durexForth

Operators Manual

Johan Kotlinski, Kevin Lee Reno, Poindexter Frink, Richard Halkyard

This manual is for durexForth, a modern Forth system for the Commodore 64.

Table of Contents

Introduction

Why Forth?

Forth is a unique language. What is so special about it? It is a small, low-level language, which can easily be extended to a high-level, domain-specific language that does anything you want it to. 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 nice thing about Forth is that you can run the full development environment on your C64, with text editor, compiler, assembler and debugger. It makes for a more interactive and fun experience than running a cross-compiler on PC.

For a Forth introduction, refer to the excellent Starting Forth by Leo Brodie.

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 can boot from cartridge.
- durexForth implements the Forth 2012 core standard.
- The durexForth editor is a vi clone.
- durexForth is open source (available at Github).

Package Contents

durexForth is packaged as a 16-kByte .crt cartridge file and a .d64 disk image. Booting from cartridge is equivalent to booting from disk, except that cartridge is faster. The disk contains various optional Forth modules, as well as some appetizer demonstration programs, as follows:

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 an image has finished drawing, press space to continue.

Fractals

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

```
include fractals
demo
```

When an image has finished drawing, press space to continue.

Music

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

```
include mmldemo
```

Sprites

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

```
include spritedemo
```

Tutorial

Meet the Interpreter

Start up durexForth. The system will greet you with a blinking yellow cursor, waiting for your input. This is the interpreter, which allows you to enter Forth code interactively.

Let us try the traditional first program: Type in `.( Hello, world! )` (and press `Return`). The system will reply `Hello, world! ok`. The `ok` means that the system is healthy and ready to accept more input.

Now, let us try some mathematics. `1 1 +` (followed by `Return`) will add 1 and 1, leaving 2 on the stack. This can be verified by entering `.s` to print the stack contents. Now enter `.` to pop the 2 and print it to screen, followed by another `.s` to verify that the stack is empty.

Let us 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!`.

Introducing the Editor

The v editor 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 `v`. 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 see a beautiful color cycle.

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

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 `end-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 )
end-code
flash
```

It is also possible to use inline assembly within regular Forth words:

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

IMPORTANT As the x register contains the parameter stack depth, your assembly code must leave it unchanged.

Console I/O Example

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

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

Printer Example

This piece of code prints a message to a printer on device #4, and then prints a message to the screen:

```
include io

: print-hello
4 device ( use device 4 )
0 0 47 7 open ioabort ( open address 7 as file 47, abort on failure )
47 chkout ioabort ( redirect output to file 47, abort on failure )
." Hello, printer!" cr
clrchn ( stop input and output redirection )
." Hello, screen!" cr
47 close ( close file 47 ) ;

print-hello
```

The device number and address may differ between printer models. Commodore MPS series printers use address 0 to print in their uppercase/graphics font, and address 7 to print in their lowercase/uppercase font.

Editor

The editor is a vi clone. Launch it by entering v foo in the interpreter (foo being the file you want to edit). You can also enter v without argument to create an unnamed buffer. For more info about vi style editing, see the Vim web site.

Inserting Text

At startup, the editor is in command mode. These commands start insert mode, which allows you to enter text. Return to command mode with ←.

i

Insert text.

R

Replace text.

a

Append text.

A

Append text at end of line.

C

Change rest of line.

S

Substitute line.

s

Substitute character.

o

Open new line after cursor line.

O

Open new line on cursor line.

cw

Change word.

Navigation

hjkl *or* ⇐□□⇒

Move cursor left, down, up, right.

-

Scroll 1 line up.

+

Scroll 1 line down.

Ctrl+u

Half page up.

Ctrl+d

Half page down.

b

Go to previous word.

w

Go to next word.

e

Go to end of word.

fx

Find char x forward.

Fx

Find char x backward.

0 ***or*** **Home**

Go to line start.

$

Go to line end.

g

Go to start of file.

G

Go to end of file.

H

Go to home window line.

L

Go to last window line.

M

Go to middle window line.

/*string*

Search forward for the next occurrence of the string.

*****

Search forward for the next occurrence of the word under the cursor.

n

Repeat the latest search.

Saving and Quitting

After quitting, the editor can be re-opened by entering v, 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. On completion, enter v to return to editor. To terminate a running program, press `RESTORE`.

Text Manipulation

r

Replace character under cursor.

x

Delete character.

X

Backspace-delete character.

dw

Delete word.

dd

Cut line.

D

Delete rest of line.

yy

Yank (copy) line.

p

Paste line below cursor position.

P

Paste line on cursor position.

J

Join lines.

Forth Words

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 … 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.

lsb *(— addr)*

The top address of the LSB parameter stack.

msb *(— addr)*

The top address of the MSB parameter stack.

Utility

. *(n —)*

Print top value of stack as signed number.

u. *(u —)*

Print top value of stack as unsigned number.

.s

See stack contents.

emit *(a —)*

Print 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 `)`. `(` is non-standard: when parsing from an `evaluate` string, it refills to accept multi-line comments.

bl *(— char)*

char is the PETSCII character for a space.

space

Display one space.

spaces *(n —)*

Display *n* spaces.

page

Clears the screen.

rvs

Reverse screen output.

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.

2* *(a — b)*

Multiply top of stack value by 2.

2/ *(a — b)*

Divide 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.

/ *(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)*

Negate a.

abs *(a — b)*

Give absolute value of a.

min *(a b — c)*

Give the lesser of a and b.

max *(a b — c)*

Give the greater of a and b.

within *(n lo hi — flag)*

Return 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.

split *(n — lsb msb)*

Byte split *n*. $1234 split gives $34 $12.

base *(— addr)*

addr is the address of a cell that holds the numerical base.

decimal

Set the numerical base to 10.

hex

Set the numerical base to 16.

Double

Double-cell (32-bit) number support is limited. Double literals (e.g. `123456.`) are not supported. This may change with public demand.

dabs *(d — ud)*

Produce the absolute value of *d*.

dnegate *(d — d)*

Negate the double-cell integer *d*.

s>d *(n — d)*

Convert 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.

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.

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*.

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.

erase *(addr len —)*

Fill range [addr, len + addr) with 0.

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 memory beginning at `dst`.

Compiling

: *("name" —)*

Define the word with the given name and enter compilation state.

:noname *(— xt)*

Create an execution token and enter compilation state.

; *(—)*

End the current definition, allow it to be found in the dictionary and go back to interpretation state.

code *("name" —)*

Start assembling a new word.

end-code

End assembly.

, *(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 >xt compile, ;

postpone *xxx*

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

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

header *("name" —)*

Create a dictionary header named *name*.

create *("name" —)* ... does>

Create a word-creating word named *name* with custom behavior specified after `does>`. For further description, see "Starting Forth."

state ***(— addr)***

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

latest ***(— value)***

Address of the latest defined header.

here ***(— value)***

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

marker ***("name" —)***

Create a word that when called, forgets itself and all words that were defined after it. Example:

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

Word List

hide ***("name" —)***

Remove *name* from the word list, while leaving its definition in place.

define ***("name" —)***

Assign `here` as the execution token of word *name* and enter the compilation state.

defcode ***("name" —)***

Like `define`, but starts a `code` segment instead.

dowords ***(xt —)***

Execute *xt* once for every word in the word list, passing the name token of the word to *xt*, until the word list is exhausted or *xt* returns false. The invoked *xt* has the stack effect *(k * x nt — l * x flag)*. If *flag* is true, `dowords` will continue on to the next name, otherwise it will return.

```
\ from debug.fs
: (words) more name>string space 1 ;
: words ['] (words) dowords ;
```

Variables

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*.

Variables

Variables are faster to write to than values.

variable _bar_

Define variable *bar*.

bar @

Fetch value of variable *bar*.

1 bar !

Set variable *bar* to *1*.

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 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 to be used inside `do` .. `loop` constructs. `i` gives inner loop index, `j` gives outer loop index.

leave

Leave the innermost loop.

unloop

Discard the loop-control parameters. Allows clean ((exit)) from within a loop.

```
: x 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 ;
```

Input

key *(— c)*

Get one character from the keyboard.

key? *(— flag)*

Return true if a character is available for `key`.

char *(— c)*

Parse the next word, delimited by a space, and puts its first character on the stack.

>in *(— addr)*

Give the address of a cell containing the offset in characters from the start of the input buffer to the start of the parse area.

refill *(— flag)*

Attempt to fill the input buffer from the input source, returning true if successful.

source *(— caddr u)*

Give the address of, and number of characters in, the input buffer.

source-id *(— n)*

Return 0 if current input is keyboard, -1 if it is a string from `evaluate`, or the current file id.

word *(char — addr)*

Read a word from input, using delimiter *char*, and put the string address on the stack. If the delimiter is the space character, non-breaking space (hex a0) will also be treated as a delimiter.

parse *(char — addr u)*

Parse a string, using delimiter *char*. *addr* is the address within the input buffer, *u* is the length of the parsed string. If the parse area was empty, the resulting string has a zero length.

parse-name *(name — caddr u)*

Read a word from input, delimited by whitespace. Skips leading spaces.

accept *(addr u — u)*

Receive a string of at most u characters into the buffer that starts at addr. Return how many characters were received.

evaluate *(addr len —)*

Evaluate the given string. Evaluate is non-standard: it interprets multi-line strings line-by-line.

abort

Clear the data stack and perform `quit`.

abort" *ccc" (f —)*

If *f* is true, print *ccc* and `abort`. Typical use: `: x ⋯ test abort" error" ⋯ ;`

quit

Enter 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 I/O.

pad *(— addr)*

addr is the address of the `pad`, a 127-byte memory region that can be used freely by user words. No built-in words will modify this region.

Strings

.(

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

."

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

s" *(— caddr u)*

Define a string. Compile-time only! Example: `s" foo"`.

count *(str — caddr u)*

Return data address and length of the counted string *str*.

type *(caddr u —)*

Print a string.

/string *(caddr u n — caddr+n u-n)*

Adjust the string by *n* characters.

Number Formatting

For more info about number formatting, read Starting Forth.

<#

Begin the number conversion process.

# *(ud — ud)*

Convert one digit and puts it in the start of the output string.

#s *(ud — ud)*

Call `#` and repeats until *ud* is zero.

hold *(ch —)*

Insert the character `ch` at the start of the output string.

sign *(a —)*

If *a* is negative, insert a minus sign at the start of the output string.

#> *(xd — addr u)*

Drop *xd* and returns the output string.

Vectored Execution

' *("name" — addr)*

Find execution token of the word named *name*.

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.

find-name *(caddr u — 0 | nt)*

Get the name token (dictionary pointer) of the word named by *caddr u*, or 0 if the word is not found.

execute *(xt —)*

Execute the execution token *xt*.

>xt *(addr — xt)*

Get execution token of word at adress *addr*.

Debugging

words

List all defined words.

size *("name" —)*

Print the size of the definition of the word named *name*.

dump *(n —)*

Memory dump starting at address *n*.

n

Continue memory dump where last one stopped.

see *("name" —)*

Print the definition of the word named *name*. Works on colon definitions only. Optionally included with `include see`.

Disk I/O

include *("filename" —)*

Open and interpret a text file. Example: `include test`

included *(filenameptr filenamelen —)*

Open and interpret a text file.

require *("filename" —)*

Like include, except that load is skipped if the file is already loaded.

required *(filenameptr filenamelen —)*

Like included, except that load is skipped if the file is already loaded.

loadb *(filenameptr filenamelen dst — endaddr)*

Load file to *dst*. Returns the address after last written byte, or 0 on failure.

saveb *(start end+1 filenameptr filenamelength —)*

Save file. *Start* = start address of memory area. *End+1* = end adress of memory area plus 1.

device *(device# —)*

Switch the current device.

save-forth *("filename" —)*

Save the forth to the given filename.

ls

Load and print disk directory with optional drive # and wildcards. Example: `ls $1:*=p` Load directory for drive 1, only prg files.

rdir *(addr —)*

Display disk directory previously loaded to addr.

rderr *(—)*

Read and print error channel of the current device.

DOS Commands

Words for sending DOS commands to drives and reading drive status are available by including the `dos` module.

send-cmd *(c-addr u —)*

Write the given string to secondary address 15 on the current device, and print the drive's response. The following example defines a word, `backup` that creates a copy of `durexforth` called `backup`:

```
: backup s" copy0:backup=durexforth" send-cmd ;
backup
```

dos *("cmd" —)*

Send *cmd* to the current device's command channel, and print the response. Note that the remainder of the line is treated as part of the command. This makes it possible to refer to file names that contain spaces, but means that `dos` and its command should be on their own line, or the last words on a line. Example: `dos scratch0:old file` will delete a file named *old file*.

Low-Level Device I/O

For more advanced uses, words corresponding to the standard Commodore Kernal IO routines are available by including the `io` module.

open *(filenameptr filenamelength file# secondary-addr — ioresult)*

Open a logical file.

chkin *(file# — ioresult)*

Use a logical file as input device.

chkout *(file# — ioresult)*

Use a logical file as output device.

clrchn *(—)*

Reset input and output to the keyboard and screen.

close *(file# —)*

Close a logical file.

readst *(— status)*

Return the status of the last IO operation. For serial-bus devices, `$01` = write timeout, `$02` = read timeout, `$40` = end of file (EOI), `$80` = device not present.

chrin *(— char)*

Read a character from the current input device. ioabort *(ioresult —)* Handle error conditions for `open`, `chkin` and `chkout`. On failure, print error message and abort.

As per the underlying Kernal routines, `chrin` does not check for end-of-file or any other error condition. `readst` should be called to ensure that the returned character is valid.

The *ioresult* value returned by `open`, `chkin` and `chkout` is 0 on success, or a Kernal error number if an error occurred.

CAUTION

Low-level device I/O may interfere with disk accesses done by durexForth and the `v` editor. The following guidelines should be followed to avoid interference:

- Avoid using file numbers 15 and below (remember, any number up to 127 can be used as a file number).
- Only use input/output redirection (`chkin` and `chkout`) within word definitions, and ensure that `clrchn` is called before exit.
- Close files as soon as they are no longer needed.
- If multiple files are open, always call `clrchn` to end any serial bus transactions before calling `open` or switching between files with `chkin` or `chkout`.

Protocol-Level IEC Device I/O

For even more advanced uses, words corresponding to the standard Commodore IEC routines are available by including the `iec` module. These words allow access to serial devices without accessing the file system, and won't hang the computer on a "Device not present" error.

listen *(dv — ioresult)*

Send IEC listen to dv.

second *(command+sa — ioresult)*

Send IEC command and secondary address after listen.

unlisten *(—)*

Send IEC unlisten to all channels.

ciout *(— u)*

Puts a data byte onto the serial bus using full handshaking.

talk *(dv — ioresult)*

Send IEC talk to dv.

tksa *(command+sa — ioresult)*

Send IEC command and secondary address after talk.

untalk *(—)*

Send IEC untalk to all channels.

acptr *(u —)*

Get a byte of data from the serial bus using full handshaking.

> Commands for `second` and `tksa` include:
>
> - $60 OPEN CHANNEL / DATA + Secondary Address / channel (0-15)
> - $E0 CLOSE + Secondary Address / channel (0-15)
> - $F0 OPEN + Secondary Address / channel (0-15)

To send a data byte to a drive, that device must first be "listened". If the Secondary address (from here referred to as: SA or channel) is 15, the drive will interpret the data as a DOS command. A DOS command is executed when the drive is UNLISTENed ($3F). If the channel is not 15, DOS will ignore it unless you first sent an OPEN. An OPEN is sent to tell DOS where you want your data to go. That is done by LISTENing the device.

> - channel = 0 is reserved for reading a PRG file.
> - channel = 1 is reserved for writing a PRG file.
> - channel = 2-14 need the filetype and the read/write flag in the filename as ",P,W" for example.
> - channel = 15 for DOS commands or device status info.

After the OPEN is sent, you can send a LISTEN using the channel used in the OPEN. DOS has a table of opened files, and will use the channel to write your data to the corresponding file.

From: IEC disected by J. Derogee

IEC examples

send-cmd *(c-addr u —)*

Demonstrates and documents sending commands and/or reading the error channel.

dos *("cmd" —)*

Same as Dos command.

bsave *(start end+1 filenameptr filenamelength —)*

Demonstrates and documents sending a file to disk.

dir

Same as `ls`, demonstrates and documents loading a file.

Compatibility

The `compat` module contains various words that are not deemed necessary for enjoyable DurexForth operation, but still must be provided to comply with the Forth 2012 core standard.

environment? *(addr u — 0)*

Environmental query.

cell+ *(n — n+2)*

2+

cells *(n — n*2)*

2*

char+ *(n — n+1)*

1+

align *(—)*

No-op

aligned *(—)*

No-op

chars *(—)*

No-op

d+ *(d1 d2 — d3)*

Adds the double-cell numbers *d1* and *d2*, giving the sum *d3*.

2@ *(addr — x1 x2)*

Fetch 32-bit value from *addr*. *x2* is stored at *addr*, and *x1* is stored at *addr* + 2.

2! ***(x1 x2 addr —)***

Store 32-bit value to *addr*. *x2* is stored at *addr*, and *x1* is stored at *addr* + 2.

2over ***(a b c d — a b c d a b)***

Copy cell pair *a b* to top of stack.

2swap ***(a b c d — c d a b)***

Exchange the top two cell pairs.

>number ***(ud addr u — ud addr2 u2)***

Convert 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.

>body ***(xt — addr)***

Return the data field address that belongs to the execution token. Example use: `' foo >body`

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.

true ***(— true)***

Return a *true* flag, a single-cell value with all bits set.

false ***(— false)***

Return a *false* flag.

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 c! $ffd2 sys` calls the CHROUT routine, which prints `0` on screen.

Turn-key Utilities

These words are available by including `turnkey`.

top ***(— addr)***

Address of the top of the dictionary, default: $9fff.

top! ***(addr —)***

Relocate the dictionary to *addr*. Example:

```
\ not using $a000 block, give all memory to dictionary
$cbff top!
```

save-pack *("filename" —)*

Save a compact version of forth to the given *filename*.

save-prg *("filename" —)*

Save a forth program with no dictionary to *filename*.

Further details on the use of these words are outlined in Turn-key Operation.

Graphics

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 *(—)*

Initialize turtle graphics.

forward *(px —)*

Move the turtle `px` pixels forward.

back *(px —)*

Move the turtle `px` pixels back.

left *(deg —)*

Rotate the turtle `deg` degrees left.

right *(deg —)*

Rotate the turtle `deg` degrees right.

penup *(—)*

Pen up (disables drawing).

pendown *(—)*

Pen down (enables drawing).

turtle@ *(— state)*

Remember turtle state.

turtle! *(state —)*

Restore turtle state as earlier read by `turtle@`.

moveto *(x y deg —)*

Move turtle to *x y* with angle *deg*.

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 *(—)*

Enter the high-resolution drawing mode.

lores *(—)*

Switch back to low-resolution text mode.

clrcol *(colors —)*

Clear 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 —)*

Change colors of the 8x8 block at given position.

plot *(x y —)*

Set the pixel at *x*, *y*.

peek *(x y — p)*

Get the pixel at *x*, *y*.

line *(x y —)*

Draw a line to *x*, *y*.

circle *(x y r —)*

Draw a circle with radius *r* around *x*, *y*.

pen *(mode —)*

Change line drawing method. `1 pen` inverts color, `0 pen` switches back to normal mode.

paint *(x y —)*

Paint the area at *x*, *y*.

text *(column row str strlen —)*

Draw a text string at the given position. E.g. `10 8 parse-name hallo text` draws the message `hallo` at column 16, row 8.

drawchar *(column row addr —)*

Draw a custom character at given column and row, using the 8 bytes long data starting at addr.

SID

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`.

Voice Control

voice! *(n —)*

Select SID voice 0-2.

freq! *(n —)*

Write 16-bit frequency.

pulse! *(n —)*

Write 16-bit pulse value.

control! *(n —)*

Write 8-bit control value.

srad! *(srad —)*

Write 16-bit ADSR value. (Bytes are swapped.)

note! *(n —)*

Play note in range [0, 94], where 0 equals C-0. The tuning is correct for PAL.

SID Control

cutoff! *(n —)*

Write 16-bit filter cutoff value.

filter! *(n —)*

Write 8-bit filter value.

volume! *(n —)*

Write 8-bit volume.

Music

Music Macro Language

Music Macro Language (MML) has been used since the 1970s to sequence music on computer and video game systems. The MML 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 MML strings, one MML melody for each of the three SID voices. An example song is as follows:

```
: frere-jaques
mml" o3l4fgaffgafab->c&c<ab->c&cl8cdc<b-l4af>l8cdc<b-l4affcf&ffcf&f"
mml" r1o3l4fgaffgafab->c&c<ab->c&cl8cdc<b-l4af>l8cdc<b-l4affcf&ffcf&f"
mml" " play-mml ;
```

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.

Assembler

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 )
end-code
```

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

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

Variables

DurexForth has a few variables that are specifically meant to be used within code words.

lsb *(— addr)*

addr points to the top of the LSB parameter stack. Together with the `x` register, it can be used to access stack contents.

msb *(— addr)*

addr points to the top of the MSB parameter stack. Together with the `x` register, it can be used to access stack contents.

w *(— addr)*

A zero-page cell that code words may use freely as work area.

w2 *(— addr)*

Second zero-page work area cell.

w3 *(— addr)*

Third zero-page work area cell.

Example usage of `lsb` and `msb`:

```
code + ( n1 n2 -- sum )
clc,           ( clear carry )
lsb 1+ lda,x   ( load n1 lsb )
lsb adc,x      ( add n2 lsb )
lsb 1+ sta,x   ( store to n1 lsb )
msb 1+ lda,x   ( load n1 msb )
msb adc,x      ( add n2 msb )
msb 1+ sta,x   ( store to n2 msb )
```

```
inx,            ( drop n2; n1 = sum )
rts,            ( return )
end-code
```

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,
```

Labels

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

@: *(n —)*

Create the assembly label *n*, where *n* is a number in range [0, 255].

@@ *(n —)*

Compile 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, rts,
end-code
```

Configuring durexForth

Stripping Modules

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

asm
: The assembler. (Required and may not be stripped.)

format
: Numerical formatting words. (Also required.)

wordlist
: Wordlist manipulation. (Required by some modules.)

labels
: Assembler labels.

doloop
: Do-loop words.

sys
: System calls.

debug
: Words for debugging.

ls
: List disk contents.

require
: The words `require` and `required`.

v
: The text editor.

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

1. Issue `---modules---` to unload all modules, or `---editor---` to unload the editor only.
2. One by one, load the modules you want included with your new Forth. (E.g. `include labels`)
3. Save the new system with e.g. `save-forth acmeforth`.

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, type e.g. `save-forth megademo`.

When writing a new program using a PC text editor, it is practical to configure durexForth to compile and execute the program at startup. That can be set up using the following snippet:

```
$a000 value buf
: go buf s" myprogramfile" buf
loadb buf - evaluate ;
' go start !
save-forth @0:durexforth
```

Turn-key Operation

Durexforth offers utilities to save your program in a turn-key fashion by including the `turnkey` module once the program is ready to be saved.

Programs can be saved in a compacted state using `save-pack`. These programs are stored with 32 bytes between `here` and `latest`. When they are first loaded, they will restore the header space to its original `top`.

If you have developed a program that has no further need of the interpreter, you can eliminate the dictionary headers entirely when saving with `save-prg`. This allows your program to use memory down to `here` plus 32 bytes for safety.

After either of these words have saved the file to disk, they will restore forth to the unpacked state, and strip the `turnkey` module from the dictionary. This allows you to continue to use forth interactively in the case of `save-pack`. As `save-prg` has stripped the dictionary headers from the system, it will no longer be usable. If you wish to test your program after saving, you can compile a call to `save-prg` instead:

```
: build save-prg mydemo start @ execute ;
build
```

This will simulate the start-up sequence after saving the packed program.

Appendix A: Assembler Mnemonics

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

Appendix B: Memory Map

3 - $3a

Parameter stack, `lsb` section.

$3b - $72

Parameter stack, `msb` section.

$8b - $8c

`w` (work area for code words).

$8d - $8e

`w2` (work area for code words).

$9e - $9f

`w3` (work area for code words).

$100 - $1ff

Return stack.

$200 - $258

Text input buffer.

$33c - $35a

`find` buffer.

$35b - $3d9

`pad` Scratch pad memory, Cassette Buffer, untouched by durexForth.

$3da - $3fb

`#>` buffer.

$801 - here

Forth Kernel followed by code and data space.

latest - $9fff

Dictionary. Grows downwards as needed.

$a000 - $cbff

Editor text buffer.

$cc00 - $cfff

Hi-res graphics colors.

$d000 - $dfff

I/O area.

$e000 - $ffff

Kernal / hi-res graphics bitmap.

Appendix C: Word Anatomy

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

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

Information about the word is split into two areas of memory, the dictionary and the code/data space. Code and data are placed in an upward-growing segment starting at $801, and the dictionary grows downward from `top`. `latest` points to the last dictionary record. A dictionary record consists of a counted string with flags, and an execution token (*xt*).

To inspect the dictionary entry, type `latest dump`. You should see something like this:

```
6228  02 42 47 fd 39 28 39 01 .bg.9(9.
...
```

For this run, the name token of `bg` is placed at address $6228. The first byte, `02`, is the name length (`bg` has two characters). After that, the string `bg` follows. ($42 = `b`, $47 = `g`). The final two bytes contain the execution token of `bg`, starting at $39fd.

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 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 and 6 are both clear.

We saw that the `bg` execution token is $39fd. To inspect the code, type `$39fd dump` or `latest >xt dump`.

The code section contains pure 6502 machine code.

```
39fd  20 15 11 20 d0 4c 0e 09  .. Pl..
...
```

20 15 11

`jsr $1115`. $1115 is the address 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 0e 09

`jmp $90e`. $90e is the address of the `c!` code.

Appendix D: Avoiding Stack Crashes

Stack overflow and underflow are common causes for errors and crashes. Simply put, the data stack must not contain too many or too few items. This section describes some techniques to avoid such errors.

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.

NOTE The Forth standard defines backslash (\) as the line comment character, but the C64 lacks a real backslash. Moreover, ASCII \ and PETSCII £ both map to $5c. Therefore, the £ character is used as a substitution on the C64.

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 ;
```

Appendix E: Internet Resources

Forth Books and Papers

- Starting Forth
- Thinking Forth
- Moving Forth: a series on writing Forth kernels
- Forth Interest Group
- 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
- Gforth
- jonesforth
- PETTIL
- volksFORTH

Forth Standards

- Forth 2012 Standard
- ANS Forth
- FORTH-83 STANDARD
- FORTH-79 STANDARD

C64 References

- ACME Cross-Assembler
- Codebase 64
- All About Your C64
- Mapping the Commodore 64

Appendix F: License

```
durexForth
Copyright (c) 2008 Johan Kotlinski, Mats Andrén
Copyright (c) 2012 Kevin Lee Reno
Copyright (c) 2017 Corey Minter
Copyright (c) 2020 Poindexter Frink, Richard Halkyard

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

```
/ MOD */MOD */ ?NEGATE ?DNEGATE DABS SM/REM
Forth Interest Group UK. Public domain.
```

```
UM/MOD
Copyright (c) 2002 Garth Wilson. Public domain.
```

```
MOVE
Based on cc65 memmove.s

This software is provided 'as-is', without any express or implied warranty.
In no event will the authors be held liable for any damages arising from
the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
claim that you wrote the original software. If you use this software in
a product, an acknowledgment in the product documentation would be
```

```
appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not
be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.
```

```
Forth Test Suite

(C) 1995 JOHNS HOPKINS UNIVERSITY / APPLIED PHYSICS LABORATORY
MAY BE DISTRIBUTED FREELY AS LONG AS THIS COPYRIGHT NOTICE REMAINS.
VERSION 1.2

This program was written by Gerry Jackson in 2007, with contributions from
others where indicated, and is in the public domain - it can be distributed
and/or modified in any way but please retain this notice.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Word Index