



durexForth

Operators Manual



Developed by
Ravelli/Durex

durex

Contents

1	Introduction	4
1.1	Forth, the Language	4
1.1.1	Why Forth?	4
1.1.2	Comparing to other Forths	4
1.2	Appetizers	5
1.2.1	Graphics	5
1.2.2	Fractals	5
1.2.3	Music	5
1.2.4	Sprites	5
2	Tutorial	6
2.1	Interpreter	6
2.2	Editor	6
2.3	Assembler	7
2.4	Console I/O Example	7
2.5	Avoiding Stack Crashes	8
2.5.1	Commenting	8
2.5.2	Stack Checks	8
2.6	Configuring durexForth	8
2.6.1	Stripping Modules	8
2.6.2	Custom Start-Up	9
2.7	How to Learn More	9
2.7.1	Internet Resources	9
2.7.2	Other	9
3	Editor	10
3.1	Key Presses	10
3.1.1	Inserting Text	10
3.1.2	Navigation	10
3.1.3	Saving & Quitting	11
3.1.4	Text Manipulation	11
4	Forth Words	12
4.1	Stack Manipulation	12
4.2	Utility	13
4.3	Mathematics	13
4.4	Signed Mathematics	14
4.5	Logic	14

4.6	Memory	15
4.7	Compiling	15
4.8	Variables	16
4.8.1	Values	16
4.8.2	Variables	16
4.8.3	Arrays	16
4.9	Control Flow	17
4.10	Input	17
4.11	Editing	18
4.12	Strings	18
4.13	Vectorized Execution	18
4.14	Debugging	18
4.15	System State	19
4.16	Disk I/O	19
4.17	Kernel Calls	19
5	Graphics	20
5.1	Turtle Graphics	20
5.2	High-Resolution Graphics	20
6	Music	22
6.1	Music Macro Language	22
6.2	Commands	22
A	Assembler Mnemonics	23
B	Memory Map	24
C	Word Anatomy	25
C.1	Inspecting a Word	25
C.2	Header	25
C.3	Code	26

Chapter 1

Introduction

1.1 Forth, the Language

1.1.1 Why Forth?

Forth is a unique language. What is so special about it? It is a very low-level and minimal language with its fair share of quirks and rough edges. At the same time, it is easy to scale it up to become a very high-level and domain-specific language, much like Lisp.

Compared to C64 Basic, Forth is more attractive in almost every way. It is a lot faster, more memory effective, and more powerful.

Compared to C, the story is a little different. It's hard to make a fair comparison. Theoretically Forth code can be very memory efficient, and it's possible to make Forth code that is leaner than C code. But it is also true that cc65 code is generally faster than Forth code.

The main advantage of Forth is that the environment runs on the actual machine. It would not be a lot of fun to use a C compiler that runs on a standard C64. But with Forth, it's possible to create an entire development suite with editor, compiler and assembler that runs entirely on the C64.

Another advantage is that Forth has an interpreter. Compared to cross-compiling, it is really nice to make small edits and tweaks without going through the entire edit-compile-link-transfer-boot-run cycle.

For a Forth introduction, please refer to the excellent [Starting Forth](#) by Leo Brodie. As a follow-up, I recommend [Thinking Forth](#) by the same author.

1.1.2 Comparing to other Forths

There are other Forths for c64, most notably Blazin' Forth. Blazin' Forth is excellent, but durexForth has some advantages:

- Store your Forth sources as text files - no crazy block file system.
- durexForth is smaller.
- durexForth is faster.
- The durexForth editor is a vi clone.

- durexForth is open source (available at [Github](#)).

1.2 Appetizers

Some demonstration files are included as appetizers.

1.2.1 Graphics

The gfxdemo package demonstrates the high-resolution graphics, with some examples adapted from the book "Step-By-Step Programming C64 Graphics" by Phil Cornes. Show the demos by entering:

```
s" gfxdemo" load
```

When a demo has finished drawing, press any key to continue.

1.2.2 Fractals

The fractal package demonstrates turtle graphics. You can load the package as follows:

```
s" fractal" load
```

The fractals can then be shown by typing `koch weed1 bush1 bush2`. When a fractal have finished drawing, press any key to continue.

1.2.3 Music

The mmldemo package demonstrates the MML music capabilities. To run:

```
s" mmldemo" load frere-jaques sarias-song
```

1.2.4 Sprites

The sprite package adds functionality for defining and displaying sprites. To run a demo:

```
s" spritedemo" load
```

Chapter 2

Tutorial

2.1 Interpreter

Start up `durexForth`. If loaded successfully, it will greet you with a friendly `ok`. You have landed in the interpreter!

Let's warm it up a little. Enter `1` (followed by return). You have now put a digit on the stack. This can be verified by the command `.s`, which will print out the stack. Now enter `.` to pop the digit and print it to screen, followed by `.s` to verify that the stack is empty.

Now some arithmetics. `1000 a * .` will calculate $a \times \$1000$ and print the result on the screen. `6502 100 / 1- .` will calculate and print $(\$6502/\$100) - 1$.

Let's define a word `bg!` for setting the border color...

```
: bg! d020 c! ;
```

Now try entering `1 bg!` to change the border color to white. Then, try changing it back again with `0 bg!`.

2.2 Editor

The editor (fully described in chapter 3) is convenient for editing larger pieces of code. With it, you keep an entire source file loaded in RAM, and you can recompile and test it easily.

Start the editor by typing `vi`. You will enter the pink editor screen.

To enter text, first press `i` to enter insert mode. This mode allows you to insert text into the buffer. You can see that it's active on the `I` that appears in the lower left corner.

This is a good start for making a program. But first, let's get rid of the `"bg!"` word we created previously. Enter:

```
forget bg!
```

...and press `←` to leave insert mode. The line you entered forgets the `bg!` word that you defined in the last section, and everything defined after it. Let's try out if it works.

First, quit the editor by pressing `:q`. You should now be back in the interpreter screen. Verify that the word `bg!` still exists by entering `0 bg!`, `1 bg!` like you did before. Then, jump back to the editor using the command `vi`. You should return to your edit buffer with the lonely `forget bg!` line.

Now, compile and run the buffer by pressing `F7`. You will be thrown out to the interpreter again. Entering `bg!` should now give you the error `bg!?`. Success — we have forgotten the `bg!` word. Now, get back into the editor by entering `vi`.

Beneath `forget bg!`, add the following lines:

```
: flash begin d020 c@ 1+ d020 c! again ;
flash
```

`flash` will cycle the border color infinitely. Before trying it out, go up and change `forget bg!` to `forget flash`. This makes sure you won't run out of RAM, no matter how many times you recompile the program. Now press `F7` to compile and run. If everything is entered right, you will be facing a wonderful color cycle.

To get back into the editor, press Restore key.

2.3 Assembler

If you need to flash as fast as possible, it is possible to use the `durexForth` assembler to generate machine code. `:asm` and `;asm` define a code word, just like `:` and `;` define Forth words. Within a code word, you can use assembler mnemonics.

```
:asm flash
here # push current addr
d020 inc,
jmp, # jump to pushed addr
;asm
flash
```

Alternatively, it is possible to use inline assembly within regular Forth words:

```
: flash begin [ d020 inc, ] again ;
flash
```

Note: As the `x` register contains the parameter stack depth, it is important that your assembly code leaves it unchanged.

2.4 Console I/O Example

This piece of code reads from keyboard and sends back the chars to screen:

```
: foo key emit recurse ;
foo
```

2.5 Avoiding Stack Crashes

durexForth should be one of the fastest and leanest Forths for the C64. To achieve this, there are not too many niceties for beginners. For example, compiled code has no checks for stack overflow and underflow. This means that the system may crash if you do too many pops or pushes. This is not much of a problem for an experienced Forth programmer, but until you reach that stage, handle the stack with care.

2.5.1 Commenting

One helpful technique to avoid stack crashes is to add comments about stack usage. In this example, we imagine a graphics word "drawbox" that draws a black box. (color --) indicates that it takes one argument on stack, and on exit it should leave nothing on the stack. The comments inside the word indicate what the stack looks like after the line has executed.

```
: drawbox ( color -- )
10 begin dup 20 < while # color x
10 begin dup 20 < while # color x y
2dup # color x y x y
4 pick # color x y x y color
blkcol # color x y
1+ repeat drop # color x
1+ repeat 2drop ;
```

Once the word is working, it may be nice to again remove the # comments as they are no longer very interesting to read.

2.5.2 Stack Checks

Another useful technique during development is to check at the end of your main loop that the stack depth is what you expect it to. This will catch stack underflows and overflows.

```
: mainloop begin
# do stuff here...
depth if ." err" exit then
again ;
```

2.6 Configuring durexForth

2.6.1 Stripping Modules

By default, durexForth boots up with all modules pre-compiled in RAM:

doloop Do-loop words.

debug Words for debugging.

asm The assembler.

vi The text editor.

ls List disk contents.

gfx Graphics module.

To reduce RAM usage, you may make a stripped-down version of `durexForth`. Do this by following these steps:

1. Issue `forget modules` to forget all modules.
2. Optionally re-add the `modules` marker with `header modules`.
3. One by one, load the modules you want included with your new Forth.
(E.g. `s" debug" load`)
4. Save the new system with e.g. `s" acmeforth" save-forth`.

2.6.2 Custom Start-Up

You may launch a word automatically at start-up by setting the variable `start` to the execution token of the word. Example: `' megademo start !`

To save the new configuration to disk, use `save-forth`.

2.7 How to Learn More

2.7.1 Internet Resources

Books and Papers

- [Starting Forth](#)
- [Thinking Forth](#)
- [Moving Forth: a series on writing Forth kernels](#)
- [Blazin' Forth — An inside look at the Blazin' Forth compiler](#)
- [The Evolution of FORTH, an unusual language](#)
- [A Beginner's Guide to Forth](#)

Other Forths

- [colorForth](#)
- [JONESFORTH](#)
- [colorForthRay.info — How.to: with Ray St. Marie](#)

2.7.2 Other

- [durexForth source code](#)

Chapter 3

Editor

The editor is a vi clone. Launch it by entering `s" foo" vi` in the interpreter (foo being the file you want to edit). You may also enter `vi` with no parameters on stack - in that case, it will create a text file named "untitled". For more info about vi style editing, see [the Vim web site](#).

The position of the editor buffer is controlled by the variable `bufstart`. The default address is \$6000.

3.1 Key Presses

3.1.1 Inserting Text

Following commands enter insert mode. Insert mode allows you to insert text. It can be exited by pressing `←`.

i Insert text.

a Append text.

o Open new line after cursor line.

O Open new line on cursor line.

cw Change word.

3.1.2 Navigation

h j k l Cursor left, down, up, right.

Cursor Keys ...also work fine.

Ctrl+u Half page up.

Ctrl+d Half page down.

b Go to previous word.

w Go to next word.

0 Go to line start.

\$ Go to line end.

g Go to start of file.

G Go to end of file.

3.1.3 Saving & Quitting

After quitting, the editor can be re-opened with Forth command **vi**, and it will resume operations with the edit buffer preserved.

ZZ Save and exit.

:q Exit.

:w Save. (Must be followed by return.)

:w!filename Save as.

F7 Compile and run editor contents. Press Restore key to return to editor.

3.1.4 Text Manipulation

r Replace character under cursor.

x Delete character.

X Backspace-delete character.

dw Delete word.

dd Cut line.

yy Yank (copy) line.

p Paste line below cursor position.

P Paste line on cursor position.

J Join lines.

Chapter 4

Forth Words

4.1 Stack Manipulation

drop (**a** –) Drop top of stack.

dup (**a** – **a a**) Duplicate top of stack.

swap (**a b** – **b a**) Swap top stack elements.

over (**a b** – **a b a**) Make a copy of the second item and push it on top.

rot (**a b c** – **b c a**) Rotate the third item to the top.

-rot (**a b c** – **c a b**) rot rot

2drop (**a b** –) Drop two topmost stack elements.

2dup (**a b** – **a b a b**) Duplicate two topmost stack elements.

?dup (**a** – **a a?**) Dup a if a differs from 0.

nip (**a b** – **b**) swap drop

tuck (**a b** – **b a b**) dup -rot

pick ($x_u \dots x_1 x_0$ **u** – $x_u \dots x_1 x_0 x_u$) Pick from stack element with depth **u** to top of stack.

>r (**a** –) Move value from top of parameter stack to top of return stack.

r> (– **a**) Move value from top of return stack to top of parameter stack.

r@ (– **a**) Copy value from top of return stack to top of parameter stack.

depth (– **n**) **n** is the number of single-cell values contained in the data stack before **n** was placed on the stack.

sp0 (– **addr**) The bottom address of the LSB section of the parameter stack.

sp1 (– **addr**) The bottom address of the MSB section of the parameter stack.

4.2 Utility

. (**a** -) Print top value of stack.
.s See stack contents.
emit (**a** -) Print top byte of stack as a PETSCII character.
Comment to end of line.
(Start multi-line comment.
) End multi-line comment.
bl Space character (value).
space Prints a space (**bl emit**).

4.3 Mathematics

These words assume that the lowest number is 0 and highest is FFFF.

1+ (**a** - **b**) Increase top of stack value by 1.
1- (**a** - **b**) Decrease top of stack value by 1.
2+ (**a** - **b**) Increase top of stack value by 2.
2* (**a** - **b**) Fast multiply top of stack value by 2.
2/ (**a** - **b**) Fast divide top of stack value by 2.
100/ (**a** - **b**) Divides top of stack value by \$100.
+! (**n** **a** -) Add n to memory address a.
+ (**a** **b** - **c**) Add a and b.
- (**a** **b** - **c**) Subtract b from a.
***** (**a** **b** - **c**) Multiply a with b.
d* (**a** **b** - **msw lsw**) 32-bit multiply a with b.
um/mod (**msw lsw d** - **r q**) Divide 32-bit number by d, giving remainder r and quotient q.
/mod (**a b** - **r q**) Divide a with b, giving remainder r and quotient q.
/ (**a b** - **q**) Divide a with b.
mod (**a b** - **r**) Remainder of a divided by b.
***/** (**a b c** - **q**) Multiply a with b, then divide by c, using a 32-bit intermediary.
***/mod** (**a b c** - **r q**) Like *//, but also keeping remainder r.

`< (a b - c)` Is a less than b?
`> (a b - c)` Is a greater than b?
`>= (a b - c)` Is a greater than or equal to b?
`<= (a b - c)` Is a less than or equal to b?
`lshift (a b - c)` Binary shift a left by b.
`rshift (a b - c)` Binary shift a right by b.
`base (value)` Numerical base.
`decimal` Sets base to 10.
`hex` Sets base to 16.

4.4 Signed Mathematics

These words treat numbers like they are signed, meaning that numbers 8000 - FFFF are negative and 0 - 7FFF are positive. (E.g., FFFF means -1.)

`0< (a - b)` Is a negative?
`negate (a - b)` Negates a.
`abs (a - b)` Gives absolute value of a.
`s< (a b - c)` Is a less than b? (Signed comparison.)
`s> (a b - c)` Is a greater than b? (Signed comparison.)

4.5 Logic

`0= (a - flag)` Is a equal to zero?
`0<> (a - flag)` Is a not equal to 0?
`= (a b - flag)` Is a equal to b?
`<> (a b - flag)` Does a differ from b?
`and (a b - c)` Binary and.
`or (a b - c)` Binary or.
`xor (a b - c)` Binary exclusive or.
`invert (a - b)` Flip all bits of a.

4.6 Memory

! (**value address** –) Store 16-bit value at address.

@ (**address** – **value**) Fetch 16-bit value from address.

c! (**value address** –) Store 8-bit value at address.

c@ (**address** – **value**) Fetch 8-bit value from address.

fill (**byte addr len** –) Fill range [addr, len + addr) with byte value.

cmove (**src dst len** –) Copies a region of memory **len** bytes long, starting at **src**, to memory beginning at **dst**. The copy begins at **src** and continues towards high memory.

cmove> (**src dst len** –) Like **cmove**, except that the copy proceeds from higher to lower memory.

forget xxx Forget Forth word **xxx** and everything defined after it.

4.7 Compiling

: Start compiling Forth word at **here** position.

; End compiling.

, (**n** –) Write word on stack to **here** position and increase **here** by 2.

c, (**n** –) Write byte on stack to **here** position and increase **here** by 1.

literal (**n** –) Compile a value from the stack as a literal value. Typical use:
 x ... [a b *] literal ... ;

[char] **c** Compile character **c** as a literal value.

[(–) Leave compile mode. Execute the following words immediately instead of compiling them.

] (–) Return to compile mode.

immediate Mark the word being compiled as immediate (i.e. inside colon definitions, it will be executed immediately instead of compiled).

no-tce Do not perform tail call elimination (the practice of substituting jsr/rts with jmp) if this word is the jsr target.

['] name (– **xt**) Place name's execution token **xt** on the stack. The execution token returned by the compiled phrase **['] x** is the same value returned by **' x** outside of compilation state. Typical use: **: x ... ['] name ... ;**

compile, (**xt** –) Append **jsr xt** to the word being compiled. Typical use: **: recurse immed latest @ >cfa compile, ;**

postpone xxx Compile the compilation semantics (instead of interpretation semantics) of xxx. Typical use:

```
: endif postpone then ; immediate
: x ... if ... endif ... ;
```

header xxx Create a dictionary header with name xxx.

create xxx/does> Create a word creating word xxx with custom behavior specified after **does>**. For further description, see "Starting Forth."

4.8 Variables

4.8.1 Values

Values are fast to read, slow to write. Use values for variables that are rarely changed.

1 value foo Create value foo and set it to 1.

foo Fetch value of foo.

0 to foo Set foo to 0.

4.8.2 Variables

Variables are faster to write to than values.

variable bar Define variable bar.

bar @ Fetch value of bar.

1 bar ! Set bar to 1.

4.8.3 Arrays

10 allot value foo Allocate 10 bytes to array foo.

1 foo 2 + ! Store 1 in position 2 of foo.

foo dump See contents of foo.

It is also possible to build arrays using **create**. The initialization is easier, but access is slightly different:

```
create 2powtable
1 c, 2 c, 4 c, 8 c,
10 c, 20 c, 40 c, 80 c,
: 2pow ( n -- 2**n ) ['] 2powtable + c@ ;
```


4.9 Control Flow

Control functions only work in compile mode, not in interpreter.

if ... then condition IF true-part THEN rest

if ... else ... then condition IF true-part ELSE false-part THEN rest

do ... loop Start a loop with index *i* and limit. Example:

```
: print0to7 8 0 do i . loop ;
```

do ... +loop Start a loop with a custom increment. Example:

```
( prints odd numbers from 1 to n )  
: printoddnnumbers (n -- ) 1 do i . 2 +loop ;
```

i, j Variables are to be used inside **do .. loop** constructs. *i* gives inner loop index, *j* gives outer loop index.

begin ... again Infinite loop.

begin ... until BEGIN loop-part condition UNTIL.

Loop until condition is true.

begin ... while ... repeat BEGIN condition WHILE loop-part REPEAT.

Repeat loop-part while condition is true.

exit Exit function. Typical use: `: X test IF EXIT THEN ... ;`

recurse Jump to the start of the word being compiled.

case ... endcase, of ... endof Switch statements.

```
: tellno ( n -- )  
case  
1 of ." one" endof  
2 of ." two" endof  
3 of ." three" endof  
." other"  
endcase
```

4.10 Input

key (– n) Reads a character from current input (can be keyboard, disk or RAM).

word (– addr) Reads a word from input and put the string address on the stack.

interpret (– value) Interprets a word from input and puts it on the stack.

compile-ram (value) Makes DurexForth interpret from RAM. E.g. `c000 compile-ram !` will make DurexForth interpret from `c000`. The string to interpret must be terminated with the bytes `20 00`.

quit Enters an endless loop where DurexForth interprets Forth commands. It is named "quit" since it can be used to quit a program.

4.11 Editing

vi (-) Enter editor. If a buffer is already open, editor will pick up where it left. Otherwise, an untitled buffer will be created.

vi (filenameptr filenamelen -) Edit a file. Try `s" ls" vi`.

4.12 Strings

.(Print a string. Example: `.(foo)`

." Compile-time version of `.(`. Example: `: foo ." bar" ;`

s" (- strptr strlen)

Define a string. Example: `s" foo"`.

tell (strptr strlen -)

Prints a string.

4.13 Vectored Execution

' xxx (- addr) Find execution token of word `xxx`.

lit xxx (- addr) Equal to `'` but used for clarity. Use `' lit , ,` to compile the (run-time) value on top of stack.

execute (xt -) Execute the execution token on top of stack.

loc xxx (- addr) Run-time only: Get adress of word `xxx`.

>cfa (addr - xt) Get execution token (a.k.a. code field adress) of word at adress `addr`.

Example: `' words execute equals loc words >cfa execute equals words`.

4.14 Debugging

Debugging words are loaded with `s" debug" load`.

words List all defined words.

size size foo prints size of `foo`.

dump (**n** –) Memory dump starting at address **n**.

n Continue memory dump where last one stopped.

see **word** Decompile Forth word and print to screen. Try **see see**.

4.15 System State

latest (**variable**) Position of latest defined word.

here (**variable**) Write position of the Forth compiler (usually first unused byte of memory). Many C64 assemblers refer to this as program counter or *****.

4.16 Disk I/O

load (**filenameptr filenamelength** –) Load and execute/compile file.

loadb (**filenameptr filenamelength dst** –) Load binary block to **dst**.

saveb (**start end filenameptr filenamelength** –) Save binary block.

scratch (**filenameptr filenamelength** –) Scratch file.

4.17 Kernel Calls

Safe kernel calls may be done from Forth words using **sys** (**addr** –). The helper variables **ar**, **xr**, **yr** and **sr** can be used to set arguments and get results through the a, x, y and status registers.

Example: **key 0 ar ! ffd2 sys** prints 0 on screen.

Chapter 5

Graphics

As of durexForth v1.2, high-resolution graphics support is included.

5.1 Turtle Graphics

Turtle graphics are mostly known from LOGO, a 1970s programming language. It enables control of a turtle that can move and turn while holding a pen. The turtle graphics library is loaded with `s" turtle" load`.

init (-) Initializes turtle graphics.

forward (**px** -) Moves the turtle **px** pixels forward.

back (**px** -) Moves the turtle **px** pixels back.

left (**deg** -) Rotates the turtle **deg** degrees left.

right (**deg** -) Rotates the turtle **deg** degrees right.

penup (-) Pen up (disables drawing).

pendown (-) Pen down (enables drawing).

5.2 High-Resolution Graphics

The high-resolution graphics library is loaded with `s" gfx" load`. It is inspired by "Step-by-Step Programming Commodore 64: Graphics Book 3." Some demonstrations can be found in `gfxdemo`.

hires (-) Enters the high-resolution drawing mode.

lores (-) Switches back to low-resolution text mode.

clrcol (**colors** -) Clears the high-resolution display using **colors**. **Colors** is a byte value with foreground color in high nibble, background color in low nibble. E.g. `15 clrcol` clears the screen with green background, white foreground.

blkcol (**col row colors** -) Changes colors of the 8x8 block at given position.

plot (**x y** –) Sets the pixel at x, y.

peek (**x y – p**) Gets the pixel at x, y.

line (**x y –**) Draws a line to x, y.

circle (**x y r –**) Draws a circle with radius r around x, y.

erase (**mode –**) Changes blit method for line drawing. **1** **erase** uses **xor** for line drawing, **0** **erase** switches back to **or**.

paint (**x y –**) Paints the area at x, y.

text (**column row str strlen –**) Draws a text string at the given position.
 E.g. `10 8 s" hallo" text` draws the message "hallo" at column 16, row 8.

drawchar (**column row char –**) Draws a custom character at given column and row.

defchar Defines an 8x8 character to use with the **drawchar** word. Example:

```
defchar sqr
00000000
00000000
00111100
00111100
00111100
00111100
00000000
00000000
2 2 sqr drawchar
```

...draws a square at column 2, row 2.

Chapter 6

Music

6.1 Music Macro Language

Music Macro Language (MML) has been used since the 1970s to sequence music on computer and video game systems. MML support is included in `durexForth`, starting with version 1.3. The package is loaded with `s" mml" load`. Two demonstration songs can be found in the `mml demo` package.

MML songs are played using the Forth word `play-mml` which takes three strings, one MML melody for each of the three SID voices. An example song is as follows:

```
: frere-jaques
s" o3l4fgaffgafab->c&c<ab->c&c18cdc<b-14af>18cdc<b-14affcf&ffcf&f"
s" r1o3l4fgaffgafab->c&c<ab->c&c18cdc<b-14af>18cdc<b-14affcf&ffcf&f"
s" " play-mml ;
```

6.2 Commands

cdefgab The letters `c` to `b` represent musical notes. Sharp notes are produced by appending a `+`, flat notes are produced by appending a `-`. The length of a note is specified by appending a number representing its length as a fraction of a whole note – for example, `c8` represents a C eight note, and `f+2` an F# half note. Valid note lengths are 1, 2, 3, 4, 6, 8, 16, 24 and 32. Appending a `.` increases the duration of the note by half of its value.

o Followed by a number, `o` selects the octave the instrument will play in.

r A rest. The length of the rest is specified in the same manner as the length of a note.

<, > Used to step down or up one octave.

l Followed by a number, specifies the default length used by notes or rests which do not explicitly specify one.

& Ties two notes together.

Appendix A

Assembler Mnemonics

adc,#	bvs,	eor,(x)	lsr,a	sbc,#
adc,	clc,	eor,(y)	lsr,	sbc,
adc,x	cld,		lsr,x	sbc,x
adc,y	cli,	inc,		sbc,y
adc,(x)	clv,	inc,x	nop,	sbc,(x)
adc,(y)				sbc,(y)
	cmp,#	inx,	ora,#	
and,#	cmp,	iny,	ora,	sec,
and,	cmp,x		ora,x	sed,
and,x	cmp,y	jmp,	ora,y	sei,
and,y	cmp,(x)	(jmp),	ora,(x)	
and,(x)	cmp,(y)		ora,(y)	sta,
and,(y)		jsr,		sta,x
	cpx,#		pha,	sta,y
asl,a	cpx,	lda,#	php,	sta,(x)
asl,		lda,	pla,	sta,(y)
asl,x	cpy,#	lda,x	plp,	
	cpy,	lda,y		stx,
bcc,		lda,(x)	rol,a	stx,y
bcs,	dec,	lda,(y)	rol,	
beq,	dec,x		rol,x	sty,
		ldx,#		sty,x
bit,	dex,	ldx,	ror,a	
	dey,	ldx,y	ror,	tax,
bmi,			ror,x	tay,
bne,	eor,#	ldy,#		tsx,
bpl,	eor,	ldy,	rti,	txa,
brk,	eor,x	ldy,x	rts,	txs,
bvc,	eor,y			tya,

Appendix B

Memory Map

3 - \$3a Parameter stack, LSB section.
\$3b - \$72 Parameter stack, MSB section.
\$8b - \$8c zptmp (temporary storage for low-level Forth words).
\$8d - \$8e zptmp2 (temporary storage for low-level Forth words).
\$9e - \$9f zptmp3 (temporary storage for low-level Forth words).
...
\$801 - here Forth Kernel followed by dictionary.
...
bufstart - eof Editor space.

Appendix C

Word Anatomy

C.1 Inspecting a Word

Let us define a word and see what it gets compiled to.

```
: bg d020 c! ;
```

When the word is defined, you can get its start address by `loc bg`, and the contents of `bg` can be dumped using `loc bg dump`. Try it, and you will get output like the following:

```
4c38  ed 4b 02 42 47 20 cf 0e .k.bg ..
4c40  20 d0 20 49 0a 60 ff ff  . i....
4c48  ff ff ff ff ff ff ff ff  ....
4c50  ...
```

Here, we can see that the "bg" word is 14 bytes long and starts at address \$4c38. It contains two parts: Header and code.

C.2 Header

```
4c38  ed 4b 02 42 47 20 cf 0e .k.bg ..
4c40  20 d0 20 49 0a 60 ff ff  . i....
```

The first two bytes contain a back-pointer to the previous word, starting at \$4bed. The next byte, "02", is the length of "bg" name string. After that, the string "bg" follows. (42 = 'b', 47 = 'g')

The name length byte is also used to store special attributes of the word. Bit 7 is "immediate" flag, which means that the word should execute immediately instead of being compiled into word definitions. ("(" is such an example of an immediate word that does not get compiled.) Bit 6 is "hidden" flag, which makes a word unfindable. Bit 5 is the "no-tail-call-elimination" flag, which makes sure that tail call elimination (the practice of replacing `jsr/rts` with `jmp`) is not performed if this word is the `jsr` target. Since `bg` does not have these flags set, bits 7-5 are all clear.

C.3 Code

```
4c38  ed 4b 02 42 47 20 cf 0e .k.bg ..
4c40  20 d0 20 49 0a 60 ff ff . i....
```

The code section contain pure 6502 machine code.

20 cf 0e (jsr \$ecf) \$ecf is the adress of the `lit` code. `lit` copies the two following bytes to parameter stack.

20 d0 (\$d020) The parameter to the `lit` word. When executed, `lit` will add \$d020 to the parameter stack.

20 49 0a (jsr \$a49) \$a49 is the address of the `c!` code.

60 (rts) Returns to the caller.