at it, and see how to actually use it.

The Interface

Let's start by examining a screenshot of the interface.



This shows the main user interface. There are two horizontal lines, broken by special symbols every four columns. Between them are eight lines, numbered 0 through 7. Below this is a line showing the current block number, and the current stack contents.

Line numbers are in red; they show which line in the block that the code is on. The code itself is shown in greeen, as is the status information below the block. The horizontal column bars are in grey.

What you type appears below the status bar. You can use any forth code in the input area; the output is displayed below your input line. This allows interactive coding and debugging while editing.

You start the editor simply by running edit. The word exit will quit the editor and return you to normal forth. Now that you have some idea as to what's being shown, let's take a look at how to load and save your code.

We provide several words for selecting, opening, and saving blockfiles. The first of these is use. You can specify a filename to load or save this way easily. Simply do:

```
use <filename>
```

And `<filename>` will become the active file. When you use r (read) or w (write), the contents of the block buffer will be written to this file. To make selecting and loading files easier, we provide a word named load. Use it like:

```
load <filename>
```

Then `<filename>` will be selected and read into the block buffer. Please note that doing load, r, or w will create the file if it does not exist.

The final word we will cover here is `blocks`. This word specifies the number of blocks to read or write. You can use it like:

```
12 blocks
```

The minimum number of blocks is 1; the maximum is 256.

Navigating

It's easy to navigate through the blocks once they're loaded. We provide a grand total of three words for this activity.

To go to the next block, run ***n***. To select the previous block, use ***p***. And finally, to select a specific block you can use ***s***. The word ***s*** takes the block number from the stack.

Editing

Now that you finally know how to start the editor, select a blockfile, and choose the block you'd like to edit, how can you get your code into it? Relax, it's not that hard.

You can use the words i and ia to insert/overwrite text. These are used like:

```
| Put the words "Hello World" on line 3, starting
| at column 0
3 i Hello World

| Put the words "Hello, World" on line 6, starting
| at column 10
9 6 ia Hello, World
```

Note that both line and column numbers start at 0.

You can delete a line using d, or the entire block using x. If you need to erase all of the blocks, use new. Examples:

```
| Delete line 3
3 d

| Delete everything in this block
x

| Delete all of the blocks
new
```

These form the core of the editing experience. It's also important to review how to actually run code in the blocks. We provide three words for this.

```
| evaluate a specific line
4 el

| evaluate the entire block (as a single line)
eb

| evaluate the entire block, one line at a time
e
```

The preferred way to evaluate a block is to use ***e***. If you need to evaluate several blocks, chain them by doing: ***n e*** at the end of each block.

Note: it is recommended that you start the actual code in block 1 and use block 0 for comments about the blockfile. If you do this you can use \ when loading a blockfile to evaluate it automatically. If you ***load*** a normal file, you can use **\f** to evaluate it.

The Library

A very important part of RetroForth is the code library. This is a collection of applications and extensions that can be quickly loaded. The following block files make up the library:

allegro	Bindings for the Allegro graphics library
assembler	Albert van der Horst's 386 assembler
ans	Some words from the DPANS CORE and CORE EXT specs
editor	Some extensions to the retroeditor
linux	Symbolic names for the Linux system calls and some related words for using retroforth as a shell
math	Enhanced math support
marker	mark/empty
memory	Dynamic memory allocation for Windows and Linux
retrospect	A debugger
strings	Enhanced string support
sockets	Basic socket I/O words for Linux
wikicore	A wiki markup to html convertor

Retrospect, a debugger

Retrospect is a debugger for RetroForth. It provides a number of useful services such as hex dumps, disassembler, assembler, and basic profiling of code.

Generally you can load it by doing:

```
load lib/retrospect \
```

Of course, you may need to adjust the path to the blockfile, depending on how you have things set up.

Hex Dumps

The ability to look into a buffer or examine a region of memory is often helpful. With Retrospect, this is provided by dump.

```
' words 42 dump
4242f4 | E8 BC D4 FD FF E8 4A FA FF FF E8 B5 FD FF FF E8 | ......J.........
424304 | 19 FA FF FF E8 77 D2 FD FF 09 00 00 00 E8 A2 FA | .....w..........
424314 | FF FF E8 06 FA FF FF 48 E8 31 | .......H.1
```

This format shows the address on the left, the hex values (16 per row) in the middle, and the ASCII symbols on the right.

```
dump ( address count -- )
```

Disassembler

Many times you'll want to take a peek at an earlier definition. Rather than searching through numerous blocks for a definition, or trying to recall that quick one-liner you wrote, you can use the word inspect to see the compiled code in a human-readable form.

Try this:

```
inspect words
```

You'll see some output like:

```
words:
4242f4: call 4017b5  ; last
4242f9: call 423d48  ; @
4242fe: call 4240b8  ; 0;
424303: call 423d21  ; dup
424308: call 401584  ; literal
42430d: dd 9         ; 9
424311: call 423db8  ; +
424316: call 423d21  ; dup
42431b: dec eax      ; 1-
42431c: call 423d52  ; c@
424321: call 4010a5  ; type
424326: call 4242b4  ; space
42432b: jmp 4242f9
```

In the first column are the addresses of each decompiled element. The assembly operation is next, and then on the right is the Forth version. The colors in the Forth section are for readability. Word names are in green, numbers in brownish-yellow, and macros are in blue.

Note: inspect is mainly intended for use with normal colon definitions. You can inspect words in the kernel, but don't expect the results to be very readable (at least at this point. We may expand the number of supported instructions in the future.)

Profiling Your Code

Sometimes you will want to optimize your code for size or performance. Retrospect can help here as well with its profiling tools. There are two ways to profile your code: for size and for performance.

```
profile.size <code to compile>
profile.speed <code to compile>
```

The size or number of clock cycles used will be reported.

A Tutorial For New Programmers

The intent of this tutorial is to provide a brief introduction to the core concepts of Forth as implemented in RetroForth. It is only a starting point. Feel free to deviate from the sequences I provide. A free form investigation that is based on your curiosity is probably the best way to learn any language. Forth is especially well adapted to this type of learning.

In the tutorials, I will print the things you need to type in monospaced font, and indent them. RetroForth is case-sensitive, unlike some other Forths; so the words which are built-in must be entered as shown, but words you create can be any combination of case you prefer.

At the end of each line, press the RETURN (or ENTER) key; this causes RetroForth to interpret what you've entered. You might also note that RetroForth doesn't prompt you with `ok` like most other Forths do.

Forth Syntax

Most of the Forth language has a very simple syntax of any computer language. Basically you have two things: words and numbers. There are also two modes of operation. Most of the time you'll be in an interpreter, but you can use the compiler to create new words as well.

Most words take their data from the stack (which is where numbers are placed when they are encountered), but you will also encounter some parsing words which take data from the input stream. (The details of this will be covered later)

Note: there is also a special class of words known as macros. These can only be used when defining new words, so we won't worry about them now. We'll take a closer look at them later in this tutorial.

Words are executed in the order they appear in the code. The following statement, for example, could appear in a Forth program:

```
WAKE.UP EAT.BREAKFAST WORK EAT.DINNER PLAY SLEEP
```

Notice that `WAKE.UP` has a dot between the `WAKE` and `UP`. The dot has no particular meaning to the Forth compiler. I simply used a dot to connect the two words together to make one word, and to make that word easier for a human to read. Forth word's names can use any combination of letters, numbers, or punctuation. We will encounter words with names like:

```
." #s swap ! @ dup . *
```

These are all called words. The word `$%%-GL7OP` is a legal Forth name, although not a very good one. It is up to the programmer to name words in a sensible manner. In general, Forth (and RetroForth in particular) give the programmer ultimate freedom to make whatever design decisions are appropriate, and does not get in the way of making bad decisions.

Now it is time to start RetroForth and begin experimenting. One of Forth's greatest strengths is its interactive, immediate nature.

The Stack

The Forth language is based on the concept of a *stack*. Imagine a stack of blocks with numbers on them. You can add or remove numbers from the top of the stack. You can also rearrange the order of the numbers. Forth uses two stacks.

The data stack is the one used for passing data between Forth words so we will concentrate our attention there. The return stack is another Forth stack that is primarily for internal system use but is often used to store temporary values. In this tutorial, when we refer to the stack, we will be referring to the data stack. For reference, we'll call the return stack RS.

The stack is initially empty. Start up RetroForth, and notice you are greeted by something like:

```
RetroForth 8.0 :: Visit http://www.retroforth.org for updates
```

The interpreter is now awaiting your command. Let's start by putting some numbers on the stack. Type in:

```
23 7 9182
```

Excellent! Now print the number on top of the stack using the Forth word `.`, which is pronounced dot. This is a hard word to write about in a manual because it is just a single period. Enter:

```
.
```

You should see the last number you entered, `9182`, printed. RetroForth has a very handy word for showing you what's on the stack. It is `.s`, which is pronounced *dot S*. The name was constructed from dot for print, and S for stack. If you enter:

```
.s
```

You will see the numbers `0 0 0 0 0 0 0 0 23 7`, in a list. The number at the far right is the one on top of the stack. The word `.s` displays the top ten entries on the stack, so if you have fewer values, the number 0 will be showed instead. Notice that `9182` is not on the stack. The word `.` removes the number on top of the stack before printing it. In contrast, `.s` leaves the stack untouched.

Forth uses the stack to hold data being operated on, and it uses the stack to pass data from word to word. Essentially, a word takes whatever it needs from the stack, and puts whatever its results are on the stack. This is a very powerful aspect of Forth, but one which requires practice to understand. It also means that documenting what each word does to the stack is important and useful.

The standard technique for documenting the effect words have on the stack is by means of a stack diagram. Stack diagrams begin with a left-parenthesis, contain the stack-effect diagram, and end with a right-parenthesis. In Forth, parentheses indicate a comment, and everything between them is ignored. So while you could put whatever you like between parentheses and treat them as ordinary comments, the usual use of parentheses is for stack-comments. For example, the stack-digram for the word 'dot' which we used before, would be:

```
. ( n -- )
```

That is to say, . takes one word off the stack (the n) and puts nothing on the stack. In other words, it consumes the top stack item (hereafter called TOS).

In the examples that follow, you should not type in the comments. When you are programming, of course, use of comments and stack diagrams may make your code more readable and maintainable. Besides the parenthesis, you may use the vertical-bar character `|` as comment to end-of-line. In other words, anything after the `|` on that line is ignored:

```
dup swap    | This is all a comment
```

Note: | comments are not supported in definitions, use the ( ) form instead.

Between examples, you may wish to clear the stack. If you enter `reset`, the stack will be cleared. Since the stack is central to Forth, it is important to be able to alter it easily. Let's look at some more words that manipulate the stack. Enter:

```
777 dup .s
```

You will notice that there are two copies of `777` on the stack. The word `dup` duplicates TOS. This is useful when you want to use the TOS and still have a copy. The stack diagram for `dup` would be:

```
dup ( n -- n n )
```

Another useful word is `swap`. Enter:

```
23 7 .s
swap .s
```

The stack should have `7 23` now. The stack diagram for `swap` would be:

```
swap ( a b -- b a )
```

Now enter:

```
over .s
```

You should see `23 7 23.` The word over causes a copy of the second item on the stack to leapfrog over the first. Its stack diagram would be:

```
over ( a b -- a b a )
```

Here is another commonly used Forth word:

```
drop ( a -- )
```

Can you guess what we will see if we enter:

```
drop .s
```

Another handy word for manipulating the stack is `rot`. Enter:

```
11 22 33 44 .s
rot .s
```

The stack diagram for `rot` is, therefore:

```
rot ( a b c -- b c a )
```

You have now learned the more important stack manipulation words. You will see these in almost every Forth program. I should caution you that if you see too many stack manipulation words being used in your code then you may want to reexamine and perhaps reorganize your code. You will often find that you can avoid excessive stack manipulations by using variables, which will be discussed later. It is also likely that factoring your code -- that is, breaking it into smaller words -- may help reduce the stack juggling.

I have included the stack diagrams for some other useful stack manipulation words. Try experimenting with them by putting numbers on the stack and calling them to get a feel for what they do. Again, the text in parentheses is just a comment and need not be entered.

```
2drop ( a b c -- a )
2dup  ( a b   -- a b a b )
nip   ( a b c -- a c )
tuck  ( a b   -- b a b )
-rot  ( a b c -- c a b )
```

Arithmetic

Simply moving numbers around on a stack can be a lot of fun. Eventually, however, you'll want to do something useful with them. This section describes how to perform arithmetic operations in Forth.

The Forth arithmetic operators work on the numbers currently on top of the stack. If you want to add the top two numbers together, use the Forth word +, pronounced plus. Enter:

```
2 3 + .
2 3 + 10 + .
```

This style of expressing arithmetic operations is called *Reverse Polish Notation,* or RPN. It will already be familiar to those of you with HP calculators. In the following examples, I have put the algebraic equivalent representation in a comment.

Some other arithmetic operators are - * /. Enter:

```
30 5 - .       | 25=30-5
30 5 / .       | 6=30/5
30 5 * .       | 150=30*5
30 5 + 7 / .   | 5=(30+5)/7
```

One thing that you should be aware of is that when you are doing division with integers using /, the remainder is lost. Enter:

```
15 5 / .
17 5 / .
```

This is true in all languages on all computers. When you need to know the remainder of a divide operation. /mod will return the remainder as well as the quotient, whereas mod will only return the remainder. Enter:

```
53 10 /mod .s
7 5 mod .s
```

Some other math words to try out:

```
negate ( a -- -a )
<< ( a n -- (a<<n) ) | Shift bits left
>> ( a n -- (a>>n) ) | Shift bits right
```

Converting Algebraic Expressions to Forth

How do we express complex algebraic expressions in Forth? For example: `20 + (3 * 4)`?

To convert this to Forth you must order the operations in the order of evaluation. In Forth, therefore, this would look like:

```
3 4 * 20 +
```

Evaluation proceeds from left to right in Forth so there is no ambiguity. Compare the following algebraic expressions and their Forth equivalents: (Do not enter these!)

```
(100+50)/2         ==> 100 50 + 2 /
((2*7) + (13*5))   ==> 2 7 * 13 5 * +
```

If any of these expressions puzzle you, try entering them one word at a time, while viewing the stack with `.s`

Defining Words

It's now time to write a small program in Forth. You can do this by defining a new word that is a combination of words we have already learned. Let's define and test a new word that takes the average of two numbers.

We will make use of two new words, **:** (colon), and **;** (semicolon). These words start and end a typical Forth definition. Enter:

```
: AVERAGE ( a b -- avg ) + 2 / ;
```

Congratulations! You have just written a Forth program. Let's look more closely at what just happened. The colon told Forth to add a new word to its list of words. This list is called the Forth dictionary. The name of the new word will be whatever name follows the colon. Any Forth words entered after the name will be compiled into the new word. This continues until the semicolon is reached which finishes the definition.

Let's test this word by entering:

```
10 20 AVERAGE .    | should print 15
```

Once a word has been defined, it can be used to define more words. Let's write a word that tests our word. Enter:

```
: TEST ( -- ) 50 60 AVERAGE . ;
TEST
```

Try combining some of the words you have learned into new Forth definitions of your choice. If you promise not to be overwhelmed, you can get a list of the words that are available for programming by entering:

```
words
```

Don't worry, only a small fraction of these will be used directly in your programs.

Character I/O

Because Forth is not a typed language, the numbers on top of the stack can represent anything. The top number might be how many blue whales are left on Earth or your weight in kilograms. It might also be an ASCII character. Try entering the following:

```
72 emit 105 emit
```

You should see the word "Hi" appear. 72 is an ASCII H and 105 is an i. The word `emit` takes a number on the stack and outputs it as a character. To get the ASCII value of a character, prepend the character with a single-quote. Enter:

```
'W .
'% dup . emit
'A dup .
32 + emit
```

The use of the single-quote character is a bit unusual. It tells the RetroForth interpreter that the character that follows should be converted to the ASCII code representing it, rather than being considered a word. There are other such modifiers in RetroForth which can make inputting numbers simpler:

```
'a     | Gives 97, the ASCII value of 'a'
%1100  | Gives 12, or 1100 binary
$ff    | 255, or hexadecimal FF
&010   | 8, or octal 10
#123   | 123 - decimal 123
```

Using `emit` to output character strings would be very tedious. Luckily there is a better way. Enter:

```
: TOFU ." Yummy bean curd!" ;
TOFU
```

The word `."`, pronounced dot quote, will take everything up to the next quotation mark and print it to the screen. Make sure you leave a space after the first quotation mark. When you want to have text begin on a new line, you can issue a carriage return using the word `cr`. Enter:

```
: SPROUTS ." Miniature vegetables." ;
: MENU cr TOFU cr SPROUTS cr ;
MENU
```

You can emit a blank space with `space`. In other Forths one may output more than one space with the word `spaces`. Let's write one for RetroForth:

```
: spaces repeat space until ;
TOFU SPROUTS
TOFU space SPROUTS
cr 10 spaces TOFU cr 20 spaces SPROUTS
```

Notice that the new word we created, spaces, uses a loop construct. The word repeat starts a series of words which will be run repeatedly. The word until subtracts one from TOS; if it's not zero then it jumps back to just after the repeat. We'll see more of these kinds of words later on.

For character input, Forth uses the word key which corresponds to the word emit for output. key waits for the user to press a key then leaves its value on the stack. Try the following.

```
: TESTKEY ( -- )
  ." Hit a key: " key cr
  ." The ASCII value = " . cr
;
TESTKEY
```

Note: On some computers, the input is buffered so you will need to hit the ENTER key after typing your character.

Text I/O

You learned earlier how to do single character I/O. This section concentrates on using strings of characters. RetroForth has several varieties of strings, and it is good to know when to use each type.

The "normal" string is a Forth string, consisting of an *address, count* pair. That is, it is represented on the stack by an address which points to the start of the character data, and a count of characters. In stack diagrams it is often listed as `( a # -- )`. To create such a string, you may use a double-quote character, `"`. That word parses until it finds another double-quote, and then it puts the address and count on the stack. Inside a colon-definition, a special version called `s"` should be used instead.

```
" This is a string" type cr
: STR s" Hi there!" ;

STR type cr
```

The word `type` prints out the Forth string on top of the stack.

The second type of string supported by RetroForth is the ASCIIZ or zero-terminated string. That is, a string which is terminated by a NUL byte (the ASCII value 0). These are the native strings for C, and both Windows and Linux API functions expect such strings. Use the word zt to temporarily convert a Forth string to an ASCIIZ string.

```
" RetroForth rocks!" zt    | Make an ASCIIZ string
```

Variables

Forth does not rely as heavily on the use of variables as other compiled languages. This is because values normally reside on the stack. There are situations, of course, where variables are required. To create a variable, use the word `variable` as follows:

```
variable MY-VAR
```

This created a variable named `MY-VAR`. A space in memory is now reserved to hold its 32-bit value. The word variable is what's known as a defining word since it creates new words in the dictionary. Now enter:

```
MY-VAR .
```

The number you see is the address, or location, of the memory that was reserved for `MY-VAR`. To store data into memory you use the word `!`, pronounced store. It looks like an exclamation point, but to a Forth programmer it is the way to write 32-bit data to memory. To read the value contained in memory at a given address, use the Forth word `@`, pronounced fetch. Try entering the following:

```
513 MY-VAR !
MY-VAR @ .
```

This sets the variable `MY-VAR` to `513`, then reads the value back and prints it. You can also create a variable and set its value at the same time:

```
513 variable: MY-VAR2
MY-VAR2 @ .
```

The stack diagrams for these words follows:

```
@ ( addr -- val )
! ( val addr -- )
variable  ( [name] -- )
variable: ( val [name] -- )
```

Imagine you are writing a game and you want to keep track of the highest score. You could keep the highest score in a variable. When you reported a new score, you could check it against the highest score. Try entering this code:

```
variable HIGH-SCORE

: max ( a b -- c ) 2dup <if nip ;; then drop ;

: REPORT.SCORE ( score -- )
  dup cr ." Your Score = " . cr
  HIGH-SCORE @ max ( calculate new high )
  dup ." Highest Score = " . cr
  HIGH-SCORE ! ( update variable )
;
```

Save the file to disk, then load this code using the word `load`. Test your word as follows:

```
123  REPORT.SCORE
9845 REPORT.SCORE
534  REPORT.SCORE
```

The Forth words `@` and `!` work on 32-bit quantities. `c@` and `c!` work on characters which are usually for 8-bit bytes. The `c` stands for character since ASCII characters are 8-bit numbers.

A word of warning about fetching and storing to memory: You have now learned enough about Forth to be dangerous. The operation of a computer is based on having the right numbers in the right place in memory. You now know how to write new numbers to any place in memory. Since an address is just a number, you could, but shouldn't, enter:

```
73 253000 ! ( Do NOT do this. )
```

The `253000` would be treated as an address and you would set that memory location to `73`. I have no idea what will happen after that, maybe nothing. This would be like firing a rifle through the walls of your apartment building. You don't know who or what you are going to hit. Since you share memory with other programs including the operating system, you could easily cause the computer to behave strangely, even crash. Don't let this bother you too much, however. Crashing a computer, unlike crashing a car, does not hurt the computer. You just have to reboot. The worst that could happen is that if you crash while the computer is writing to a disk, you could lose a file. That's why we make backups. This same potential problem exists in any powerful language, not just Forth. This might be less likely in BASIC, however, because BASIC protects you from a lot of things, including the danger of writing powerful programs.

Constants

If you have a number that is appearing often in your program, it's recommended you define it as a constant. We do this just like defining a word. Enter:

```
: MAX_CHARS 128 ;
MAX_CHARS .
```

We just defined a word called `MAX_CHARS` that returns a specific value. It cannot be directly changed unless you edit the program and recompile. Using constants can improve the readability of your programs and reduce some bugs. Imagine if you refer to the number 128 very often in your program, say 8 times. Then you decide to change this number to 256. If you globally change 128 to 256 you might change something you didn't intend. If you change it by hand you might miss one, especially if your program occupies more than one file. Using constant will make it easy to change. The code that results is equally as fast and small as putting the numbers in directly. I recommend defining a constant for almost any number used more than two or three times.

Loops

We've mentioned one of the looping constructs before. Now we'll examine them more closely. RetroForth has a completely different set of loop constructs than ANS Forth, so you'll want to pay attention.

Most loops begin with the word repeat. The first kind of loop is between `repeat ... until`. This is a simple counted loop that exits when the TOS is zero. If it's not zero, it subtracts one from it and loops back to repeat. Try this:

```
: COUNTDOWN ( N -- ) repeat dup . cr until ;
16 COUNTDOWN
```

This word will count down from N to zero. The second type of loop RetroForth has is an unconditional loop, `repeat ... again`. This keeps going until you break out of it, perhaps by using `;;` or pressing Ctrl-C :

```
: MAIN-LOOP repeat ." looping again ..." cr again ;
```

Consider the following word for doing character graphics. Enter:

```
: PLOT# ( n -- ) repeat '- emit until ;
cr 9 PLOT# 37 PLOT#
```

RetroForth also has a `for ... next` construct similar to ANS Forth. It is used like this:

```
: looper 10 for ." Iteration #" r . cr next ;
```

for ... next puts the counter on the return stack. You can use the word r to obtain this value. Like repeat ... until it counts down to zero.

Conditionals

This section is concerned with decision making. How can your program react to values on the stack and decide how to manipulate them?

Most Forths provide a few conditionals and a word named if to handle this. However, in RetroForth, the conditionals are bound to if. RetroForth therefore has special forms of `if`, which are used directly with a condition. In California, the drinking age for alcohol is 21. You could write a simple word now to help bartenders. Enter:

```
: DRINK? 20 >if ." OK" cr ;; then ." Underage!" cr ;

20 DRINK?
21 DRINK?
43 DRINK?
```

Here you are introduced to the `if/;;/then` structure of RetroForth's conditional statements. The word `;;` is used to force an exit to a word. It's useful in both conditional constructs and looping constructs. Other useful "if" constructs are:

```
<if     If second stack item is less than TOS
=if     Two top stack items are equal
<>if    Top two stack items are not equal
?if     Is the conditional flag TRUE?
true    Set the conditional flag to TRUE
false   Set the conditional flag to FALSE
;;      Exit the current word
0;      Exit if TOS=0, otherwise continue execution
```