durexForth

Operators Manual

Developed by

Ravelli/Durex

durex

Contents

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:

- durexForth uses text files instead of a custom block file system.
- durexForth is smaller.
- durexForth is faster.
- durexForth fully implements the Forth 2012 core standard.

- 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:

```
include gfxdemo
```

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

1.2.2 Fractals

The fractals package demonstrates turtle graphics by generating fractal images. Run it by entering:

```
include fractals
```

When an image has finished drawing, press any key to continue.

1.2.3 Music

The mmldemo package demonstrates the MML music capabilities. To play some music:

```
include mmldemo
```

1.2.4 Sprites

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

```
include spritedemo
```

Exit the demo by pressing any key.

Chapter 2

Tutorial

2.1 Interpreter

Start up durexForth. When loaded, it will greet you with a blinking yellow cursor, waiting for your input. 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 the stack contents without modifying it. Now enter `.` to pop the digit and print it to screen, followed by `.s` to verify that the stack is empty.

Now something about numbers. The default input mode in DurexForth is hexadecimal. As an example, `1000 a * u.` will calculate `a` × `1000` and print the result `a000`. If you wish, it is possible to switch numerical base using `decimal` and `hex`. Or, you can prefix `$`, `#` or `%` to your number to set a base for it: like `$d020`, `#1234` or `%11010101`.

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 red 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 creating a program!

Now, enter the following lines...

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

...and then press ← to leave insert mode. Press F7 to compile and run. If everything is entered right, you will be facing a wonderful color cycle.

When you finished watching, press RESTORE to quit your program, then enter `vi` to reopen the editor.

2.3 Assembler

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

```
code flash
here ( push current addr )
d020 inc,
jmp, ( jump to pushed addr )
;code
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 (starting with £) 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 as supposed, 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 abort" depth not 0"
again ;
```

2.6 Configuring durexForth

2.6.1 Stripping Modules

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

asm The assembler. (Required and not forgettable.)

labels Assembler labels.

doloop Do-loop words.

format Numerical formatting words.

sys System calls.

debug Words for debugging.

ls List disk contents.

gfx Graphics module.

vi The text editor.

require The words require and required.

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

1. Issue `modules` to forget all modules.

2. Optionally re-add the `modules` marker with `marker modules`.
3. One by one, load the modules you want included with your new Forth. (E.g. `include debug`)
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 `vi foo` 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 "noname" if no buffer is already open. 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 $7000.

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

hjkl 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 by entering `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.

2over (**a b c d** – **a b c d a b**) Copies cell pair a b to top of stack.

2swap (**a b c d** – **c d a b**) Exchanges the top two cell pairs.

?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 **...** x_1 x_0 u – x_u **...** 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

. **(n –)** Prints top value of stack as signed number.

u. **(u –)** Prints top value of stack as unsigned number.

.s See stack contents.

emit **(a –)** Prints top value of stack as a PETSCII character. Example: `'q' emit`

£ Comment to end of line. (Used on C64/PETSCII.)

\ Comment to end of line. (Used when cross-compiling from PC/ASCII.)

(Multiline comment. Ignores everything until a).

bl **(– char)** Gives the PETSCII character for a space.

space Displays one space.

spaces **(n –)** Displays n spaces.

page Clears the screen.

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)** Multiply top of stack value by 2.

2/ **(a – b)** 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.

/ **(a b – q)** Divide a with b using floored division.

/mod **(a b – r q)** Divide a with b, giving remainder r and quotient q.

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.

0< (a – b) Is a negative?

negate (a – b) Negates a.

abs (a – b) Gives absolute value of a.

min (a b – c) Gives the lesser of a and b.

max (a b – c) Gives the greater of a and b.

within (n lo hi – flag) Returns true if lo <= n < hi.

< (n1 n2 – flag) Is n1 less than n2? (Signed.)

> (n1 n2 – flag) Is n1 greater than n2? (Signed.)

u< (u1 u2 – flag) Is u1 less than u2? (Unsigned.)

u> (u1 u2 – flag) Is u1 greater than u2? (Unsigned.)

lshift (a b – c) Binary shift a left by b.

rshift (a b – c) Binary shift a right by b.

base (variable) Points to the cell that holds the numerical base.

decimal Sets the numerical base to 10.

hex Sets the numerical base to 16.

4.4 Double

The following words use double-cell integers. On the stack, the cell containing the most significant part of a double-cell integer is above the cell containing the least significant part.

dabs (d – ud) Produces the absolute value of d.

dnegate (d – d) Negates the double-cell integer d.

s>d (n – d) Converts the number n to the double-cell number d.

m+ (d n – d) Add n to double-cell number d.

m* (a b – d) Multiply a with b, producing a double-cell value.

um* (a b – ud) Multiply a with b, giving the unsigned double-cell number ud.

um/mod (ud n – r q) Divide double-cell number ud by n, giving remainder r and quotient q. Values are unsigned.

sm/rem (d n – r q) Divide double-cell number d by n, giving the symmetric quotient q and the remainder r. Values are signed.

fm/mod (d n – r q) Divide double-cell number d by n, giving the floored quotient q and the remainder r. Values are signed.

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.

2@ (address – x1 x2) Fetch 32-bit value from address. x2 is stored at address, and x1 is stored at address + 2.

2! (x1 x2 address –) Store 32-bit value to address. x2 is stored at address, and x1 is stored at address + 2.

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

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

cell+ (n – n+2) Adds the cell size (which is 2).

cells (n – n*2) Multiplies with the cell size (which is 2).

char+ (n – n+1) Adds the char size (which is 1).

align (–) No-op, required by Forth 2012 standard.

aligned (–) No-op, required by Forth 2012 standard.

chars (–) No-op, required by Forth 2012 standard.

fill (addr len char –) Fill range [addr, len + addr) with char.

move (src dst len –) Copies a region of memory `len` bytes long, starting at `src`, to emory beginning at `dst`.

4.7 Compiling

: ("<spaces>name" –) Start compiling a new Forth word.

; End compiling Forth word.

code ("<spaces>name" –) Start assembling a new word.

;code End assembler.

, (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.

allot (n –) Add n bytes to the body of the most recently defined word.

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 most recently defined word as immediate (i.e. inside colon definitions, it will be executed immediately instead of compiled).

['] 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."

>body (xt – addr) Returns the data field address that belongs to the execution token. Example use: `' foo >body`

state (– addr) addr is the address of a cell containing the compilation-state flag. It is 1 when compiling, otherwise 0.

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.

2 constant bar Create constant value bar and set it to 2.

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.

4.9 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 )
: printoddnumbers (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.

leave Leaves the innermost loop.

unloop Discards the loop-control parameters. Allows clean `exit` from within a loop.

```
: xx 0 0 do unloop exit loop ;
```

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 (– c) Gets one character from the keyboard.

key? (– flag) Returns true if a character is available for `key`.

getc (– c) Consumes the next character from the input buffer and increases `>in` by one. If no characters are available, the input buffer is refilled as needed.

char (– c) Parses the next word, delimited by a space, and puts its first character on the stack.

>in (– addr) Gives the address of a cell containing the offset in characters from the start of the input buffer to the start of the parse area.

source (– caddr u) Gives the address of, and number of characters in, the input buffer.

source-id (– n) Returns 0 if current input is keyboard, -1 if it is a string from `evaluate`, or the current file id.

word (delim – addr) Reads a word from input, using delimiter `delim`, and puts the string address on the stack.

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

accept (addr u – u) Receive a string of at most u characters into the buffer that starts at addr. Returns how many characters were received.

evaluate (addr len –) Makes DurexForth evaluate the given string.

abort Empties the data stack and performs `quit`.

abort" ccc" (**f** –) If `f` is true, print `ccc` and abort.

Typical use: `: x ... test abort" error" ... ;`

quit Enters an endless loop where DurexForth interprets Forth commands from the keyboard. The word is named "quit" since it can be used to quit a program. It also does cleanup tasks like resetting input.

4.11 Editing

vi filename Opens text editor and starts editing the file named "filename". If filename is empty and a buffer is already open, editor will pick up where it left. Otherwise, an untitled buffer will be created.

4.12 Strings

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

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

s" (– **strptr strlen**) Define a string. Example: `s" foo"`.

count (**str** – **caddr u**) Returns data address and length of the counted string str.

type (**caddr u** –) Prints a string.

4.13 Number Formatting

For more info about number formatting, read Starting Forth!

>number (**ud addr u** – **ud addr2 u2**) Converts the string in `addr u` to digits, using `BASE`, and adds each digit into ud after multiplying it with `BASE`. `addr2 u2` contains the part of the string that was not converted.

<# Begins the number conversion process.

(**ud** – **ud**) Converts one digit and puts it in the start of the output string.

s# (**ud** – **ud**) Calls # once, and repeats until the number is zero.

hold (**ch** –) Inserts the char at the start of the output string.

sign (**a** –) If `a` is negative, inserts a minus sign at the start of the output string.

#> (**xd** – **addr u**) Drops xd and returns the output string.

4.14 Vectored Execution

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

find (cstr – cstr 0 | xt -1 | xt 1) Find the definition named in the counted string cstr. If the definition is not found, return cstr and 0, otherwise return the execution token. If the definition is immediate, also return 1, otherwise also return -1.

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

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

4.15 Debugging

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`.

4.16 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 `*`.

marker name (–) Creates a word that when called, forgets itself and all words that were defined after it. Example:

```
marker forget
: x ;
forget
```

4.17 Disk I/O

include filename (–) Load and parse file. Example: `include test`

included (filenameptr filenamelength –) Load and parse file.

require filename (–) Like include, except that load is skipped if the file is already loaded.

required (filenameptr filenamelength –) Like included, except that load is skipped if the file is already loaded.

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

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

device (device# –) Switches the active device. 8 to 11 are valid device#'s.

4.18 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: `'0' ar ! ffd2 sys` calls the CHROUT routine, which 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 `include turtle`.

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 `include gfx`. 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 addr –) Draws a custom character at given column and row, using the 8 bytes long data starting at addr.

Chapter 6

SID

6.1 Introduction

The `sid` module contains low-level words for controlling the SID chip. To load it, type `include sid`. To test that it works, run `sid-demo`.

6.1.1 Voice Control

voice! (**n** –) Selects SID voice 0-2.

freq! (**n** –) Writes 16-bit frequency.

pulse! (**n** –) Writes 16-bit pulse value.

control! (**n** –) Writes 8-bit control value.

srad! (**srad** –) Writes 16-bit ADSR value. (Bytes are swapped.)

note! (**n** –) Plays note in range [0, 94], where 0 equals C-0. The tuning is correct for PAL.

6.1.2 SID Control

cutoff! (**n** –) Writes 16-bit filter cutoff value.

filter! (**n** –) Writes 8-bit filter value.

volume! (**n** –) Writes 8-bit volume.

Chapter 7

Music

7.1 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 `include mml`. Two demonstration songs can be found in the `mmldemo` 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&cl8cdc<b-l4af>l8cdc<b-l4affcf&ffcf&f"
s" r1o3l4fgaffgafab->c&c<ab->c&cl8cdc<b-l4af>l8cdc<b-l4affcf&ffcf&f"
s" " play-mml ;
```

7.2 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.

Chapter 8

Assembler

8.1 Introduction

DurexForth features a simple but useful 6510 assembler with support for branches and labels. Assembly code is typically used within a `code` word, as in the tutorial example:

```
code flash
here ( push current addr )
d020 inc, ( inc $d020 )
jmp, ( jump to pushed addr )
;code
```

It is also possible to inline assembly code into a regular Forth word, as seen in the tutorial:

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

8.2 Branches

The assembler supports forward and backward branches. These branches cannot overlap each other, so their usage is limited to simple cases.

+branch (– **addr**) Forward branch.

:+ (**addr** –) Forward branch target.

:- (– **addr**) Backward branch target.

-branch (**addr** –) Backward branch.

Example of a forward branch:

```
foo lda,
+branch beq,
bar inc, :+
```

Example of a backward branch:

```
:- d014 lda, f4 cmp,#
-branch bne,
```

8.3 Labels

The `labels` module adds support for more complicated flows where branches can overlap freely. These branches are resolved by the `;code` word, so it is not possible to branch past it.

@: (n –) Creates the assembly label n, where n is a number in range [0, 255].

@@ (n –) Compiles a branch to the label n.

Example:

```
code checkers
7f lda,# 0 ldy,# 'l' @:
400 sta,y 500 sta,y
600 sta,y 700 sta,y
dey, 'l' @@ bne, ;code
```

8.4 Assembler Mnemonics

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

Appendix A

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. Default `bufstart` is $7000.

Appendix B

Word Anatomy

B.1 Inspecting a Word

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

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

After the word is defined, you can get its start address by `latest @`, and the contents of bg can be dumped using `latest @ 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 4c 49 0a ff ff ff  .li....
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.

B.2 Header

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

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.

B.3 Code

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

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.

4c 49 0a (jmp $a49) $a49 is the address of the `c!` code.