FORTH(1)

Richard Howe

November 2016

Contents

NAME **2**

SYNOPSIS **2**

DESCRIPTION **2**

OPTIONS **2**

EXAMPLES 4

LICENSE **5**

EXIT STATUS **5**

SEE ALSO **5**

BUGS **5**

MANUAL **5**

Other documentation 7

Using the interpreter 7

A Forth Word 9

Memory Map and Special Registers 12

Registers 12

Dictionary 13

Glossary of Forth words 13
Internal words 14
Defined words 21
Library of Forth words 24
Glossary of Forth terminology 24
Porting this interpreter 25
Standards compliance 26
Bugs 27
To-Do 27

include "libforth.h" 28

ifdef __WIN32 #include #include #include #endif 28
Notes 28

NAME

forth - a forth interpreter

SYNOPSIS

forth [**-s** file] [**-e** string] [**-l** file] [**-m** size] [**-VthvL**] [**-**] [**files**]

DESCRIPTION

A Forth interpreter built around a library, libforth, that implements a complete Forth interpreter.

This interpreter is available at here.

OPTIONS

Command line switches must be given before any files, unless that switch takes a file as an argument.

- -s file

This saves the working memory of the Forth interpreter to a file, which can later be loaded with the "-l" option. If a core file has been invalidated this will not be saved, invalidation occurs when an unrecoverable error has been detected that would prevent any recovery or meaningful execution with the current image.

- -e string

Evaluate a Forth string passed in as an argument.

- -t

After all the files have been read from and any core files have been loaded this will make the Forth interpreter read from stdin, the core file will be saved after stdin has been read from and there is no more work to do, if the "-d" or "-s" flags have been specified.

- -h

Print out a short help message and exit unsuccessfully.

- -v

Turn verbose mode on, more information will be printed out, to stderr, about what is happening in the interpreter. Usually the interpreter is as silent as possible.

- -m size

Specify the virtual machines memory size in kilobytes, overriding the default memory size. This is mutually exclusive with "-l".

- -l file

This option loads a forth core file generated from the "-d" option of a previous run. This core file is not portable and must be generated on the same platform as it was generated. It can only be specified once per run of the interpreter.

- '-'

Stop processing any more command line options and treat all arguments after this as files to be executed, if there are any.

- -V

Print version and interpreter information and exit successfully.

- -L

If the line editing library is compiled into the executable, which is a compile time option, then when reading from stdin this will use a line editor to read in a line at a time. This option implies *-t*.

- file…

If a file, or list of files, is given, read from them one after another and execute them. The dictionary and any stored Forth blocks will persist, as will values on the stack.

If no files are given to execute stdin will be read from.

EXAMPLES

```
./forth
```

Execute any commands given from stdin

```
./forth -t file1.4th file2.4th
```

Execute file "file1.4th", then "file2.4th", then read from stdin

```
./forth file1.4th
```

Execute file "file1.4th".

```
./forth -s file1.4th
```

Execute file "file1.4th", the produce a "forth.core" save file.

```
./forth -s -l forth.core
```

Load a "forth.core" file, read from stdin and execute any commands given, then dump the new core file to "forth.core".

The interpreter returns zero on success and non zero on failure.

LICENSE

The Forth interpreter and the library that implements it are released under the MIT license. Copyright (c) Richard Howe, 2016.

EXIT STATUS

This program will return a non-zero value on failure, and zero on success.

SEE ALSO

libforth(3)

BUGS

If you find a bug, or would like to request a new feature, please Email me at:

```
howe.r.j.89 [ at ] gmail . com
```

The interpreter has not been battle hardened yet so there is likely behavior that is non-standard (for no reason) or just outright incorrect.

MANUAL

This small Forth interpreter is based on a de-obfuscated entrant into the IOCCC by *buzzard*. The entry described a Forth like language which this derives from. You can use this library to evaluate Forth strings or as an embeddable interpreter. Work would need to be done to get useful information after doing those evaluations, but the library works quite well.

main.c is simply a wrapper around one the functions that implements a simple REPL.

This project implements a Forth interpreter library which can be embedded in other projects, it is incredibly minimalistic, but usable. To build the project a C compiler is needed, and a copy of Make, type:

```
make help
```

For a list of build options. By running:

```
make run
```

Will build the interpreter and run it, it will then read from stdin.

To build the documentation other programs may be needed, such as pandoc and the markdown script, but these steps are optional.

Forth is an odd language that has a loyal following groups, but it is admittedly not the most practical of language as it lacks nearly everything the modern programmer wants in a language; safety, garbage collection, modularity and clarity. It is however possible to implement a fully working interpreter in a one to two kilobytes of assembly, those kilobytes can make a functional and interactive programming environment, giving a high ratio of utility memory used.

From the Wikipedia article we can neatly summarize the language:

```
"Forth is an imperative stack-based computer programming language
and programming environment.

Language features include structured programming, reflection (the
ability to modify the program structure during program execution),
concatenative programming (functions are composed with juxtaposition)
and extensibility (the programmer can create new commands).

...

A procedural programming language without type checking, Forth features
both interactive execution of commands (making it suitable as a shell
for systems that lack a more formal operating system) and the ability
to compile sequences of commands for later execution."
```

Given the nature of the Forth language it does not make for a terribly good embeddable scripting language, but it is simple to implement and can be fun to use. This interpreter is based off a previous IOCCC in a file called buzzard.2.c, it is a descendant of that file.

Before using and understanding this library/interpreter it is useful to checkout more literature on Forth such as Thinking Forth by Leo Brodie for a philosophy of the language, Starting Forth (same Author), Jonesforth which is a specific implementation of the language in x86 assembly and Gforth, a more modern and portable implementation of the language.

It is important to realize that Forth is really more a philosophy and collection of ideas than a specific reference implementation or standard. It has been said that an intermediate Forth user is one who has implemented a Forth interpreter,

something which cannot be said about other languages nor is possible given their complexity.

The saying "if you have seen one Forth implementation, you have seen one Forth implementation" applies, nearly every single Forth implementation has its own idea of how to go about things despite standardization efforts - in keeping with this, this library has its own idiosyncrasies.

This implementation, written in C, can be thought of as a hybrid between a fairly dumb stack based virtual machine with instructions such as "pop two values off the stack, add them, and push the result" and a small interpreter/compiler for the virtual machine. This simple kernel is then used to build a more compliant and usable Forth implementation by defining words that build upon those provided by the base system.

Other documentation

Apart from this file there are other sources of information about the project:

As can the code, which is small enough to be comprehensible:

- libforth.c (contains the core interpreter)
- libforth.h (contains the API documentation)

And the forth startup code:

- forth.fth

The startup code is well commented and shows how the core interpreter is extended to a more function Forth environment.

The source file libforth.c can be converted to a more readable webpage by first converting the source to markdown with convert script, the converting that to HTML in the usual fashion

Using the interpreter

main.c simple calls the function *main_forth()* in *libforth.c*, this function initializes a Forth environment and puts the user in a REPL where you can issue commands and define words. See the manual pages for list of command line options and library calls. All commands are given using Reverse Polish Notation (or RPN),

So:

```
2+(2*4)
```

Becomes:

```
4 2 * 2 +
```

And brackets are no longer needed. Numbers of pushed on to the variable stack automatically and commands (such as '*' and '+') take their operands off the stack and push the result. Juggling variables on the stack becomes easier over time. To pop a value from the stack and print it there is the '.' word.

So:

```
2 2 + .
```

Prints:

```
4
```

The simplicity of the language allows for a small interpreter, the loop looks something like this:

```
1) Read in a space delimited Forth WORD.
2) Is this WORD in the dictionary?
   FOUND)      Are we in IMMEDIATE mode?
               IMMEDIATE-MODE) Execute WORD.
                               goto 1;
               COMPILE-MODE)   Compile WORD into the dictionary.
                               goto 1;
   NOT-FOUND)  Is this actually a number?
               YES) Are we in IMMEDIATE mode?
                    IMMEDIATE-MODE) Push Number onto the stack.
                                    goto 1;
                    COMPILE-MODE)   Compile a literal number.
                                    goto 1;
               NO)  Error! Handle error
                    goto 1;
```

Given that we are reading in *space delimited words* if follows that the above expression:

```
2 2 + .
```

Would not work if we did:

```
2 2+ .
```

Or:

```
2 2 +.
```

As "2+" and "+." would be parsed as words, which may or may not be defined and if they are do not have the behavior that we want. This is more apparent when we do any kind of string handling.

A Forth Word

The Forth execution model uses Threaded Code, the layout of a word header follows from this.

A Forth word is defined in the dictionary and has a particular format that varies between implementations. A dictionary is simply a linked list of Forth words, the dictionary is usually contiguous and can only grow. The format for our Forth words is as follows:

Briefly:

- Word Header:
- field <0 = Word Name (the name is stored before the main header)
- field 0 = Previous Word
- field 1 = Code Word (bits 0 - 7) | Hidden Flag (bit 8) | Word Name Offset (bit 9 - 15)
- field 3 = Code Word or First Data field Entry
- field 4+ = Data Field

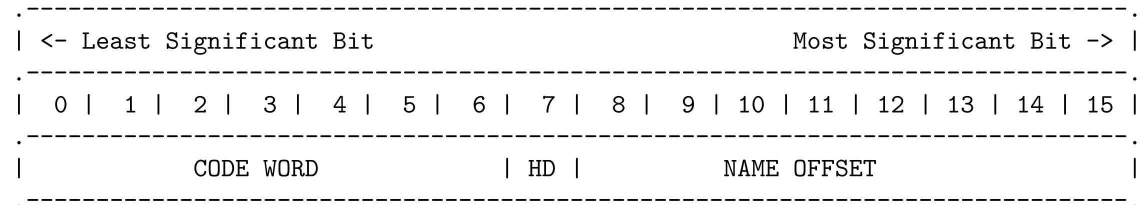
And in more detail:

```
.----------------------------------------------.----------------.
|                  Word Header                 |   Word Body    |
.---------------.-----.------.-------------------.----------------.
| NAME ...      | PWD | MISC | CODE WORD or DATA | DATA ...       |
.---------------.-----.------.-------------------.----------------.


----
NAME        = The name, or the textual representation, of a Forth
              word, it is a variable length field that is ASCII NUL
              terminated, the MISC field has an offset that points
              to the begining of this field if taken off the PWD
              position (not value). The offset is in machine words,
              not characters.

---
```

```
PWD         = A pointer to the previously declared word.

----
MISC        = A complex field that can contains a CODE WORD, a
              "hide" bit and the offset from the PWD field to the
              beginning of NAME

---------    ----
CODE WORD or DATA = This will be RUN if the following DATA is a pointer
              to the CODE WORDs of previously defined words. But it
              could be any CODE WORD.

----
DATA        = This could be anything, but it is most likely to be
              a list of pointers to CODE WORDs of previously defined
              words if this optional DATA field is present.
```

All fields are aligned on the Forth virtual machines word boundaries.

The MISC field is laid out as so:

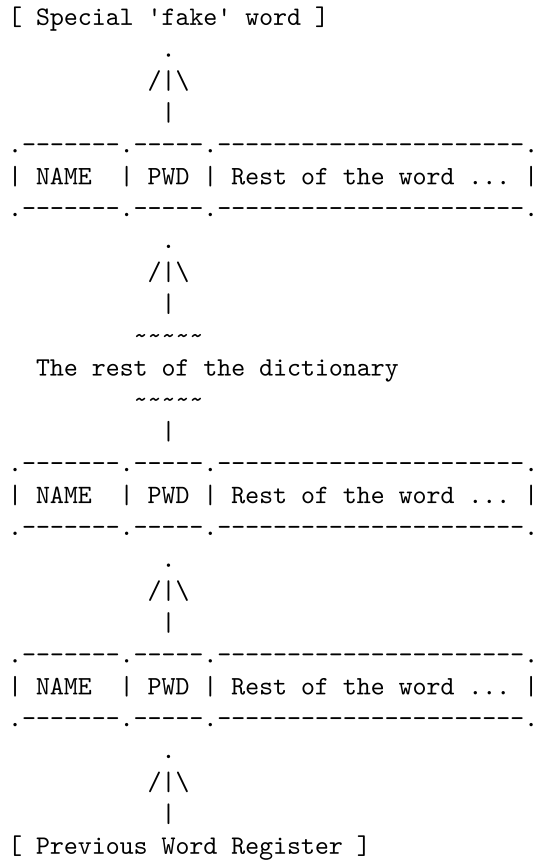


```
.-------------------------------------------------------------------------------.
| <- Least Significant Bit                              Most Significant Bit -> |
.-------------------------------------------------------------------------------.
|  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 |
.-------------------------------------------------------------------------------.
|            CODE WORD              | HD |          NAME OFFSET                  |
.-------------------------------------------------------------------------------.
```

```
---------
CODE WORD    = Bits 0-6 are a code word, this code word is always run
               reguardless of whether we are in compiling or command
               mode

--
HD           = Bit 7 is the Hide Bit, if this is true then when
               compiling or executing words the word will be hidden from the
               search.

-----------
NAME OFFSET  = Bits 8 to 15 are the offset to the words name. To find the
               beginning of the words name we take this value away from
               position of this words PWD header. This value is in
               machine words, and so the beginning of the NAME must be aligned
               to the virtual machine words boundaries and not character, or byte,
               aligned. The length of this field, and the size of the input buffer,
               limit the maximum size of a word.
```

Depending on the virtual machine word size, or cell size, there may be more bits above bit '15', the most significant bit, in the MISC field. These bits are not used and should be set to zero.

And the dictionary looks like this:



Searching of the dictionary starts from the *Previous Word Register* and ends at a special 'fake' word.

Defining words adds them to the dictionary, we can defined words with the ':' words like this:

```
: two-times 2 * ;
```

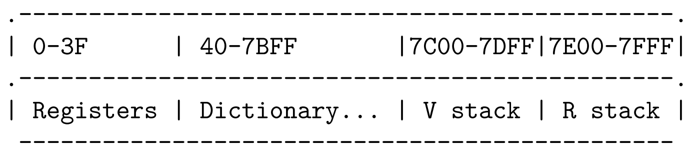
Which defined the word "two-times", a word that takes a value from the stack, multiplies it by two and pushes the results back onto the stack.

The word ':' performs multiple actions; it is an immediate word that reads in the next space delimited word from the input stream and creates a header for that word. It also switches the interpreter into compile mode, compiling words will be compiled into that word definition instead of being executed, immediate words are executed as normal. ';' is also an immediate word, it compiles a special word exit into the dictionary which returns from a word call and switches the interpreter back into command mode. This type of behavior is typical of Forth implementations.

Memory Map and Special Registers

The way this interpreter works is that is emulates an idealized machine, one built for executing Forth directly. As such it has to make compromises and treats certain sectors of memory as being special, as shown below (numbers are given in *hexadecimal* and are multiples of the virtual machines word-size which is either 16, 32 or 64 bit depending on compile time options.

Where the dictionary ends and the variable and return stacks begin depends on how much memory was allocated to the interpreter (with a minimum of 2048 words), the default is 32768 words, and the following diagram assumes this:




```
V stack = The Variable Stack
R stack = The Return Stack
```

Each may be further divided into special sections:

Registers

At the beginning of the Forth virtual machine there is a section used for registers, modifying them arbitrary can cause undefined behavior to occur which will most likely cause the virtual machine to be terminated.

NAME	LOCATION		DESCRIPTION
	DECIMAL	HEX	
	0-1	0-1	Unused
	2-5	2-5	Push integer word
DIC	6	6	Dictionary pointer
RSTK	7	7	Return stack pointer
STATE	8	8	Interpreter state; compile or command mode
BASE	9	9	Base conversion variable
PWD	10	A	Pointer to last defined word
SOURCE_ID	11	B	Input source selector (-1 = string input, 0 = file input)
SIN	12	C	String input pointer
SIDX	13	D	String input index (index into SIN)
SLEN	14	E	String input length (length of SIN)
START_ADDR	15	F	Pointer to start of VM
FIN	16	10	File input pointer

FOUT	17	11	File output pointer
STDIN	18	12	File pointer to stdin, if available
STDOUT	19	13	File pointer to stdout, if available
STDERR	20	14	File pointer to stderr, if available
ARGC	21	15	Count of arguments passed to program, if available
ARGV	22	16	An array of pointers to NUL terminated ASCII strings, if available, of ARGC length
DEBUG	23	17	Turn debugging on/off if enabled
INVALID	24	18	If non zero, this interpreter is invalid
TOP	25	19	Stored version of top of stack
INSTRUCTION	26	1A	Stored version of instruction pointer
STACK_SIZE	27	1B	Size of the variable stack
ERROR_HANDLER	28	1C	Action to take on error
THROW	29	1D	Used for throw/catch
SCRATCH_X	30	1E	Fixed scratch variable for the user
SCRATCH_Y	31	1F	Fixed scratch variable for the user

Dictionary

Apart from the constraints that the dictionary begins after where the registers are and before where V stack is there are no set demarcations for each region, although currently the defined word region ends before 0x200 leaving room between that and 0x7BFF for user defined words.

```
.------------------------------------------------------------------.
| 40-???            | ???-???          | ???-7BFF                  |
.------------------------------------------------------------------.
| Special read word | Interpreter word | Defined word ...          |
.------------------------------------------------------------------.

Special read word = A word called on entrance to the interpreter,
                    it calls itself recursively (as a tail call). This
                    word cannot be 'found', it does not have a name.
Interpreter word  = Any named (not 'invisible' ones) interpreter word
                    gets put here.
Defined word      = A list of words that have been defined with ':'
```

Glossary of Forth words

Each word is also given with its effect on the variable stack, any other effects are documented (including the effects on other stacks). Each entry looks like this:

- word (y – z)

Where ‘word’ is the word being described, the contents between the parenthesis describe the stack effects, this word expects one number to be one the stack, ‘y’, and returns a number to the stack ‘z’.

Internal words

There are three types of words.

‘Invisible’ words These invisible words have no name but are used to implement the Forth. They are all *immediate* words.

- push (– x)

Push the next value in the instruction stream onto the variable stack, advancing the instruction stream.

- compile (–)

Compile a pointer to the next instruction stream value into the dictionary.

- run (–)

Save the current instruction stream pointer onto the return stack and set the pointer instruction stream pointer to point to value after *run*.

Immediate words These words are named and are *immediate* words.

- ‘:’ (–)

Read in a new word from the input stream and compile it into the dictionary.

- ‘immediate’ (–)

Make the previously declared word immediate. Unlike in most Forth implementations this is used after the words name is given not after the final ‘;’ has been reached.

So:

```
: word immediate ... ;
```

Instead of:

```
: word ... ; immediate
```

- '\' (–)

A comment, ignore everything until the end of the line.

Compiling words

- 'read' (–)

read is a complex word that implements most of the input interpreter, it reads in a Forth *word* (up to 31 characters), if this *word* is in the *dictionary* it will either execute the word if we are in *command mode* or compile a pointer to the executable section of the word if in *compile mode*. If this *word* is not in the *dictionary* it is checked if it is a number, if it is then in *command mode* we push this value onto the *variable stack*, if in *compile mode* then we compile a *literal* into the *dictionary*. If it is none of these we print an error message and attempt to read in a new word.

- '@' (address – x)

Pop an address and push the value at that address onto the stack.

- '!' (x address –)

Given an address and a value, store that value at that address.

- 'c@' (char-address – char)

Pop a character address and push the character value at that address onto the stack. Note that this access is not checked for being within range of the virtual machines memory, but it is still relative to the start address of virtual machine memory.

- 'c!' (char char-address –)

Given a character address, store a character value at that address, like 'c@' the address is relative to the virtual machines starting address.

- ‘-’ (x y – z)

Pop two values, subtract ‘y’ from ‘x’ and push the result onto the stack.

- ‘+’ (x y – z)

Pop two values, add ‘y’ to ‘x’ and push the result onto the stack.

- ‘and’ (x y – z)

Pop two values, compute the bitwise ‘AND’ of them and push the result on to the stack.

- ‘or’ (x y – z)

Pop two values, compute the bitwise ‘OR’ of them and push the result on to the stack.

- ‘xor’ (x y – z)

Pop two values, compute the bitwise ‘XOR’ of them and push the result on to the stack.

- ‘invert’ (x y – z)

Perform a bitwise negation on the top of the stack.

- ‘lshift’ (x y – z)

Pop two values, compute ‘y’ shifted by ‘x’ places to the left and push the result on to the stack.

- ‘rshift’ (x y – z)

Pop two values, compute ‘y’ shifted by ‘x’ places to the right and push the result on to the stack.

- ‘*’ (x y – z)

Pop two values, multiply them and push the result onto the stack.

- ‘/’ (x y – z)

Pop two values, divide ‘x’ by ‘y’ and push the result onto the stack. If ‘y’ is zero and error message is printed and ‘x’ and ‘y’ will remain on the stack, but execution will continue on as normal.

- ‘u<’ (x y – z)

Pop two unsigned values, compare them (y < x) and push the result onto the stack, the comparison will be unsigned.

- ‘u>’ (x y – z)

Pop two values, compare them (y > x) and push the result onto the stack. The comparison will be unsigned.

- ‘exit’ (–)

Pop the return stack and set the instruction stream pointer to that value.

- ‘key’ (– char)

Get a value from the input and put it onto the stack.

- ‘_emit’ (char – status)

Put a character to the output stream returning a success value.

- ‘r>’ (– x)

Pop a value from the return stack and push it to the variable stack.

- ‘>r’ (x –)

Pop a value from the variable stack and push it to the return stack.

- ‘branch’ (–)

Jump unconditionally to the destination next in the instruction stream.

- ‘?branch’ (bool –)

Pop a value from the variable stack, if it is zero the jump to the destination next in the instruction stream, otherwise skip over it.

- 'pnum' (x – status)

Pop a value from the variable stack and print it to the output either as a ASCII decimal or hexadecimal value depending on the BASE register. A return status is pushed onto the stack, greater or equal to zero is a success, negative is a failure. Failure can occur because of an invalid base in the BASE register, or because the output could not be written to.

- '" (–)

Push the next value in the instruction stream onto the variable stack and advance the instruction stream pointer over it.

- ',' (x –)

Write a value into the dictionary, advancing the dictionary pointer.

- '=' (x y – z)

Pop two values, perform a test for equality and push the result.

- 'swap' (x y – y z)

Swap two values on the stack.

- 'dup' (x – x x)

Duplicate a value on the stack.

- 'drop' (x –)

Drop a value.

- 'over' (x y – x y x)

Duplicate the value that is next on the stack.

- 'find' (– execution-token)

Find a word in the dictionary pushing a pointer to that word onto the variable stack.

- 'depth' (– depth)

Push the current stack depth onto the stack, the value is the depth of the stack before the depth value was pushed onto the variable stack.

- 'sp@' (– addr)

Push the address of the stack pointer onto the stack, before **sp@** was executed:

```
1 2 sp@ . . .
```

Prints:

```
2 2 1
```

- 'sp!' (addr –)

Set the address of the stack pointer.

- 'clock' (– x)

Push the difference between the startup time and now, in milliseconds. This can be used for timing and implementing sleep functionality, the counter will not increase the interpreter is blocking and waiting for input, although this is implementation dependent.

- 'evaluator' (c-addr u 0 | file-id 0 1 – x)

This word is a primitive used to implement 'evaluate' and 'include-file', it takes a boolean to decide whether it will read from a file (1) or a string (0), and then takes either a forth string, or a **file-id**.

- 'system' (c-addr u – status)

Execute a command with the systems command interpreter.

File Access Words The following compiling words are part of the File Access Word set, a few of the fields need explaining in the stack comments. “file-id” refers to a previously opened file as returned by “open-file”, “ior” refers to a return status provided by the file operations. “fam” is a file access method,

- ‘close-file’ (file-id – ior)

Close an already opened file.

- ‘open-file’ (c-addr u fam – file-id ior)

Open a file, given a Forth string (the ‘c-addr’ and the ‘u’ arguments), and a file access method, which is defined within “forth.fth”. Possible file access methods are “w/o”, “r/w” and “r/o” for read only, read-write and write only respectively.

- ‘delete-file’ (c-addr u – ior)

Delete a file on the file system given a Forth string.

- ‘read-file’ (c-addr u file-id – ior)

Read in ‘u’ characters into ‘c-addr’ given a file identifier.

- ‘write-file’ (c-addr u file-id – ior)

Write ‘u’ characters from ‘c-addr’ to a given file identifier.

- ‘file-position’ (file-id – ud ior)

Get the file position offset from the beginning of the file given a file identifier.

- ‘reposition-file’ (ud file-id – ior)

Reposition a files offset relative to the beginning of the file given a file identifier.

- flush-file (file-id – ior)

Attempt to flush any buffered information written to a file.

- rename-file (c-addr1 u1 c-addr2 u2 – ior)

Rename a file on the file system named by the first string (‘c-addr1’ and ‘u1’) to the second string (‘c-addr2’ and ‘u2’).

Defined words

Defined words are ones which have been created with the ':' word, some words get defined before the user has a chance to define their own to make their life easier.

- 'state' (– addr)

Push the address of the register that controls the interpreter state onto the stack, this value can be written to put the interpreter into compile or command modes.

- ';' (–)

Write 'exit' into the dictionary and switch back into command mode.

- 'base' (– addr)

This pushes the address of a variable used for input and output conversion of numbers, this address can be written to and read, valid numbers to write are 0 and 2 to 36 (*not* 1).

- 'pwd' (– pointer)

Pushes a pointer to the previously define word onto the stack.

- 'h' (– pointer)

Push a pointer to the dictionary pointer register.

- 'r' (– pointer)

Push a pointer to the register pointer register.

- 'here' (– dictionary-pointer)

Push the current dictionary pointer (equivalent to "h @").

- '[' (–)

Immediately switch into command mode.

- ‘]’ (–)

Switch into compile mode

- ‘>mark’ (– location)

Write zero into the head of the dictionary and advance the dictionary pointer, push a address to the zero written into the dictionary. This is usually used after in a word definition that changes the control flow, after a branch for example.

- ‘:noname’ (– execution-token)

This creates a word header for a word without a name and switches to compile mode, the usual ‘;’ finishes the definition. It pushes a execution token onto the stack that can be written into the dictionary and run, or executed directly.

- ‘if’ (bool –)

Begin an if-else-then statement. If the top of stack is true then we execute all between the if and a corresponding ‘else’ or ‘then’, otherwise we skip over it.

Abstract Examples:

```
: word ... bool if do-stuff ... else do-other-stuff ... then ... ;

: word ... bool if do-stuff ... then ... ;
```

and a concrete examples:

```
: test-word if 2 2 + . cr else 3 3 * . cr ;
0 test-word
4             # prints 4
1 test-word
9             # prints 9
```

Is a simple and contrived example.

- ‘else’ (–)

See ‘if’.

- ‘then’ (–)

See ‘if’.

- ‘begin’ (–)

This marks the beginning of a loop.

- ‘until’ (bool –)

Loop back to the corresponding ‘begin’ if the top of the stack is zero, continue on otherwise.

- “ ‘)’ ” (– char)

Push the number representing the ‘)’ character onto the stack.

- ‘tab’ (–)

Print a tab.

- ‘cr’ (–)

Prints a newline.

- ‘(’ (–)

This will read the input stream until encountering a ‘)’ character, it is used for comments.

- ‘allot’ (amount –)

Allocate a number of cells in the dictionary.

- ‘tuck’ (x y – y x y)

The stack comment documents this word entirely.

- ‘nip’ (x y – y)

The stack comment documents this word entirely.

- ‘rot’ (x y z – z x y)

The stack comment documents this word entirely. This word rotates three items on the variable stack.

- '-rot' (x y z – y z x)

The stack comment documents this word entirely. This word rotates three items on the variable stack, in the opposite direction of "rot".

- 'emit' (x –)

Write a single character out to the output stream.

Library of Forth words

The file forth.fth contains many defined words, however those words are documented within that file and so as to avoid duplication will not be mentioned here. This file is *not* loaded automatically, and so should be run like this:

Unix:

```
./forth -t forth.fth
```

Windows

```
forth.exe -t forth.fth
```

Glossary of Forth terminology

- Word vs Machine-Word

Usually in computing a 'word' refers to the natural length of integer in a machine, the term 'machine word' is used to invoke this specific meaning, a word in Forth is more analogous to a function, but there are different types of Forth words; *immediate* and *compiling* words, *internal* and *defined* words and finally *visible* and *invisible* words.

The distinction between a machine word and a Forth word can lead to some confusion.

- *The* dictionary

There is only one dictionary in a normal Forth implementation, it is a data structure that can only grow in size (or at least it can in this implementation) and holds all of the defined words.

- *The* stack

When we referring to a stack, or the stack, we refer to the variable stack unless otherwise stated (such as the return stack). The variable, or the stack, holds the result of recent operations such as addition or subtraction.

- The return stack

Forth implementations are two (or more) stack machines. The second stack is the return stack which holds the usual function call return values as well as temporary variables.

- Defined Words

A defined word is one that is not implement directly by the interpreter but has been create with the ':' word. It can be an *immediate* word, but does not have to be.

- Compile mode

In this mode we *compile* words unless those words are *immediate* words, if the are then we immediately execute them.

- Command mode

In this mode, regardless of whether we are in *command* or *compile* mode we execute words or push them on to the stack.

- A block.

A Forth block is primitive way of managing persistent storage and this version of block interface is more primitive than most. A block is a contiguous range of bytes, usually 1024 of them as in this instance, and they can be written or read from disk. Flushing of dirty blocks is not performed in this implementation and must be done 'manually'.

Porting this interpreter

The interpreter code is written in C99, and is written to be portable, however porting it to embedded platforms that lack a C standard library (which is most of them) would mean replacing the most of the C standard library functions used, and implementing a new I/O mechanism for reading, printing and block storage.

The interpreter has been tested on the following platforms:

- Linux x86-64 with,
- Debian Jessie (8.x)
- GCC version 4.9.2
- Windows 7 x86-64 (not recently)
- Linux ARM 32-bit Little Endian (not recently)
- OSX Sierra 10.12.1 (tested by Rikard Lang).

And the different virtual machine word size options (32 and 64 bit machine words) have been tested. There is no reason it should not also work on 16-bit platforms.

libforth is also available as a Linux Kernel Module, on a branch of libforth, see https://github.com/howerj/libforth/tree/linux-kernel-module. This is module is very experimental, and it is quite possible that it will make your system unstable.

Standards compliance

This Forth interpreter is in no way compliant with any of the standards relating to Forth, such as ANS Forth, or previous Forth standardization efforts. However attempts to support words and behavior typical of these standards are made.

Some important deviations are:

- immediate

In most Forths the "immediate" word goes after a words definition instead of inside it like this:

```
: word ... ; immediate
```

Instead this interpreter does it:

```
: word immediate ... ;
```

This behavior will not be changed for the foreseeable future, although it is the biggest difference. This is due to how the internals of the interpreter work.

- recursion and definition hiding

A word can be called immediately before the terminating semi-colon has been reached, in the middle of a word definition. This makes the recurse keyword redundant but means using a previous definition of a word with the same name more difficult (but can be done). This might be a candidate for behavior that should be made more compliant.

- ok

'ok' is not printed after a successful command execution , this is for two reasons, firstly because of limitations in the implementation, and secondly there is no reason for cluttering up the output window with this. The implementation should be silent by default.

Bugs

As mentioned in the standards compliance section, this Forth does things in a non-standard way. Apart from that:

- Passing invalid pointers to instructions like **open-file** or **system** can cause undefined behavior (your program will most likely crash). There is no simple way to handle this (apart from not doing it).
- The core interpreter does not currently make use of the throw and catch mechanism when handling certain errors (like division by zero), in effect there are two error handlers. These mechanisms need unifying.

To-Do

- Port this to a micro controller, and a Linux kernel module device pointer to the forth object, a pointer to the stack and the stack depth.
- A few environment variables could be used to specify start up files for the interpreter and user specific startup files.
- Add save-core, number, parse, load-core, more-core to the virtual machine.
- Signal handling should be added, so the Forth program can handle them.
- Add loading in a Forth image from a memory structure, this will need to be in a portable Format.
- Error handling could be improved - the latest word definition should be erased if an error occurs before the terminating ';'
- For a Forth only related "To-Do" list see the end of the file forth.fth.
- A compiler for the virtual machine itself should be made, as a separate program. This could be used to make a more advanced read-evaluate loop.
- Core files are currently not portable across machines of different words sizes or endianess, which needs addressing.

- Character addressing should be used throughout the interpreter, instead of cell addressing and conversion to/from character addresses. Real address could also be used, but this would make core files non-portable.
- The unit tests in unit.c could be integrated with the main program.
- Due to the way Windows opens stdin, stdout, and stderr, there are problems reading in binary files, this can be remedied quite easily with some code that executes before **main_forth** is called.

include "libforth.h"

ifdef _WIN32 #include #include #include #endif

int main(int argc, char **argv) { #ifdef *WIN32* setmode(*fileno(stdin),* O_BINARY); *setmode(*fileno(stdout), *O*BINARY); #endif return main_forth(argc, argv); }

Notes

- The compilation should result in a small executable, and when statically linked against musl under Linux (x86-84), the stripped executable is around 50kb in size.
- It is quite possible to make Forth programs that corrupt memory that they should, this is not a design flaw in this interpreter but more part of the Forth philosophy. If you want memory safety (and most of the time you should) you should use a different language, or implementation.