Contents

libforth.c.md

- libforth.c
- Richard James Howe.
- Copyright 2015,2016 Richard James Howe.
- MIT
- howe.r.j.89@gmail.com

A FORTH library, written in a literate style.

License

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.

Introduction

This file implements the core Forth interpreter, it is written in portable C99. The file contains a virtual machine that can interpret threaded Forth code and a simple compiler for the virtual machine, which is one of its instructions. The interpreter can be embedded in another application and there should be no problem instantiating multiple instances of the interpreter.

For more information about Forth see:

- https://en.wikipedia.org/wiki/Forth/_%28programming/_language%29
- Thinking Forth by Leo Brodie
- Starting Forth by Leo Brodie

A glossary of words for FIG FORTH 79:

- http://www.dwheeler.com/6502/fig-forth-glossary.txt

And the more recent and widespread standard for ANS Forth:

- http://lars.nocrew.org/dpans/dpans.htm

The antecedent of this interpreter:

- http://www.ioccc.org/1992/buzzard.2.c

cxxforth, a literate Forth written in C++

- https://github.com/kristopherjohnson/cxxforth

Jones Forth, a literate Forth written in x86 assembly:

- https://rwmj.wordpress.com/2010/08/07/jonesforth-git-repository/
- https://github.com/AlexandreAbreu/jonesforth (backup)

A Forth processor:

- http://www.excamera.com/sphinx/fpga-j1.html

And my Forth processor based on this one:

- https://github.com/howerj/fyp

The repository should also contain:

- "readme.md" : a Forth manual, and generic project information
- "forth.fth" : basic Forth routines and startup code
- "libforth.h" : The header contains the API documentation

The structure of this file is as follows:

1) Headers and configuration macros
2) Enumerations and constants
3) Helping functions for the compiler
4) API related functions and Initialization code
5) The Forth virtual machine itself
6) An example main function called **main_forth** and support functions

Each section will be explained in detail as it is encountered.

An attempt has been made to make this document flow, as both a source code document and as a description of how the Forth kernel works. This is helped by the fact that the program is small and compact without being written in obfuscated C. It is, as mentioned, compact, and can be difficult to understand regardless of code quality. Some of the semantics of Forth will not be familiar to C programmers.

A basic understanding of how to use Forth would help as this document is meant to describe how a Forth implementation works and not as an introduction to the language. A quote about the language from Wikipedia best sums the language up:

```
"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."
```

Forth has a philosophy like most languages, one of simplicity, compactness and of trying only to solve the problem at hand, even going as far as to try to simplify the problem or replace the problem (which may span multiple domains, not just software) with a simpler one. This is often not a realistic way of tackling things and Forth has fallen out of favor, it is nonetheless an interesting language which can be implemented and understood by a single programmer (another integral part of the Forth philosophy).

The core of the concept of the language - simplicity I would say - is achieved by the following:

1) The language uses Reverse Polish Notation to enter expressions and parsing is simplified to the extreme with space delimited words and numbers being the most complex terms. This means a abstract syntax tree does not need to be constructed and terms can be executed as soon as they are parsed. The *parser* can described in only a handful of lines of C.

2) The language uses concatenation of Forth words (called functions in other language) to create new words, this allows for small programs to be created and encourages *factoring* definitions into smaller words.

3) The language is untyped.

4) Forth functions, or words, take their arguments implicitly and return variables implicitly via a variable stack which the programmer explicitly interacts with. A comparison of two languages behavior best illustrates the point, we will define a function in C and in Forth that simply doubles a number. In C this would be:

   int double_number(int x) { return x << 1; }

And in Forth it would be:

```
: 2* 1 lshift ;
```

No types are needed, and the arguments and the return values are not stated, unlike in C. Although this has the advantage of brevity, it is now up to the programmer to manages those variables.

5) The input and output facilities are set up and used implicitly as well. Input is taken from **stdin** and output goes to **stdout**, by default. Words that deal with I/O uses these file steams internally.
6) Error handling is traditionally non existent or limited.
7) This point is not a property of the language, but part of the way the Forth programmer must program. The programmer must make their factored word definitions *flow*. Instead of reordering the contents of the stack for each word, words should be made so that the reordering does not have to take place (ie. Manually performing the job of a optimizing compile another common theme in Forth, this time with memory reordering).

The implicit behavior relating to argument passing and I/O really reduce program size, the type of implicit behavior built into a language can really define what that language is good for. For example AWK is naturally good for processing text, thanks in large part to sensible defaults for how text is split up into lines and records, and how input and output is already set up for the programmer.

An example of this succinctness in AWK is the following program, which can be typed in at the command line. It will read from the standard input if no files are given, and print any lines longer than eighty characters along with the line number of that line:

```
awk '{line++}length > 80 {printf "%04u: %s\n", line, $0}' file.txt ...
```

For more information about AWK see:

- http://www.grymoire.com/Unix/Awk.html
- https://en.wikipedia.org/wiki/AWK
- http://www.pement.org/awk/awk1line.txt

Forth likewise can achieve succinctness and brevity because of its implicit behavior.

Naturally we try to adhere to Forth philosophy, but also to Unix philosophy (which most Forths do not do), this is described later on.

Glossary of Terms:

```
VM              - Virtual Machine
Cell            - The Virtual Machines natural Word Size, on a 32 bit
            machine the Cell will be 32 bits wide
Word            - In Forth a Word refers to a function, and not the
            usual meaning of an integer that is the same size as
            the machines underlying word size, this can cause confusion
API             - Application Program Interface
```

```
interpreter    - as in byte code interpreter, synonymous with virtual
           machine.
REPL           - Read-Evaluate-Print-Loop, this Forth actually provides
           something more like a "REL", or Read-Evaluate-Loop (as printing
           has to be done explicitly), but the interpreter is interactive
           which is the important point
RPN            - Reverse Polish Notation (see
           <https://en.wikipedia.org/wiki/Reverse_Polish_notation>).
           The Forth interpreter uses RPN to enter expressions.
The stack      - Forth implementations have at least two stacks, one for
           storing variables and another for control flow and temporary
           variables, when the term *stack* is used on its own and with
           no other context it refers to the *variable stack* and not
           the *return stack*. This *variable stack* is used for
           passing parameters into and return values to functions.
Return stack   - Most programming languages have a call stack, C has one
           but not one that the programmer can directly access, in
           Forth manipulating the return stack is often used.
factor         - factoring is splitting words into smaller words that
           perform a specific function. To say a word is a natural
           factor of another word is to say that it makes sense to take
           some functionality of the word to be factored and to create
           a new word that encapsulates that functionality. Forth
           encourages heavy factoring of definitions.
Command mode   - This mode executes both compiling words and immediate
           words as they are encountered
Compile mode   - This mode executes immediate words as they are
           encountered, but compiling words are compiled into the
           dictionary.
Primitive      - A word whose instruction is built into the VM.
```

Headers and configurations macros

This file implements a Forth library, so a Forth interpreter can be embedded in another application, as such a subset of the functions in this file are exported, and are documented in the *libforth.h* header

```
0001 #include "libforth.h"
```

We try to make good use of the C library as even microcontrollers have enough space for a reasonable implementation of it, although it might require some setup. The only time allocations are explicitly done is when the virtual machine image is initialized, after this the VM does not allocate any more memory.

```
0002 #include <assert.h>
```

```
0003 #include <stdarg.h>
0004 #include <ctype.h>
0005 #include <errno.h>
0006 #include <limits.h>
0007 #include <stdlib.h>
0008 #include <string.h>
0009 #include <setjmp.h>
0010 #include <time.h>
```

Some forward declarations are needed for functions relating to logging.

```
0011 static const char *emsg(void);
0012 static int logger(const char *prefix, const char *func,
0013        unsigned line, const char *fmt, ...);
```

Some macros are also needed for logging. As an aside, **___VA__ARGS___** should be prepended with '##' in case zero extra arguments are passed into the variadic macro, to swallow the extra comma, but it is not *standard* C, even if most compilers support the extension.

```
0014 #define fatal(FMT,...)   logger("fatal",  __func__, __LINE__, FMT, __VA_ARGS__)
0015 #define error(FMT,...)   logger("error",  __func__, __LINE__, FMT, __VA_ARGS__)
0016 #define warning(FMT,...) logger("warning",__func__, __LINE__, FMT, __VA_ARGS__)
0017 #define note(FMT,...)    logger("note",   __func__, __LINE__, FMT, __VA_ARGS__)
0018 #define debug(FMT,...)   logger("debug",  __func__, __LINE__, FMT, __VA_ARGS__)
```

Traditionally Forth implementations were the only program running on the (micro)computer, running on processors orders of magnitude slower than this one, as such checks to make sure memory access was in bounds did not make sense and the implementation had to have access to the entire machines limited memory.

To aide debugging and to help ensure correctness the **ck** macro, a wrapper around the function **check__bounds**, is called for most memory accesses that the virtual machine makes.

```
0019 #ifndef NDEBUG
```

This is a wrapper around **check__bounds**, so we do not have to keep typing in the line number, as so the name is shorter (and hence the checks are out of the way visually when reading the code).

```
0020 #define ck(C) check_bounds(o, &on_error, (C), __LINE__, o->core_size)
```

This is a wrapper around **check_bounds**, so we do not have to keep typing in the line number, as so the name is shorter (and hence the checks are out of the way visually when reading the code). This will check character pointers instead of cell pointers, like **ck** does.

```
0021 #define ckchar(C) check_bounds(o, &on_error, (C), __LINE__, \
0022            o->core_size * sizeof(forth_cell_t))
```

This is a wrapper around **check_depth**, to make checking the depth short and simple.

```
0023 #define cd(DEPTH) check_depth(o, &on_error, S, (DEPTH), __LINE__)
```

This macro makes sure any dictionary pointers never cross into the stack area.

```
0024 #define dic(DPTR) check_dictionary(o, &on_error, (DPTR))
```

This macro wraps up the tracing function, which we may want to remove.

```
0025 #define TRACE(ENV,INSTRUCTION,STK,TOP) trace(ENV,INSTRUCTION,STK,TOP)
0026 #else
```

The following are defined only if we remove the checking and the debug code.

```
0027 #define ck(C) (C)
0028 #define ckchar(C) (C)
0029 #define cd(DEPTH) ((void)DEPTH)
0030 #define dic(DPTR) check_dictionary(o, &on_error, (DPTR))
0031 #define TRACE(ENV, INSTRUCTION, STK, TOP)
0032 #endif
```

Default VM size which should be large enough for any Forth application.

```
0033 #define DEFAULT_CORE_SIZE   (32 * 1024)
```

When we are reading input to be parsed we need a space to hold that input, the offset to this area is into a field called **m** in **struct forth**, defined later, the offset is a multiple of cells and not chars.

```
0034 #define STRING_OFFSET       (32u)
```

This defines the maximum length of a Forth words name, that is the string that represents a Forth word, this number is in cells (or machine words).

```
0035 #define MAXIMUM_WORD_LENGTH (32u)
```

The minimum stack size of both the variable and return stack, the stack size should not be made smaller than this otherwise the built in code and code in *forth.fth* will not work.

```
0036 #define MINIMUM_STACK_SIZE  (64u)
```

The start of the dictionary is after the registers and the **STRING_OFFSET**, this is the area where Forth definitions are placed.

@note The string offset could be placed after the end of the dictionary to save space, in the area between the end of the dictionary and the beginning of the pad area.

```
0037 #define DICTIONARY_START (STRING_OFFSET+MAXIMUM_WORD_LENGTH)
```

Later we will encounter a field called **MISC**, a field in every Word definition and is always present in the Words header. This field contains multiple values at different bit offsets, only the lower 16 bits of this cell are ever used. The next macros are helper to extract information from the **MISC** field.

The bit offset for word length start.

```
0038 #define WORD_LENGTH_OFFSET  (8)
```

WORD_LENGTH extracts the length of a Forth words name so we know where it is relative to the **PWD** field of a word.

```
0039 #define WORD_LENGTH(MISC) (((MISC) >> WORD_LENGTH_OFFSET) & 0xff)
```

Test if a word is a **hidden** word, one that is not in the search order for the dictionary.

```
0040 #define WORD_HIDDEN(MISC) ((MISC) & 0x80)
```

The lower 7 bits of the MISC field are used for the VM instruction, limiting the number of instructions the virtual machine can have in it, the higher bits are used for other purposes.

```
0041 #define INSTRUCTION_MASK    (0x7f)
```

A mask that the VM uses to extract the instruction.

```
0042 #define instruction(k)      ((k) & INSTRUCTION_MASK)
```

VERIFY is our assert macro that will always been defined regardless of whether **NDEBUG** is defined.

```
0043 #define VERIFY(X)          do { if(!(X)) { abort(); } } while(0)
```

The **IS_BIG_ENDIAN** macro looks complicated, however all it does is determine the endianess of the machine using trickery.

See:

- https://stackoverflow.com/questions/2100331
- https://en.wikipedia.org/wiki/Endianness

For more information and alternatives.

```
0044 #define IS_BIG_ENDIAN (!(union { uint16_t u16; uint8_t c; }){ .u16 = 1 }.c)
```

Enumerations and Constants

This following string is a forth program that gets called when creating a new Forth environment, it is not actually the first program that gets run, but it is run before the user gets a chance to do anything.

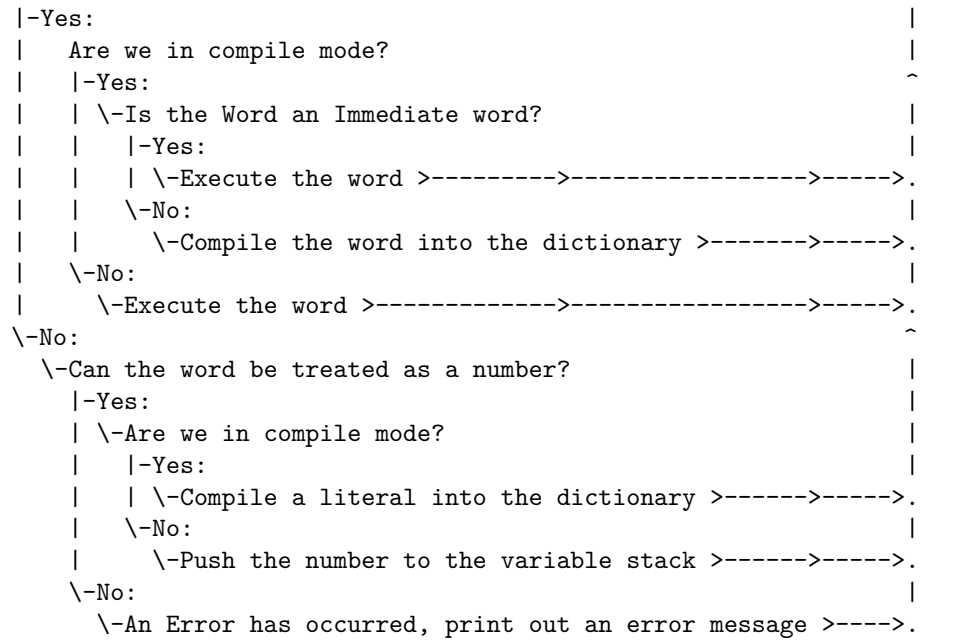
The program is kept as small as possible, but is dependent on the virtual machine image being set up correctly with other, basic, words being defined first, they will be described as they are encountered. Suffice to say, before this program is executed the following happens:

1) The virtual machine image is initialized
2) All the virtual machine primitives are defined
3) All registers are named and some constants defined
4) **;** is defined

Of note, words such as **if**, **else**, **then**, and even comments - **(** -, are not actually Forth primitives, there are defined in terms of other Forth words.

The Forth interpreter is a simple loop that does the following:

```
Start the interpreter loop <-----------<-----------------<---.
Get a space delimited word                                    \
Attempt to look up that word in the dictionary                 \
Was the word found?                                             ^
```



As you can see, there is not too much too it, however there are still a lot of details left out, such as how exactly the virtual machine executes words and how this loop is formed.

A short description of the words defined in **initial_forth_program** follows, bear in mind that they depend on the built in primitives, the named registers being defined, as well as **state** and **;**.

```
here      - push the current dictionary pointer
[         - immediately enter command mode
]         - enter compile mode
>mark     - make a hole in the dictionary and push a pointer to it
:noname   - make an anonymous word definition, push token to it, the
    definition is terminated by ';' like normal word definitions.
if        - immediate word, begin if...else...then clause
else      - immediate word, optional else clause
then      - immediate word, end if...else...then clause
begin     - immediate word, start a begin...until loop
until     - immediate word, end begin...until loop, jump to matching
    begin at run time if top of stack is zero.
')'       - push a ")" character to the stack
(         - begin a Forth comment, terminated by a )
rot       - perform stack manipulation: x y z => y z x
-rot      - perform stack manipulation: x y z => z x y
tuck      - perform stack manipulation: x y   => y x y
```

```
nip       - perform stack manipulation: x y   => y
allot     - allocate space in the dictionary
bl        - push the space character to the stack
space     - print a space
.         - print out current top of stack, followed by a space

0045 static const char *initial_forth_program =
0046 ": here h @ ; \n"
0047 ": [ immediate 0 state ! ; \n"
0048 ": ] 1 state ! ; \n"
0049 ": >mark here 0 , ; \n"
0050 ": :noname immediate -1 , here 2 , ] ; \n"
0051 ": if immediate ' ?branch , >mark ; \n"
0052 ": else immediate ' branch , >mark swap dup here swap - swap ! ; \n"
0053 ": then immediate dup here swap - swap ! ; \n"
0054 ": begin immediate here ; \n"
0055 ": until immediate ' ?branch , here - , ; \n"
0056 ": ')' 41 ; \n"
0057 ": ( immediate begin key ')' = until ; \n"
0058 ": rot >r swap r> swap ; \n"
0059 ": -rot rot rot ; \n"
0060 ": tuck swap over ; \n"
0061 ": nip swap drop ; \n"
0062 ": allot here + h ! ; \n"
0063 ": 2drop drop drop ; \n"
0064 ": bl 32 ; \n"
0065 ": emit _emit drop ; \n"
0066 ": space bl emit ; \n"
0067 ": evaluate 0 evaluator ; \n"
0068 ": . pnum drop space ; \n";
```

This is a string used in number to string conversion in **number_printer**, which is dependent on the current base.

```
0069 static const char conv[] = "0123456789abcdefghijklmnopqrstuvwxzy";
```

int to **char*** map for file access methods.

```
0070 enum fams {
0071    FAM_WO,   /**< write only */
0072    FAM_RO,   /**< read only */
0073    FAM_RW,   /**< read write */
0074    LAST_FAM  /**< marks last file access method */
0075 };
```

These are the file access methods available for use when the virtual machine is up and running, they are passed to the built in primitives that deal with file input and output (such as open-file).

@note It might be worth adding more *fams*, which **fopen** can accept.

```
0076 static const char *fams[] = {
0077    [FAM_WO] = "wb",
0078    [FAM_RO] = "rb",
0079    [FAM_RW] = "w+b",
0080    NULL
0081 };
```

The following are different reactions errors can take when using **longjmp** to a previous **setjump**.

```
0082 enum errors
0083 {
0084    INITIALIZED, /**< setjmp returns zero if returning directly */
0085    OK,          /**< no error, do nothing */
0086    FATAL,       /**< fatal error, this invalidates the Forth image */
0087    RECOVERABLE, /**< recoverable error, this will reset the interpreter */
0088 };
```

We can serialize the Forth virtual machine image, saving it to disk so we can load it again later. When saving the image to disk it is important to be able to identify the file somehow, and to identify properties of the image.

Unfortunately each image is not portable to machines with different cell sizes (determined by "sizeof(forth_cell_t)") and different endianess, and it is not trivial to convert them due to implementation details.

enum header names all of the different fields in the header.

The first four fields (**MAGIC0**...**MAGIC3**) are magic numbers which identify the file format, so utilities like *file* on Unix systems can differentiate binary formats from each other.

CELL_SIZE is the size of the virtual machine cell used to create the image.

VERSION is used to both represent the version of the Forth interpreter and the version of the file format.

ENDIAN is the endianess of the VM

MAGIC7 is the last magic number.

When loading the image the magic numbers are checked as well as compatibility between the saved image and the compiled Forth interpreter.

```
0089 enum header { /**< Forth header description enum */
0090    MAGIC0,     /**< Magic number used to identify file type */
0091    MAGIC1,     /**< Magic number ... */
0092    MAGIC2,     /**< Magic number ... */
0093    MAGIC3,     /**< Magic number ...  */
0094    CELL_SIZE,  /**< Size of a Forth cell, or virtual machine word */
0095    VERSION,    /**< Version of the image */
0096    ENDIAN,     /**< Endianess of the interpreter */
0097    MAGIC7      /**< Final magic number */
0098 };
```

The header itself, this will be copied into the **forth_t** structure on initialization, the **ENDIAN** field is filled in then as it seems impossible to determine the endianess of the target at compile time.

```
0099 static const uint8_t header[MAGIC7+1] = {
0100    [MAGIC0]    = 0xFF,
0101    [MAGIC1]    = '4',
0102    [MAGIC2]    = 'T',
0103    [MAGIC3]    = 'H',
0104    [CELL_SIZE] = sizeof(forth_cell_t),
0105    [VERSION]   = FORTH_CORE_VERSION,
0106    [ENDIAN]    = -1,
0107    [MAGIC7]    = 0xFF
0108 };
```

The main structure used by the virtual machine is **forth_t**.

The structure is defined here and not in the header to hide the implementation details it, all API functions are passed an opaque pointer to the structure (see https://en.wikipedia.org/wiki/Opaque/_pointer).

Only three fields are serialized to the file saved to disk:

1) **header**
2) **core_size**
3) **m**

And they are done so in that order, **core_size** and **m** are save in whatever endianess the machine doing the saving is done in, however **core_size** is converted to a **uint64_t** before being save to disk so it is not of a variable size. **m** is a flexible array member **core_size** number of members.

The **m** field is the virtual machines working memory, it has its own internal structure which includes registers, stacks and a dictionary of defined words.

The **m** field is laid out as follows, assuming the size of the virtual machine is 32768 cells big:

```
.-----------------------------------------------.
| 0-3F      | 40-7BFF       |7C00-7DFF|7E00-7FFF|
.-----------------------------------------------.
| Registers | Dictionary... | V stack | R stack |
.-----------------------------------------------.

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

The dictionary has its own complex structure, and it always starts just after the registers. It includes scratch areas for parsing words, start up code and empty space yet to be consumed before the variable stack. The sizes of the variable and returns stack change depending on the virtual machine size. The structures within the dictionary will be described later on.

In the following structure, **struct forth**, values marked with a '~~' are serialized, the serialization takes place in order. Values are written out as they are with the exception of **core_size** which is converted to a **uint64_t** before serialization (it being a fixed width makes reading it back in from a file easier).

```
0109 struct forth { /**< FORTH environment */
0110    uint8_t header[sizeof(header)]; /**< ~~ header for core file */
0111    forth_cell_t core_size;  /**< ~~ size of VM */
0112    uint8_t *s;          /**< convenience pointer for string input buffer */
0113    char hex_fmt[16];    /**< calculated hex format */
0114    char word_fmt[16];   /**< calculated word format */
0115    forth_cell_t *S;     /**< stack pointer */
0116    forth_cell_t *vstart;/**< index into m[] where variable stack starts*/
0117    forth_cell_t *vend;  /**< index into m[] where variable stack ends*/
0118    const struct forth_functions *calls; /**< functions for CALL instruction */
0119    forth_cell_t m[];    /**< ~~ Forth Virtual Machine memory */
0120 };
```

This enumeration describes the possible actions that can be taken when an error occurs, by setting the right register value it is possible to make errors halt the interpreter straight away, or even to make it invalidate the core.

This does not override the behavior of the virtual machine when it detects an error that cannot be recovered from, only when it encounters an error such as a divide by zero or a word not being found, not when the virtual machine executes and invalid instruction (which should never normally happen unless something has been corrupted).

```
0121 enum actions_on_error
0122 {
0123    ERROR_RECOVER,    /**< recover when an error happens, like a call to ABORT */
0124    ERROR_HALT,       /**< halt on error */
0125    ERROR_INVALIDATE, /**< halt on error and invalid the Forth interpreter */
0126 };
```

These are the possible options for the debug registers.

```
0127 enum trace_level
0128 {
0129    DEBUG_OFF,         /**< tracing is off */
0130    DEBUG_FORTH_CODE,  /**< used within the forth interpreter */
0131    DEBUG_NOTE,        /**< print notes */
0132    DEBUG_INSTRUCTION, /**< instructions and stack are traced */
0133    DEBUG_CHECKS,      /**< bounds checks are printed out */
0134    DEBUG_ALL,         /**< trace everything that can be traced */
0135 };
```

There are a small number of registers available to the virtual machine, they are actually indexes into the virtual machines main memory, this is so that the programs running on the virtual machine can access them.

There are other registers that are in use that the virtual machine cannot access directly (such as the program counter or instruction pointer). Some of these registers correspond directly to well known Forth concepts, such as the dictionary and return stack pointers, others are just implementation details.

X-Macros are an unusual but useful method of making tables of data. We use this to store the registers name, it's address within the virtual machine and the enumeration for it.

More information about X-Macros can be found here:

- https://en.wikipedia.org/wiki/X/_Macro
- http://www.drdobbs.com/cpp/the-x-macro/228700289
- https://stackoverflow.com/questions/6635851

  0136 #define XMACRO_REGISTERS
  0137 X("h", DIC, 6, "dictionary pointer")
  0138 X("r", RSTK, 7, "return stack pointer")
  0139 X("state", STATE, 8, "interpreter state")\ 0140 X("base", BASE, 9, "base conversion variable")\ 0141 X("pwd", PWD, 10, "pointer to previous word")\ 0142 X("source-id", SOURCE_ID, 11, "input source selector")

0143 X(“sin", SIN, 12, "string input pointer")\ 0144 X("sidx”, SIDX, 13, “string input index”)
0145 X(“slen", SLEN, 14, "string input length")\ 0146 X("start-address”, START_ADDR, 15, “pointer to start of VM”)
0147 X(“fin", FIN, 16, "file input pointer")\ 0148 X("fout”, FOUT, 17, “file output pointer”)
0149 X(“stdin", STDIN, 18, "file pointer to stdin")\ 0150 X("stdout”, STDOUT, 19, “file pointer to stdout”)
0151 X(“stderr", STDERR, 20, "file pointer to stderr")\ 0152 X("argc”, ARGC, 21, “argument count”)
0153 X(“argv", ARGV, 22, "arguments")\ 0154 X("debug”, DEBUG, 23, “turn debugging on/off if enabled”)
0155 X(“invalid", INVALID, 24, "non-zero on serious error")\ 0156 X("top”, TOP, 25, “*stored* version of top of stack”)
0157 X(“instruction", INSTRUCTION, 26, "start up instruction")\ 0158 X("stack-size”, STACK_SIZE, 27, “size of the stacks”)
0159 X(“error-handler", ERROR_HANDLER, 28, "actions to take on error")\ 0160 X("handler”, THROW, 29, “exception handler is stored here”)
0161 X(“x", SCRATCH_X, 30, "scratch variable x")\ 0162 X("y”, SCRATCH_Y, 31, “scratch variable y”)

0163 enum registers { /**< virtual machine registers */ 0164 #define X(NAME, ENUM, VALUE, HELP) ENUM = VALUE, 0165 XMACRO_REGISTERS 0166 #undef X 0167 };

0168 static const char *register_names[] = { /**< names of VM registers */ 0169 #define X(NAME, ENUM, VALUE, HELP) NAME, 0170 XMACRO_REGISTERS 0171 #undef X 0172 NULL 0173 };

The enum **input_stream** lists values of the **SOURCE_ID** register.

Input in Forth systems traditionally (tradition is a word we will keep using here, generally in the context of programming it means justification for cruft) came from either one of two places, the keyboard that the programmer was typing at, interactively, or from some kind of non volatile store, such as a floppy disk. Our C program has no portable way of interacting directly with the keyboard, instead it could interact with a file handle such as **stdin**, or read from a string. This is what we do in this interpreter.

A word in Forth called **SOURCE-ID** can be used to query what the input device currently is, the values expected are zero for interactive interpretation, or minus one (minus one, or all bits set, is used to represent truth conditions in most Forths, we are a bit more liberal in our definition of true) for string input. These are the possible values that the **SOURCE_ID** register can take. The **SOURCE-ID** word, defined in *forth.fth*, then does more processing of this word.

Note that the meaning is slightly different in our Forth to what is meant traditionally, just because this program is taking input from **stdin** (or possibly another file handle), does not mean that this program is being run interactively, it could possibly be part of a Unix pipe, which is the reason the interpreter defaults to being as silent as possible.

```
0174 enum input_stream {
0175    FILE_IN,       /**< file input; this could be interactive input */
0176    STRING_IN = -1 /**< string input */
0177 };
```

enum instructions contains each virtual machine instruction, a valid instruction is less than LAST. One of the core ideas of Forth is that given a small set of primitives it is possible to build up a high level language, given only these primitives it is possible to add conditional statements, case statements, arrays and strings, even though they do not exist as instructions here.

Most of these instructions are simple (such as; pop two items off the variable stack, add them and push the result for **ADD**) however others are a great deal more complex and will require paragraphs to explain fully (such as **READ**, or how **IMMEDIATE** interacts with the virtual machines execution).

The instruction name, enumeration and a help string, are all stored with an X-Macro.

Some of these words are not necessary, that is they can be implemented in Forth, but they are useful to have around when the interpreter starts up for debugging purposes (like **pnum**).

```
0178 #define XMACRO_INSTRUCTIONS\
0179  X(PUSH,      "push",       " -- x : push a literal")\
0180  X(COMPILE,   "compile",    " -- : compile a pointer to a Forth word")\
0181  X(RUN,       "run",        " -- : run a Forth word")\
0182  X(DEFINE,    "define",     " -- : make new Forth word, set compile mode")\
0183  X(IMMEDIATE, "immediate",  " -- : make a Forth word immediate")\
0184  X(READ,      "read",       " -- : read in a Forth word and execute it")\
0185  X(LOAD,      "@",          "addr -- x : load a value")\
0186  X(STORE,     "!",          "x addr -- : store a value")\
0187  X(CLOAD,     "c@",         "c-addr -- x : load character value")\
0188  X(CSTORE,    "c!",         "x c-addr -- : store character value")\
0189  X(SUB,       "-",          "x1 x2 -- x3 : subtract x2 from x1 yielding x3")\
0190  X(ADD,       "+",          "x x -- x : add two values")\
0191  X(AND,       "and",        "x x -- x : bitwise and of two values")\
0192  X(OR,        "or",         "x x -- x : bitwise or of two values")\
0193  X(XOR,       "xor",        "x x -- x : bitwise exclusive or of two values")\
0194  X(INV,       "invert",     "x -- x : invert bits of value")\
0195  X(SHL,       "lshift",     "x1 x2 -- x3 : left shift x1 by x2")\
```

```
0196  X(SHR,       "rshift",     "x1 x2 -- x3 : right shift x1 by x2")\
0197  X(MUL,       "*",          "x x -- x : multiply to values")\
0198  X(DIV,       "/",          "x1 x2 -- x3 : divide x1 by x2 yielding x3")\
0199  X(ULESS,     "u<",         "x x -- bool : unsigned less than")\
0200  X(UMORE,     "u>",         "x x -- bool : unsigned greater than")\
0201  X(EXIT,      "exit",       " -- : return from a word defition")\
0202  X(KEY,       "key",        " -- char : get one character of input")\
0203  X(EMIT,      "_emit",      " char -- status : get one character of input")\
0204  X(FROMR,     "r>",         " -- x, R: x -- : move from return stack")\
0205  X(TOR,       ">r",         "x --, R: -- x : move to return stack")\
0206  X(BRANCH,    "branch",     " -- : unconditional branch")\
0207  X(QBRANCH,   "?branch",    "x -- : branch if x is zero")\
0208  X(PNUM,      "pnum",       "x -- : print a number")\
0209  X(QUOTE,     "'",          " -- addr : push address of word")\
0210  X(COMMA,     ",",          "x -- : write a value into the dictionary")\
0211  X(EQUAL,     "=",          "x x -- bool : compare two values for equality")\
0212  X(SWAP,      "swap",       "x1 x2 -- x2 x1 : swap two values")\
0213  X(DUP,       "dup",        "x -- x x : duplicate a value")\
0214  X(DROP,      "drop",       "x -- : drop a value")\
0215  X(OVER,      "over",       "x1 x2 -- x1 x2 x1 : copy over a value")\
0216  X(TAIL,      "tail",       " -- : tail recursion")\
0217  X(FIND,      "find",       "c\" xxx\" -- addr | 0 : find a Forth word")\
0218  X(DEPTH,     "depth",      " -- x : get current stack depth")\
0219  X(SPLOAD,    "sp@",        " -- addr : load current stack pointer ")\
0220  X(SPSTORE,   "sp!",        " addr -- : modify the stack pointer")\
0221  X(CLOCK,     "clock",      " -- x : push a time value")\
0222  X(EVALUATOR, "evaluator", "c-addr u 0 | file-id 0 1 -- x : evaluate file/str")\
0223  X(PSTK,      ".s",         " -- : print out values on the stack")\
0224  X(RESTART,   "restart",    " error -- : restart system, cause error")\
0225  X(CALL,      "call",       "x1...xn c -- x1...xn c : call a function")\
0226  X(SYSTEM,    "system",     "c-addr u -- bool : execute system command")\
0227  X(FCLOSE,    "close-file", "file-id -- ior : close a file")\
0228  X(FOPEN,     "open-file",  "c-addr u fam -- open a file")\
0229  X(FDELETE,   "delete-file",     "c-addr u -- : delete a file")\
0230  X(FREAD,     "read-file",       "c-addr u file-id -- u ior : write block")\
0231  X(FWRITE,    "write-file",      "c-addr u file-id -- u ior : read block")\
0232  X(FPOS,      "file-position",   "file-id -- u : get the file position")\
0233  X(FSEEK,     "reposition-file", "file-id u -- ior : reposition file")\
0234  X(FFLUSH,    "flush-file",      "file-id -- ior : flush a file")\
0235  X(FRENAME,   "rename-file",     "c-addr1 u1 c-addr2 u2 -- ior : rename file")\
0236  X(TMPFILE,   "temporary-file",  "-- file-id ior : open a temporary file")\
0237  X(LAST_INSTRUCTION, NULL, "")

0238 enum instructions { /**< instruction enumerations */
0239 #define X(ENUM, STRING, HELP) ENUM,
0240    XMACRO_INSTRUCTIONS
```

```
0241 #undef X
0242 };
```

So that we can compile programs we need ways of referring to the basic programming constructs provided by the virtual machine, theses words are fed into the C function **compile** in a process described later.

LAST_INSTRUCTION is not an instruction, but only a marker of the last enumeration used in **enum instructions**, so it does not get a name.

```
0243 static const char *instruction_names[] = { /**< instructions with names */
0244 #define X(ENUM, STRING, HELP) STRING,
0245    XMACRO_INSTRUCTIONS
0246 #undef X
0247 };
```

The help strings are made available in the following array:

```
0248 static const char *instruction_help_strings[] = {
0249 #define X(ENUM, STRING, HELP) HELP,
0250    XMACRO_INSTRUCTIONS
0251 #undef X
0252 };
```

Helping Functions For The Compiler

emsg returns a possible reason for a failure in a library function, in the form of a string

```
0253 static const char *emsg(void)
0254 {
0255    static const char *unknown = "unknown reason";
0256    const char *r = errno ? strerror(errno) : unknown;
0257    if(!r)
0258        r = unknown;
0259    return r;
0260 }
```

The logging function is used to print error messages, warnings and notes within this program.

```
0261 static int logger(const char *prefix, const char *func,
0262        unsigned line, const char *fmt, ...)
```

```
0263 {
0264    int r;
0265    va_list ap;
0266    assert(prefix && func && fmt);
0267    fprintf(stderr, "[%s %u] %s: ", func, line, prefix);
0268    va_start(ap, fmt);
0269    r = vfprintf(stderr, fmt, ap);
0270    va_end(ap);
0271    fputc('\n', stderr);
0272    return r;
0273 }
```

Get a char from string input or a file

This Forth interpreter only has a limited number of mechanisms for I/O, one of these is to fetch an individual character of input from either a string or a file which can be set either with knowledge of the implementation from within the virtual machine, or via the API presented to the programmer.

The C functions **forth_init**, **forth_set_file_input** and **forth_set_string_input** set up and manipulate the input of the interpreter. These functions act on the following registers:

```
SOURCE_ID - The current input source (SIN or FIN)
SIN       - String INput
SIDX      - String InDeX
SLEN      - String LENgth
FIN       - File   INput
```

Note that either SIN or FIN might not both be valid, one will be but the other might not, this makes manipulating these values hazardous. The input functions **forth_get_char** and **forth_ge_word** both take their input streams implicitly via the registers contained within the Forth execution environment passed in to those functions.

```
0274 static int forth_get_char(forth_t *o)
0275 {
0276    switch(o->m[SOURCE_ID]) {
0277    case FILE_IN:   return fgetc((FILE*)(o->m[FIN]));
0278    case STRING_IN: return o->m[SIDX] >= o->m[SLEN] ?
0279                EOF :
0280                ((char*)(o->m[SIN]))[o->m[SIDX]++];
0281    default:        return EOF;
0282    }
0283 }
```

get a word (space delimited, up to 31 chars) from a FILE* or string-in

This function reads in a space delimited word, limited to **MAXIMUM_WORD_LENGTH**, the word is put into the pointer ***p**, due to the simple nature of Forth this is as complex as parsing and lexing gets. It can either read from a file handle or a string, like forth_get_char()

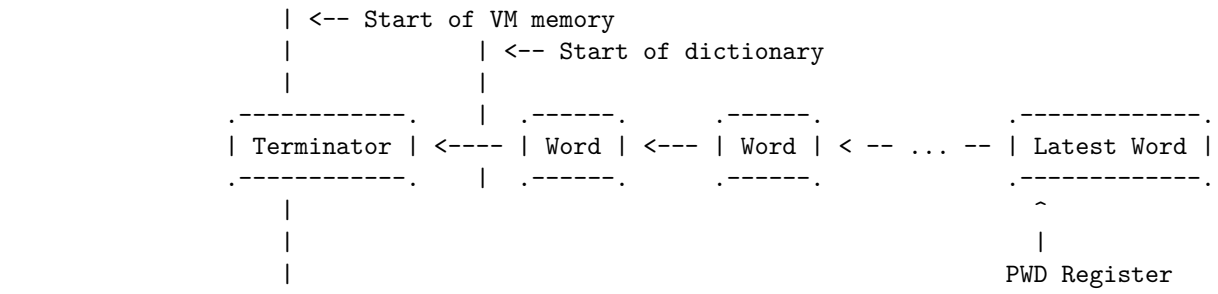
```
0284 static int forth_get_word(forth_t *o, uint8_t *p)
0285 {
0286    int n = 0;
0287    switch(o->m[SOURCE_ID]) {
0288    case FILE_IN:   return fscanf((FILE*)(o->m[FIN]), o->word_fmt, p, &n);
0289    case STRING_IN:
0290        if(sscanf((char *)&(((char*)(o->m[SIN]))[o->m[SIDX]]),
0291                    o->word_fmt, p, &n) <= 0)
0292            return EOF;
0293        o->m[SIDX] += n;
0294        return n;
0295    default:       return EOF;
0296    }
0297 }
```

Compile a Forth word header into the dictionary

The function **compile** is not that complicated in itself, however it requires an understanding of the structure of a Forth word definition and the behavior of the Forth run time.

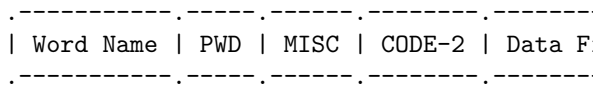
In all Forth implementations there exists a concept of *the dictionary*, although they may be implemented in different ways the usual way is as a linked list of words, starting with the latest defined word and ending with a special terminating value. Words cannot be arbitrarily deleted, deletions have to occur in the reverse order that they are defined.

Each word or Forth function that has been defined can be looked up in this dictionary, and dependent on whether it is an immediate word or a compiling word, and whether we are in command or compile mode different actions are taken when we have found the word we are looking for in our Read-Evaluate-Loop.



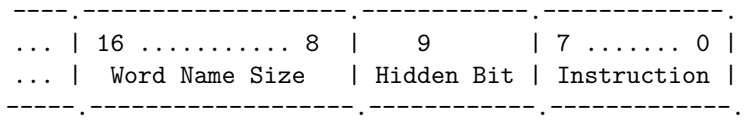
The **PWD** registers points to the latest defined word, a search starts from here and works it way backwards (allowing us replace old definitions by appending new ones with the same name only), the terminator

Our word header looks like this:



- **CODE-2** and the **Data Field** are optional and the **Data Field** is of variable length.
- **Word Name** is a variable length field whose size is recorded in the MISC field.

And the **MISC** field is a composite to save space containing a virtual machine instruction, the hidden bit and the length of the Word Name string as an offset in cells from **PWD** field. The field looks like this:



The maximum value for the Word Name field is determined by the width of the Word Name Size field.

The hidden bit is not used in the **compile** function, but is used elsewhere (in **forth_find**) to hide a word definition from the word search. The hidden bit is not set within this program at all, however it can be set by a running Forth virtual machine (and it is, if desired).

The **Instruction** tells the interpreter what to do with the Word definition when it is found and how to interpret **CODE-2** and the **Data Field** if they exist.

```
0298 static void compile(forth_t *o, forth_cell_t code, const char *str)
0299 {
0300    assert(o && code < LAST_INSTRUCTION);
0301    forth_cell_t *m = o->m, header = m[DIC], l = 0;
0302    /*FORTH header structure */
0303    /*Copy the new FORTH word into the new header */
0304    strcpy((char *)(o->m + header), str);
0305    /* align up to size of cell */
0306    l = strlen(str) + 1;
0307    l = (l + (sizeof(forth_cell_t) - 1)) & ~(sizeof(forth_cell_t) - 1);
```

```
0308    l = l/sizeof(forth_cell_t);
0309    m[DIC] += l; /* Add string length in words to header (STRLEN) */

0310    m[m[DIC]++] = m[PWD]; /*0 + STRLEN: Pointer to previous words header */
0311    m[PWD] = m[DIC] - 1;  /*Update the PWD register to new word */
0312    /*size of words name and code field*/
0313    m[m[DIC]++] = (l << WORD_LENGTH_OFFSET) | code;
0314 }
```

This function turns a string into a number using a base and returns an error code to indicate success or failure, the results of the conversion are stored in **n**, even if the conversion failed.

```
0315 static int numberify(int base, forth_cell_t *n, const char *s)
0316 {
0317    char *end = NULL;
0318    errno = 0;
0319    *n = strtol(s, &end, base);
0320    return errno || *s == '\0' || *end != '\0';
0321 }
```

Forths are usually case insensitive and are required to be (or at least accept only uppercase characters only) by the majority of the standards for Forth. As an aside I do not believe case insensitivity is a good idea as it complicates interfaces and creates as much confusion as it tries to solve (not only that, but different case letters do convey information). However, in keeping with other implementations, this Forth is also made insensitive to case **DUP** is treated the same as **dup** and **Dup**.

This comparison function, **istrcmp**, is only used in one place however, in the C function **forth_find**, replacing it with **strcmp** will bring back the more logical, case sensitive, behavior.

```
0322 static int istrcmp(const char *a, const char *b)
0323 {
0324    for(; ((*a == *b) || (tolower(*a) == tolower(*b))) && *a && *b; a++, b++)
0325        ;
0326    return tolower(*a) - tolower(*b);
0327 }
```

The **match** function returns true if the word is not hidden and if a case sensitive case sensitive has succeeded.

```
0328 static int match(forth_cell_t *m, forth_cell_t pwd, const char *s)
0329 {
```

```
0330    forth_cell_t len = WORD_LENGTH(m[pwd + 1]);
0331    return !WORD_HIDDEN(m[pwd+1]) && !istrcmp(s, (char*)(&m[pwd-len]));
0332 }
```

forth_find finds a word in the dictionary and if it exists it returns a pointer to its **PWD** field. If it is not found it will return zero, also of notes is the fact that it will skip words that are hidden, that is the hidden bit in the **MISC** field of a word is set. The structure of the dictionary has already been explained, so there should be no surprises in this word. Any improvements to the speed of this word would speed up the text interpreter a lot, but not the virtual machine in general.

```
0333 forth_cell_t forth_find(forth_t *o, const char *s)
0334 {
0335    forth_cell_t *m = o->m, pwd = m[PWD];
0336    for (;pwd > DICTIONARY_START && !match(m, pwd, s);)
0337        pwd = m[pwd];
0338    return pwd > DICTIONARY_START ? pwd + 1 : 0;
0339 }
```

Print a number in a given base to an output stream

```
0340 static int print_unsigned_number(forth_cell_t u, forth_cell_t base, FILE *out)
0341 {
0342    assert(base > 1 && base < 37);
0343    int i = 0, r = 0;
0344    char s[64 + 1] = "";
0345    do
0346        s[i++] = conv[u % base];
0347    while ((u /= base));
0348    for(; i >= 0 && r >= 0; i--)
0349        r = fputc(s[i], out);
0350    return r;
0351 }
```

Print out a forth cell as a number, the output base being determined by the **BASE** registers:

```
0352 static int print_cell(forth_t *o, FILE *output, forth_cell_t f)
0353 {
0354    unsigned base = o->m[BASE];
0355    if(base == 10 || base == 0)
0356        return fprintf(output, "%"PRIdCell, f);
0357    if(base == 16)
0358        return fprintf(output, o->hex_fmt, f);
```

```
0359    if(base == 1 || base > 36)
0360        return -1;
0361    return print_unsigned_number(f, base, output);
0362 }
```

check_bounds is used to both check that a memory access performed by the virtual machine is within range and as a crude method of debugging the interpreter (if it is enabled). The function is not called directly but is instead wrapped in with the **ck** macro, it can be removed with compile time defines, removing the check and the debugging code.

```
0363 static forth_cell_t check_bounds(forth_t *o, jmp_buf *on_error,
0364        forth_cell_t f, unsigned line, forth_cell_t bound)
0365 {
0366    if(o->m[DEBUG] >= DEBUG_CHECKS)
0367        debug("0x%"PRIxCell " %u", f, line);
0368    if(f >= bound) {
0369        fatal("bounds check failed (%"PRIdCell" >= %zu) line %u",
0370                f, (size_t)bound, line);
0371        longjmp(*on_error, FATAL);
0372    }
0373    return f;
0374 }
```

check_depth is used to check that there are enough values on the stack before an operation takes place. It is wrapped up in the **cd** macro.

```
0375 static void check_depth(forth_t *o, jmp_buf *on_error,
0376        forth_cell_t *S, forth_cell_t expected, unsigned line)
0377 {
0378    if(o->m[DEBUG] >= DEBUG_CHECKS)
0379        debug("0x%"PRIxCell " %u", (forth_cell_t)(S - o->vstart), line);
0380    if((uintptr_t)(S - o->vstart) < expected) {
0381        error("stack underflow %p -> %u", S, line);
0382        longjmp(*on_error, RECOVERABLE);
0383    } else if(S > o->vend) {
0384        error("stack overflow %p -> %u", S - o->vend, line);
0385        longjmp(*on_error, RECOVERABLE);
0386    }
0387 }
```

Check that the dictionary pointer does not go into the stack area:

```
0388 static forth_cell_t check_dictionary(forth_t *o, jmp_buf *on_error,
```

```
0389        forth_cell_t dptr)
0390 {
0391    if((o->m + dptr) >= (o->vstart)) {
0392        fatal("dictionary pointer is in stack area %"PRIdCell, dptr);
0393        o->m[INVALID] = 1;
0394        longjmp(*on_error, FATAL);
0395    }
0396    return dptr;
0397 }
```

This checks that a Forth string is *NUL* terminated, as required by most C functions, which should be the last character in string (which is s+end). There is a bit of a mismatch between Forth strings (which are pointer to the string and a length) and C strings, which a pointer to the string and are *NUL* terminated. This function helps to correct that.

```
0398 static void check_is_asciiz(jmp_buf *on_error, char *s, forth_cell_t end)
0399 {
0400    if(*(s + end) != '\0') {
0401        error("not an ASCIIZ string at %p", s);
0402        longjmp(*on_error, RECOVERABLE);
0403    }
0404 }
```

This function gets a string off the Forth stack, checking that the string is *NUL* terminated. It is a helper function used when a Forth string has to be converted to a C string so it can be passed to a C function.

```
0405 static char *forth_get_string(forth_t *o, jmp_buf *on_error,
0406        forth_cell_t **S, forth_cell_t f)
0407 {
0408    forth_cell_t length = f + 1;
0409    char *string = ((char*)o->m) + **S;
0410    (*S)--;
0411    check_is_asciiz(on_error, string, length);
0412    return string;
0413 }
```

Forth file access methods (or *fam*s) must be held in a single cell, this requires a method of translation from this cell into a string that can be used by the C function **fopen**

```
0414 static const char* forth_get_fam(jmp_buf *on_error, forth_cell_t f)
0415 {
```

```
0416    if(f >= LAST_FAM) {
0417        error("Invalid file access method %"PRIdCell, f);
0418        longjmp(*on_error, RECOVERABLE);
0419    }
0420    return fams[f];
0421 }
```

This prints out the Forth stack, which is useful for debugging.

```
0422 static void print_stack(forth_t *o, FILE *out, forth_cell_t *S, forth_cell_t f)
0423 {
0424    forth_cell_t depth = (forth_cell_t)(S - o->vstart);
0425    fprintf(out, "%"PRIdCell": ", depth);
0426    if(!depth)
0427        return;
0428    print_cell(o, out, f);
0429    fputc(' ', out);
0430    while(o->vstart + 1 < S) {
0431        print_cell(o, out, *(S--));
0432        fputc(' ', out);
0433    }
0434 }
```

This function allows for some more detailed tracing to take place, reading the logs is difficult, but it can provide *some* information about what is going on in the environment. This function will be compiled out if **NDEBUG** is defined by the C preprocessor.

```
0435 static void trace(forth_t *o, forth_cell_t instruction,
0436        forth_cell_t *S, forth_cell_t f)
0437 {
0438    if(o->m[DEBUG] < DEBUG_INSTRUCTION)
0439        return;
0440    if(instruction > LAST_INSTRUCTION) {
0441        error("traced invalid instruction %"PRIdCell, instruction);
0442        return;
0443    }
0444    fprintf(stderr, "\t( %s\t ", instruction_names[instruction]);
0445    print_stack(o, stderr, S, f);
0446    fputs(" )\n", stderr);
0447 }
```

API related functions and Initialization code

```
0448 void forth_set_file_input(forth_t *o, FILE *in)
```

```
0449 {
0450    assert(o && in);
0451    o->m[SOURCE_ID] = FILE_IN;
0452    o->m[FIN]       = (forth_cell_t)in;
0453 }

0454 void forth_set_file_output(forth_t *o, FILE *out)
0455 {
0456    assert(o && out);
0457    o->m[FOUT] = (forth_cell_t)out;
0458 }

0459 void forth_set_string_input(forth_t *o, const char *s)
0460 {
0461    assert(o && s);
0462    o->m[SIDX] = 0;              /* m[SIDX] == current character in string */
0463    o->m[SLEN] = strlen(s) + 1;  /* m[SLEN] == string len */
0464    o->m[SOURCE_ID] = STRING_IN; /* read from string, not a file handle */
0465    o->m[SIN] = (forth_cell_t)s; /* sin  == pointer to string input */
0466 }

0467 int forth_eval(forth_t *o, const char *s)
0468 {
0469    assert(o && s);
0470    forth_set_string_input(o, s);
0471    return forth_run(o);
0472 }

0473 int forth_define_constant(forth_t *o, const char *name, forth_cell_t c)
0474 {
0475    char e[MAXIMUM_WORD_LENGTH+32] = {0};
0476    assert(o && strlen(name) < MAXIMUM_WORD_LENGTH);
0477    sprintf(e, ": %31s %" PRIdCell " ; \n", name, c);
0478    return forth_eval(o, e);
0479 }
```

This function defaults all of the registers in a Forth environment and sets up the input and output streams.

forth_make_default default is called by **forth_init** and **forth_load_core_file**, it is a routine which deals that sets up registers for the virtual machines memory, and especially with values that may only be valid for a limited period (such as pointers to **stdin**).

```
0480 static void forth_make_default(forth_t *o, size_t size, FILE *in, FILE *out)
0481 {
```

```
0482    assert(o && size >= MINIMUM_CORE_SIZE && in && out);
0483    o->core_size     = size;
0484    o->m[STACK_SIZE] = size / MINIMUM_STACK_SIZE > MINIMUM_STACK_SIZE ?
0485                size / MINIMUM_STACK_SIZE :
0486                MINIMUM_STACK_SIZE;

0487    o->s             = (uint8_t*)(o->m + STRING_OFFSET); /*skip registers*/
0488    o->m[FOUT]       = (forth_cell_t)out;
0489    o->m[START_ADDR] = (forth_cell_t)&(o->m);
0490    o->m[STDIN]      = (forth_cell_t)stdin;
0491    o->m[STDOUT]     = (forth_cell_t)stdout;
0492    o->m[STDERR]     = (forth_cell_t)stderr;
0493    o->m[RSTK] = size - o->m[STACK_SIZE]; /* set up return stk ptr */
0494    o->m[ARGC] = o->m[ARGV] = 0;
0495    o->S       = o->m + size - (2 * o->m[STACK_SIZE]); /* v. stk pointer */
0496    o->vstart  = o->m + size - (2 * o->m[STACK_SIZE]);
0497    o->vend    = o->vstart + o->m[STACK_SIZE];
0498    VERIFY(sprintf(o->hex_fmt, "0x%%0%d"PRIxCell,
0499                (int)sizeof(forth_cell_t)*2) > 0);
0500    VERIFY(sprintf(o->word_fmt, "%%%ds%%n", MAXIMUM_WORD_LENGTH - 1) > 0);
0501    forth_set_file_input(o, in);  /* set up input after our eval */
0502 }
```

This function simply copies the current Forth header into a byte array, filling in the endianess which can only be determined at run time.

```
0503 static void make_header(uint8_t *dst)
0504 {
0505    memcpy(dst, header, sizeof header);
0506    /*fill in endianess, needs to be done at run time */
0507    dst[ENDIAN] = !IS_BIG_ENDIAN;
0508 }
```

forth_init is a complex function that returns a fully initialized forth environment we can start executing Forth in, it does the usual task of allocating memory for the object to be returned, but it also does has the task of getting the object into a runnable state so we can pass it to **forth_run** and do useful work.

```
0509 forth_t *forth_init(size_t size, FILE *in, FILE *out,
0510        const struct forth_functions *calls)
0511 {
0512    assert(in && out);
0513    forth_cell_t *m, i, w, t;
0514    forth_t *o;
0515    assert(sizeof(forth_cell_t) >= sizeof(uintptr_t));
```

There is a minimum requirement on the **m** field in the **forth_t** structure which is not apparent in its definition (and cannot be made apparent given how flexible array members work). We need enough memory to store the registers (32 cells), the parse area for a word (**MAXIMUM_WORD_LENGTH** cells), the initial start up program (about 6 cells), the initial built in and defined word set (about 600-700 cells) and the variable and return stacks (**MINIMUM_STACK_SIZE** cells each, as minimum).

If we add these together we come up with an absolute minimum, although that would not allow us define new words or do anything useful. We use **MINIMUM_STACK_SIZE** to define a useful minimum, albeit a restricted on, it is not a minimum large enough to store all the definitions in *forth.fth* (a file within the project containing a lot of Forth code) but it is large enough for embedded systems, for testing the interpreter and for the unit tests within the *unit.c* file.

We **VERIFY** that the size has been passed in is equal to or about minimum as this has been documented as being a requirement to this function in the C API, if we are passed a lower number the programmer has made a mistake somewhere and should be informed of this problem.

```
0516    VERIFY(size >= MINIMUM_CORE_SIZE);
0517    if(!(o = calloc(1, sizeof(*o) + sizeof(forth_cell_t)*size)))
0518        return NULL;
```

Default the registers, and input and output streams:

```
0519    forth_make_default(o, size, in, out);
```

o->header needs setting up, but has no effect on the run time behavior of the interpreter:

```
0520    make_header(o->header);

0521    o->calls = calls; /* pass over functions for CALL */
0522    m = o->m;         /* a local variable only for convenience */
```

The next section creates a word that calls **READ**, then **TAIL**, then itself. This is what the virtual machine will run at startup so that we can start reading in and executing Forth code. It creates a word that looks like this:

```
| <-- start of dictionary          |
.------.------.-----.----.----.----.
| TAIL | READ | RUN | P1 | P2 | P2 | Rest of dictionary ...
.------.------.-----.----.----.----.
```

```
|     end of this special word --> |

P1 is a pointer to READ
P2 is a pointer to TAIL
P2 is a pointer to RUN
```

The effect of this can be described as "make a function which performs a **READ** then calls itself tail recursively". The first instruction run is **RUN** which we save in **o->m[INSTRUCTION]** and restore when we enter **forth_run**.

```
0523    o->m[PWD]   = 0;  /* special terminating pwd value */
0524    t = m[DIC] = DICTIONARY_START; /* initial dictionary offset */
0525    m[m[DIC]++] = TAIL; /* add a TAIL instruction that can be called */
0526    w = m[DIC];         /* save current offset, which will contain READ */
0527    m[m[DIC]++] = READ; /* populate the cell with READ */
0528    m[m[DIC]++] = RUN;  /* call the special word recursively */
0529    o->m[INSTRUCTION] = m[DIC]; /* stream points to the special word */
0530    m[m[DIC]++] = w;    /* call to READ word */
0531    m[m[DIC]++] = t;    /* call to TAIL */
0532    m[m[DIC]++] = o->m[INSTRUCTION] - 1; /* recurse*/
```

DEFINE and **IMMEDIATE** are two immediate words, the only two immediate words that are also virtual machine instructions, we can make them immediate by passing in their code word to **compile**. The created word looks like this

```
.------.-----.------.
| NAME | PWD | MISC |
.------.-----.------.
```

The **MISC** field here contains either **DEFINE** or **IMMEDIATE**, as well as the hidden bit field and an offset to the beginning of name.

```
0533    compile(o, DEFINE,    ":");
0534    compile(o, IMMEDIATE, "immediate");
```

All of the other built in words that use a virtual machine instruction to do work are instead compiling words, and because there are lots of them we can initialize them in a loop

The created word looks like this:

```
.------.-----.------.----------------.
| NAME | PWD | MISC | VM-INSTRUCTION |
.------.-----.------.----------------.
```

The MISC field here contains the **COMPILE** instructions, which will compile a pointer to the **VM-INSTRUCTION**, as well as the other fields it usually contains.

```
0535    for(i = READ, w = READ; instruction_names[i]; i++) {
0536        compile(o, COMPILE, instruction_names[i]);
0537        m[m[DIC]++] = w++; /*This adds the actual VM instruction */
0538    }
```

The next eval is the absolute minimum needed for a sane environment, it defines two words **state** and **;**

```
0539    VERIFY(forth_eval(o, ": state 8 exit : ; immediate ' exit , 0 state ! ;") >= 0);
```

We now name all the registers so we can refer to them by name instead of by number, this is not strictly necessary but is good practice.

```
0540    for(i = 0; register_names[i]; i++)
0541        VERIFY(forth_define_constant(o, register_names[i], i+DIC) >= 0);
```

More constants are now defined:

```
0542    VERIFY(forth_define_constant(o, "size", sizeof(forth_cell_t)) >= 0);
0543    VERIFY(forth_define_constant(o, "stack-start", size - (2 * o->m[STACK_SIZE])) >= 0);
0544    VERIFY(forth_define_constant(o, "max-core", size) >= 0);
0545    VERIFY(forth_define_constant(o, "r/o",      FAM_RO) >= 0);
0546    VERIFY(forth_define_constant(o, "w/o",      FAM_WO) >= 0);
0547    VERIFY(forth_define_constant(o, "r/w",      FAM_RW) >= 0);
0548    VERIFY(forth_define_constant(o, "dictionary-start",  DICTIONARY_START) >= 0);
0549    VERIFY(forth_define_constant(o, "tib",  STRING_OFFSET * sizeof(forth_cell_t)) >= 0);
0550    VERIFY(forth_define_constant(o, "#tib", MAXIMUM_WORD_LENGTH * sizeof(forth_cell_t))
```

Now we finally are in a state to load the slightly inaccurately named **initial_forth_program**, which will give us basic looping and conditional constructs

```
0551    VERIFY(forth_eval(o, initial_forth_program) >= 0);
```

All of the calls to **forth_eval** and **forth_define_constant** have set the input streams to point to a string, we need to reset them to they point to the file **in**

```
0552    forth_set_file_input(o, in);  /*set up input after our eval */
0553    return o;
0554 }
```

This is a crude method that should only be used for debugging purposes, it simply dumps the forth structure to disk, including any padding which the compiler might have inserted. This dump cannot be reloaded

```
0555 int forth_dump_core(forth_t *o, FILE *dump)
0556 {
0557    assert(o && dump);
0558    size_t w = sizeof(*o) + sizeof(forth_cell_t) * o->core_size;
0559    return w != fwrite(o, 1, w, dump) ? -1: 0;
0560 }
```

We can save the virtual machines working memory in a way, called serialization, such that we can load the saved file back in and continue execution using this save environment. Only the three previously mentioned fields are serialized; **m**, **core_size** and the **header**.

```
0561 int forth_save_core_file(forth_t *o, FILE *dump)
0562 {
0563    assert(o && dump);
0564    uint64_t r1, r2, r3, core_size = o->core_size;
0565    if(o->m[INVALID])
0566        return -1;
0567    r1 = fwrite(o->header,  1, sizeof(o->header), dump);
0568    r2 = fwrite(&core_size, sizeof(core_size), 1, dump);
0569    r3 = fwrite(o->m,       1, sizeof(forth_cell_t) * core_size, dump);
0570    if(r1+r2+r3 != (sizeof(o->header) + 1 + sizeof(forth_cell_t)*core_size))
0571        return -1;
0572    return 0;
0573 }
```

Logically if we can save the core for future reuse, then we must have a function for loading the core back in, this function returns a reinitialized Forth object. Validation on the object is performed to make sure that it is a valid object and not some other random file, endianess, **core_size**, cell size and the headers magic constants field are all checked to make sure they are correct and compatible with this interpreter.

forth_make_default is called to replace any instances of pointers stored in registers which are now invalid after we have loaded the file from disk.

```
0574 forth_t *forth_load_core_file(FILE *dump)
0575 {
0576    uint8_t actual[sizeof(header)] = {0},   /* read in header */
0577        expected[sizeof(header)] = {0}; /* what we expected */
0578    forth_t *o = NULL;
```

```
0579    uint64_t w = 0, core_size = 0;
0580    assert(dump);
0581    make_header(expected);
0582    if(sizeof(actual) != fread(actual, 1, sizeof(actual), dump)) {
0583        goto fail; /* no header */
0584    }
0585    if(memcmp(expected, actual, sizeof(header))) {
0586        goto fail; /* invalid or incompatible header */
0587    }
0588    if(1 != fread(&core_size, sizeof(core_size), 1, dump)) {
0589        goto fail; /* no header */
0590    }
0591    if(core_size < MINIMUM_CORE_SIZE) {
0592        error("core size of %"PRIdCell" is too small", core_size);
0593        goto fail;
0594    }
0595    w = sizeof(*o) + (sizeof(forth_cell_t) * core_size);
0596    errno = 0;
0597    if(!(o = calloc(w, 1))) {
0598        error("allocation of size %"PRId64" failed, %s", w, emsg());
0599        goto fail;
0600    }
0601    w = sizeof(forth_cell_t) * core_size;
0602    if(w != fread(o->m, 1, w, dump)) {
0603        error("file too small (expected %"PRId64")", w);
0604        goto fail;
0605    }
0606    o->core_size = core_size;
0607    memcpy(o->header, actual, sizeof(o->header));
0608    forth_make_default(o, core_size, stdin, stdout);
0609    return o;
0610 fail:
0611    free(o);
0612    return NULL;
0613 }
```

The following function allows us to load a core file from memory:

```
0614 forth_t *forth_load_core_memory(forth_cell_t *m, forth_cell_t size)
0615 {
0616    assert(m && (size / sizeof(forth_cell_t)) >= MINIMUM_CORE_SIZE);
0617    forth_t *o;
0618    size /= sizeof(forth_cell_t);
0619    size_t w = sizeof(*o) + (sizeof(forth_cell_t) * size);
0620    errno = 0;
```

```
0621    o = calloc(w, 1);
0622    if(!o) {
0623        error("allocation of size %zu failed, %s", w, emsg());
0624        return NULL;
0625    }
0626    make_header(o->header);
0627    memcpy(o->m, m, size * sizeof(forth_cell_t));
0628    forth_make_default(o, size, stdin, stdout);
0629    return o;
0630 }
```

And likewise we will want to be able to save to memory as well, the load and save functions for memory expect headers *not* to be present.

```
0631 forth_cell_t *forth_save_core_memory(forth_t *o, forth_cell_t *size)
0632 {
0633    assert(o && size);
0634    forth_cell_t *m;
0635    *size = 0;
0636    errno = 0;
0637    m = malloc(o->core_size * sizeof(forth_cell_t));
0638    if(!m) {
0639        error("allocation of size %zu failed, %s",
0640                o->core_size * sizeof(forth_cell_t), emsg());
0641        return NULL;
0642    }
0643    memcpy(m, o->m, o->core_size);
0644    *size = o->core_size * sizeof(forth_cell_t);
0645    return m;
0646 }
```

Free the Forth interpreter, we make sure to invalidate the interpreter in case there is a use after free.

```
0647 void forth_free(forth_t *o)
0648 {
0649    assert(o);
0650    /* invalidate the forth core, a sufficiently "smart" compiler
0651     * might optimize this out */
0652    o->m[INVALID] = 1;
0653    free(o);
0654 }
```

Unfortunately C disallows the static initialization of structures with flexible array member, GCC allows this as an extension.

```
0655 struct forth_functions *forth_new_function_list(forth_cell_t count)
0656 {
0657    struct forth_functions *ff = NULL;
0658    errno = 0;
0659    ff = calloc(sizeof(*ff) + sizeof(ff->functions[0]) * count + 1, 1);
0660    if(!ff)
0661        warning("calloc failed: %s", emsg());
0662    else
0663        ff->count = count;
0664    return ff;
0665 }

0666 void forth_delete_function_list(struct forth_functions *calls)
0667 {
0668    free(calls);
0669 }
```

forth_push, **forth_pop** and **forth_stack_position** are the main ways an application programmer can interact with the Forth interpreter. Usually this tutorial talks about how the interpreter and virtual machine work, about how compilation and command modes work, and the internals of a Forth implementation. However this project does not just present an ordinary Forth interpreter, the interpreter can be embedded into other applications, and it is possible be running multiple instances Forth interpreters in the same process.

The project provides an API which other programmers can use to do this, one mechanism that needs to be provided is the ability to move data into and out of the interpreter, these C level functions are how this mechanism is achieved. They move data between a C program and a paused Forth interpreters variable stack.

```
0670 void forth_push(forth_t *o, forth_cell_t f)
0671 {
0672    assert(o && o->S < o->m + o->core_size);
0673    *++(o->S) = o->m[TOP];
0674    o->m[TOP] = f;
0675 }

0676 forth_cell_t forth_pop(forth_t *o)
0677 {
0678    assert(o && o->S > o->m);
0679    forth_cell_t f = o->m[TOP];
0680    o->m[TOP] = *(o->S)--;
0681    return f;
0682 }
```

```
0683 forth_cell_t forth_stack_position(forth_t *o)
0684 {
0685    assert(o);
0686    return o->S - o->vstart;
0687 }
```

The Forth Virtual Machine

The largest function in the file, which implements the forth virtual machine, everything else in this file is just fluff and support for this function. This is the Forth virtual machine, it implements a threaded code interpreter (see https://en.wikipedia.org/wiki/Threaded/_code, and https://www.complang.tuwien.ac.at/forth/threaded-code.html).

```
0688 int forth_run(forth_t *o)
0689 {
0690    int errorval = 0;
0691    assert(o);
0692    jmp_buf on_error;
0693    if(o->m[INVALID]) {
0694        fatal("refusing to run an invalid forth, %"PRIdCell, o->m[INVALID]);
0695        return -1;
0696    }

0697    /* The following code handles errors, if an error occurs, the
0698     * interpreter will jump back to here.
0699     *
0700     * @todo This code needs to be rethought to be made more compliant with
0701     * how "throw" and "catch" work in Forth. */
0702    if ((errorval = setjmp(on_error)) || o->m[INVALID]) {
0703        /* if the interpreter is invalid we always exit*/
0704        if(o->m[INVALID])
0705            return -1;
0706        switch(errorval) {
0707            default:
0708            case FATAL:
0709                return -(o->m[INVALID] = 1);
0710            /* recoverable errors depend on o->m[ERROR_HANDLER],
0711             * a register which can be set within the running
0712             * virtual machine. */
0713            case RECOVERABLE:
0714                switch(o->m[ERROR_HANDLER]) {
0715                case ERROR_INVALIDATE:
0716                    o->m[INVALID] = 1;
0717                case ERROR_HALT:
```

```
0718                    return -(o->m[INVALID]);
0719                case ERROR_RECOVER:
0720                    o->m[RSTK] = o->core_size - o->m[STACK_SIZE];
0721                    break;
0722                }
0723            case OK:
0724                break;
0725        }
0726    }

0727    forth_cell_t *m = o->m,  /* convenience variable: virtual memory */
0728             pc,         /* virtual machines program counter */
0729             *S = o->S,  /* convenience variable: stack pointer */
0730             I = o->m[INSTRUCTION], /* instruction pointer */
0731             f = o->m[TOP], /* top of stack */
0732             w,          /* working pointer */
0733             clk;        /* clock variable */

0734    clk = (1000 * clock()) / CLOCKS_PER_SEC;
```

The following section will explain how the threaded virtual machine interpreter works. Threaded code is a simple concept and Forths typically compile their code to threaded code, it suites Forth implementations as word definitions consist of juxtaposition of previously defined words until they reach a set of primitives.

This means a function like **square** will be implemented like this:

```
call dup   <- duplicate the top item on the variable stack
call *     <- push the result of multiplying the top two items
call exit  <- exit the definition of square
```

Each word definition is like this, a series of calls to other functions. We can optimize this by removing the explicit **call** and just having a series of code address to jump to, which will become:

```
address of "dup"
address of "*"
address of "exit"
```

We now have the problem that we cannot just jump to the beginning of the definition of **square** in our virtual machine, we instead use an instruction (**RUN** in our interpreter, or **DOLIST** as it is sometimes known in most other implementations) to determine what to do with the following data, if there is any. This system also allows us to encode primitives, or virtual machine instructions, in the same way as we encode words. If our word does not have the **RUN**

instruction as its first instruction then the list of addresses will not be interpreted but only a simple instruction will be executed.

The for loop and the switch statement here form the basis of our thread code interpreter along with the program counter register (**pc**) and the instruction pointer register (**I**).

To explain how execution proceeds it will help to refer to the internal structure of a word and how words are compiled into the dictionary.

Above we saw that a words layout looked like this:

```
.-----------.-----.------.--------.------------.
| Word Name | PWD | MISC | CODE-2 | Data Field |
.-----------.-----.------.--------.------------.
```

During execution we do not care about the **Word Name** field and **PWD** field. Also during execution we do not care about the top bits of the **MISC** field, only what instruction it contains.

Immediate words looks like this:

```
.-------------.---------------------.
| Instruction | Optional Data Field |
.-------------.---------------------.
```

And compiling words look like this:

```
.---------.-------------.---------------------.
| COMPILE | Instruction | Optional Data Field |
.---------.-------------.---------------------.
```

If the data field exists, the **Instruction** field will contain **RUN**. For words that only implement a single virtual machine instruction the **Instruction** field will contain only that single instruction (such as ADD, or SUB).

Let us define a series of words and see how the resulting word definitions are laid out, discounting the **Word Name**, **PWD** and the top bits of the **MISC** field.

We will define two words **square** (which takes a number off the stack, multiplies it by itself and pushes the result onto the stack) and **sum-of-products** (which takes two numbers off the stack, squares each one, adds the two results together and pushes the result onto the stack):

```
: square           dup * ;
: sum-of-products  square swap square + ;
```

Executing these:

```
9 square .            => prints '81 '
3 4 sum-of-products . => prints '25 '
```

1) **square** refers to two built in words **dup** and ***,**
2) **sum-of-products** to the word we just defined and two built in words
3) **swap** and **+**. We have also used the immediate word **:** and **;**.

Definition of **dup**, a compiling word:

```
.---------.------.
| COMPILE | DUP  |
.---------.------.
```

Definition of **+**, a compiling word:

```
.---------.------.
| COMPILE | +    |
.---------.------.
```

Definition of **swap**, a compiling word:

```
.---------.------.
| COMPILE | SWAP |
.---------.------.
```

Definition of **exit**, a compiling word:

```
.---------.------.
| COMPILE | EXIT |
.---------.------.
```

Definition of **:**, an immediate word:

```
.---.
| : |
.---.
```

Definition of **;**, a defined immediate word:

```
.-----.----.-------.----.-----------.--------.-------.
| RUN | $' | $exit | $, | literal 0 | $state | $exit |
.-----.----.-------.----.-----------.--------.-------.
```

Definition of **square**, a defined compiling word:

```
.---------.-----.------.----.-------.
| COMPILE | RUN | $dup | $* | $exit |
.---------.-----.------.----.-------.
```

Definition of **sum-of-products**, a defined compiling word:

```
.---------.-----.---------.-------.---------.----.-------.
| COMPILE | RUN | $square | $swap | $square | $+ | $exit |
.---------.-----.---------.-------.---------.----.-------.
```

All of these words are defined in the dictionary, which is a separate data structure from the variable stack. In the above definitions we use $square**or**$***** to mean a pointer to the words run time behavior, this is never the **COMPILE** field. **literal 0** means that at run time the number 0 is pushed to the variable stack, also the definition of **state** is not shown, as that would complicate things.

Imagine we have just typed in "sum-of-products" with "3 4" on the variable stack. Our **pc** register is now pointing the **RUN** field of sum of products, the virtual machine will next execute the **RUN** instruction, saving the instruction pointer to the return stack for when we finally exit **sum-of-products** back to the interpreter. **square** will now be called, it's **RUN** field encountered, then **dup**. **dup** does not have a **RUN** field, it is a built in primitive, so the instruction pointer will not be touched nor the return stack, but the **DUP** instruction will now be executed.

After this has run the instruction pointer will now be moved to executed *****, another primitive, then **exit** - which pops a value off the return stack and sets the instruction pointer to that value. The value points to the $swap**fieldin**$$**sum-of-products**$$, whichwillinturnbeexecuteduntilthefinal$$**exit** field is encountered. This exits back into our special read-and-loop word defined in the initialization code.

The **READ** routine must make sure the correct field is executed when a word is read in which depends on the state of the interpreter (held in **STATE** register).

It should be noted that for compatibility with future versions of the virtual machine that instructions can be added to the end (after the last defined instruction) but not removed.

```
0735    for(;(pc = m[ck(I++)]);) {
```

```
0736    INNER:
0737        w = instruction(m[ck(pc++)]);
0738        TRACE(o, w, S, f);
0739        switch (w) {
```

When explaining words with example Forth code the instructions enumeration will not be used (such as **ADD** or **SUB**), but its name will be used instead (such as + or -)

```
0740        case PUSH:    *++S = f;     f = m[ck(I++)];          break;
0741        case COMPILE: m[dic(m[DIC]++)] = pc;                 break;
0742        case RUN:     m[ck(++m[RSTK])] = I; I = pc;          break;
0743        case DEFINE:
```

DEFINE backs the Forth word **:**, which is an immediate word, it reads in a new word name, creates a header for that word and enters into compile mode, where all words (baring immediate words) are compiled into the dictionary instead of being executed.

The created header looks like this:

```
 .------.-----.------.-----.----
 | NAME | PWD | MISC | RUN |    ...
 .------.-----.------.-----.----
                  ^
                  |
               Dictionary Pointer
```

```
0744            m[STATE] = 1; /* compile mode */
0745            if(forth_get_word(o, o->s) < 0)
0746                goto end;
0747            compile(o, COMPILE, (char*)o->s);
0748            m[dic(m[DIC]++)] = RUN;
0749            break;
0750        case IMMEDIATE:
```

IMMEDIATE makes the current word definition execute regardless of whether we are in compile or command mode. Unlike most Forths this needs to go right after the word to be defined name instead of after the word definition itself. I prefer this behavior, however the reason for this is due to implementation reasons and not because of this preference.

So our interpreter defines immediate words:

```
: name immediate ... ;
```

versus, as is expected:

```
: name ... ; immediate
```

The way this word works is when **DEFINE** (or **:**) runs it creates a word header that looks like this:

```
.------.-----.------.-----.----
| NAME | PWD | MISC | RUN |    ...
.------.-----.------.-----.----
                ^
                |
              Dictionary Pointer
```

Where the **MISC** field contains **COMPILE**, we want it to look like this:

```
.------.-----.------.----
| NAME | PWD | MISC |    ...
.------.-----.------.----
              ^
              |
              Dictionary Pointer
```

With the **MISC** field containing **RUN**.

```
0751            m[DIC] -= 2; /* move to first code field */
0752            m[m[DIC]] &= ~INSTRUCTION_MASK; /* zero instruction */
0753            m[m[DIC]] |= RUN; /* set instruction to RUN */
0754            dic(m[DIC]++); /* compilation start here */
0755            break;
0756        case READ:
```

The **READ** instruction, an instruction that usually does not belong in a virtual machine, forms the basis of Forths interactive nature. In order to move this word outside of the virtual machine a compiler for the virtual machine would have to be made, which would complicate the implementation, but simplify the virtual machine and make it more like a 'normal' virtual machine.

It attempts to do the follow:

a) Lookup a space delimited string in the Forth dictionary, if it is found and we are in command mode we execute it, if we are in compile mode and the word is a compiling word we compile a pointer to it in the dictionary, if not we execute it.

b) If it is not a word in the dictionary we attempt to treat it as a number, if it is numeric (using the **BASE** register to determine the base) then if we are in command mode we push the number to the variable stack, else if we are in compile mode we compile the literal into the dictionary.
c) If it is neither a word nor a number, regardless of mode, we emit a diagnostic.

This is the most complex word in the Forth virtual machine, there is a good case for it being moved outside of it, and perhaps this will happen. You will notice that the above description did not include any looping, as such there is a driver for the interpreter which must be made and initialized in **forth_init**, a simple word that calls **READ** in a loop (actually tail recursively).

```
0757            if(forth_get_word(o, o->s) < 0)
0758                goto end;
0759            if ((w = forth_find(o, (char*)o->s)) > 1) {
0760                pc = w;
0761                if (!m[STATE] && instruction(m[ck(pc)]) == COMPILE)
0762                    pc++; /* in command mode, execute word */
0763                goto INNER;
0764            } else if(numberify(o->m[BASE], &w, (char*)o->s)) {
0765                error("'%s' is not a word", o->s);
0766                longjmp(on_error, RECOVERABLE);
0767                break;
0768            }
0769            if (m[STATE]) { /* must be a number then */
0770                m[dic(m[DIC]++)] = 2; /*fake word push at m[2] */
0771                m[dic(m[DIC]++)] = w;
0772            } else { /* push word */
0773                *++S = f;
0774                f = w;
0775            }
0776            break;
```

Most of the following Forth instructions are simple Forth words, each one with an uncomplicated Forth word which is implemented by the corresponding instruction (such as LOAD and "@", STORE and "!", EXIT and "exit", and ADD and "+").

However, the reason for these words existing, and under what circumstances some of the can be used is a different matter, the COMMA and TAIL word will require some explaining, but ADD, SUB and DIV will not.

```
0777        case LOAD:    cd(1); f = m[ck(f)];                    break;
0778        case STORE:   cd(2); m[ck(f)] = *S--; f = *S--;       break;
0779        case CLOAD:   cd(1); f = *(((uint8_t*)m) + ckchar(f)); break;
```

```
0780        case CSTORE:  cd(2); ((uint8_t*)m)[ckchar(f)] = *S--; f = *S--; break;
0781        case SUB:     cd(2); f = *S-- - f;                    break;
0782        case ADD:     cd(2); f = *S-- + f;                    break;
0783        case AND:     cd(2); f = *S-- & f;                    break;
0784        case OR:      cd(2); f = *S-- | f;                    break;
0785        case XOR:     cd(2); f = *S-- ^ f;                    break;
0786        case INV:     cd(1); f = ~f;                          break;
0787        case SHL:     cd(2); f = *S-- << f;                   break;
0788        case SHR:     cd(2); f = *S-- >> f;                   break;
0789        case MUL:     cd(2); f = *S-- * f;                    break;
0790        case DIV:
0791            cd(2);
0792            if(f) {
0793                f = *S-- / f;
0794            } else {
0795                error("divide %"PRIdCell" by zero ", *S--);
0796                longjmp(on_error, RECOVERABLE);
0797            }
0798            break;
0799        case ULESS:   cd(2); f = *S-- < f;                        break;
0800        case UMORE:   cd(2); f = *S-- > f;                        break;
0801        case EXIT:    I = m[ck(m[RSTK]--)];                       break;
0802        case KEY:     *++S = f; f = forth_get_char(o);            break;
0803        case EMIT:    f = fputc(f, (FILE*)o->m[FOUT]);            break;
0804        case FROMR:   *++S = f; f = m[ck(m[RSTK]--)];             break;
0805        case TOR:     cd(1); m[ck(++m[RSTK])] = f; f = *S--;      break;
0806        case BRANCH:  I += m[ck(I)];                              break;
0807        case QBRANCH: cd(1); I += f == 0 ? m[I] : 1; f = *S--;    break;
0808        case PNUM:    cd(1);
0809                  f = print_cell(o, (FILE*)(o->m[FOUT]), f); break;
0810        case QUOTE:   *++S = f;     f = m[ck(I++)];               break;
0811        case COMMA:   cd(1); m[dic(m[DIC]++)] = f; f = *S--;      break;
0812        case EQUAL:   cd(2); f = *S-- == f;                       break;
0813        case SWAP:    cd(2); w = f;  f = *S--;   *++S = w;        break;
0814        case DUP:     cd(1); *++S = f;                            break;
0815        case DROP:    cd(1); f = *S--;                            break;
0816        case OVER:    cd(2); w = *S; *++S = f; f = w;             break;
```

TAIL is a crude method of doing tail recursion, it should not be used generally but is useful at startup, there are limitations when using it in word definitions.

The following tail recursive definition of the greatest common divisor, called **(gcd)** will not work correctly when interacting with other words:

```
: (gcd) ?dup if dup rot rot mod tail (gcd) then ;
```

If we define a word:

```
: uses-gcd 50 20 (gcd) . ;
```

We might expect it to print out "10", however it will not, it will calculate the GCD, but not print it out with ".", as GCD will have popped off where it should have returned.

Instead we must wrap the definition up in another definition:

```
: gcd (gcd) ;
```

And the definition **gcd** can be used. There is a definition of **tail** within *forth.fth* that does not have this limitation, in fact the built in definition is hidden in favor of the new one.

```
0817        case TAIL:
0818            m[RSTK]--;
0819            break;
```

FIND is a natural factor of READ, we add it to the Forth interpreter as it already exits, it looks up a Forth word in the dictionary and returns a pointer to that word if it found.

```
0820        case FIND:
0821            *++S = f;
0822            if(forth_get_word(o, o->s) < 0)
0823                goto end;
0824            f = forth_find(o, (char*)o->s);
0825            f = f < DICTIONARY_START ? 0 : f;
0826            break;
```

DEPTH is added because the stack is not directly accessible by the virtual machine, normally it would have no way of knowing where the variable stack pointer is, which is needed to implement Forth words such as **.s** - which prints out all the items on the stack.

```
0827        case DEPTH:
0828            w = S - o->vstart;
0829            *++S = f;
0830            f = w;
0831            break;
```

SPLOAD (**sp@**) loads the current stack pointer, which is needed because the stack pointer does not live within any of the virtual machines registers.

```
0832        case SPLOAD:
0833            *++S = f;
0834            f = (forth_cell_t)(S - o->m);
0835            break;
```

SPSTORE (**sp!**) modifies the stack, setting it to the value on the top of the stack.

```
0836        case SPSTORE:
0837            w = *S--;
0838            S = (forth_cell_t*)(f + o->m - 1);
0839            f = w;
0840            break;
```

CLOCK allows for a primitive and wasteful (depending on how the C library implements "clock") timing mechanism, it has the advantage of being portable:

```
0841        case CLOCK:
0842            *++S = f;
0843            f = ((1000 * clock()) - clk) / CLOCKS_PER_SEC;
0844            break;
```

EVALUATOR is another complex word which needs to be implemented in the virtual machine. It saves and restores state which we do not usually need to do when the interpreter is not running (the usual case for **forth_eval** when called from C). It can read either from a string or from a file.

```
0845        case EVALUATOR:
0846        {
0847            /* save current input */
0848            forth_cell_t sin    = o->m[SIN],  sidx = o->m[SIDX],
0849                slen   = o->m[SLEN], fin  = o->m[FIN],
0850                source = o->m[SOURCE_ID], r = m[RSTK];
0851            char *s = NULL;
0852            FILE *file = NULL;
0853            int file_in = 0;
0854            cd(3);
0855            file_in = f; /*get file/string in bool*/
0856            f = *S--;
0857            if(file_in) {
0858                file = (FILE*)(*S--);
0859                f = *S--;
0860            } else {
0861                s = forth_get_string(o, &on_error, &S, f);
```

```
0862                f = *S--;
0863            }
0864            /* save the stack variables */
0865            o->S = S;
0866            o->m[TOP] = f;
0867            /* push a fake call to forth_eval */
0868            m[RSTK]++;
0869            if(file_in) {
0870                forth_set_file_input(o, file);
0871                w = forth_run(o);
0872            } else {
0873                w = forth_eval(o, s);
0874            }
0875            /* restore stack variables */
0876            m[RSTK] = r;
0877            S = o->S;
0878            *++S = o->m[TOP];
0879            f = w;
0880            /* restore input stream */
0881            o->m[SIN]  = sin;
0882            o->m[SIDX] = sidx;
0883            o->m[SLEN] = slen;
0884            o->m[FIN]  = fin;
0885            o->m[SOURCE_ID] = source;
0886            if(o->m[INVALID])
0887                return -1;
0888        }
0889        break;
0890        case PSTK:    print_stack(o, (FILE*)(o->m[STDOUT]), S, f);   break;
0891        case RESTART: cd(1); longjmp(on_error, f);                   break;
```

CALL allows arbitrary C functions to be passed in and used within the interpreter, allowing it to be extended. The functions have to be passed in during initialization and then they become available to be used by CALL.

The structure **forth_functions** is a list of function pointers that can be populated by the user of the libforth library, CALL indexes into that structure (after performing bounds checking) and executes the function.

```
0892        case CALL:
0893        {
0894            cd(1);
0895            if(!(o->calls) || !(o->calls->count)) {
0896                /* no call structure, or count is zero */
0897                f = -1;
0898                break;
```

```
0899            }
0900            forth_cell_t i = f;
0901            if(i >= (o->calls->count)) {
0902                f = -1;
0903                break;
0904            }

0905            assert(o->calls->functions[i].function);
0906            /* check depth of function */
0907            cd(o->calls->functions[i].depth);
0908            /* pop call number */
0909            f = *S--;
0910            /* save stack state */
0911            o->S = S;
0912            o->m[TOP] = f;
0913            /* call arbitrary C function */
0914            w = o->calls->functions[i].function(o);
0915            /* restore stack state */
0916            S = o->S;
0917            f = o->m[TOP];
0918            /* push call success value */
0919            *++S = f;
0920            f = w;
0921            break;
0922        }
```

Whilst loathe to put these in here as virtual machine instructions (instead a better mechanism should be found), this is the simplest way of adding file access words to our Forth interpreter.

The file access methods *should* all be wrapped up so it does not matter if a file or a piece of memory (a string for example) is being read or written to. This would allow the KEY to be removed as a virtual machine instruction, and would be a useful abstraction.

```
0923        case SYSTEM:  cd(2); f = system(forth_get_string(o, &on_error, &S, f)); break;
0924        case FCLOSE:  cd(1);
0925                  errno = 0;
0926                  f = fclose((FILE*)f) ? errno : 0;
0927                  break;
0928        case FDELETE: cd(2);
0929                  errno = 0;
0930                  f = remove(forth_get_string(o, &on_error, &S, f)) ? errno : 0;
0931                  break;
0932        case FFLUSH:  cd(1);
0933                  errno = 0;
```

```
0934                f = fflush((FILE*)f) ? errno : 0;
0935                break;
0936        case FSEEK:
0937            {
0938                cd(2);
0939                errno = 0;
0940                int r = fseek((FILE*)f, *S--, SEEK_SET);
0941                *++S = r;
0942                f = r == -1 ? errno : 0;
0943                break;
0944            }
0945        case FPOS:
0946            {
0947                cd(1);
0948                errno = 0;
0949                int r = ftell((FILE*)f);
0950                *++S = r;
0951                f = r == -1 ? errno : 0;
0952                break;
0953            }
0954        case FOPEN:
0955            cd(3);
0956            {
0957                const char *fam = forth_get_fam(&on_error, f);
0958                f = *S--;
0959                char *file = forth_get_string(o, &on_error, &S, f);
0960                errno = 0;
0961                *++S = (forth_cell_t)fopen(file, fam);
0962                f = errno;
0963            }
0964            break;
0965        case FREAD:
0966            cd(3);
0967            {
0968                FILE *file = (FILE*)f;
0969                forth_cell_t count = *S--;
0970                forth_cell_t offset = *S--;
0971                *++S = fread(((char*)m)+offset, 1, count, file);
0972                f = ferror(file);
0973                clearerr(file);
0974            }
0975            break;
0976        case FWRITE:
0977            cd(3);
0978            {
0979                FILE *file = (FILE*)f;
```

```
0980                forth_cell_t count = *S--;
0981                forth_cell_t offset = *S--;
0982                *++S = fwrite(((char*)m)+offset, 1, count, file);
0983                f = ferror(file);
0984                clearerr(file);
0985            }
0986            break;
0987        case FRENAME:
0988            cd(3);
0989            {
0990                const char *f1 = forth_get_fam(&on_error, f);
0991                f = *S--;
0992                char *f2 = forth_get_string(o, &on_error, &S, f);
0993                errno = 0;
0994                f = rename(f2, f1) ? errno : 0;
0995            }
0996            break;
0997        case TMPFILE:
0998            {
0999                *++S = f;
1000               errno = 0;
1001               *++S = (forth_cell_t)tmpfile();
1002               f = errno ? errno : 0;
1003           }
1004           break;
```

This should never happen, and if it does it is an indication that virtual machine memory has been corrupted somehow.

```
1005        default:
1006            fatal("illegal operation %" PRIdCell, w);
1007            longjmp(on_error, FATAL);
1008        }
1009    }
```

We must save the stack pointer and the top of stack when we exit the interpreter so the C functions like "forth_pop" work correctly. If the **forth_t** object has been invalidated (because something went wrong), we do not have to jump to *end* as functions like **forth_pop** should not be called on the invalidated object any longer.

```
1010 end:   o->S = S;
1011    o->m[TOP] = f;
1012    return 0;
1013 }
```

An example main function called main_forth and support functions

This section is not needed to understand how Forth works, or how the C API into the Forth interpreter works. It provides a function which uses all the functions available to the API programmer in order to create an example program that implements a Forth interpreter with a Command Line Interface.

This program can be used as a filter in a Unix pipe chain, or as a standalone interpreter for Forth. It tries to follow the Unix philosophy and way of doing things (see http://www.catb.org/esr/writings/taoup/html/ch01s06.html and https://en.wikipedia.org/wiki/Unix/_philosophy). Whether this is achieved is a matter of opinion. There are a things this interpreter does differently to most Forth interpreters that support this philosophy however, it is silent by default and does not clutter up the output window with "ok", or by printing a banner at start up (which would contain no useful information whatsoever). It is simple, and only does one thing (but does it do it well?).

```
1014 static void fclose_input(FILE **in)
1015 {
1016    if(*in && (*in != stdin))
1017        fclose(*in);
1018    *in = stdin;
1019 }

1020 void forth_set_args(forth_t *o, int argc, char **argv)
1021 { /* currently this is of little use to the interpreter */
1022    assert(o);
1023    o->m[ARGC] = argc;
1024    o->m[ARGV] = (forth_cell_t)argv;
1025 }
```

main_forth implements a Forth interpreter which is a wrapper around the C API, there is an assumption that main_forth will be the only thing running in a process (it does not seem sensible to run multiple instances of it at the same time - it is just for demonstration purposes), as such the only error handling should do is to die after printing an error message if an error occurs, the **fopen_or_die** is an example of this philosophy, one which does not apply to functions like **forth_run** (which makes attempts to recover from a sensible error).

```
1026 static FILE *fopen_or_die(const char *name, char *mode)
1027 {
1028    errno = 0;
1029    FILE *file = fopen(name, mode);
1030    if(!file) {
```

```
1031        fatal("opening file \"%s\" => %s", name, emsg());
1032        exit(EXIT_FAILURE);
1033    }
1034    return file;
1035 }
```

It is customary for Unix programs to have a usage string, which we can print out as a quick reminder to the user as to what the command line options are.

```
1036 static void usage(const char *name)
1037 {
1038    fprintf(stderr,
1039        "usage: %s "
1040        "[-(s|l) file] [-e expr] [-m size] [-Vthv] [-] files\n",
1041        name);
1042 }
```

We try to keep the interface to the example program as simple as possible, so there are limited, uncomplicated options. What they do should come as no surprise to an experienced Unix programmer, it is important to pick option names that they would expect (for example *-l* for loading, *-e* for evaluation, and not using *-h* for help would be a hanging offense).

```
1043 static void help(void)
1044 {
1045    static const char help_text[] =
1046 "Forth: A small forth interpreter build around libforth\n\n"
1047 "\t-h        print out this help and exit unsuccessfully\n"
1048 "\t-e string evaluate a string\n"
1049 "\t-s file   save state of forth interpreter to file\n"
1050 "\t-d        save state to 'forth.core'\n"
1051 "\t-l file   load previously saved state from file\n"
1052 "\t-m size   specify forth memory size in KiB (cannot be used with '-l')\n"
1053 "\t-t        process stdin after processing forth files\n"
1054 "\t-v        turn verbose mode on\n"
1055 "\t-V        print out version information and exit\n"
1056 "\t-         stop processing options\n\n"
1057 "Options must come before files to execute.\n\n"
1058 "The following words are built into the interpreter:\n\n";
1059 ;
1060    fputs(help_text, stderr);

1061    for(unsigned i = 0; i < LAST_INSTRUCTION; i++)
1062        fprintf(stderr, "%s\t\t%s\n",
1063                instruction_names[i],
```

```
1064                instruction_help_strings[i]);
1065 }

1066 static void version(void)
1067 {
1068    fprintf(stdout,
1069        "libforth:\n"
1070        "\tversion:     %d\n"
1071        "\tsize:        %u\n"
1072        "\tendianess:   %u\n"
1073        "initial forth program:\n%s\n",
1074        FORTH_CORE_VERSION,
1075        (unsigned)sizeof(forth_cell_t) * CHAR_BIT,
1076        (unsigned)IS_BIG_ENDIAN,
1077        initial_forth_program);
1078 }
```

main_forth is the second largest function is this file, but is not as complex as **forth_run** (currently the largest and most complex function), it brings together all the API functions offered by this library and provides a quick way for programmers to implement a working Forth interpreter for testing purposes.

This make implementing a Forth interpreter as simple as:

```
==== main.c ============================

#include "libforth.h"

int main(int argc, char **argv)
{
   return main_forth(argc, argv);
}

==== main.c ============================
```

To keep things simple options are parsed first then arguments like files, although some options take arguments immediately after them.

A library for parsing command line options like *getopt* should be used, this would reduce the portability of the program. It is not recommended that arguments are parsed in this manner.

```
1079 int main_forth(int argc, char **argv)
1080 {
1081    FILE *in = NULL, *dump = NULL;
1082    int rval = 0, c = 0, i = 1;
```

```
1083            int save = 0,            /* attempt to save core if true */
1084        eval = 0,            /* have we evaluated anything? */
1085        verbose = 0,         /* verbosity level */
1086        readterm = 0,        /* read from standard in */
1087        mset = 0;            /* memory size specified */
1088    static const size_t kbpc = 1024 / sizeof(forth_cell_t); /*kilobytes per cell*/
1089    static const char *dump_name = "forth.core";
1090    char *optarg = NULL;
1091    forth_cell_t core_size = DEFAULT_CORE_SIZE;
1092    forth_t *o = NULL;
```

This loop processes any options that may have been passed to the program, it looks for arguments beginning with '-' and attempts to process that option, if the argument does not start with '-' the option processing stops. It is a simple mechanism for processing program arguments and there are better ways of doing it (such as "getopt" and "getopts"), but by using them we sacrifice portability.

```
1093    for(i = 1; i < argc && argv[i][0] == '-'; i++)
1094        switch(argv[i][1]) {
1095        case '\0': goto done; /* stop processing options */
1096        case 'h':  usage(argv[0]);
1097               help();
1098               return -1;
1099        case 't':  readterm = 1;
1100               break;
1101        case 'e':
1102            if(i >= (argc - 1))
1103                goto fail;
1104            errno = 0;
1105            if(!(o = o ? o : forth_init(core_size, stdin, stdout, NULL))) {
1106                fatal("initialization failed, %s", emsg());
1107                return -1;
1108            }
1109            o->m[DEBUG] = verbose;
1110            optarg = argv[++i];
1111            if(verbose >= DEBUG_NOTE)
1112                note("evaluating '%s'", optarg);
1113            if(forth_eval(o, optarg) < 0)
1114                goto end;
1115            eval = 1;
1116            break;
1117        case 's':
1118            if(i >= (argc - 1))
1119                goto fail;
1120            dump_name = argv[++i];
```

```
1121        case 'd':  /*use default name */
1122            if(verbose >= DEBUG_NOTE)
1123                note("saving core file to '%s' (on exit)", dump_name);
1124            save = 1;
1125            break;
1126        case 'm':
1127            if(o || (i >= argc - 1) || numberify(10, &core_size, argv[++i]))
1128                goto fail;
1129            if((core_size *= kbpc) < MINIMUM_CORE_SIZE) {
1130                fatal("-m too small (minimum %zu)", MINIMUM_CORE_SIZE / kbpc);
1131                return -1;
1132            }
1133            if(verbose >= DEBUG_NOTE)
1134                note("memory size set to %zu", core_size);
1135            mset = 1;
1136            break;
1137        case 'l':
1138            if(o || mset || (i >= argc - 1))
1139                goto fail;
1140            optarg = argv[++i];
1141            if(verbose >= DEBUG_NOTE)
1142                note("loading core file '%s'", optarg);
1143            if(!(o = forth_load_core_file(dump = fopen_or_die(optarg, "rb")))) {
1144                fatal("%s, core load failed", optarg);
1145                return -1;
1146            }
1147            o->m[DEBUG] = verbose;
1148            fclose(dump);
1149            break;
1150        case 'v':
1151            verbose++;
1152            break;
1153        case 'V':
1154            version();
1155            return EXIT_SUCCESS;
1156            break;
1157        default:
1158        fail:
1159            fatal("invalid argument '%s'", argv[i]);
1160            usage(argv[0]);
1161            return -1;
1162        }
1163 done:
1164    /* if no files are given, read stdin */
1165    readterm = (!eval && i == argc) || readterm;
1166    if(!o) {
```

```
1167        errno = 0;
1168        if(!(o = forth_init(core_size, stdin, stdout, NULL))) {
1169            fatal("forth initialization failed, %s", emsg());
1170            return -1;
1171        }
1172        o->m[DEBUG] = verbose;
1173    }
1174    forth_set_args(o, argc, argv);
1175    for(; i < argc; i++) { /* process all files on command line */
1176        if(verbose >= DEBUG_NOTE)
1177            note("reading from file '%s'", argv[i]);
1178        forth_set_file_input(o, in = fopen_or_die(argv[i], "rb"));
1179        /* shebang line '#!', core files could also be detected */
1180        if((c = fgetc(in)) == '#')
1181            while(((c = forth_get_char(o)) > 0) && (c != '\n'));
1182        else if(c == EOF)
1183            goto close;
1184        else
1185            ungetc(c, in);
1186        if((rval = forth_run(o)) < 0)
1187            goto end;
1188 close:
1189        fclose_input(&in);
1190    }
1191    if(readterm) { /* if '-t' or no files given, read from stdin */
1192        if(verbose >= DEBUG_NOTE)
1193            note("reading from stdin (%p)", stdin);
1194        forth_set_file_input(o, stdin);
1195        rval = forth_run(o);
1196    }
1197 end:
1198    fclose_input(&in);
```

If the save option has been given we only want to save valid core files, we might want to make an option to force saving of core files for debugging purposes, but in general we do not want to over write valid previously saved state with invalid data.

```
1199    if(save) { /* save core file */
1200        if(rval || o->m[INVALID]) {
1201            fatal("refusing to save invalid core, %u/%"PRIdCell, rval, o->m[INVALID]);
1202            return -1;
1203        }
1204        if(verbose >= DEBUG_NOTE)
1205            note("saving for file to '%s'", dump_name);
```

```
1206        if(forth_save_core_file(o, dump = fopen_or_die(dump_name, "wb"))) {
1207            fatal("core file save to '%s' failed", dump_name);
1208            rval = -1;
1209        }
1210        fclose(dump);
1211    }
```

Whilst the following **forth_free** is not strictly necessary, there is often a debate that comes up making short lived programs or programs whose memory use stays either constant or only goes up, when these programs exit it is not necessary to clean up the environment and in some case (although not this one) it can slow down the exit of the program for no reason. However not freeing the memory after use does not play nice with programs that detect memory leaks, like Valgrind. Either way, we free the memory used here, but only if no other errors have occurred before hand.

```
1212    forth_free(o);
1213    return rval;
1214 }
```

And that completes the program, and the documentation describing it.