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 different language. It's aged and a little weird.

What's cool about it? It's a very low-level and minimal language without any automatic memory management. At the same time, it easily scales 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 more fast, memory effective and powerful.

Compared to C, specifically cc65, 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 much faster than Forth code.

The main advantage of Forth is that the compiler can run on the actual machine. It would hardly be possible to write 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. It can be really nice to make small edits and tests without going through the entire edit-compile-link-run-test loop.

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 less bloated.
- The editor is a vi clone.
- It's open source (available at Google Code).

1.1.3 Stack Checking

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, there are no checks for stack overflow and underflow. Since the parameter stack is placed in zeropage for best performance, this means that the system will blow up 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 status, handle the stack with care.

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! ;
```

`0 bg`, `1 bg` and so on will let you set border color. Cool! Now let's head on to making our first "real" program...

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 `s" foo" vi`.[1] It will try to open the (non-existant) file "foo." After a little while you will be presented to the nice pink screen of the editor.

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 junk we created in the last section. Enter:

```
forget bg
```

[1] Yes, the space should be between `s"` and `foo"`. The `s"` word creates a string by reading characters until it hits a `"`.

... and press ← to leave insert mode. This line 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 `fg` (foreground). 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 with `fg` again.

Under `forget bg`, add the following lines:

```
: flash d020 c@ 1+ d020 c! recurse ;
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 stop the program, press RESTORE to get to the interpreter. Then enter `fg` to get back into the editor. Let's see how we can factor the program to get something more Forth'y:

```
forget bg
: bg d020 ; # border color addr
: 1+c! dup c@ 1+ swap c! ; ( addr -- ) # inc addr by 1
: flash dup 1+c! recurse ; ( addr -- ) # inc addr forever
bg flash
```

(Note: Parentheses are used for multi-line comments or describing arguments and return values. `#` is used for single-line comments.)

Of course, it is a matter of taste which version you prefer. Press `F7` to see if the new version runs faster or slower.

2.3 Assembler

If you need to flash as fast as possible, use the assembler:

```
:asm flash
here @ # push curr addr
d020 inc,
jmp, # jmp to pushed addr
;asm
flash
```

`:asm` and `;asm` define a code word, just like `:` and `;` define Forth words. Within a code word, you can use assembler mnemonics.

Note: As the x register contains the durexForth stack depth, it is important that it remains unchanged at the end of the code word.

2.4 Console I/O Example

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

```
: init 0 linebuf c! ; # turn off key buffering
: foo init begin key emit again ;
foo
```

2.5 Configuring durexForth

2.5.1 Stripping Modules

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

debug Words for debugging.

asm The assembler.

edit The text editor.

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 `: 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`.

2.5.2 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: `loc megademo >cfa start !`

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

2.6 How to Learn More

2.6.1 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

2.6.2 Other

- durexForth source code

Quality coders may contact a Durex representative for assistance.

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). 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 $4000.

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.

U Half page up.

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 `fg`, 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.

3.1.4 Text Manipulation

r Replace character under cursor.

x Delete character.

X Backspace-delete character.

dw Delete word.

dd Cut 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.

2swap (**a b c d** – **c d a b**) Swap topmost double 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 **...** 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**) Fetch value of top of return stack (without lifting it).

rdrop (–) Drop value on top of return stack.

4.2 Utility

. **(a –)** Print top value of stack.

c. **(a –)** Print top byte 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.

4.3 Mathematics

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.

+! **(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.

/mod **(a b – r q)** Divide a with b. r = rest, q = quotient.

/ **(a b – q)** Divide a with b.

mod **(a b – r)** Rest of a divided by b.

4.4 Logic

0> **(a – b)** Is a greater than zero?

0= **(a – b)** Is a equal to zero?

= **(a b – c)** Is a equal to b?

<> **(a b – c)** Does a differ from b?

< **(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?

and **(a b – c)** Binary and.

or (a b – c) Binary or.

xor (a b – c) Binary exclusive or.

not (a – b) Flip all bits of a.

4.5 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 –) Copy len bytes from src to dst. The move begins with the contents of src and proceeds towards high memory.

cmove> **(src dst len –)** Byte-to-byte copy like `cmove`, but starts with address src + len - 1 and proceeds towards src.

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

4.6 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.

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

] (–) Return to compile mode.

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

[compile] xxx Compile the immediate word `xxx` instead of executing it.

['] xxx Compile the execution token of word `xxx` as a literal value.

create xxx Create a dictionary header with name `xxx`.

4.7 Variables

4.7.1 Values

Values are fast to read, slow to write.

: foo 1 ; Define value foo.

1 value foo Equivalent to the above.

foo Fetch value of foo.

0 to foo Set foo to 0.

4.7.2 Variables

Variables are faster to write to than values.

var foo Define variable foo.

foo @ Fetch value of foo.

1 foo ! Set value of foo to 1.

4.7.3 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 0 c, 1 c, 2 c, 4 c, 8 c,
10 c, 20 c, 40 c, 80 c,
: 2pow ( n -- 2pown ) ['] 2powtable + c@ ;
```

4.8 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

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.

recurse Jump to the start of the word being compiled.

4.9 Keyboard Input

key (– n) Read a character from input. Buffered/unbuffered reading is controlled by the `linebuf` variable.

word (– addr) Read a word from input and put it on the stack.

linebuf This variable switches between buffered/unbuffered input. Disable input buffering with `0 linebuf c!`, enable with `1 linebuf c!`.

4.10 Editing

vi (s –) Open editor. Try `s" foo" vi`.

fg Re-open editor to pick up where it left.

4.11 Strings

." Print a string. E.g. `." foo"`

s" Run time: (– strptr strlen) Compile time: (– str)

Define a string and put it on the stack. E.g. `s" foo"`.

In run time, it puts the string length and a text pointer on the stack. In compile time (inside a colon definition), it puts a pointer to a Pascal-string on the stack.

4.12 Vectored Execution

' xxx (– addr) Compile-time only: 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.

exec (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: `f = loc f >cfa exec`

4.13 Debugging

words List all defined words.

sizes List sizes of all defined words.

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`.

4.14 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 `*`.

sp@ (– addr) Address of stack top before `sp@` is executed.

sp0 (value) Address of stack bottom.

blink (status –) Disable/enable cursor blink. (0 = off, 1 = on)

4.15 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.

4.16 Kernel Calls

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

Example: `30 ar !  ffd2 jsr-wrap` prints `0` on screen.

4.17 "Missing" Words

The following words might be expected in a "normal" Forth, but are not included in durexForth for the sake of keeping it lean:

- do ... loop, i, j
- */, */mod
- abs

Also, I do not have time to describe every word defined. Please refer to a Forth reference manual and/or the source.

Appendix A

Assembler Mnemonics

```
adc,#       bvs,        eor,(x)     lsra,       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,#       cpx,        lda,#       php,        sta,(x)
asl,                    lda,        pla,        sta,(y)
asl,x       cpy,#       lda,x       plp,
            cpy,        lda,y                   stx,
bcc,                    lda,(x)     rola,       stx,y
bcs,        dec,        lda,(y)     rol,
beq,        dec,x                   rol,x       sty,
                        ldx,#                   sty,x
bit,        dex,        ldx,        rora,
            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

... - $83 Parameter stack (grows downwards).

$84 - $89 Temporary registers.

$8a - $8b Instruction pointer.

...

$801 - `here` Forth Kernel followed by dictionary.

...

`bufstart - eof` Editor space.

`compile-addr - ...` Temporary buffer for `load` (grows upwards).