Contents

```
0001| 0 <ok> ! hex ( Turn off *ok* prompt, go into hex mode )
```

meta.fth

```
| Project    | A Small Forth VM/Implementation   |
| ---------- | --------------------------------- |
| Author     | Richard James Howe                |
| Copyright  | 2017 Richard James Howe           |
| License    | MIT                               |
| Email      | howe.r.j.89@gmail.com             |
| Repository | <https://github.com/howerj/embed> |
```

A Meta-compiler, an implementation of eForth, and a tutorial on both.

Introduction

In this file a meta-compiler (or a cross compiler written in Forth) is described and implemented, and after that a working Forth interpreter is both described and implemented. This new interpreter can be used in turn to meta-compile the original program, ad infinitum. The design decisions and the philosophy behind Forth and the system will also be elucidated.

This files source is a Forth program which the meta-compiler can read, it is also used as source for a simple document generation system using AWK which feeds into either Pandoc for PDF or the original Markdown script for HTML output. The AWK script is crude and requires that the Forth source file, meta.fth be formatted in a specific way.

Lines beginning with a back-slash are turned into normal Markdown text, with some characters needed to be escaped. Other lines are assumed to be Forth source code and are turned into Markdown code/literal-text blocks with a number and '|' symbol preceding them, numbering starts from line '0001'.

What is Forth?

Forth is a stack based procedural language, which uses Reverse Polish Notation (RPN) to enter expressions. It is a minimal language with no type checking and little to no error handling depending on the implementation. Despite its small size and simplicity it has various features usually found in higher level languages, such as reflection, incremental compilation and an interactive read-evaluate-print loop.

It is still at heart a language that is close to the machine with low level capabilities and direct access to memory. Memory manage itself is mostly manual, with preallocation of all needed memory the preferred method of program writing.

Forth has mostly fallen out of favor in recent years, it performed admirably on the microcomputer systems available in the 1980s and is still suitable for very memory constrained embed systems (having on a few kilobytes of memory available to them), but lacks a lot of features modern languages provide.

A catalogue of deficiencies hamper Forth adoption; poor string handling, lack of libraries and poor integration with the operating system (on hosted platforms), mutually incompatible and wildly different Forth implementations and as mentioned - little error detection and handling.

Despite this fact it has a core of adherents who find uses for the language, in fact some of its deficiencies are actually trade-offs. Having no type checking means there is no type checking to do, having very little in the way of error detection means errors do not have to be detected. This off loads the complexity of the problem to the programmer and means a Forth implementation can be minimal and terse.

The saying "Once you have seen one Forth implementation, you have seen one Forth implementation" comes about because of how easy it is to implement a Forth, which is a double edged sword. It is possible to completely understand a Forth system, the software, the hardware and the problems you are trying to solve and optimize everything towards this goal. This is oft not possible with modern systems, a single person cannot totally understand even subcomponents of modern systems in its entirety (such as compilers or the operating system kernels we use).

Another saying from the creator of Forth, Charles Moore, "Forth is Sudoku for programmers". The reason the author uses Forth is because it is fun, no more justification is needed.

Project Origins

This project derives from a simulator for a CPU written in VHDL, designed to execute Forth primitives directly. The CPU and Forth interpreter themselves have their own sources, which all makes for a confusing pedigree. The CPU, called the H2, was derived from a well known Forth CPU written in Verilog, called the J1 CPU, and the Forth running on the H2 comes from an adaption of eForth written for the J1 and from 'The Zen of eForth' by C. H. Ting.

Instead of a metacompiler written in Forth a cross compiler for a Forth like language was made, which could create an image readable by both the simulator, and the FPGA development tools. The simulator was cut down and modified for use on a computer, with new instructions for input and output.

This system, with a cross compiler and virtual machine written in C, was used to develop the present system which consists of only the virtual machine, a binary image containing a Forth interpreter, and this metacompiler with the meta-compiled Forth. These changes and the discarding of the cross compiler written in C can be seen in the Git repository this project comes in (<https://github.com/howerj/embed>).

The project, documentation and Forth images are under an MIT license.

The Virtual Machine

The virtual machine is incredibly simple and cut down at around 200 lines of C code, with most of the code being not being the virtual machine itself, but code to get data in and out of the system correctly, or setting the machine up. It is described in the appendix (at the end of this file), which also contains an example implementation of the virtual machine.

The virtual machine is 16-bit dual stack machine with an instruction set encoding which allows for many Forth words to be implemented in a single instruction. As the CPU is designed to execute Forth, Subroutine Threaded Code (STC) is the most efficient method of running Forth upon it.

What you are reading is itself a Forth program, all the explanatory text is are Forth comments. The file is fed through a preprocessor to turn it into a Markdown file for further processing.

Many Forths are written in an assembly language, especially the ones geared towards microcontrollers, although it is more common for new Forth interpreters to be written in C. A metacompiler is a Cross Compiler written in Forth.

References

- 'The Zen of eForth' by C. H. Ting
- <https://github.com/howerj/embed> (This project)
- <https://github.com/howerj/libforth>
- <https://github.com/howerj/forth-cpu>

Jones Forth:

- <https://rwmj.wordpress.com/2010/08/07/jonesforth-git-repository/>
- <https://github.com/AlexandreAbreu/jonesforth>

J1 CPU:

- <http://excamera.com/files/j1.pdf>
- <http://excamera.com/sphinx/fpga-j1.html>
- <https://github.com/jamesbowman/j1>
- <https://github.com/samawati/j1eforth>

Meta-compilation/Cross-Compilation:

- <http://www.ultratechnology.com/meta1.html>
- <https://en.wikipedia.org/wiki/Compiler-compiler#FORTH_metacompiler>

The Virtual Machine is specifically designed to execute Forth, it is a stack machine that allows many Forth words to be encoded in one instruction but does not contain any high level Forth words, just words like @, 'r>' and a few basic words for I/O. A full description of the virtual machine is in the appendix.

Metacompilation wordset

This section defines the metacompilation wordset as well as the assembler. The metacompiler, or cross compiler, requires some assembly instructions to be defined so the two word sets are interlinked.

A clear understanding of how Forth vocabularies work is needed before proceeding with the tutorial. Vocabularies are the way Forth manages namespaces and are generally talked about that much, they are especially useful (in fact pretty much required) for writing a metacompiler.

```
0002| only forth definitions hex
0003| variable meta          ( Metacompilation vocabulary )
0004| meta +order definitions
0005| variable assembler.1   ( Target assembler vocabulary )
0006| variable target.1      ( Target dictionary )
0007| variable tcp           ( Target dictionary pointer )
0008| variable tlast         ( Last defined word in target )
0009| variable tdoVar        ( Location of doVar in target )
0010| variable tdoConst      ( Location of doConst in target )
0011| variable tdoNext       ( Location of doNext in target )
0012| variable tdoPrintString ( Location of .string in target )
0013| variable tdoStringLit  ( Location of string-literal in target )
0014| variable fence         ( Do not peephole optimize before this point )
0015| 1984 constant #version ( Version number )
0016| 5000 constant #target  ( Memory location where the target image will be built )
0017| 2000 constant #max     ( Max number of cells in generated image )
0018| 2    constant =cell    ( Target cell size )
0019| -1   constant optimize ( Turn optimizations on [-1] or off [0] )
0020| 0    constant swap-endianess ( if true, swap the endianess )
0021| $4280 constant pad-area      ( area for pad storage )
0022| variable header -1 header !  ( if true target headers generated )
0023| $7FFF constant (rp0)   ( start of return stack )
0024| $4400 constant (sp0)   ( start of variable stack )
0025| 1   constant verbose   ( verbosity level, higher is more verbose )
0026| #target #max 0 fill    ( Erase the target memory location )
0027| : ]asm assembler.1 +order ; immediate        ( -- )
0028| : a: get-current assembler.1 set-current : ; ( "name" -- wid link )
```

```
0029| : a; [compile] ; set-current ; immediate     ( wid link -- )
0030| : ( [char] ) parse 2drop ; immediate ( "comment" -- discard until parenthesis )
0031| : \ source drop @ >in ! ; immediate  ( "comment" -- discard until end of line )
0032| : there tcp @ ;         ( -- a : target dictionary pointer value )
0033| : tc! #target + c! ;    ( u a -- : store character in target )
0034| : tc@ #target + c@ ;    ( a -- u : retrieve character from target )
0035| : [last] tlast @ ;      ( -- a : last defined word in target )
0036| : low  swap-endianess 0= if 1+ then ; ( b -- b : low byte at address )
0037| : high swap-endianess    if 1+ then ; ( b -- b : high byte at address)
0038| : t! over $FF and over high tc! swap 8 rshift swap low tc! ; ( u a -- )
0039| : t@ dup high tc@ swap low tc@ 8 lshift or ; ( a -- u )
0040| : 2/ 1 rshift ;               ( u -- u : non-standard definition divide by 2 )
0041| : talign there 1 and tcp +! ;  ( -- : align target dictionary pointer value )
0042| : tc, there tc! 1 tcp +! ;     ( c -- : write byte into target dictionary )
0043| : t,  there t!  =cell tcp +! ; ( u -- : write cell into target dictionary )
0044| : tallot tcp +! ;              ( n -- : allocate memory in target dictionary )
0045| : update-fence there fence ! ; ( -- : update optimizer fence location )
0046| : $literal                     ( <string>, -- )
0047|   [char] " word count dup tc, 1- for count tc, next drop talign update-fence ;
0048| : tcells =cell * ;             ( u -- a )
0049| : tbody 1 tcells + ;           ( a -- a )
0050| : meta! ! ;                    ( u a --  )
0051| : dump-hex #target there $10 + dump ; ( -- )
0052| : locations ( -- : list all words and locations in target dictionary )
0053|   target.1 @
0054|   begin
0055|     ?dup
0056|   while
0057|     dup
0058|     nfa count type space dup ( <- @warning should use target nfa! )
0059|     cfa tbody @ u. cr        ( <- @warning should use target cfa! )
0060|     $3FFF and @
0061|   repeat ;
0062| : display ( -- : display metacompilation and target information )
0063|   verbose 0= if exit then
0064|   hex
0065|   ." COMPILATION COMPLETE" cr
0066|   verbose 1 u> if
0067|     dump-hex cr
0068|     ." TARGET DICTIONARY: " cr
0069|     locations
0070|   then
0071|   ." HOST: "       here        . cr
0072|   ." TARGET: "     there       . cr
0073|   ." HEADER: "     #target $30 dump cr ;
0074| : checksum #target there crc ; ( -- u : calculate CRC of target image )
0075| : save-hex ( -- : save target binary to file )
0076|    #target #target there + (save) throw ;
0077| : finished ( -- : save target image and display statistics )
0078|    display
0079|    only forth definitions hex
0080|    ." SAVING... " save-hex ." DONE! " cr
0081|    ." STACK> " .s cr ;
0082| : [a] ( "name" -- : find word and compile an assembler word )
0083|   token assembler.1 search-wordlist 0= abort" [a]? "
0084|   cfa compile, ; immediate
0085| : asm[ assembler.1 -order ; immediate ( -- )
```

There are five types of instructions, which are differentiated from each other by the top bits of the instruction.

```
0086| a: #literal $8000 a; ( literal instruction - top bit set )
0087| a: #alu     $6000 a; ( ALU instruction, further encoding below... )
0088| a: #call    $4000 a; ( function call instruction )
0089| a: #?branch $2000 a; ( branch if zero instruction )
0090| a: #branch  $0000 a; ( unconditional branch )
```

An ALU instruction has a more complex encoding which can be seen in the table in the appendix, it consists of a few flags for moving values to different registers before and after the ALU operation to perform, an ALU operation, and a return and variable stack increment/decrement.

Some of these operations are more complex than they first appear, either because they do more than a single line explanation allows for, or because they are not typical instructions that you would find in an actual processors ALU and are only possible within the context of a virtual machine. Operations like '#um/mod' are an example of the former, '#save' is an example of the later.

The most succinct description of these operations, and the virtual machine, is the source code for it which weighs in at under two hundred lines of C code. Unfortunately this would not include that rationale that led to the virtual machine being the way it is. ALU Operations

```
0091| a: #t      $0000 a; ( T = t )
0092| a: #n      $0100 a; ( T = n )
0093| a: #r      $0200 a; ( T = Top of Return Stack )
0094| a: #[t]    $0300 a; ( T = memory[t] )
0095| a: #n->[t] $0400 a; ( memory[t] = n )
0096| a: #t+n    $0500 a; ( n = n+t, T = carry )
0097| a: #t*n    $0600 a; ( n = n*t, T = upper bits of multiplication )
0098| a: #t&n    $0700 a; ( T = T and N )
0099| a: #t|n    $0800 a; ( T = T  or N )
0100| a: #t^n    $0900 a; ( T = T xor N )
0101| a: #~t     $0A00 a; ( Invert T )
0102| a: #t-1    $0B00 a; ( T == t - 1 )
0103| a: #t==0   $0C00 a; ( T == 0? )
0104| a: #t==n   $0D00 a; ( T = n == t? )
0105| a: #nu<t   $0E00 a; ( T = n < t )
0106| a: #n<t    $0F00 a; ( T = n < t, signed version )
0107| a: #n>>t   $1000 a; ( T = n right shift by t places )
0108| a: #n<<t   $1100 a; ( T = n left  shift by t places )
0109| a: #sp@    $1200 a; ( T = variable stack depth )
0110| a: #rp@    $1300 a; ( T = return stack depth )
0111| a: #sp!    $1400 a; ( set variable stack depth )
0112| a: #rp!    $1500 a; ( set return stack depth )
0113| a: #save   $1600 a; ( Save memory disk: n = start, T = end, T' = error )
0114| a: #tx     $1700 a; ( Transmit Byte: t = byte, T' = error )
0115| a: #rx     $1800 a; ( Block until byte received, T = byte/error )
0116| a: #u/mod  $1900 a; ( Remainder/Divide: )
0117| a: #/mod   $1A00 a; ( Signed Remainder/Divide: )
0118| a: #bye    $1B00 a; ( Exit Interpreter )
```

The Stack Delta Operations occur after the ALU operations have been executed. They affect either the Return or the Variable Stack. An ALU instruction without one of these operations (generally) do not affect the stacks.

```
0119| a: d+1     $0001 or a; ( increment variable stack by one )
0120| a: d-1     $0003 or a; ( decrement variable stack by one )
0121| ( a: d-2   $0002 or a; ( decrement variable stack by two )
0122| a: r+1     $0004 or a; ( increment variable stack by one )
0123| a: r-1     $000C or a; ( decrement variable stack by one )
0124| ( a: r-2   $0008 or a; ( decrement variable stack by two )
```

All of these instructions execute after the ALU and stack delta operations have been performed except r->pc, which occurs before. They form part of an ALU operation.

```
0125| a: r->pc   $0010 or a; ( Set Program Counter to Top of Return Stack )
0126| a: n->t    $0020 or a; ( Set Top of Variable Stack to Next on Variable Stack )
0127| a: t->r    $0040 or a; ( Set Top of Return Stack to Top on Variable Stack )
0128| a: t->n    $0080 or a; ( Set Next on Variable Stack to Top on Variable Stack )
```

There are five types of instructions; ALU operations, branches, conditional branches, function calls and literals. ALU instructions comprise of an ALU operation, stack effects and register move bits. Function returns are part of the ALU operation instruction set.

```
0129| : ?set dup $E000 and abort" argument too large " ;
0130| a: branch  2/ ?set [a] #branch  or t, a; ( a -- : an Unconditional branch )
0131| a: ?branch 2/ ?set [a] #?branch or t, a; ( a -- : Conditional branch )
0132| a: call    2/ ?set [a] #call    or t, a; ( a -- : Function call )
0133| a: ALU        ?set [a] #alu     or    a; ( u -- : Make ALU instruction )
0134| a: alu                    [a] ALU  t, a; ( u -- : ALU operation )
0135| a: literal ( n -- : compile a number into target )
0136|   dup [a] #literal and if   ( numbers above $7FFF take up two instructions )
0137|     invert recurse  ( the number is inverted, an literal is called again )
0138|     [a] #~t [a] alu ( then an invert instruction is compiled into the target )
0139|   else
0140|     [a] #literal or t, ( numbers below $8000 are single instructions )
0141|   then a;
0142| a: return ( -- : Compile a return into the target )
0143|    [a] #t [a] r->pc [a] r-1 [a] alu a;
```

The following words implement a primitive peephole optimizer, which is not the only optimization done, but is the major one. It performs tail call optimizations and merges the return instruction with the previous instruction if possible. These simple optimizations really make a lot of difference in the size of meta-compiled program. It means proper tail recursive procedures can be constructed.

The optimizer is wrapped up in the *exit,* word, it checks a fence variable first, then the previously compiled cell to see if it can replace the last compiled cell.

The fence variable is an address below which the peephole optimizer should not look, this is to prevent the optimizer looking at data and merging with it, or breaking control structures.

An exit can be merged into an ALU instruction if it does not contain any return stack manipulation, or information from the return stack. This includes operations such as *r->pc*, or *r+1*.

A call then an exit can be replaced with an unconditional branch to the call.

If no optimization can be performed an *exit* instruction is written into the target.

The optimizer can be held off manually be inserting a *nop*, which is a call or instruction which does nothing, before the *exit*.

Other optimizations performed by the metacompiler, but not this optimizer, include; inlining constant values and addresses, allowing the creation of headerless words which are named only in the metacompiler and not in the target, and the 'fallthrough;' word which allows for greater code sharing. Some of these optimizations have a manual element to them, such as 'fallthrough;'.

```
0144| : previous there =cell - ;                      ( -- a )
0145| : lookback previous t@ ;                        ( -- u )
0146| : call? lookback $E000 and [a] #call = ;       ( -- t )
0147| : call>goto previous dup t@ $1FFF and swap t! ; ( -- )
0148| : fence? fence @  previous u> ;                 ( -- t )
0149| : safe? lookback $E000 and [a] #alu = lookback $001C and 0= and ; ( -- t )
0150| : alu>return previous dup t@ [a] r->pc [a] r-1 swap t! ; ( -- )
0151| : exit-optimize                                ( -- )
0152|   fence? if [a] return exit then
0153|   call?  if call>goto  exit then
0154|   safe?  if alu>return exit then
0155|   [a] return ;
0156| : exit, exit-optimize update-fence ;           ( -- )
```

compile-only and *immediate* set bits in the latest defined word for making a word a "compile only" word (one which can only be executed from within a word definition) and "immediate" respectively. None of these are relevant to the execution of the metacompiler so are not checked by it, but are needed when the target Forth is up and running.

These words affect the target dictionaries word definitions and not the meta-compilers definitions.

```
0157| : compile-only tlast @ t@ $8000 or tlast @ t! ; ( -- )
0158| : immediate tlast @ t@ $4000 or tlast @ t! ;    ( -- )
```

mcreate creates a word in the metacompilers dictionary, not the targets. For each word we create in the meta-compiled Forth we will need to create at least one word in the meta-compilers dictionary which contains an address of the Forth in the target.

```
0159| : mcreate get-current >r target.1 set-current create r> set-current ;
```

thead compiles a word header into the target dictionary with a name given a string. It is used by *t:*.

```
0160| : thead ( b u -- : compile word header into target dictionary )
0161|   header @ 0= if 2drop exit then
0162|   talign
0163|   there [last] t, tlast !
0164|   there #target + pack$ c@ 1+ aligned tcp +! talign ;
```

lookahead parses the next word but leaves it in the input stream, pushing a string to the parsed word. This is needed as we will be creating two words with the same name with a word defined later on called *t:*, it creates a word in the meta-compilers dictionary and compiles a word with a header into the target dictionary.

```
0165| : lookahead ( -- b u : parse a word, but leave it in the input stream )
0166|   >in @ >r bl parse r> >in ! ;
```

The word *h:* creates a headerless word in the target dictionary for space saving reasons and to declutter the target search order. Ideally it would instead add the word to a different vocabulary, so it is still accessible to the programmer, but there is already very little room on the target.

h: does not actually affect the target dictionary, it can be used by itself and is called by *t:*. *h:* is used in conjunction with either *fallthrough;* or *t;* (*t;* calls *fallthrough;*). *h:* but does several things:

1. It sets *<literal>* to the meta-compilers version of *literal* so that when we are compiling words within a meta-compiled word definition it does the right thing, which is compile a number literal into the target dictionary.
2. It pushes a magic number *$F00D* onto the stack, this popped off and checked for by *fallthrough;*, if it is not present we have messed up a word definition some how.
3. It creates a meta-compiler word in *target.1*, this word-list consists of pointers into the target word definitions. The created word when called compiles a pointer to the word it represents into the target image.
4. It updates the *fence* variable to hold off the optimizer.

fallthrough; allows words to be created which instead of exiting just carry on into the next word, which is a space saving measure and provides a minor speed boost. Due to the way this Forth stores word headers *fallthrough;* cannot be used to fall through to a word defined with a word header.

t; does everything *fallthrough;* does except it also compiles an exit into the dictionary, which is how a normal word definition is terminated.

```
0167| : literal [a] literal ;                      ( u -- )
0168| : h: ( -- : create a word with no name in the target dictionary )
0169|  ' literal <literal> !
0170|  $F00D mcreate there , update-fence does> @ [a] call ;
```

t: does everything *h:* does but also compiles a header for that word into the dictionary using *thead*. It does affect the target dictionary directly.

```
0171| : t: ( "name", -- : creates a word in the target dictionary )
0172|   lookahead thead h: ;
```

@warning: Only use *fallthrough* to fallthrough to words defined with *h:*.

```
0173| : fallthrough;
0174|   ' (literal) <literal> !
0175|   $F00D <> if source type cr 1 abort" unstructured! " then ;
0176| : t; fallthrough; optimize if exit, else [a] return then ;
```

fetch-xt is used to check that a variable contains a valid execution token, to implement certain functionality we will need to refer to functions yet to be defined in the target dictionary. We will not be able to use these features until we have defined these functions. For example we cannot use *tconstant*, which defines a constant in the target dictionary, until we have defined the target versions of *doConst*. A reference to this function will be stored in *tdoConst* once it has been defined in the target dictionary.

```
0177| : fetch-xt @ dup 0= abort" (null) " ; ( a -- xt )
```

tconstant as mentioned defines a constant in the target dictionary which is visible in that target dictionary (that is, it has a header and when *words* is run in the target it will be in that list).

tconstant behaves like *constant* does, it parses out a name and pops a variable off of the stack. As mentioned, it cannot be used until *tdoConst* has been filled with a reference to the targets *doConst*. *tconstant* makes a word in the meta-compiler which points to a word it makes in the target. This words purpose when run in the target is to push a constant onto the stack. When the constant is referenced when compiling words with the meta-compiler it does not compile references to the constant, but instead it finds out what the constant was and compiles it in as a literal - which is a small optimization.

```
0178| : tconstant ( "name", n --, Run Time: -- )
0179|   >r
0180|   lookahead
0181|   thead
0182|   there tdoConst fetch-xt [a] call r> t, >r
0183|   mcreate r> ,
0184|   does> @ tbody t@ [a] literal ;
```

tvariable is like *tconstant* expect for variables. It requires *tdoVar* is set to a reference to targets version of *doVar* which pushes a pointer to the targets variable location when run in the target. It does a similar optimization as *tconstant*, it does not actually compile a call to the created variables *doVar* field but instead compiles the address as a literal in the target when the word is called by the meta-compiler.

```
0185| : tvariable ( "name", n -- , Run Time: -- a )
0186|   >r
0187|   lookahead
0188|   thead
0189|   there tdoVar fetch-xt [a] call r> t, >r
0190|   mcreate r> ,
0191|   does> @ tbody [a] literal ;
```

tlocation just reserves space in the target.

```
0192| : tlocation ( "name", n -- : Reserve space in target for a memory location )
0193|   there swap t, mcreate , does> @ [a] literal ;
0194| : [t] ( "name", -- a : get the address of a target word )
0195|   token target.1 search-wordlist 0= abort" [t]? "
0196|   cfa >body @ ;
```

@warning only use *[v]* on variables, not *tlocations*

```
0197| : [v] [t] =cell + ; ( "name", -- a )
```

xchange takes two vocabularies defined in the target by their variable names, "name1" and "name2", and updates "name1" so it contains the previously defined words, and makes "name2" the vocabulary which subsequent definitions are added to.

```
0198| : xchange ( "name1", "name2", -- : exchange target vocabularies )
0199|   [last] [t] t! [t] t@ tlast meta! ;
```

These words implement the basic control structures needed to make applications in the meta-compiled program, they are no immediate words and they do not need to be, *t:* and *t;* do not change the interpreter state, once the actual metacompilation begins everything is command mode.

```
0200| : begin  there update-fence ;                ( -- a )
0201| : until  [a] ?branch ;                       ( a -- )
0202| : if     there update-fence 0 [a] ?branch  ; ( -- a )
0203| : skip   there update-fence 0 [a] branch ;   ( -- a )
0204| : then   begin 2/ over t@ or swap t! ;       ( a -- )
0205| : else   skip swap then ;                    ( a -- a )
0206| : while  if swap ;                           ( a -- a a )
0207| : repeat [a] branch then update-fence ;      ( a -- )
0208| : again  [a] branch update-fence ;           ( a -- )
0209| : aft    drop skip begin swap ;              ( a -- a )
0210| : constant mcreate , does> @ literal ;       ( "name", a -- )
0211| : [char] char literal ;                       ( "name" )
0212| : postpone [t] [a] call ;                    ( "name", -- )
0213| : next tdoNext fetch-xt [a] call t, update-fence ; ( a -- )
0214| : exit exit, ;                               ( -- )
0215| : ' [t] literal ;                            ( "name", -- )
```

@bug maximum string length is 32 bytes, not 255 as it should be.

```
0216| : ." tdoPrintString fetch-xt [a] call $literal ; ( "string", -- )
0217| : $" tdoStringLit   fetch-xt [a] call $literal ; ( "string", -- )
```

The following section adds the words implementable in assembly to the metacompiler, when one of these words is used in the meta-compiled program it will be implemented in assembly.

```
0218| : nop     ]asm  #t       alu asm[ ;
0219| : dup     ]asm  #t       t->n   d+1   alu asm[ ;
0220| : over    ]asm  #n       t->n   d+1   alu asm[ ;
0221| : invert  ]asm  #~t      alu asm[ ;
0222| : um+     ]asm  #t+n     alu asm[ ;
0223| : +       ]asm  #t+n     n->t   d-1   alu asm[ ;
0224| : um*     ]asm  #t*n     alu asm[    ;
0225| : *       ]asm  #t*n     n->t   d-1   alu asm[ ;
0226| : swap    ]asm  #n       t->n   alu asm[ ;
0227| : nip     ]asm  #t       d-1    alu asm[ ;
0228| : drop    ]asm  #n       d-1    alu asm[ ;
0229| : >r      ]asm  #n       t->r   d-1   r+1   alu asm[ ;
0230| : r>      ]asm  #r       t->n   d+1   r-1   alu asm[ ;
0231| : r@      ]asm  #r       t->n   d+1   alu asm[ ;
0232| : @       ]asm  #[t]     alu asm[ ;
0233| : !       ]asm  #n->[t]  d-1    alu asm[ ;
0234| : rshift  ]asm  #n>>t    d-1    alu asm[ ;
0235| : lshift  ]asm  #n<<t    d-1    alu asm[ ;
0236| : =       ]asm  #t==n    d-1    alu asm[ ;
0237| : u<      ]asm  #nu<t    d-1    alu asm[ ;
0238| : <       ]asm  #n<t     d-1    alu asm[ ;
```

```
0239| : and     ]asm  #t&n     d-1    alu asm[ ;
0240| : xor     ]asm  #t^n     d-1    alu asm[ ;
0241| : or      ]asm  #t|n     d-1    alu asm[ ;
0242| : sp@     ]asm  #sp@     t->n   d+1   alu asm[ ;
0243| : sp!     ]asm  #sp!     alu asm[ ;
0244| : 1-      ]asm  #t-1     alu asm[ ;
0245| : rp@     ]asm  #rp@     t->n   d+1   alu asm[ ;
0246| : rp!     ]asm  #rp!     d-1    alu asm[ ;
0247| : 0=      ]asm  #t==0    alu asm[ ;
0248| : yield?  ]asm  #bye     alu asm[ ;
0249| : rx?     ]asm  #rx      t->n   d+1   alu asm[ ;
0250| : tx!     ]asm  #tx      n->t   d-1   alu asm[ ;
0251| : (save)  ]asm  #save    d-1    alu asm[ ;
0252| : um/mod  ]asm  #u/mod   t->n   alu asm[ ;
0253| : /mod    ]asm  #/mod    t->n   alu asm[ ;
0254| : /       ]asm  #/mod    d-1    alu asm[ ;
0255| : mod     ]asm  #/mod    n->t   d-1   alu asm[ ;
0256| : rdrop   ]asm  #t       r-1    alu asm[ ;
```

Some words can be implemented in a single instruction which have no analogue within Forth.

```
0257| : dup@    ]asm  #[t]     t->n   d+1 alu asm[ ;
0258| : dup0=   ]asm  #t==0    t->n   d+1 alu asm[ ;
0259| : dup>r   ]asm  #t       t->r   r+1 alu asm[ ;
0260| : 2dup=   ]asm  #t==n    t->n   d+1 alu asm[ ;
0261| : 2dupxor ]asm  #t^n     t->n   d+1 alu asm[ ;
0262| : 2dup<   ]asm  #n<t     t->n   d+1 alu asm[ ;
0263| : rxchg   ]asm  #r       t->r       alu asm[ ;
0264| : over-and  ]asm  #t&n      alu asm[ ;
0265| : over-xor  ]asm  #t^n      alu asm[ ;
```

for needs the new definition of *>r* to work correctly.

```
0266| : for >r begin ;
0267| : meta: : ;
0268| ( : :noname h: ; )
0269| : : t: ;
0270| meta: ; t; ;
0271| hide meta:
0272| hide t:
0273| hide t;
0274| ]asm #~t              ALU asm[ constant =invert ( invert instruction )
0275| ]asm #t  r->pc    r-1 ALU asm[ constant =exit   ( return/exit instruction )
0276| ]asm #n  t->r d-1 r+1 ALU asm[ constant =>r     ( to r. stk. instruction )
0277| $20   constant =bl         ( blank, or space )
0278| $D    constant =cr         ( carriage return )
0279| $A    constant =lf         ( line feed )
0280| $8    constant =bs         ( back space )
0281| $1B   constant =escape     ( escape character )
0282| $10   constant dump-width  ( number of columns for *dump* )
0283| $50   constant tib-length  ( size of terminal input buffer )
0284| $40   constant word-length ( maximum length of a word )
0285| $40   constant c/l         ( characters per line in a block )
0286| $10   constant l/b         ( lines in a block )
0287| $F    constant l/b-1       ( lines in a block, less one )
0288| (rp0) constant rp0         ( start of return stack )
0289| (sp0) constant sp0         ( start of variable stack )
0290| $2BAD constant magic       ( magic number for compiler security )
0291| $F    constant #highest    ( highest bit in cell )
0292| ( Volatile variables )
```

$4000 Unused

```
0293| $4002 constant last-def    ( last, possibly unlinked, word definition )
0294| $4006 constant id          ( used for source id )
0295| $4008 constant seed        ( seed used for the PRNG )
0296| $400A constant handler     ( current handler for throw/catch )
0297| $400C constant block-dirty ( -1 if loaded block buffer is modified )
0298| $4010 constant <key>       ( -- c : new character, blocking input )
0299| $4012 constant <emit>      ( c -- : emit character )
0300| $4014 constant <expect>    ( "accept" vector )
0301| ( $4016 constant <tap>     ( "tap" vector, for terminal handling )
0302| ( $4018 constant <echo>    ( c -- : emit character )
0303| ( $4020 constant <ok>      ( -- : display prompt )
0304| ( $4022 constant _literal  ( u -- u | : handles literals )
0305| $4110 constant context     ( holds current context for search order )
0306| $4122 constant #tib        ( Current count of terminal input buffer )
0307| $4124 constant tib-buf     ( ... and address )
0308| $4126 constant tib-start   ( backup tib-buf value )
```

$4280 == pad-area

```
0309| $C    constant vm-options     ( Virtual machine options register )
0310| $1A   constant header-length  ( location of length in header )
0311| $1C   constant header-crc     ( location of CRC in header )
0312| $22   constant header-options ( location of options bits in header )
0313| target.1 +order         ( Add target word dictionary to search order )
0314| meta -order meta +order ( Reorder so *meta* has a higher priority )
0315| forth-wordlist   -order ( Remove normal Forth words to prevent accidents )
```

The Target Forth

With the assembler and meta compiler complete, we can now make our target application, a Forth interpreter which will be able to read in this file and create new, possibly modified, images for the Forth virtual machine to run.

The Image Header

The following *t,* sequence reserves space and partially populates the image header with file format information, based upon the PNG specification. See <http://www.fadden.com/tech/file-formats.html> and <https://stackoverflow.com/questions/323604> for more information about how to design binary formats.

The header contains enough information to identify the format, the version of the format, and to detect corruption of data, as well as having a few other nice properties - some having to do with how other systems and programs may deal with the binary (such as have a string literal *FTH* to help identify the binary format, and the first byte being outside the ASCII range of characters so it is obvious that the file is meant to be treated as a binary and not as text).

```
0316| 0       t, \  $0: PC: program counter, jump to start / reset vector
0317| 0       t, \  $2: T, top of stack
0318| (rp0)   t, \  $4: RP0, return stack pointer
0319| (sp0)   t, \  $6: SP0, variable stack pointer
0320| 0       t, \  $8: Instruction exception vector
0321| $8000   t, \  $A: VM Memory Size in cells
0322| $0000   t, \  $C: VM Options
0323| $0000   t, \  $E: VM Reserved
0324| $0000   t, \ $10: VM Reserved
0325| $4689   t, \ $12: 0x89 'F'
0326| $4854   t, \ $14: 'T'  'H'
0327| $0A0D   t, \ $16: '\r' '\n'
```

```
0328| $0A1A    t, \ $18: ^Z   '\n'
0329| 0        t, \ $1A: For Length of Forth image, different from VM size
0330| 0        t, \ $1C: For CRC of Forth image, not entire VM memory
0331| $0001    t, \ $1E: Endianess check
0332| #version t, \ $20: Version information
0333| $0001    t, \ $22: Header options
```

@bug There is something very weird going on with the initial header size It should be possible to allocate memory how I want here, up to a point, but it causes things to fail. Sometimes '0 t,' does not work, but '0 t, 0t,' does. ## First Word Definitions

After the header two short words are defined, visible only to the meta compiler and used by its internal machinery. The words are needed by *tvariable* and *tconstant*, and these constructs cannot be used without them. This is an example of the metacompiler and the meta-compiled program being intermingled, which should be kept to a minimum.

```
0334| h: doVar   r> ;    ( -- a : push return address and exit to caller )
0335| h: doConst r> @ ;  ( -- u : push value at return address and exit to caller )
```

Here the address of *doVar* and *doConst* in the target is stored in variables accessible by the metacompiler, so *tconstant* and *tvariable* can compile references to them in the target.

```
0336| [t] doVar   tdoVar   meta!
0337| [t] doConst tdoConst meta!
```

Next some space is reserved for variables which will have no name in the target and are not on the target Forths search order. We do this with *tlocation*. These variables are needed for the internal working of the interpreter but the application programmer using the target Forth can make do without them, so they do not have names within the target.

A short description of the variables and their uses:

cp is the dictionary pointer, which usually is only incremented in order to reserve space in this dictionary. Words like *,* and *:* advance this variable.

root-voc, *editor-voc*, *assembler-voc*, and *__forth-wordlist* are variables which point to word lists, they can be used with *set-order* and pointers to them may be returned by *get-order*. By default the only vocabularies loaded are the root vocabulary (which contains only a few vocabulary manipulation words) and the forth vocabulary are loaded (which contains most of the words in a standard Forth).

current contains a pointer to the vocabulary which new words will be added to when the target is up and running, this will be the forth vocabulary, or *__forth-wordlist*.

None of these variables are set to any meaningful values here and will be updated during the metacompilation process.

```
0338| 0 tlocation cp                ( Dictionary Pointer: Set at end of file )
0339| 0 tlocation root-voc          ( root vocabulary )
0340| 0 tlocation editor-voc        ( editor vocabulary )
0341| ( 0 tlocation assembler-voc   ( assembler vocabulary )
0342| 0 tlocation _forth-wordlist   ( set at the end near the end of the file )
0343| 0 tlocation current           ( WID to add definitions to )
```

Target Assembly Words

The first words added to the target Forths dictionary are all based on assembly instructions. The definitions may seem like nonsense, how does the definition of + work? It appears that the definition calls itself, which obviously would not work. The answer is in the order new words are added into the dictionary. In Forth, a word definition is not placed in the search order until the definition of that word is complete, this allows the previous definition of a word to be use within that definition, and requires a separate word (*recurse*) to implement recursion.

However, the words *:* and *;* are not the normal Forth define and end definitions words, they are the meta-compilers and they behave differently, *:* is implemented with *t:* and *;* with *t;*.

t: uses *create* to make a new variable in the meta-compilers dictionary that points to a word definition in the target, it also creates the words header in the target (*h:* is the same, but without a header being made in the target). The word is compilable

into the target as soon as it is defined, yet the definition of + is not recursive because the meta-compilers search order, *meta*, is higher that the search order for the words containing the meta-compiled target addresses, *target.1*, so the assembly for + gets compiled into the definition of +.

Manipulation of the word search order is key in understanding how the metacompiler works.

The following words will be part of the main search order, in *forth-wordlist* and in the assembly search order.

```
0344| ( : nop      nop      ; ( -- : do nothing )
0345| : dup      dup      ; ( n -- n n : duplicate value on top of stack )
0346| : over     over     ; ( n1 n2 -- n1 n2 n1 : duplicate second value on stack )
0347| : invert   invert   ; ( u -- u : bitwise invert of value on top of stack )
0348| : um+      um+      ; ( u u -- u carry : addition with carry )
0349| : +        +        ; ( u u -- u : addition without carry )
0350| : um*      um*      ; ( u u -- ud : multiplication  )
0351| : *        *        ; ( u u -- u : multiplication )
0352| : swap     swap     ; ( n1 n2 -- n2 n1 : swap two values on stack )
0353| : nip      nip      ; ( n1 n2 -- n2 : remove second item on stack )
0354| : drop     drop     ; ( n -- : remove item on stack )
0355| : @        @        ; ( a -- u : load value at address )
0356| : !        !        ; ( u a -- : store *u* at address *a* )
0357| : rshift   rshift   ; ( u1 u2 -- u : shift u2 by u1 places to the right )
0358| : lshift   lshift   ; ( u1 u2 -- u : shift u2 by u1 places to the left )
0359| : =        =        ; ( u1 u2 -- t : does u2 equal u1? )
0360| : u<       u<       ; ( u1 u2 -- t : is u2 less than u1 )
0361| : <        <        ; ( u1 u2 -- t : is u2 less than u1, signed version )
0362| : and      and      ; ( u u -- u : bitwise and )
0363| : xor      xor      ; ( u u -- u : bitwise exclusive or )
0364| : or       or       ; ( u u -- u : bitwise or )
0365| ( : sp@    sp@      ; ( ??? -- u : get stack depth )
0366| ( : sp!    sp!      ; ( u -- ??? : set stack depth )
0367| : 1-       1-       ; ( u -- u : decrement top of stack )
0368| : 0=       0=       ; ( u -- t : if top of stack equal to zero )
0369| h: yield?  yield?   ; ( u -- !!! : exit VM with *u* as return value )
0370| h: rx?     rx?      ; ( -- c | -1 : fetch a single character, or EOF )
0371| h: tx!     tx!      ; ( c -- : transmit single character )
0372| : (save)   (save)   ; ( u1 u2 -- u : save memory from u1 to u2 inclusive )
0373| : um/mod   um/mod   ; ( d  u2 -- rem div : mixed unsigned divide/modulo )
0374| : /mod     /mod     ; ( u1 u2 -- rem div : signed divide/modulo )
0375| : /        /        ; ( u1 u2 -- u : u1 divided by u2 )
0376| : mod      mod      ; ( u1 u2 -- u : remainder of u1 divided by u2 )
```

Forth Implementation of Arithmetic Functions

As an aside, the basic arithmetic functions of Forth can be implemented in terms of simple addition, some tests and bit manipulation, if they are not available for your system. Division, the remainder operation and multiplication are provided by the virtual machine in this case, but it is interesting to see how these words are put together. The first task it to implement an add with carry, or *um+*. Once this is available, *um/mod* and *um** are coded.

```
: um+ ( w w -- w carry )
  over over + &gt;r
  r@ 0 &lt; invert &gt;r
  over over and
  0 &lt; r&gt; or &gt;r
  or 0 &lt; r&gt; and invert 1 +
  r&gt; swap ;

\$F constant #highest
: um/mod ( ud u -- r q )
  ?dup 0= if \$A -throw exit then
```

```
    2dup u&lt;
    if negate #highest
      for &gt;r dup um+ &gt;r &gt;r dup um+ r&gt; + dup
        r&gt; r@ swap &gt;r um+ r&gt; or
        if &gt;r drop 1+ r&gt; else drop then r&gt;
      next
      drop swap exit
    then drop 2drop [-1] dup ;

  : m/mod ( d n -- r q ) \ floored division
    dup 0&lt; dup&gt;r
    if
      negate &gt;r dnegate r&gt;
    then
    &gt;r dup 0&lt; if r@ + then r&gt; um/mod r&gt;
    if swap negate swap exit then ;

  : um* ( u u -- ud )
    0 swap ( u1 0 u2 ) #highest
    for dup um+ &gt;r &gt;r dup um+ r&gt; + r&gt;
      if &gt;r over um+ r&gt; + then
    next rot-drop ;
```

The other arithmetic operations follow from the previous definitions almost trivially:

```
  : /mod  over 0&lt; swap m/mod ; ( n n -- r q )
  : mod  /mod drop ;           ( n n -- r )
  : /    /mod nip ;            ( n n -- q )
  : *    um* drop ;            ( n n -- n )
  : m* 2dup xor 0&lt; &gt;r abs swap abs um* r&gt; if dnegate then ; ( n n -- d )
  : */mod  &gt;r m* r&gt; m/mod ;    ( n n n -- r q )
  : */  */mod nip ;            ( n n n -- q )
```

Inline Words

These words can also be implemented in a single instruction, yet their definition is different for multiple reasons. These words should only be use within a word definition begin defined within the running target Forth, so they have a bit set in their header indicating as such.

Another difference is how these words are compiled into a word definition in the target, which is due to the fact they manipulate the return stack. These words are *inlined*, which means the instruction they contain is written directly into a definition being defined in the running target Forth instead of a call to a word that contains the assembly instruction, calls obviously change the return stack, so these words would either have to take that into account or the assembly instruction could be inlined, the later option has been taken.

As these words are never actually called, as they are only of use in compile mode, and then they are inlined instead of being called, we can leave off *;* which would write an exit instruction on the end of the definition.

```
0377| there constant inline-start
0378| ( : rp@ rp@   fallthrough; compile-only ( -- u )
0379| ( : rp! rp!   fallthrough; compile-only ( u --, R: --- ??? )
0380| : exit  exit  fallthrough; compile-only ( -- )
0381| : >r    >r    fallthrough; compile-only ( u --, R: -- u )
0382| : r>    r>    fallthrough; compile-only ( -- u, R: u -- )
0383| : r@    r@    fallthrough; compile-only ( -- u )
0384| : rdrop rdrop fallthrough; compile-only ( --, R: u -- )
0385| there constant inline-end
```

Finally we can set the *assembler-voc* to variable, we will add to the assembly vocabulary later, but all of the words defined so far belong in the assembly vocabulary. Unfortunately, the assembler when run in the target Forth interpreter will compile

calls to the instructions like + or *xor*, only a few words will be inlined. There are potential solutions to this problem, but they are not worth further complicating the Forth just yet. [last] [t] assembler-voc t!

```
0386| $2       tconstant cell  ( size of a cell in bytes )
0387| $0       tvariable >in   ( Hold character pointer when parsing input )
0388| $0       tvariable state ( compiler state variable )
0389| $0       tvariable hld   ( Pointer into hold area for numeric output )
0390| $10      tvariable base  ( Current output radix )
0391| $0       tvariable span  ( Hold character count received by expect   )
0392| $8       tconstant #vocs ( number of vocabularies in allowed )
0393| $400     tconstant b/buf ( size of a block )
0394| 0        tvariable blk   ( current blk loaded, set in *cold* )
0395| #version constant  ver   ( eForth version )
0396| pad-area tconstant pad   ( pad variable - offset into temporary storage )
0397| $ffff    tvariable dpl   ( number of places after fraction )
0398| 0        tvariable <literal> ( holds execution vector for literal )
0399| 0        tvariable <boot>  ( -- : execute program at startup )
0400| 0        tvariable <ok>
```

The following execution vectors would/will be added if there is enough space, it is very useful to have hooks into the system to change how the interpreter behaviour works. Being able to change how the Forth interpreter handles number parsing allows the processing of double or floating point numbers in a system that could otherwise not handle them.

```
0401| ( 0        tvariable <error>      ( execution vector for interpreter error )
0402| ( 0        tvariable <interpret>  ( execution vector for interpreter )
0403| ( 0        tvariable <abort>      ( execution vector for abort handler )
0404| ( 0        tvariable <at-xy>      ( execution vector for at-xy )
0405| ( 0        tvariable <page>       ( execution vector for page )
0406| ( 0        tvariable <number>     ( execution vector for >number )
0407| ( 0        tvariable hidden       ( vocabulary for hidden words )
```

Basic Word Set

The following section of words is purely a space saving measure, or they allow for other optimizations which also save space. Examples of this include *[-1]*; any number about $7FFF requires two instructions to encode, numbers below only one, -1 is a commonly used number so this allows us to save on space.

This does not explain the creation of a word to push the number zero though, this only takes up one instruction. This is instead explained by the interaction of the peephole optimizer with function calls, calls to function can be turned into a branch if that instruction were to be followed by an exit instruction because it is at the end of a word definition. This cannot be said of literals. This allows us to save space under special circumstances.

The following example illustrates this:

```
| FORTH CODE                   | PSEUDO ASSEMBLER         |
| ---------------------------- | ------------------------ |
| : push-zero 0 literal ;      | LITERAL(0) EXIT          |
| : example-1 drop 0 literal ; | DROP LITERAL(0) EXIT     |
| : example-2 drop 0 literal ; | DROP BRANCH(push-zero)   |
```

Where *example-1* being unoptimized requires three instructions, whereas *example-2* requires only two, with the two instruction overhead of *push-zero*.

Optimizations like this explain some of the structure of the Forth code, it is better to exit early and heavily factorize code if space is at a premium, which it is due to the way the virtual machine works (both it being 16-bit only, and only allowing the first 16KiB to be used for program storage). Factoring code like this is similar to performing LZW compression, or similar dictionary related compression schemes.

Whilst factoring words into smaller, cleaner, definitions is highly encouraged for Forth code (it is often an art coming up with the right word name and associated concept it encapsulates), making words like *2drop-0* is not. It hurts readability as there is no reason or idea backing a word like *2drop-0*, even if it is fairly clear what it does from its name.

```
0408| h: [-1] -1 ;                ( -- -1 : space saving measure, push -1 )
0409| h: 0x8000 $8000 ;           ( -- $8000 : space saving measure, push $8000 )
0410| h: 2drop-0 drop fallthrough; ( n n -- 0 )
0411| h: drop-0 drop fallthrough;  ( n -- 0 )
0412| h: 0x0000 $0000 ;           ( -- $0000 : space/optimization, push $0000 )
0413| h: state@ state @ ;         ( -- u )
0414| h: first-bit 1 and ;        ( u -- u )
0415| h: in! >in ! ;              ( u -- )
0416| h: in@ >in @ ;              ( -- u )
0417| h: base@ base @ ;           ( -- u )
0418| h: base! base ! ;           ( u -- )
```

Now the implementation of the Forth interpreter without the apologies for the words in the prior section. This group of words implement some of the basic words expected in Forth; simple stack manipulation, tests, and other one, or two line definitions that do not really require an explanation of how they work - only why they are useful. Some of the words are described by their stack comment entirely, like *2drop*, other like *cell+* require a reason for such a simple word (they embody a concept or they help hide implementation details).

```
0419| : 2drop drop drop ;         ( n n -- )
0420| : 1+ 1 + ;                  ( n -- n : increment a value  )
0421| : negate invert 1+ ;        ( n -- n : negate a number )
0422| : - negate + ;              ( n1 n2 -- n : subtract n1 from n2 )
0423| h: over- over - ;           ( u u -- u u )
0424| h: over+ over + ;           ( u1 u2 -- u1 u1+2 )
0425| : aligned dup first-bit + ; ( b -- a )
0426| : bye -1 0 yield? nop ( $38 -throw ) ; ( -- : leave the interpreter )
0427| h: cell- cell - ;           ( a -- a : adjust address to previous cell )
0428| : cell+  cell + ;           ( a -- a : move address forward to next cell )
0429| : cells 1 lshift ;          ( n -- n : convert cells count to address count )
0430| : chars 1 rshift ;          ( n -- n : convert bytes to number of cells )
0431| : ?dup dup if dup exit then ; ( n -- 0 | n n : duplicate non zero value )
0432| : >  swap  < ;              ( n1 n2 -- t : signed greater than, n1 > n2 )
0433| : u> swap u< ;              ( u1 u2 -- t : unsigned greater than, u1 > u2 )
0434| h: u>= u< invert ;          ( u1 u2 -- t : unsigned greater/equal )
0435| :  <>  = invert ;           ( n n -- t : not equal )
0436| : 0<> 0= invert ;           ( n n -- t : not equal  to zero )
0437| : 0> 0 > ;                  ( n -- t : greater than zero? )
0438| : 0< 0 < ;                  ( n -- t : less than zero? )
0439| : 2dup over over ;          ( n1 n2 -- n1 n2 n1 n2 )
0440| : tuck swap over ;          ( n1 n2 -- n2 n1 n2 )
0441| : +! tuck @ +  fallthrough; ( n a -- : increment value at *a* by *n* )
0442| h: swap! swap ! ;           ( a u -- )
0443| h: zero  0 swap! ;          ( a -- : zero value at address )
0444| : 1+!   1  swap +! ;        ( a -- : increment value at address by 1 )
0445| : 1-! [-1] swap +! ;        ( a -- : decrement value at address by 1 )
0446| : 2! ( d a -- ) tuck ! cell+ ! ;      ( n n a -- )
0447| : 2@ ( a -- d ) dup cell+ @ swap @ ;  ( a -- n n )
0448| : get-current current @ ;             ( -- wid )
0449| : set-current current ! ;             ( wid -- )
0450| : bl =bl ;                            ( -- c )
0451| : within over- >r - r> u< ;           ( u lo hi -- t )
0452| h: s>d dup 0< ;                       ( n -- d )
0453| : abs s>d if negate exit then ;       ( n -- u )
0454| : source #tib 2@ ;                    ( -- a u )
0455| h: tib source drop ;                  ( -- a )
0456| : source-id id @ ;                    ( -- 0 | -1 )
0457| : rot >r swap r> swap ;               ( n1 n2 n3 -- n2 n3 n1 )
0458| : -rot rot rot ;                      ( n1 n2 n3 -- n3 n1 n2 )
0459| h: rot-drop rot drop ;               ( n1 n2 n3 -- n2 n3 )
0460| h: d0= or 0= ;                       ( d -- t )
```

```
0461| h: dnegate invert >r invert 1 um+ r> + ; ( d -- d )
0462| h: d+  >r swap >r um+ r> + r> + ;      ( d d -- d )
0463| ( : 2swap >r -rot r> -rot ; ( n1 n2 n3 n4 -- n3 n4 n1 n2 )
```

: d< rot
2dup > if = nip nip if 0 exit then [-1] exit then 2drop u< ; (d – f)

```
0464| ( : d>  2swap d< ;                      ( d -- t )
0465| ( : du> 2swap du< ;                     ( d -- t )
0466| ( : d= rot xor -rot xor xor 0= ;      ( d d -- t )
0467| ( : d- dnegate d+ ;                    ( d d -- d )
0468| ( : dabs  s>d if dnegate exit then ;  ( d -- ud )
0469| ( : even first-bit 0= ;               ( u -- t )
0470| ( : odd even 0= ;                     ( u -- t )
0471| ( : pow2? dup dup 1- and 0= and ;     ( u -- u|0 : is u a power of 2? )
0472| ( : opposite? xor 0< ;                ( n n -- f : true if opposite signs )
```

execute requires an understanding of the return stack, much like *doConst* and *doVar*, when given an execution token of a word, a pointer to its Code Field Address (or CFA), *execute* will call that word. This allows us to call arbitrary function and change, or vector, execution at run time. All *execute* needs to do is push the address onto the return stack and when *execute* exits it will jump to the desired word, the callers address is still on the return stack, so when the called word exit it will jump back to *executes* caller.

@execute is similar but it only executes the token if it is non-zero.

```
0473| : execute >r ;                   ( cfa -- : execute a function )
0474| h: @execute @ ?dup if >r then ;  ( cfa -- )
```

As the virtual machine is only addressable by cells, and not by characters, the words *c@* and *c!* cannot be defined as simple assembly primitives, they must be defined in terms of *@* and *!*. It is not difficult to do, but does mean these two primitives are slower than might be first thought.

```
0475| : c@ dup@ swap first-bit 3 lshift rshift $FF and ; ( b --c : char load )
0476| : c! ( c b -- : store character at address )
0477|   tuck first-bit 3 lshift dup>r
0478|   lshift over @
0479|   $FF r> 8 xor lshift and or swap! ;
```

command? will be used later for words that are state aware. State awareness and whether the interpreter is in command mode, or compile mode, as well as immediate words, will require a lot of explanation for the beginner until they are understood. This is best done elsewhere. *command?* is used to determine if the interpreter is in command mode or not, it returns true if it is.

```
0480| h: command? state@ 0= ;              ( -- t )
```

here, *align*, *cp!* and *allow* all manipulate the dictionary pointer, which is a common operation. *align* aligns the pointer up to the next cell boundary, and *cp!* sets the dictionary pointer to a value whilst enforcing that the value written is aligned.

here retrieves the current dictionary pointer, which is needed for compiling control structures into words, so is used by words like *if*, and has other uses as well.

```
0481| : here cp @ ;                        ( -- a )
0482| : align here fallthrough;            ( -- )
0483| h: cp! aligned cp ! ;                ( n -- )
```

allot is used in Forth to allocate memory after using *create* to make a new word. It reserves space in the dictionary, space can be deallocated by giving it a negative number, however this may trash whatever data is written there and should not generally be done.

Typical usage:

```
create x $20 allot ( Create an array of 32 values )
$cafe  x $10 !     ( Store '$cafe' at element 16 in array )
```

```
0484| : allot cp +! ;                          ( n -- )
```

2>r and *2r>* are like *rot* and *-rot*, useful but they should not be overused. The words move two values to and from the return stack. Care should exercised when using them as the return stack can easily be corrupted, leading to unpredictable behavior, much like all the return stack words. The optimizer might also change a call to one of these words into a jump, which should be avoided as it could cause problems in edge cases, so do not use these words directly before an *exit* or *;*.

```
0485| h: 2>r rxchg swap >r >r ;              ( u1 u2 --, R: -- u1 u2 )
0486| h: 2r> r> r> swap rxchg nop ;          ( -- u1 u2, R: u1 u2 -- )
```

doNext needs to be defined before we can use *for... next* loops, the metacompiler compiles a reference to this word when *next* is encountered.

It is worth explaining how this word works, it is a complex word that requires an understanding of how the return stack words, as well as how both *for* and *next* work.

The *for... next* loop accepts a value, *u* and runs for *u+1* times. *for* puts the loop counter onto the return stack, meaning the loop counter value is available to us as the first stack element, but also meaning if we want to exit from within a *for... next* loop we must pop off the value from the return stack first.

The *next* word compiles a *doNext* into the dictionary, and then the address to jump back to, just after the *for* has compiled a *>r* into the dictionary.

doNext has two possible actions, it either takes the branch back to the place after *for* if the loop counter non zero, being careful to decrement the loop counter and restoring it the correct place, or it jumps over the place where the back jump address is stored in the case when the loop counter is zero, removing the loop counter from the return stack.

This is all possible, because when *doNext* is called (and it must be called, not jumped to), the return address points to the cell after *doNext* is compiled into. By manipulating the return stack correctly it can change the program flow to either jump over the next cell, or jump to the address contained in the next cell. Understanding the simpler *doConst* and *doVar* helps in understanding this word.

```
0487| h: doNext 2r> ?dup if 1- >r @ >r exit then cell+ >r ;
0488| [t] doNext tdoNext meta!
```

min and *max* are standard operations in many languages, they operate on signed values. Note how they are factored to use the *mux* word, with *min* falling through into it, and *max* calling *mux*, all to save on space.

```
0489| : min 2dup< fallthrough;              ( n n -- n )
0490| h: mux if drop exit then nip ;        ( n1 n2 b -- n : multiplex operation )
0491| : max 2dup > mux ;                    ( n n -- n )
0492| ( : 2over 2>r 2dup 2r> 2swap ; )
0493| ( : 2nip 2>r 2drop 2r> nop ; )
0494| ( : 4dup 2over 2over ; )
0495| ( : dmin 4dup d< if 2drop exit else 2nip ; )
0496| ( : dmax 4dup d> if 2drop exit else 2nip ; )
```

key retrieves a single character of input, it is a vectored word so the method used to get data can be changed.

It calls *bye* if the End of File character is returned (-1, which is outside the normal byte range).

```
0497| : key <key> @execute dup [-1] = if bye then ; ( -- c )
```

/string, *+string* and *count* are for manipulating strings, *count* words best on counted strings which have a length prefix, but can be used to advance through an array of byte data, whereas */string* is used with a address and length pair that is already on the stack. */string* will not advance the pair beyond the end of the string.

```
0498| : /string over min rot over+ -rot - ;  ( b u1 u2 -- b u : advance string u2 )
0499| h: +string 1 /string ;                 ( b u -- b u : )
0500| : count dup 1+ swap c@ ;               ( b -- b u )
0501| h: string@ over c@ ;                   ( b u -- b u c )
```

crc computes the 16-bit CCITT CRC over a segment of memory, and *ccitt* is the word that does the polynomial checking. It can be also be used as a crude Pseudo Random Number Generator. CRC routines are useful for detecting memory corruption in the Forth image.

```
0502| h: ccitt ( crc c -- crc : crc polynomial $1021 AKA "x16 + x12 + x5 + 1" )
0503|   over $8 rshift xor   ( crc x )
0504|   dup  $4 rshift xor   ( crc x )
0505|   dup  $5 lshift xor   ( crc x )
0506|   dup  $C lshift xor   ( crc x )
0507|   swap $8 lshift xor ; ( crc )
0508| : crc ( b u -- u : calculate ccitt-ffff CRC )
0509|   [-1] ( -1 = 0xffff ) >r
0510|   begin
0511|     dup
0512|   while
0513|    string@ r> swap ccitt >r +string
0514|   repeat 2drop r> ;
0515| ( : random ( -- u : pseudo random number )
0516| (  seed @ 0= seed toggle seed @ 0 ccitt dup seed ! ; )
```

address and *@address* are for use with the previous word point in the word header, the top two bits for other purposes (a *compile-only* and an *immediate* word bit). This makes traversing the dictionary a little more tricker than normal, and affects functions like *words* and *search-wordlist* Code can only exist in the first 16KiB of space anyway, so the top two bits cannot be used for anything else. It is debatable as to what the best way of marking words as immediate is, to use unused bits in a word header, or to place *immediate* words in a special vocabulary which a minority of Forths do. Most Forths use the *unused-bits-in-word-header* approach.

```
0517| h: @address @ fallthrough;             ( a -- a )
0518| h: address $3FFF and ;                 ( a -- a : mask off address bits )
```

last gets a pointer to the most recently defined word, which is used to implement words like *recurse*, as well as in words which must traverse the current word list.

```
0519| h: last get-current @address ;         ( -- pwd )
```

A few character emitting words will now be defined, it should be obvious what these words do, *emit* is the word that forms the basis of all these words. By default it is set to the primitive virtual machine instruction *tx!*. In eForth another character emitting primitive was defined alongside *emit*, which was *echo*, which allowed for more control in how the interpreter interacts with the programmer and other programs. It is not necessary in a hosted Forth to have such a mechanism, but it can be added back in as needed, we will see commented out relics of this features later, when we see *^h* and *ktap*.

```
0520| ( h: echo <echo> @execute ; )          ( c -- )
0521| : emit <emit> @execute ;               ( c -- : write out a char )
0522| : cr =cr emit =lf emit ;               ( -- : emit a newline )
0523| : space     1 fallthrough;             ( -- : emit a space )
0524| h: spaces =bl fallthrough;             ( +n -- )
0525| h: nchars                              ( +n c -- : emit c n times )
0526|    swap 0 max for aft dup emit then next drop ;
0527| h: colon-space [char] : emit space ;               ( -- )
```

depth and *pick* require knowledge of how this Forth implements its stacks. *sp0* contains the location of the stack pointer when there is nothing on the stack, and *sp@* contains the current position. Using this it is possible to work out how many items, or

how deep it is. *pick* is used to pick an item at an arbitrary depth from the return stack. This version of *pick* does this by indexing into the correct position and using a memory load operation to do this.

In some systems Forth is implemented on this is not possible to do, for example some Forths running on stack CPUs specifically designed to run Forth have stacks made in hardware which are not memory mapped. This is not just a hypothetical, the H2 Forth CPU (based on the J1 CPU) available at <https://github.com/howerj/forth-cpu> has stacks whose only operations are to increment and decrement them by a small number, and to get the current stack depth. On a platform like this, *pick* can still be implemented, but it is more complex and can be done like this:

: pick ?dup if swap >r 1- pick r> swap exit then dup ;

```
0528| h: vrelative cells sp@ swap - ;  ( u -- u : position relative to sp )
0529| : depth sp0 vrelative chars 1- ; ( -- u : get current depth )
0530| : pick vrelative @ ;             ( vn...v0 u -- vn...v0 vu )
```

ndrop removes a variable number of items off the variable stack, which is sometimes needed for cleaning things up before exiting a word.

```
0531| h: ndrop vrelative sp! drop ; ( 0u...nu n -- : drop n cells )
```

>char takes a character and converts it to an underscore if it is not printable, which is useful for printing out arbitrary sections of memory which may contain spaces, tabs, or non-printable. *list* and *dump* use this word. *typist* can either use *>char* to print out a memory range or it can print out the value regardless if it is printable.

```
0532| h: >char dup $7F =bl within if drop [char] _ then ; ( c -- c )
0533| : type 0 fallthrough;                  ( b u -- )
0534| h: typist                             ( b u f -- : print a string )
0535|   >r begin dup while
0536|     swap count r@
0537|     if
0538|       >char
0539|     then
0540|     emit
0541|     swap 1-
0542|   repeat
0543|   rdrop 2drop ;
0544| h: print count type ;                   ( b -- )
0545| h: $type [-1] typist ;                  ( b u --  )
```

cmove and *fill* are generic memory related functions for moving blocks of memory around and setting blocks of memory to a specific value respectively.

```
0546| : cmove for aft >r dup c@ r@ c! 1+ r> 1+ then next 2drop ; ( b b u -- )
0547| : fill swap for swap aft 2dup c! 1+ then next 2drop ; ( b u c -- )
```

Exception Handling

catch and *throw* are complex and very useful words, these words are minor modifications to the ones provided in the ANS Forth standard.

The standard describes the stack effects of *catch* and *throw* as so:

```
catch ( i*x xt -- j*x 0 | i*x n )
throw ( k*x n -- k*x | i*x n )
```

Which is a difficult stack effect to digest, although their use is quite simple.

catch accepts an execution token, which it executes, if throw is somehow invoked when the execution token is run, then *catch* catches the exception and returns the exception number thrown. If no exception was thrown then *catch* returns zero.

throw is used to throw exceptions, it can be used anyway to indicate something has gone wrong, or to affect the control throw of a program in a portable way (instead of messing around with the return stack using *>r* and *r>*, which is non-portable), although it is ill advised to use exceptions for control flow purposes. *throw* only throws an exception if the value provided to it was non-zero, otherwise execution continues as normal.

Example usage:

```
: word-the-might-fail 2 2 + 5 = if -9001 throw then ;
: foo
   ' word-the-might-fail catch
   ?dup if abort" Something has gone terribly wrong..." then
   ." Everything went peachy" cr ;
```

```
0548| : catch ( i*x xt -- j*x 0 | i*x n )
0549|   sp@       >r
0550|   handler @ >r
0551|   rp@ handler !
0552|   execute
0553|   r> handler !
0554|   r> drop-0 ;
0555| : throw ( k*x n -- k*x | i*x n )
0556|   ?dup if
0557|     handler @ rp!
0558|     r> handler !
0559|     rxchg ( *rxchg* is equivalent to 'r> swap >r' )
0560|     sp! drop r>
0561|   then ;
```

Negative numbers take up two cells in a word definition when compiled into one, whilst positive words only take up one cell, as a space saving measure we define *-throw*. Which negates a number before calling *throw*.

We then do something curious, we set the second cell in the target virtual machine image to a branch to *-throw*. This is because it is the the virtual machine sets the program counter to the second cell (at address $2, or 1 cell in) whenever an exception is raised when executing an instruction, such as a division by zero. By setting the cell to the execution token divided by two we are setting that cell to an unconditional branch to *-throw*. The virtual machine also puts a number indicating what the exception was on the top of the stack so it is possible to determine what went wrong.

```
0562| h: -throw negate throw ;  ( u -- : negate and throw )
0563| [t] -throw 2/ 4 tcells t!
```

?ndepth throws an exception if a certain number of items on the stack do not exist. It is possible to use this primitive to implement some basic checks to make sure that words are passed the correct number of arguments.

By using *?ndepth* strategically it is possible to catch errors quite quickly with minimal overhead in speed and size by selecting only a few words to put depth checking in.

```
0564| h: 1depth 1 fallthrough; ( ??? -- : check depth is at least one )
0565| h: ?ndepth depth 1- u> if 4 -throw exit then ; ( ??? n -- check depth )
0566| h: 2depth 2 ?ndepth ;    ( ??? -- :  check depth is at least two )
```

Numeric Output

With the basic word set in place and exception handling, as well as some variables being defined, it is possible to define the numeric output wordset, this word set will allow numbers to be printed or strings to the numeric representation of a number defined.

Numeric output in Forth, as well as input, is controlled by the *base* variable, it is possible to enter numbers and print them out in binary, octal, decimal, hexadecimal or any base from base 2 to base 36 inclusive. This section only deals with numeric

output. The numeric output string is formed within a temporary buffer which can either be printed out or copied to another location as needed, the method for doing this is known as Pictured Numeric Output. The words *<#*, *#*, *#>* and *hold* form the kernel from which the other numeric output words are formed.

As a note, Base-36 can be used as a fairly compact textual encoding of binary data if needed, like Base-64 encoding but without the need for a special encoding and decoding scheme.

First, a few utility words are formed, *decimal* and *hex* set the base to known values. These are needed so the programmer can get the interpret back to a known state when they have set the radix to a value (remember, *10* is valid and different value in every base!), as well as a quick shorthand.

The word *radix* is then defined, which is used to check that the *base* variable is set to a meaningful value before it is used, between 2 and 36, inclusive as previously mentioned. Radixes or bases higher than 10 uses the alphabet after 0-9 to represent higher numbers, so *a* corresponds to *10*, there are only 36 alphanumeric characters (ignoring character case) meaning higher numbers cannot be represented. If the radix is outside of this range, then base is set back to its default, decimal, and an exception is thrown.

```
0567| : decimal  $A base! ;                        ( -- )
0568| : hex     $10 base! ;                        ( -- )
0569| h: radix base@ dup 2 - $22 u> if decimal $28 -throw exit then ; ( -- u )
```

digit converts a number to its character representation, but it only deals with numbers less than 36, it does no checking for the output base, but is a straight forward conversion utility. It assumes that the ASCII character set is being used.

extract is used to divide the number being converted by the current output radix, extracting a single digit which can be passed to *digit* for conversion.

hold then takes this extracted and converted digit and places in the hold space, this is where the string representing the number to be output is held. It adds characters in reverse order to the hold space, which is due to how *extract* works, *extract* gets the lowest digit first.

Lets look at how conversion would work for the number 1234 in base 10. We start of the number to convert and extract the first digit:

```
1234 / 10 = 123, remainder 4 (hold ASCII 4)
```

We then add *4* to the hold space, which is the last digit that needs to be output, hence storage in reverse order. We then continue on with the output conversion:

```
123 / 10  = 12,  remainder 3 (hold ASCII 3)
12  / 10  = 1,   remainder 2 (hold ASCII 2)
1   / 10  = 0,   remainder 1 (hold ASCII 1)
0 <- conversion complete!
```

If we did not store the string in reverse order, we would end up printing *4321*.

The words *<#*, *#*, and *#>* call these words to do the work of numeric conversion, but before this is described, Notice the checking that is done within each of these words, *hold* makes sure that too many characters are not stored in the hold space. *#* checks the depth of the stack before it is called and the base variable is only accessed with the *radix* word. This combination of checking catches most errors that occur and makes sure they do not propagate.

```
0570| : hold  hld @ 1- dup hld ! c! fallthrough;             ( c -- )
0571| h: ?hold pad $100 - hld @ u> if $11 -throw exit then ; ( -- )
0572| h: extract dup>r um/mod rxchg um/mod r> rot ;          ( ud ud -- ud u )
0573| h: digit 9 over < 7 and + [char] 0 + ;                ( u -- c )
```

The quartet formed by "<# # #s #>" look intimidating to the new comer, but is quite simple. They allow the format of numeric output to be controlled quite easily by the programmer. A description of these words:

- *<#* resets the hold space and initializes the conversion.
- *#* extracts, converts and holds a single character.

- *#s* repeatedly calls *#* until the number has been fully converted.
- *#>* finalizes the conversion and pushes a string.

Anywhere within this conversion arbitrary characters can be added to the hold space with *hold*. These words should always be used together, whenever you see *<#*, you should see an enclosing *#>*. Once the *#>* has been called the number provided has been fully converted to a number string in the current output base, and can be printed or copied.

These words will go on to form more convenient words for numeric output, like *.*, or *u.r*.

```
0574| : #> 2drop hld @ pad over- ;              ( w -- b u )
0575| : # 2depth 0 base@ extract digit hold ;   ( d -- d )
0576| : #s begin # 2dup d0= until ;             ( d -- 0 )
0577| : <# pad hld ! ;                          ( -- )
```

sign is used with the Pictured Numeric Output words to add a sign character if the number is negative, *str* is then defined to convert a number in any base to its signed representation, in actual use however we will only use *str* for base 10 numeric output, we usually do not want to print the sign of a number when operating with a non-decimal base.

```
0578| : sign  0< if [char] - hold exit then ;     ( n -- )
0579| h: str ( n -- b u : convert a signed integer to a numeric string )
0580|   dup>r abs 0 <# #s r> sign #> ;
```

.r, *u.r* are used as the basis of further words with a more controlled output format. They implement right justification of a number (signed for *.r*, unsigned for *u.r*) by the number of spaces specified by the first argument provided to them. This is useful for printing out columns of data that is for human consumption.

u. prints out an unsigned number with a trailing space, and *.* prints out a signed number when operating in decimal, and an unsigned one otherwise, again with a trailing space. Neither adds leading spaces.

```
0581| h: (u.) 0 <# #s #> ;             ( u -- b u : turn *u* into number string )
0582| : u.r >r (u.) r> fallthrough;    ( u +n -- : print u right justified by +n)
0583| h: adjust over- spaces type ;    ( b n n -- )
0584| h: 5u.r 5 u.r ;                  ( u -- )
0585| ( :  .r >r str r> adjust ;       ( n n -- : print n, right justified by +n )
0586| : u.  (u.) space type ;          ( u -- : print unsigned number )
0587| :  .  radix $A xor if u. exit then str space type ; ( n -- print number )
0588| ( : >base swap base @ >r base ! execute r> base ! ; )
0589| ( : d. $a  '  . >base ; )
0590| ( : h. $10 ' u. >base ; )
```

holds and *.base* can be defined as so:

```
: holds begin dup while 1- 2dup + c@ hold repeat 2drop ;
: .base base@ dup decimal base! ; ( -- )
```

If needed. *?* is another common utility for printing out the contents at an address:

```
: ? @ . ; ( a -- : display the contents in a memory cell )
```

Words that use the numeric output words can be defined, *.free* can now be defined which prints out the amount of space left in memory for programs

```
0591| h: unused $4000 here - ;         ( -- u : unused program space )
0592| h: .free unused u. ;             ( -- : print unused program space )
```

String Handling and Input

pack$ and *=string* are more advanced string handling words for copying strings to a new location, turning it into a counted string, which *pack$* does, or for comparing two string, which *=string* does.

expect, *query* and *accept* are used for fetching a line of input which can be further processed. Input in Forth is fundamentally line based, a line is fetch from the terminal, which is then parsed into tokens delimited by whitespace called words, which are then processed by the Forth interpreter. On hosted systems the input typed into a terminal is usually line buffered, you type in a line and hit return, then the line is passed to the program reading from the terminal. This is not the case on all systems, often it is the case that the Forth will be on a different target and the programmer is talking to it via a serial port, the programmer sends a character - not an entire line - and the Forth interpreter echos back a character, or not. This means that the Forth interpreter has to mimic the line handling normally provided by the terminal emulator. As this is a hosted Forth it does not have to do this, however the capability to handle input in this way is left commented out so the system is easier to port across to such platforms. Either way, a line of text needs to be fetched from an input stream.

First to describe *pack$* and *=string*.

pack$ takes a string, its length, and a place which is assumed to be large enough to store the string in as a final argument. Creates a counted string and places it in the target location, this word is especially useful for the creation of word headers which will be needed later.

=string checks for string equality, as it has the length of both strings it checks their lengths first before proceeding to check all of the string, this saves a little time. Only a yes/no to the comparison is returned as a boolean answer, unlike strcmp in C.

down aligns a cell down to the next aligned address.

```
0593| h: down cell negate and ; ( a -- a : align down )
0594| : pack$ ( b u a -- a ) \ null fill
0595|   aligned dup>r over
0596|   dup down
0597|   - over+ zero 2dup c! 1+ swap cmove r> ;
0598| : =string ( a1 u2 a1 u2 -- t : string equality )
0599|   >r swap r> ( a1 a2 u1 u2 )
0600|   over-xor if drop 2drop-0 exit then
0601|   for ( a1 a2 )
0602|     aft
0603|       count >r swap count r> xor
0604|       if rdrop 2drop-0 exit then
0605|     then
0606|   next 2drop [-1] ;
```

^h and *ktap* are commented out as they are not needed, they deal with line based input, some sections of *tap* and *accept* are commented out as well as they are not needed on a hosted system, thus the job of *accept* is simplified. It is still worth talking about what these words do however, in case they need to be added back in.

accept is the word that does all of the work and calls *^h* and *ktap* when needed, *query* and *expect* a wrappers around it. Image we are talking to the Forth interpreter over a serial line, if the programmer sends a character to the Forth interpreter it is up to the Forth interpreter how to respond, the remote interpreter needs to echo back characters to the programmer otherwise they will be greeted with a blank terminal. Likewise the remote Forth interpreter needs to process not only new lines, indicating a new line of input should be accepted for processing, but it also needs to deal with backspace characters, which are used to delete characters in the current input line. It also needs to make sure it does not overflow the input buffer, or when deleting characters go beyond the beginning of the input buffer.

accept takes an address of an input buffer and a length of it, the words *^h*, *tap* and *ktap* are free to move around inside that buffer and do so based on the latest character received.

^h takes a pointer to the bottom of the input buffer, one to the end of the input buffer and the current position. It decrements the current position within the buffer and uses *echo* - not *emit* - to output a backspace to move the terminal cursor back one place, a space to erase the character and a backspace to move the cursor back one space again. It only does this if it will not go below the bottom of the input buffer.

```
: ^h ( bot eot cur -- bot eot cur )
  &gt;r over r@ &lt; dup
```

```
    if
      =bs dup echo =bl echo echo
    then r&gt; + ;
```

ktap processes the current character received, and has the same buffer information that *^h* does, in fact it passes that information to *^h* to process if the character is a backspace. It also handles the case where a line feed, indicating a new line, has been entered. It returns a modified buffer.

```
  : ktap ( bot eot cur c -- bot eot cur )
    dup =lf ( &lt;-- was =cr ) xor
    if =bs xor
      if =bl tap else ^h then
      exit
    then drop nip dup ;
```

tap is used to store a character in the current line, *ktap* can also call this word, *tap* uses echo to echo back the character - it is currently commented out as this is handled by the terminal emulator but would be needed for serial communication.

```
0607| h: tap ( dup echo ) over c! 1+ ; ( bot eot cur c -- bot eot cur )
```

accept takes an buffer input buffer and returns a pointer and line length once a line of text has been fully entered. A line is considered fully entered when either a new line is received or the maximum number of input characters in a line is received, *accept* checks for both of these conditions. It calls *tap* to do the work. In an alternate version it calls the execution vector *<tap>*, which is usually set to *ktap*, which is shown as a commented out line.

```
0608| : accept ( b u -- b u )
0609|   over+ over
0610|   begin
0611|     2dupxor
0612|   while
0613|     key dup =lf xor if tap else drop nip dup then
0614|     \ The alternative *accept* code replaces the line above:
0615|     \
0616|     \   key  dup =bl - 95 u< if tap else <tap> @execute then
0617|     \
0618|   repeat drop over- ;
```

<expect> is set to *accept* later on in this file, but can be changed if needed. *expect* and *query* both call *accept* this way. *query* uses the systems input buffer and stores the result in a know location so that the rest of the interpreter can use the results. *expect* gets a buffer from the use and stores the length of the resulting string in *span*.

```
0619| : expect <expect> @execute span ! drop ;                      ( b u -- )
0620| : query tib tib-length <expect> @execute #tib ! drop-0 in! ; ( -- )
```

query stores its results in the Terminal Input Buffer (TIB), which is way the word *tib* gets its name. The TIB is a simple data structure which contains a pointer to the buffer itself and the length of the most current input line.

Now we have a line based input system, and from the previous chapters we also have numeric output, the Forth interpreter is starting to take shape.

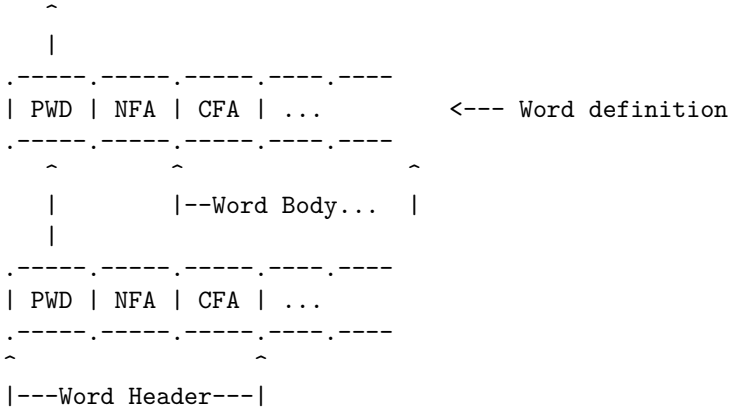
Dictionary Words

These words either navigate around the word header, or search through the dictionary to find a word, both word sets are related. This section now requires an understanding on how this Forth lays out its word headers, each Forth tends to do this in a slightly different way and more modern Forths use more advanced (and more complex, perhaps un-forth-like) techniques.

The dictionary is organized into word lists, and these words lists are simply linked lists of all the words in that list. To find a definition by name we search through all of the word lists in the current search order following the linked lists until they terminate.

This Forth, like Jones Forth (another literate Forth designed to teach how a Forth interpreter works), lays out its dictionary in a way that is simpler than having a separate storage area for word names, but complicates other parts of the system. Most other Forths have a separate space for string storage where names of words and pointers to their code field are kept. This is how things were done in the original eForth model.

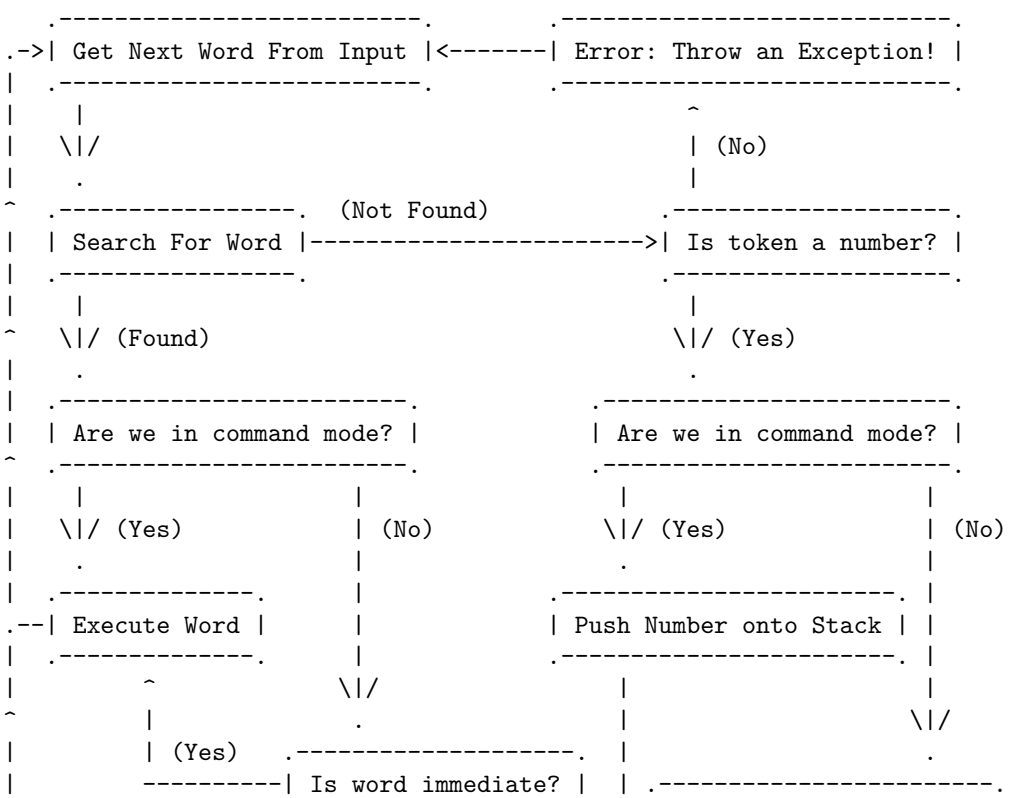
A word list looks like this..

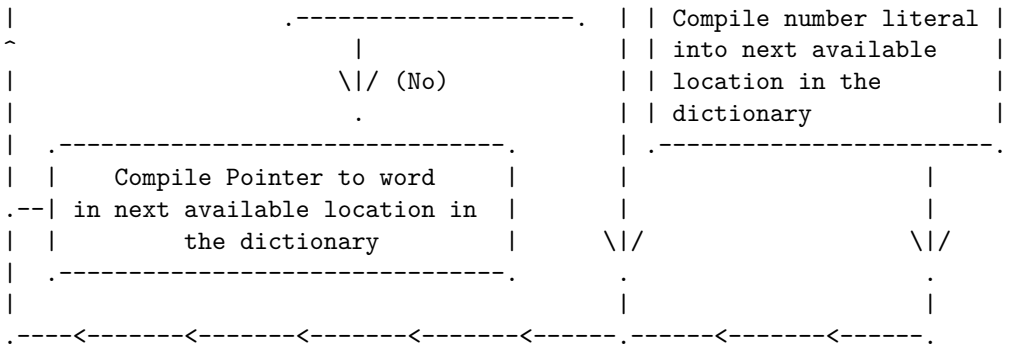


PWD is the Previous Word pointer, *NFA* is the Name Field Address, and the *CFA* is the first code word in the Forth word. In other Forths this is a special field, but as this is a Subroutine Threaded forth, designed to execute on a Forth machine, this field does not need to do anything. It is simply the first bit of code that gets executed in a Forth definition, and things point to the *CFA*. The rest of the word definition follows and includes the *CFA*. The *NFA* is a variable length field containing the name of the Forth word, it is stored as a counted string, which is what *pack$* was defined for.

The *PWD* field is not just a simple pointer, as already mentioned, it is also used to store information about the word, the top two most bits store whether a word is compile only (the top most bit) meaning the word should only be used within a word definition, being compiled into it, and whether or not the word is an immediate word or not (the second highest bit in the *PWD* field). Another property of a word is whether it is an inline word or not, which is a property of the location of where the word is defined, and applies to words like *r@* and *r>*, this is done by checking to see if the word is between two sentinel values.

Immediate and compiling words are a very important concept in Forth, and it is possible for the programmer to define their own immediate words. The Forth Read-Evaluate loop is quite simple, as this diagram shows:





No matter how complex the Forth system may appear, this loop forms the heart of it: parse a word and execute it or compile it, with numbers handled as a special case. Immediate words are always executed, whereas compiling words may be executed depending on whether we are in command mode, or in compile mode, which is stored in the *state* variable. There are several words that affect the interpreter state, such as *:*, *;*, *[*, and *]*.

We should now talk about some examples of immediate and compiling words, *:* is a compiling word that when executed reads in a single token and creates a new word header with this token. It does not add the new word to the definition just yet. It also does one other things, it switches the interpreter to compile mode, words that are not immediate are instead compiled into the dictionary, as are numbers. This allows us to define new words. However, we need immediate words so that we can break out of compile mode, and enter back into command word so we can actually execute something. *;* is an example of an immediate word, it is executed instead of compiled into the dictionary, it switches the interpreter back into command mode, as well as finishing off the word definition by compiling an exit instructions into the dictionary, and adding the word into the current definitions word list.

The control structure words, like *if*, *for*, and *begin*, are just words as well, they are immediate words and will be explained later.

nfa and *cfa* both take a pointer to the *PWD* field and adjust it to point to different sections of the word.

```
0621| : nfa address cell+ ; ( pwd -- nfa : move to name field address)
0622| : cfa nfa dup c@ + cell+ down ; ( pwd -- cfa )
```

.id prints out a words name field.

```
0623| h: .id dup nfa print space ;          ( pwd -- pwd : print out a word )
```

immediate?, *compile-only?* and *inline?* are the tests words that all take a pointer to the *PWD* field.

```
0624| h: immediate? $4000 fallthrough;  ( pwd -- t : is word immediate? )
0625| h: set? swap @ and 0<> ;          ( a u -- t : it any of 'u' set? )
0626| h: compile-only? 0x8000 set? ;    ( pwd -- t : is word compile only? )
0627| h: inline? inline-start inline-end within ; ( pwd -- t : is word inline? )
```

Now we know the structure of the dictionary we can define some words that work with it. We will define *find* and *search-wordlist*, which will be derived from *finder* and *searcher*. *searcher* will attempt to find a word in a specific word list, and *find* will attempt to find a word in all the word lists in the current order.

searcher and *finder* both return as much information about the words they find as possible, if they find them. *searcher* returns the *PWD* of the word that points to the found word, the *PWD* of the found word and a number indicating whether the found word is immediate (1) or a compiling word (-1). It returns zero, the original counted string, and another zero if the word could not be found in its first argument, a word list (wid). The word is provided as a counted string in the second argument to *searcher*.

finder wraps up *searcher* and applies it to all the words in the search order. It returns the same values as *searcher* if a word is found, returns the original counted word string address if it was not found, as well as zeros.

```
0628| h: searcher ( a wid -- pwd pwd 1 | pwd pwd -1 | 0 a 0 : find a word in a WID )
0629|   swap >r dup
```

```
0630|   begin
0631|     dup
0632|   while
0633|     dup nfa count r@ count =string
0634|     if ( found! )
0635|       rdrop
0636|       dup immediate? 1 or negate exit
0637|     then
0638|     nip dup @address
0639|   repeat
0640|   rdrop 2drop-0 ;
0641| h: finder ( a -- pwd pwd 1 | pwd pwd -1 | 0 a 0 : find a word dictionary )
0642|   >r
0643|   context
0644|   begin
0645|     dup@
0646|   while
0647|     dup@ @ r@ swap searcher ?dup
0648|     if
0649|       >r rot-drop r> rdrop exit
0650|     then
0651|     cell+
0652|   repeat drop-0 r> 0x0000 ;
```

search-wordlist and *find* are simple applications of *searcher* and *finder*, there is a difference between this Forths version of *find* and the standard one, this version of *find* does not return an execution token but a pointer in the word header which can be turned into an execution token with *cfa*.

```
0653| : search-wordlist searcher rot-drop ; ( a wid -- pwd 1 | pwd -1 | a 0 )
0654| : find ( a -- pwd 1 | pwd -1 | a 0 : find a word in the dictionary )
0655|   finder rot-drop ;
```

Numeric Input

Numeric input is handled next, converting a string into a number, which is similar to numeric output.

digit? takes a character and the current base and returns a character converted to the number it represents in the base and a boolean indicating whether or not the conversion was successful. An ASCII character set is assumed.

```
0656| : digit? ( c base -- u f )
0657|   >r [char] 0 - 9 over <
0658|   if
0659|     7 -
0660|     =bl invert and ( handle lower case, as well as upper case )
0661|     dup $A < or
0662|   then dup r> u< ;
```

>number does the work of the numeric conversion, getting a character from an input array, converting the character to a number, multiplying it by the current input base and adding in to the number being converted. It stops on the first non-numeric character.

>number accepts a string as an address-length pair which are the first two arguments, and a starting number for the number conversion (which is usually zero). *>number* returns a string containing the unconverted characters, if any, as well as the converted number.

>number operates on unsigned double cell values, not single cell values.

```
0663| : >number ( ud b u -- ud b u : convert string to number )
0664|   begin
```

```
0665|     ( get next character )
0666|     2dup 2>r drop c@ base@ digit?
0667|     0= if                                  ( d char )
0668|       drop                                 ( d char -- d )
0669|       2r>                                  ( restore string )
0670|       nop exit                             ( ..exit )
0671|     then                                   ( d char )
0672|     swap base@ um* drop rot base@ um* d+   ( accumulate digit )
0673|     2r>                                    ( restore string )
0674|     +string dup0=                          ( advance string and test for end )
0675|   until ;
```

negative? and *base?* should be thought of as working in conjunction with *>number* only. Numbers can have certain prefixes which change the interpretation of the number, for example a prefix of - means the number is negative, a *$* prefix means the number is hexadecimal regardless of the current input base, and a *#* prefix (which is a less common prefix) means the number is decimal. *negative?* handles the negative case, and could potentially be used as standalone word, however using *base?* comes with caveats, it changes the current base if one of the base changing prefixes is encountered, *>number* is careful to restore the base back to what it was when it uses *base?*.

```
h: ?exit if rdrop then ;

h: negative? ( b u -- b u t : is >number negative? )
  [char] - ' nop fallthrough;
h: base-match ( b u c xt -- b u t )
   2>r string@ r> = if +string r> execute [-1] exit then rdrop 0x0000 ;

h: base? ( b u -- b u  )
  [char] $ ' hex     base-match ?exit
  [char] # ' decimal base-match drop ;
```

```
0676| h: negative? ( b u -- b u t : is >number negative? )
0677|    string@ [char] - = if +string [-1] exit then 0x0000 ;
0678| h: base? ( b u -- )
0679|   string@ [char] $ = if +string hex     exit then ( $hex )
0680|   string@ [char] # = if +string decimal exit then ; ( #decimal )
```

>number is a generic word, but awkward to use, *number?* does some processing of the results to *>number* and handles other input processing expected for numbers. For example numbers can be prefixed by *$* to specify they are in hex, or *#* for decimal, regardless of the current base. The decimal point is handled for double cell input. The minus sign, -, can be used as a prefix for all of these different formats to specify that the number is negative. The negative prefix must come before any base specifier. *dpl* is used to store where the decimal place occurred in the input.

```
0681| h: number? ( b u -- d f : is number? )
0682|   [-1] dpl !
0683|   radix     >r
0684|   negative? >r
0685|   base?
0686|   0 -rot 0 -rot
0687|   >number
0688|   string@ [char] . = if +string
0689|     dup>r >number nip if 0 rdrop else r> dpl ! [-1] then
0690|   else
0691|     nip 0=
0692|   then
0693|   r> if >r dnegate r> then
0694|   r> base! ;
```

Parsing

After a line of text has been fetched the line needs to be tokenized into space delimited words, which is as complex as parsing gets in Forth. Technically Forth has no fixed grammar as any Forth word is free to parse the following input stream however it likes, but it is rare that to deviate from the norm and words which do complex processing are highly discouraged.

```
0695| h: -trailing ( b u -- b u : remove trailing spaces )
0696|   for
0697|     aft =bl over r@ + c@ <
0698|       if r> 1+ exit then
0699|     then
0700|   next 0x0000 ;
0701| h: lookfor ( b u c xt -- b u : skip until *xt* test succeeds )
0702|   swap >r -rot
0703|   begin
0704|     dup
0705|   while
0706|     string@ r@ - r@ =bl = 4 pick execute
0707|     if rdrop rot-drop exit then
0708|     +string
0709|   repeat rdrop rot-drop ;
0710| h: no-match if 0> exit then 0<> ; ( n f -- t )
0711| h: match no-match invert ;        ( n f -- t )
0712| h: parser ( b u c -- b u delta )
0713|   >r over r> swap 2>r
0714|   r@ ' no-match lookfor 2dup
0715|   r> ' match    lookfor swap r> - >r - r> 1+ ;
0716| : parse ( c -- b u ; <string> )
0717|    >r tib in@ + #tib @ in@ - r@ parser >in +!
0718|    r> =bl = if -trailing then 0 max ;
0719| : ) ; immediate ( -- : do nothing )
0720| :  ( [char] ) parse 2drop ; immediate \ ) ( parse until matching paren )
0721| : .( [char] ) parse type ; ( print out text until matching parenthesis )
0722| : \ #tib @ in! ; immediate ( comment until new line )
0723| h: ?length dup word-length u> if $13 -throw exit then ;
0724| : word 1depth parse ?length here pack$ ; ( c -- a ; <string> )
0725| : token =bl word ;                        ( -- a )
0726| : char token count drop c@ ;             ( -- c; <string> )
```

The Interpreter

The interpreter consists of two main words, *literal* and *interpret*. Other useful words are also defined, such as *,* or *immediate*, but they are relatively trivial in comparison.

interpret takes a pointer to a counted string and interprets the results, with this word we have the main interpreter. It searches for words, executes them or compiles them if found, or if it is not a word it attempts to convert it to a number, if this fails an error is signaled.

interpret also handles single cell and double cell conversion and compilation. It does not know what to do with numbers directly, instead it executes the execution token stored in *<literal>*, which contains *(literal)*. *(literal)* has two different behaviors depending on the execution state, in compiling mode the number is compiled into the dictionary with *literal*.

literal takes a number and compiles a number literal into the dictionary which when run will push the number provided to it. Numbers below $7FFF can be compiled to a single instruction, numbers between $FFFF and $8000 are compiled as the bitwise inversion of the number followed by and invert instruction. Single cell values take between 1-2 instructions to represent and double cell values between 2 and 4 instructions.

As said, the other words a pretty straight forward and behave the same as you would expect in most other Forths, *,* writes a value in the next available location in the dictionary, *c,* does the same but for character sized values (potentially making the dictionary unaligned). They both check there is enough room left in the dictionary to proceed.

compile, is needed as this is a subroutine threaded Forth, execution tokens should be handed to *compile,* and not straight to *,* which would be ideal. This is because addresses need to be converted into a call instruction before they are written into the dictionary. On other Forths you might get away with using *,*.

immediate, *compile-only* and *smudge* work by setting bits in the word header. *immediate* and *compile-only* should be familiar by now, they set bits in the *PWD* field of a words header. *smudge* works by toggling a bit in the name fields length byte, which by virtue of how searching works means that the word will not be found in the dictionary any more. *smudge* is *not* used to hide the word being currently defined until the matching *;* is met, however that is it's most common use in other Forths if it is defined. It hides the latest define word.

```
0727| h: ?dictionary dup $3F00 u> if 8 -throw exit then ;
0728| : , here dup cell+ ?dictionary cp! ! ; ( u -- : store *u* in dictionary )
0729| : c, here ?dictionary c! cp 1+! ;      ( c -- : store *c* in the dictionary )
0730| h: doLit 0x8000 or , ;                 ( n+ -- : compile literal )
0731| : literal ( n -- : write a literal into the dictionary )
0732|   dup 0x8000 and ( n > $7FFF ? )
0733|   if
0734|     invert doLit =invert , exit ( store inversion of n the invert it )
0735|   then
0736|   doLit ; compile-only immediate ( turn into literal, write into dictionary )
0737| h: make-callable chars $4000 or ; ( cfa -- instruction )
0738| : compile, make-callable , ; ( cfa -- : compile a code field address )
0739| h: $compile dup inline? if cfa @ , exit then cfa compile, ; ( pwd -- )
0740| h: not-found source type $D -throw ; ( -- : throw 'word not found' )
0741| h: ?compile dup compile-only? if source type $E -throw exit then ;
0742| : (literal) state@ if postpone literal exit then ; ( u -- u | )
0743| : interpret ( ??? a -- ??? : The command/compiler loop )
0744|   \ dup count type space ( <- for tracing the parser )
0745|   find ?dup if
0746|     state@
0747|     if
0748|       0> if cfa execute exit then \ <- immediate word are executed
0749|       $compile exit               \ <- compiling word are...compiled.
0750|     then
0751|     drop ?compile     \ <- check it's not a compile only word word
0752|     cfa execute exit  \ <- if its not, execute it, then exit *interpreter*
0753|   then
0754|   \ not a word
0755|   dup>r count number? if rdrop \ it's a number!
0756|     dpl @ 0< if \ <- dpl will -1 if its a single cell number
0757|        drop     \ drop high cell from 'number?' for single cell output
0758|     else        \ <- dpl is not -1, it's a double cell number
0759|        state@ if swap then
0760|        <literal> @execute \ <literal> is executed twice if it's a double
0761|     then
0762|     <literal> @execute exit
0763|   then
0764|   r> not-found ; \ not a word or number, it's an error!
```

NB. *compile* only works for words, instructions, and numbers below $8000

```
0765| : compile  r> dup@ , cell+ >r ; compile-only ( --:Compile next compiled word )
0766| : immediate $4000 last fallthrough; ( -- : previous word immediate )
0767| h: toggle tuck @ xor swap! ;        ( u a -- : xor value at addr with u )
0768| ( : compile-only 0x8000 last toggle ; )
0769| ( : smudge last fallthrough; )
0770| h: (smudge) nfa $80 swap toggle ; ( pwd -- )
```

Strings

The string word set is quite small, there are words already defined for manipulating strings such *c@*, and *count*, but they are not exclusively used for strings. These woulds will allow string literals to be embedded within word definitions.

Forth uses counted strings, at least traditionally, which contain the string length as the first byte of the string. This limits the string length to 255 characters, which is enough for our small Forth but is quite limiting.

More modern Forths either use NUL terminated strings, or larger counts for their counted strings. Both methods have trade-offs. NUL terminated strings allow for arbitrary lengths, and are often used by programs written in C, or built upon the C runtime, along with their libraries. NUL terminated strings cannot hold binary data however, and have an overhead for string length related operations.

Using a larger count prefix obviously allows for larger strings, but it is not standard and new words would have to be written, or new conventions followed, when dealing with these strings. For example the *count* word can be used on the entire string if the string size is a single byte in size. Another complication is how big should the length prefix be? 16, 32, or 64-bit? This might depend on the intended use, the preferences of the programmer, or what is most natural on the platform.

Another complication in modern string handling is UTF-8, and other character encoding schemes, which is something to be aware of, but not a subject we will go in to.

The issues described only talk about problems with Forths representation of strings, nothing has even been said about the difficulty of using them for string heavy applications!

Only a few string specific words will be defined, for compiling string literals into a word definition, and for returning an address to a compiled string. There are two things the string words will have to do; parse a string from the input stream and compile the string and an action into the dictionary. The action will be more complicated than it might seem as the string literal will be compiled into a word definition, like code is, so the action will have also manipulate the return stack so the string literal is jumped over.

```
0771| h: count+ count + ;
0772| h: do$ r> r@ r> count+ aligned >r swap >r ; ( -- a )
0773| h: string-literal do$ nop ; ( -- a : do string NB. nop to fool optimizer )
0774| h: .string do$ print ; ( -- : print string  )
```

To allow the meta-compilers version of *.“* and *$“* to work we will need to populate two variables in the metacompiler with the correct actions.

```
0775| [t] .string        tdoPrintString meta!
0776| [t] string-literal tdoStringLit   meta!
0777| h: parse-string [char] " word count+ cp! ; ( -- )
0778| ( <string>, --, Run: -- b )
0779| : $"  compile string-literal parse-string ; immediate compile-only
0780| : ."  compile .string parse-string ; immediate compile-only ( <string>, -- )
0781| : abort [-1] [-1] yield? ;                                    ( -- )
0782| h: ?abort swap if print cr abort then drop ;                  ( u a -- )
0783| h: (abort) do$ ?abort ;                                       ( -- )
0784| : abort" compile (abort) parse-string ; immediate compile-only ( u -- )
```

Evaluator

With the following few words extra words we will have a working Forth interpreter, capable of reading in a line of text and executing it. More words will need to be defined to make the system usable, as well as some house keeping.

quit is the name for the word which implements the interpreter loop, which is an odd name for its function, but this is what it is traditionally called. *quit* calls a few minor functions:

- *<ok>*, which is the execution vector for the Forth prompt, its contents are executed by *eval*.
- *(ok)* defines the default prompt, it only prints *ok* if we are in command mode.
- *[* and *]* for changing the interpreter state to command and compile mode respectively. Note that *[* must be an immediate word.

- *preset* which resets the input line variables
- *?error* handles an error condition from *throw*, it forms the error handling part of the error handler of last resort. It prints out the non-zero error number that occurred and attempts to return the interpreter into a sensible state.
- *?depth* checks for a stack underflow
- *eval* tokenizes the current input line which *quit* has fetched and calls *interpret* on each token until there is no more. Any errors that occur within *interpreter* or *eval* will be caught by *quit*.

quit will be called from the boot word later on.

```
0785| h: preset tib-start #tib cell+ ! 0 in! id zero ;  ( -- : reset input )
0786| : ] [-1] state ! ;                                ( -- : compile mode )
0787| : [   state zero ; immediate                      ( -- : command mode )
0788| h: ?error ( n -- : perform actions on error )
0789|   ?dup if
0790|     .             ( print error number )
0791|     [char] ? emit ( print '?' )
0792|     cr            ( and terminate the line )
0793|     sp0 cells sp! ( empty the stack )
0794|     preset        ( reset I/O streams )
0795|     postpone [    ( back into interpret mode )
0796|     exit
0797|   then ;
0798| h: (ok) command? if ."  ok  " cr exit then ;  ( -- )
0799| ( : ok <ok> @execute ; )
0800| h: ?depth sp@ sp0 u< if 4 -throw exit then ;  ( u -- : depth check )
0801| h: eval ( -- )
0802|   begin
0803|     token dup c@
0804|   while
0805|     interpret ?depth
0806|   repeat drop <ok> @execute ;
```

quit can now be defined, it sets up the input line variables, sets the interpreter into command mode, enters an infinite loop it does not exit from, and within that loop it reads in a line of input, evaluates it and processes any errors that occur, if any.

```
0807| ( : @echo source type cr ; ( -- : can be used to monitor input )
0808| : quit preset [ begin query ( @echo ) ' eval catch ?error again ; ( -- )
```

evaluate is used to evaluate a string of text as if it were typed in by the programmer. It takes an address and its length, this is used in the word *load*. It needs two new words apart from *eval*, which are *get-input* and *set-input*, these two words are used to retrieve and set the input input stream, which *evaluate* needs to manipulate. *evaluate* saves the current input stream specification and changes it to the new string, restoring to what it was before the evaluation took place. It is careful to catch errors and always perform the restore, any errors that thrown are re-thrown after the input is restored.

```
0809| h: get-input source in@ source-id <ok> @ ;    ( -- n1...n5 )
0810| h: set-input <ok> ! id ! in! #tib 2! ;   ( n1...n5 -- )
0811| : evaluate ( a u -- )
0812|   get-input 2>r 2>r >r
0813|   0 [-1] 0 set-input
0814|   ' eval catch
0815|   r> 2r> 2r> set-input
0816|   throw ;
```

I/O Control

The I/O control section is a relic from eForth that is not really needed in a hosted Forth, at least one where the terminal emulator used handles things like line editing. It is left in here so it can be quickly be added back in if this Forth were to be ported to an embed environment, one in which communications with the Forth took place over a UART.

Open and reading from different files is also not needed, it is handled by the virtual machine.

```
0817| h: io! preset fallthrough;  ( -- : initialize I/O )
0818| h: console ' rx? <key> ! ' tx! <emit> ! fallthrough;
0819| h: hand ' (ok)  ( ' drop <-- was emit )  ( ' ktap ) fallthrough;
0820| h: xio  ' accept <expect> ! ( <tap> ! ) ( <echo> ! ) <ok> ! ;
0821| ( h: pace 11 emit ; )
0822| ( : file ' pace ' drop ' ktap xio ; )
```

Control Structures and Defining words

Companions the creator seeks, not corpses, not herds and believers. Fellow creators the creator seeks – those who write new values on new tablets. Companions the creator seeks, and fellow harvesters; for everything about him is ripe for the harvest. - Friedrich Nietzsche

The next set of words defines relating to control structures and defining new words, along with some basic error checking for the control structures (which is known as 'compiler security' in the Forth community). Some of the words defined here are quite complex, even if their definitions are only a few lines long.

Control structure words include *if*, *else*, *then*, *begin*, *again*, *until*, *for*, *next*, *recurse* and *tail*. These words are all immediate words and are also compile only, most words visible in the target dictionary in this section have these properties.

New control structure words can be defined by the user quite easily, for example a word called *unless* can be made which executes a clause if the test provided is false, which is the opposite of *if*. This is one of the many unique features of Forth, that the language itself can be extended and customized. This is possible in languages like *lisp*, but not in *C*, to do the same in *C* would require the modification of the *C* compiler and recompilation. In *forth* (and *lisp*) it is a natural part of the language.

Using our *unless* example:

```
: unless ' 0= compile, postpone if ; immediate
```

We this newly defined control structure like so:

```
: x unless 99 . cr then ;
0 x ( prints *99* )
1 x ( prints nothing )
```

Not only are the control structures defined in this section, but so are several defining words. These Nietzschean words create new values and words in the dictionary. These words include *:*, and *;*, as well as the more complex *create* and *does>*. To define a new word they need to parse the next word in input stream, so they take over from interpreter loop from one token. They then have differing actions on what they do with this token.

: and *;* should need no explanation as to what they do by now, only the *how* needs to be elucidated. *:* parses the next word in the input stream and puts the *state* variable into compile mode, the *interpret* word will pick this state change up when the word after the one just read in. *:* also makes sure the dictionary is aligned before compilation takes place, it prints out a warning if a word is redefined, compiles the word header for the new word and pushes a magic constant value onto the stack which *;* will pull off. Apart from the word header it does no compilation itself, this is left for the interpreter to do. *;* checks for the magic number *:* left on the stack and throws an error if this is not found. It compiles an *exit* instruction into the dictionary as the final instruction for the word being defined and finally puts the interpreter back into command mode - to do this it needs to be an immediate word. These words also need set the last definition variable and *;* links the new word into the dictionary making it visible.

create and *does>* are often used as pairs, *create* can be used alone, but *does>* should only be used on words that have been make with *create*. *create* defines a new word with a default action of pushes the address of the next location available after the current definition, more memory can be allocated after this with *allot*, *create* can therefore be used to make new data structures. However the default behavior is quite simple, it would be more useful if we could change it, or extend it. This is what *does>* allows us to do.

```
0823| h: ?check ( magic-number -- : check for magic number on the stack )
0824|    magic <> if $16 -throw exit then ;
0825| h: ?unique ( a -- a : print a message if a word definition is not unique )
0826|   dup last @ searcher
0827|   if
```

```
0828|     ( source type )
0829|     space
0830|     2drop last-def @ nfa print  ."  redefined " cr exit
0831|   then ;
0832| h: ?nul ( b -- : check for zero length strings )
0833|    count 0= if $A -throw exit then 1- ;
0834| h: find-token token find fallthrough; ( -- pwd,  <string> )
0835| h: ?not-found 0= if not-found exit then ; ( t -- )
0836| h: find-cfa find-token cfa ;                          ( -- xt, <string> )
0837| : ' find-cfa state@ if postpone literal exit then ; immediate
0838| : [compile] find-cfa compile, ; immediate compile-only  ( --, <string> )
0839| : [char] char postpone literal ; immediate compile-only ( --, <string> )
0840| ( h: ?quit command? if $38 -throw exit then ; )
0841| : ; ( ?quit ) ?check =exit , postpone [ fallthrough; immediate compile-only
0842| h: get-current! ?dup if get-current ! exit then ; ( -- wid )
0843| : : align here dup last-def ! ( "name", -- colon-sys )
0844|     last , token ?nul ?unique count+ cp! magic ] ;
0845| : begin here  ; immediate compile-only      ( -- a )
0846| : until chars $2000 or , ; immediate compile-only  ( a -- )
0847| : again chars , ; immediate compile-only ( a -- )
0848| h: here-0 here 0x0000 ;
0849| h: >mark here-0 postpone again ;
0850| : if here-0 postpone until ; immediate compile-only
0851| ( : unless ' 0= compile, postpone if ; immediate compile-only )
0852| : then fallthrough; immediate compile-only
0853| h: >resolve here chars over @ or swap! ;
0854| : else >mark swap >resolve ; immediate compile-only
0855| : while postpone if ; immediate compile-only
0856| : repeat swap postpone again postpone then ; immediate compile-only
0857| h: last-cfa last-def @ cfa ;  ( -- u )
0858| : recurse last-cfa compile, ; immediate compile-only
0859| ( : tail last-cfa postpone again ; immediate compile-only )
0860| : create postpone : drop compile doVar get-current ! postpone [ ;
0861| : >body cell+ ; ( a -- a )
0862| h: doDoes r> chars here chars last-cfa dup cell+ doLit ! , ;
0863| : does> compile doDoes nop ; immediate compile-only
0864| : variable create 0 , ;
0865| : constant create ' doConst make-callable here cell- ! , ;
0866| : :noname here-0 magic ] ;
0867| : for =>r , here ; immediate compile-only
0868| : next compile doNext , ; immediate compile-only
0869| : aft drop >mark postpone begin swap ; immediate compile-only
0870| : hide find-token (smudge) ; ( --, <string> : hide word by name )
0871| ( : name? find-token nfa ; )
```

DOER/MAKE

doer and *make* which are useful for implementing deferred execution and for making words with vectored execution, found in Thinking Forth. They are not needed for the meta-compiler nor for any of the example programs so they are left commented out. They are useful for reference however.

```
: doer create =exit last-cfa ! =exit ,  ;
: make ( "name1", "name2", -- : make name1 do name2 )
 find-cfa find-cfa make-callable
 state@
 if
   postpone literal postpone literal compile ! nop exit
 then
 swap! ; immediate
```

doer creates a new word which by default does nothing. *make* is a state aware word that parses two names, a word created with *doer* which is the first name, and then the name of another word which will become the new action that the word does.

do loops

This Forth does not define the 'do...loop' wordset, but it is possible to add them in if needed:

```
h: (do) r@ swap rot >r >r cell+ >r ; compile-only ( hi lo -- index )
: do compile (do) 0 , here ; compile-only immediate ( hi lo -- )
h: (leave) rdrop rdrop rdrop ; compile-only
: leave compile (leave) nop ; compile-only immediate
h: (loop)
   r> r> 1+ r> 2dupxor if
    >r >r @ >r exit
   then >r 1- >r cell+ >r ; compile-only
h: (unloop) r> rdrop rdrop rdrop >r ; compile-only
: unloop compile (unloop) nop ; compile-only immediate
h: (?do)
  2dupxor if r@ swap rot >r >r cell+ >r exit then 2drop ; compile-only
: ?do compile (?do) 0 , here ; compile-only immediate ( hi lo -- )
: loop  compile (loop) dup , compile (unloop) cell- here chars ( -- )
    swap! ; compile-only immediate
h: (+loop)
   r> swap r> r> 2dup - >r
   2 pick r@ + r@ xor 0< 0=
   3 pick r> xor 0< 0= or if
    >r + >r @ >r exit
   then >r >r drop cell+ >r ; compile-only
: +loop ( n -- ) compile (+loop) dup , compile
  (unloop) cell- here chars swap! ; compile-only immediate
h: (i)  2r> tuck 2>r nop ; compile-only ( -- index )
: i  compile (i) nop ; compile-only immediate ( -- index )
```

These define the standard *do*-loop words.

Tracing

The Virtual Machine has options (or has bits reserved in the VM header) for enabling tracing. Tracing output is quite voluminous when the virtual machine does it. It is possible to add tracing functionality to the Forth interpreter at a higher level as well, but sometimes it is useful to have a low level instruction by instruction view. To enable an easy way to trace word execution the word *trace* is defined, this parses a word name and executes it, turning tracing on just before it executed and turning it off when it returns. The VM prints out; the Program Counter, Instruction, Top Of Stack register, Variable and Return Stack Pointers. This does not give a full view of what is going on, but it helps. If more information is needed it is always possible to edit the VM's C source and recompile it.

```
0872| h: trace-execute vm-options ! >r ; ( u xt -- )
0873| : trace ( "name" -- : trace a word )
0874|   find-cfa vm-options @ dup>r 3 or trace-execute r> vm-options ! ;
```

Annotated example output of running *trace* on *+* is shown:

```
2 2 trace +              ( '+' has the address $5c )
[  7b9 6147   5c  8  3 ] ( 'trace-execute' executing '>r' )
[  7ba 601c    2  9  2 ] ( 'trace-execute' returns to '+' )
[   2e 653f    2  8  2 ] ( '+' executing as instruction 653f )
[  7c6 628d    4  7  1 ] ( 'r>' in 'trace', Top of Stack = 4 from 2+2 )
[  7c7 800c    0  6  2 ] ( push address of VM options )
[  7c8 641f    c  6  3 ] ( disable tracing with '!' )
 ok                      ( trace complete and turned off )
```

Vocabulary Words

The vocabulary word set should already be well understood, if the metacompiler has been. The vocabulary word set is how Forth organizes words and controls visibility of words.

```
0875| h: find-empty-cell 0 fallthrough; ( a -- )
0876| h: find-cell >r begin dup@ r@ <> while cell+ repeat rdrop ; ( u a -- a )
```

get-order retrieves the current search order by pushing a list of *wid* values onto the stack *n* long which can be modified before being passed to *set-order*.

```
0877| : get-order ( -- widn ... wid1 n : get the current search order )
0878|   context
0879|   find-empty-cell
0880|   dup cell- swap
0881|   context - chars dup>r 1- s>d if $32 -throw exit then
0882|   for aft dup@ swap cell- then next @ r> ;
```

xchange is used to place the next few words into the root vocabulary, it will be set back to the default shortly.

```
0883| xchange _forth-wordlist root-voc
0884| : forth-wordlist _forth-wordlist ;
```

set-order does the opposite of *get-order*, it can be used to set the current search order. It has a special case however, when the number of word lists is set to *-1* it loads the minimal word set, which consists of very few words, namely:

```
 words forth set-order forth-wordlist
```

These words can in turn be used to load the default word list, and find out what words are in the current search order. This is useful to create new words which consist of the minimal set of words to do a job, and once that has been achieved go back to the normal Forth environment.

set-order also checks that the number of word lists is not too high, there should not be more than *#vocs* word lists given to *set-order*, and exception is thrown otherwise.

set-order does not set the definitions word list however, that is the word list which new definitions are added to, *definitions* and *set-current* can be used to do this.

```
0885| ( @warning recursion without using recurse is used )
0886| : set-order ( widn ... wid1 n -- : set the current search order )
0887|   dup [-1] = if drop root-voc 1 set-order exit then
0888|   dup #vocs > if $31 -throw exit then
0889|   context swap for aft tuck ! cell+ then next zero ;
```

forth loads the root vocabulary containing the minimal word set, and also loads the normal *forth* vocabulary.

```
0890| : forth root-voc forth-wordlist 2 set-order ; ( -- )
```

words is used to display all of the words defined in the dictionary in the current search order. *.words* handles a specific word list, and *not-hidden?* looks at an individual words header to check whether a word is to be shown or now, a bit is set in the words name fields counted string byte, in the highest bit of the count byte, if the word is to be hidden or shown.

```
0891| h: not-hidden? nfa c@ $80 and 0= ; ( pwd -- )
0892| h: .words
0893|     begin
0894|       ?dup
0895|     while dup not-hidden? if .id then @address repeat cr ;
0896| : words
0897|   get-order begin ?dup while swap dup cr u. colon-space @ .words 1- repeat ;
```

xchange is used to place words back into the forth word list again.

```
0898| xchange root-voc _forth-wordlist
```

The *previous*/*also*/*only* mechanism for Forth word list manipulation are quite awkward, the only one word keeping is *only* which is just a shortcut for '-1 set-order'. The more powerful words *-order* and *+order* are much easier to use.

Of note is the idiom 'wid -order wid +order' which reorders the search order so *wid* now has a higher priority than the other word lists. This can be done before *definitions* is used to make *wid* the current definitions word list as well.

Another idiom is:

```
only forth definitions
```

Which restores the search order to the default and sets the *forth-wordlist* vocabulary as the one which new definitions are added to.

Anonymous word lists can be created with the *anonymous* word, this is useful to make programs which only a few words exported to another word list, making things less cluttered.

For example:

```
only forth
anonymous definitions
: x 99 . ;
: y 88 . ;
: reorder dup -order +order ;
forth-wordlist reorder definitions
: z x y cr ;
only forth definitions
```

In this example new words *x*, *y* and *reorder* are created and added to an anonymous wordlist, *z* uses words *x* and *y* from this word this and then the words in the anonymous word lists are hidden, but not the newly defined word *z*, that is available in the default Forth word list.

```
0899| ( : previous get-order swap drop 1- set-order ; ( -- )
0900| ( : also get-order over swap 1+ set-order ;     ( wid -- )
0901| : only [-1] set-order ;                         ( -- )
0902| ( : order get-order for aft . then next cr ;    ( -- )
0903| ( : anonymous get-order 1+ here 1 cells allot swap set-order ; ( -- )
0904| : definitions context @ set-current ;           ( -- )
0905| h: (order)                                      ( w wid*n n -- wid*n w n )
0906|   dup if
0907|     1- swap >r (order) over r@ xor
0908|     if
0909|       1+ r> -rot exit
0910|     then rdrop
0911|   then ;
0912| : -order get-order (order) nip set-order ;             ( wid -- )
0913| : +order dup>r -order get-order r> swap 1+ set-order ; ( wid -- )
```

editor is a word which loads the editor vocabulary, which will be defined later, it requires *+order* to work.

```
0914| : editor editor-voc +order ;                   ( -- )
0915| ( : assembler root-voc assembler-voc 2 set-order ;     ( -- )
0916| ( : ;code assembler ; immediate                         ( -- )
0917| ( : code postpone : assembler ;                         ( -- )
0918| ( xchange _forth-wordlist assembler-voc )
0919| ( : end-code forth postpone ; ; immediate ( -- )
0920| ( xchange assembler-voc _forth-wordlist )
```

Block Word Set

The block word set abstracts out how access to mass storage works in just a handful of words. The main word is *block*, with the words *update*, *flush* and the variable *blk* also integral to the working of the block word set. All of the other words can be implemented upon these.

Block storage is an outdated, but simple, method of accessing mass storage that demands little from the hardware or the system it is implemented under, just that data can be transfered from memory to disk somehow. It has no requirements that there be a file system, which is perfect for embedded devices as well as upon the microcomputers it originated on.

A 'Forth block' is 1024 byte long buffer which is backed by a mass storage device, which we will refer to as *disk*. Very compact programs can be written that have their data stored persistently. Source can and data can be stored in blocks and evaluated, which will be described more in the 'Block editor' section of this document.

The *block* word does most of the work. The way it is usually implemented is as follows:

1. A user provides a block number to the *block* word. The block number is checked to make sure it is valid, and an exception is thrown if it is not.
2. If the block if already loaded into a block buffer from disk, the address of the memory it is loaded into is returned.
3. If it was not, then *block* looks for a free block buffer, loads the 1024 byte section off disk into the block buffer and returns an address to that.
4. If there are no free block buffers then it looks for a block buffer that is marked as being dirty with *update* (which marks the previously loaded block as being dirty when called), then transfers that dirty block to disk. Now that there is a free block buffer, it loads the data that the user wants off of disk and returns a pointer to that, as in the previous bullet point. If none of the buffers are marked as dirty then then any one of them could be reused - they have not been marked as being modified so their contents could be retrieved off of disk if needed.
5. Under all cases, before the address of the loaded block has been returned, the variable *blk* is updated to contain the latest loaded block.

This word does a lot, but is quite simple to use. It implements a simple cache where data is transfered back to disk only if needed, and multiple sections of memory from disk can be loaded into memory at the same time. The mechanism by which this happens is entirely hidden from the user.

This Forth implements *block* in a slightly different way. The entire virtual machine image is loaded at start up, and can be saved back to disk (or small sections of it) with the *(save)* instruction. *update* marks the entire image as needing to be saved back to disk, whilst *flush* calls *(save)*. *block* then only has to check the block number is within range, and return a pointer to the block number multiplied by the size of a block - so this means that this version of *block* is just an index into main memory. This is similar to how *colorForth* implements its block word.

```
0921| : update [-1] block-dirty ! ; ( -- )
0922| h: blk-@ blk @ ;              ( -- k : retrieve current loaded block )
0923| h: +block blk-@ + ;           ( -- )
0924| : save 0 here (save) throw ;  ( -- : save blocks )
0925| : flush block-dirty @ if 0 [-1] (save) throw exit then ; ( -- )
0926| : block ( k -- a )
0927|   1depth
0928|   dup $3F u> if $23 -throw exit then
0929|   dup blk !
0930|   $A lshift ( <-- b/buf * ) ;
```

The block word set has the following additional words, which augment the set nicely, they are *list*, *load* and *thru*. The *list* word is used for displaying the contents of a block, it does this by splitting the block into 16 lines each, 64 characters long. *load* evaluates a given block, and *thru* evaluates a range of blocks.

This is how source code was stored and evaluated during the microcomputer era, as opposed to storing the source in named byte stream oriented files as is common nowadays.

It is more difficult, but possible, to store and edit source code in this manner, but it requires that the programmer(s) follow certain conventions when editing blocks, both in how programs are split up, and how they are formatted.

A block shown with *list* might look like the following:

```
  ------------------------------------------------------------
 0|( Simple Math Routines 31/12/1989 RJH 3/4               #20  ) |
 1|                                                              |
 2|: square dup * ; ( u -- u )                                   |
 3|: sum-of-squares square swap square + ; ( u u -- u )          |
 4|: even 1 and 0= ; ( u -- b )                                  |
 5|: odd even 0= ;   ( u -- b )                                  |
 6|: log >r 0 swap ( u base -- u )                               |
 7|  begin swap 1+ swap r@ / dup 0= until                        |
 8|  drop 1- rdrop ;                                             |
 9|: signum ( n -- -1 | 0 | 1 )                                  |
10|    dup 0> if drop 1 exit then                                |
11|        0< if     -1 exit then                                |
12|        0 ;                                                   |
13|: >< dup 8 rshift swap 8 lshift or ; ( u -- u : swap bytes )  |
14|: #digits dup 0= if 1+ exit then base@ log 1+ ;               |
15|                                                              |
  ------------------------------------------------------------
```

Longer comments for a source code block were stored in a 'shadow block', which is only enforced by convention. One possible convention is to store the source in even numbered blocks, and the comments in the odd numbered blocks.

By storing a comment in the first line of a block a word called *index* could be used to make a table of contents all of the blocks available on disk, *index* simply displays the first line of each block within a block range. The common theme is that convention is key to successfully using blocks.

An example of using blocks to store data is how some old Forths used to store their error messages. Instead of storing strings of error messages in memory (either in RAM or ROM) they were stored in block storage, with one error message per line. The error messages were stored in order, and a word *message* is used to convert an error code like *-4* to its error string (the numbers and what they mean are standardized, there is a table in the appendix), which was then printed out. The full list of the standardized error messages can be stored in four blocks, and form a neat and easy to use database.

```
0931| h: c/l* ( c/l * ) 6 lshift ;            ( u -- u )
0932| h: c/l/ ( c/l / ) 6 rshift ;            ( u -- u )
0933| h: line c/l* swap block + c/l ;          ( k u -- a u )
0934| h: loadline line evaluate ;              ( k u -- )
0935| : load 0 l/b-1 for 2dup 2>r loadline 2r> 1+ next 2drop ; ( k -- )
0936| h: pipe [char] | emit ;                       ( -- )
0937| ( h: .line line -trailing $type ;        ( k u -- )
0938| h: .border 3 spaces c/l [char] - nchars cr ; ( -- )
0939| h: #line dup 2 u.r ;                     ( u -- u : print line number )
0940| ( : thru over- for dup load 1+ next drop ; ( k1 k2 -- )
0941| h: blank =bl fill ;                      ( b u -- )
0942| ( : message l/b extract .line cr ;       ( u -- )
0943| h: retrieve block drop ;                 ( k -- )
0944| : list                                   ( k -- )
0945|   dup retrieve
0946|   cr
0947|   .border
0948|   0 begin
0949|     dup l/b <
0950|   while
0951|     2dup #line pipe line $type pipe cr 1+
0952|   repeat .border 2drop ;
```

t: index (k1 k2 – : show titles for block k1 to k2) over- cr for dup 5u.r space pipe space dup 0 .line cr 1+ next drop ;

Booting

We are now nearing the end of this tutorial, after the boot sequence word set has been completed we will have a working Forth system. The boot sequence consists of getting the Forth system into a known working state, checking for corruption in

the image, and printing out a welcome message. This behaviour can be changed if needed.

The boot sequence is as follows:

1. The virtual machine starts execution at address 0 which will be set to point to the word *boot-sequence*.
2. The word *boot-sequence* is executed, which will run the word *cold* to perform the system setup.
3. The word *cold* checks that the image length and CRC in the image header match the values it calculates, zeros blocks of memory, and initializes the systems I/O.
4. *boot-sequence* continues execution by executing the execution token stored in the variable *<boot>*. This is set to *normal-running* by default.
5. *normal-running* prints out the welcome message by calling the word *hi*, and then entering the Forth Read-Evaluate Loop, known as *quit*. This should not normally return.
6. If the function returns, *bye* is called, halting the virtual machine.

The boot sequence is modifiable by the user by either writing an execution token to the *<boot>* variable, or by writing to a jump to a word into memory location zero, if the image is saved, the next time it is run execution will take place at the new location.

It should be noted that the self-check routine, *bist*, disables checking in generated images by manipulating the word header once the check has succeeded. This is because after the system boots up the image is going to change in RAM soon after *cold* has finished, this could be organized better so only sections of memory that do not (or should not change) get checked, one way this could be achieved is by locating all variables in specific sections, and checking only code. This has the advantage that the system image could be stored in Read Only Memory (ROM), which would facilitate porting the system to low memory systems, such as a microcontroller as they only have a few kilobytes of RAM to work with. The virtual machine primitives for load and store could be changed so that reads get mapped to RAM or ROM based on their address, and writes to RAM also (with attempted writes to ROM throwing an exception). The image is only 6KiB in size, and only a few kilobytes are needed in RAM for a working forth image, perhaps as little as ~2-4 KiB.

A few other interesting things could be done during the execution of *cold*, the image could be compressed by the metacompiler and the boot sequence could decompress it (which would require the decompressor to be one of the first things to run). Experimenting with various schemes it appears Adaptive Huffman Coding produces the best results, followed by lzss.c). The schemes are also simple and small enough (both in size and memory requirements) that they could be implemented in Forth.

Another possibility is to obfuscate the image by exclusive or'ing it with a value to frustrate the reserve engineering of binaries, or even to encrypt it and ask for the decryption key on startup.

Obfuscation and compression are more difficult to implement than a simple CRC check and require a more careful organization and control of the boot sequence than is provided. It is possible to make *turn-key* applications that start execution at an arbitrary point (call *turn-key* because all the user has to do is 'turn the key' and the program is ready to go) by setting the boot variable to the desired word and saving the image with *save*.

```
0953| h: check-header? header-options @ first-bit 0= ; ( -- t )
0954| h: disable-check 1 header-options toggle ;       ( -- )
```

bist checks the length field in the header matches *here* and that the CRC in the header matches the CRC it calculates in the image, it has to zero the CRC field out first.

```
0955| h: bist ( -- u : built in self test )
0956|   check-header? if 0x0000 exit then       ( is checking disabled? Success? )
0957|   header-length @ here xor if 2 exit then ( length check )
0958|   header-crc @ header-crc zero            ( retrieve and zero CRC )
0959|   0 here crc xor if 3 exit then           ( check CRC )
0960|   disable-check 0x0000 ;                  ( disable check, success )
```

cold performs the self check, and exits if it fails. It then goes on to zero memory, set the initial value of *blk*, set the I/O vectors to sensible values, set the vocabularies to the default search order and reset the variable stack.

```
0961| h: cold ( -- : performs a cold boot  )
0962|    bist ?dup if negate dup yield? exit then
```

$10 retrieve x

```
0963|    $10 block b/buf 0 fill
0964|    $12 retrieve io!
0965|    forth sp0 cells sp!
0966|    ( rp0 cells rp! )
0967|    <boot> @execute bye ;
```

hi prints out the welcome message, the version number, sets the numeric base to decimal, and prints out the amount of memory used and free.

normal-running is the boot word, that is *<boot>* is set to.

```
0968| h: hi hex cr ." eFORTH v" ver 0 u.r cr decimal here . .free cr ;       ( -- )
0969| h: normal-running hi quit ;                                  ( -- : boot word )
```

See : The Forth Disassembler

This section defines *see*, the Forth disassembler. This is meant only as a rough debugging tool and not meant to reproduce the exact source. The output format is not defined in any standard, it is just meant to be useful for debugging purposes and perhaps for programmers to quickly determine how a word works.

There are many ways *see* could be improved but each minor improvement would make the disassembler more complicated and prone to breaking, the disassembler as it stands is probably the best trade-off in terms of complexity and utility. Despite it being small it is still quite useful and does a lot, for example it attempts to look up words in the current word search orders, it prints out the type of instruction, the addresses of instructions and the raw instruction. It also gives information about if a word is compile only or immediate.

One of the most difficult thing is determining the end of a word, the disassembler uses a crude method by disassembling everything from the start of the word definition in the dictionary to the start of the next one in the same word list - this means that *:noname* definitions (of which the metacompiler makes many in the target) and words not in the same word list that are between the word to be disassembled and the next word in the same word as the word being disassembled will be disassembled in their entirety, their headers and all. As such the output should be viewed only as an aid and as not being definitive and absolutely correct.

Three words will be defined that do most of the complex work of the disassembler, they are *validate*, *search-for-cfa* and *name*, which should be understood together. To make a disassembler we will need a way to look up compiled calls to words and retrieve their name, this is what *name* does.

search-for-cfa searches a given word list for a CFA (or a Code Field Address, the address of the executable section of a Forth word), it does this by following the dictionary linked list assuming that if a CFA points in between the current word and the previous word in the list then this is word that we are looking for. It uses *validate* to attempt to check this, if we have found a candidate word address by calling the word *cfa* on it, it should move to the same address we provided.

```
0970| h: validate over cfa <> if drop-0 exit then nfa ; ( pwd cfa -- nfa | 0 )
0971| h: search-for-cfa ( wid cfa -- nfa : search for CFA in a word list )
0972|   address cells >r
0973|   begin
0974|     dup
0975|   while
0976|     address dup @address over r@ -rot within
0977|     if dup @address r@ validate ?dup if rdrop nip exit then then
0978|     address @
0979|   repeat rdrop ;
0980| h: name ( cwf -- a | 0 )
0981|    >r
0982|    get-order
0983|    begin
0984|      dup
0985|    while
0986|      swap r@ search-for-cfa ?dup if >r 1- ndrop r> rdrop exit then
0987|    1- repeat rdrop ;
0988| h: .name name ?dup 0= if $" ?" then print ;
```

```
0989| h: ?instruction ( i m e -- i 0 | e -1 )
0990|   >r over-and r> tuck = if nip [-1] exit then drop-0 ;
0991| h: .instruction ( u -- u )
0992|    0x8000  0x8000 ?instruction if [char] L emit exit then
0993|    $6000   $6000  ?instruction if [char] A emit exit then
0994|    $6000   $4000  ?instruction if [char] C emit exit then
0995|    $6000   $2000  ?instruction if [char] Z emit exit then
0996|    drop-0 [char] B emit ;
0997| h: decompile ( u -- : decompile instruction )
0998|    dup .instruction $BFFF and if drop exit then space .name ;
0999| h: decompiler ( previous current -- : decompile starting at address )
1000|   >r
1001|   begin dup r@ u< while
1002|     dup 5u.r colon-space
1003|     dup@
1004|     dup 5u.r space decompile cr cell+
1005|   repeat rdrop drop ;


: disassembler ( a u -- : disassemble a range of code )
 1 rshift for aft
    dup   5u.r [char] : emit
    dup @ 5u.r space
    dup @ decompile cr cell+
 then next drop ;
```

see is the Forth disassembler, it takes a word and (attempts) to turn it back into readable Forth source code. The disassembler is only a few hundred bytes in size, which is a testament to the brevity achievable with Forth.

If the word *see* was good enough we could potentially dispense with the source code entirely: the entire dictionary could be disassembled and saved to disk, modified, then recompiled yielding a modified Forth. Although comments would not be present, meaning this would be more of an intellectual exercise than of any utility.

One way of improving the output would be to move the assembler in the metacompiler to the target image, the same assembler words like *d-1* and *#t/n* could then be used to disassemble ALU instructions, which are currently only displayed in their raw format.

```
1006| : see ( --, <string> : decompile a word )
1007|   token finder ?not-found
1008|   swap      2dup= if drop here then >r
1009|   cr colon-space .id dup cr
1010|   cfa r> decompiler space [char] ; emit
1011|   dup compile-only? if ."  compile-only " then
1012|   dup inline?       if ."  inline "       then
1013|       immediate?    if ."  immediate "    then cr ;
```

A few useful utility words will be added next, which are not strictly necessary but are useful. Those are *.s* for examining the contents of the variable stack, and *dump* for showing the contents of a section of memory

The *.s* word must be careful not to alter that variable stack whilst trying to print it out. It uses the word *pick* to achieve this, otherwise there is nothing special about this word, and it is very useful for debugging code interactively to see what its stack effects are.

The dump keyword is fairly useful for the implementer so that they can use Forth to debug if the compilation is working, or if a new word is producing the correct assembly. It can also be used as a utility to export binary sections of memory as text.

The programmer might want to edit the *dump* word to customize its output, the addition of the *dc+* word is one way it could be extended, which is commented out below. Like *dm+*, *dc+* operates on a single line to be displayed, however *dc+* decompiles the memory into human readable instructions instead of numbers, unfortunately the lines it produces are too long.

Normally the word *dump* outputs the memory contents in hexadecimal, however a design decision was taken to output the contents of memory in what the current numeric output base is instead. This makes the word more flexible, more consistent and shorter, than it otherwise would be as the current output base would have to be saved and then restored.

```
1014| : .s cr depth for aft r@ pick . then next ."  <sp" ;          ( -- )
1015| h: dm+ chars for aft dup@ space 5u.r cell+ then next ;   ( a u -- a )
1016| ( h: dc+ chars for aft dup@ space decompile cell+ then next ; ( a u -- a )
1017| : dump ( a u -- )
1018|   $10 + \ align up by dump-width
1019|   4 rshift ( <-- equivalent to "dump-width /" )
1020|   for
1021|     aft
1022|       cr dump-width 2dup
1023|       over 5u.r colon-space
1024|       dm+ ( dump-width dc+ ) \ <-- dc+ is optional
1025|       -rot
1026|       2 spaces $type
1027|     then
1028|   next drop ;
```

The standard Forth dictionary is now complete, but the variables containing the word list need to be updated a final time. The next section implements the block editor, which is in the *editor* word set. Their are two variables that need updating, *__forth-wordlist*, a vocabulary we have already encountered. An *current*, which contains a pointer to a word list, this word list is the one new definitions (defined by *:*, or *create*) are added to. It will be set to *__forth-wordlist* so new definitions are added to the default vocabulary.

```
1029| [last]              [t] _forth-wordlist t!
1030| [t] _forth-wordlist [t] current         t!
```

Block Editor

This block editor is an excellent example of a Forth application; it is small, terse, and uses the facilities already built into Forth to do all of the heavy lifting, specifically vocabularies, the block word set and the text interpreter.

Forth blocks used to be the canonical way of storing both source code and data within a Forth system, it is a simply way of abstracting out how mass storage works and worked well on the microcomputers available in the 1980s. With the rise of computers with a more capable operating systems the Block Block Word Set Word Set fell out of favour, being replaced instead by the File Access Word Set, allowing named files to be accessed as a byte stream.

To keep things simple this editor uses the block word set and in typical Forth fashion simplifies the problem to the extreme, whilst also sacrificing usability and functionally - the block editor allows for the editing of programs but it is more difficult and more limited than traditional editors. It has no spell checking, or syntax highlighting, and does little in the way of error checking. But it is very small, compact, easy to understand, and if needed could be extended.

The way this editor works is by replacing the current search order with a set of words that implement text editing on the currently loaded block, as well as managing what block is loaded. The defined words are short, often just a single letter long.

The act of editing text is simplified as well, instead of keeping track of variable width lines of text and files, a single block (1024 characters) is divided up into 16 lines, each 64 characters in length. This is the essence of Forth, radically simplifying the problem from all possible angels; the algorithms used, the software itself and where possible the hardware. Not every task can be approached this way, nor would everyone be happy with the results, the editor being presented more as a curiosity than anything else.

We have a way of loading and saving data from disks (the *block*, *update* and *flush* words) as well as a way of viewing the data in a block (the *list* word) and evaluating the text within a block (with the *load* word). The variable *blk* is also of use as it holds the latest block we have retrieved from disk. By defining a new word set we can skip the part of reading in a parsing commands and numbers, we can use text interpreter and line oriented input to do the work for us, as discussed.

Only one extra word is actually need given the words we already have, one which can destructively replace a line starting at a given column in the currently loaded block. All of the other commands are simple derivations of existing words. This word is called *ia*, short for 'insert at', which takes two numeric arguments (starting line as the first, and column as the second) and reads all text on the line after the *ia* and places it at the specified line/column.

The command description and their definitions are the best descriptions of how this editor works. Try to use the word set interactively to get a feel for it:

A1	A2	Command	Description
#2	#1	ia	insert text into column #1 on line #2
	#1	i	insert text into column 0 on line #1
	#1	l	load block number #1
	#1	k	blank line number #1
		z	blank currently loaded block
		v	redisplay currently loaded block
		q	remove editor word set from search order
		n	load next block
		p	load previous block
		s	save changes to disk
		x	evaluate block

An example command session might be:

Command Sequence	Description
editor	add the editor word set to search order
$20 l v	load block $20 (hex) and display it
x	blank block $20
0 i .(Hello, World) cr	Put ".(Hello, World) cr" on line 0
1 i 2 2 + . cr	Put "2 2 + . cr" on line 1
v	list block $20 again
x	evaluate block $20
s	save contents
q	unload block word set

This editor is based on block editor originally written for Retro Forth.

```
1031| 0 tlast meta!
1032| h: [block] blk-@ block ;       ( k -- a : loaded block address )
1033| h: [check] dup b/buf c/l/ u>= if $18 -throw exit then ;
1034| h: [line] [check] c/l* [block] + ; ( u -- a )
1035| : l retrieve ;                 ( k -- : Load a block )
1036| : v blk-@ list ;               ( -- : View current block )
1037| : n   1  +block l v ;          ( -- : load and list Next block )
1038| : p [-1] +block l v ;          ( -- : load and list Previous block )
1039| : k [line] c/l blank ;         ( u -- : delete/Kill line )
1040| : z [block] b/buf blank ;      ( -- : Zero/blank loaded block )
1041| : s update flush ;             ( -- : Save changes to disk )
1042| : q editor-voc -order ;        ( -- : Quit editor )
1043| : x q blk-@ load editor ;      ( -- : eXecute/evaluate block )
1044| : ia c/l* + [block] + source drop in@ + ( u u -- Insert At )
1045|    swap source nip in@ - cmove postpone \ ;
1046| : i 0 swap ia ;                ( u -- : Insert line )
1047| ( : u update ;                 ( -- : set block set as dirty )
1048| ( : w words ; )
1049| ( : yank pad c/l ; )
1050| ( : c [line] yank >r swap r> cmove ; )
1051| ( : y [line] yank cmove ; )
1052| ( : ct swap y c ; )
1053| ( : xa [line] c/l evaluate ; )
1054| ( : sw 2dup y [line] swap [line] swap c/l cmove c ; )
1055| [last] [t] editor-voc t! 0 tlast meta!
```

Final Touches

```
1056| there [t] cp t!
```

```
1057| [t] (literal) [v] <literal> t!   ( set literal execution vector )
1058| [t] cold 2/ 0 t!                 ( set starting word )
1059| [t] normal-running [v] <boot> t!
1060| there    $D tcells t! \ Set Length First!
1061| checksum $E tcells t! \ Calculate image CRC
1062| finished
1063| bye
```

Appendix

The Virtual Machine

The Virtual Machine is a 16-bit stack machine based on the H2 CPU, itself a derivative of the J1 CPU, but adapted for use on a computer.

Its instruction set allows for a fairly dense encoding, and the project goal is to be fairly small whilst still being useful. It is small enough that is should be easily understandable with little explanation, and it is hackable and extensible by modification of the source code.

Virtual Machine Memory Map

There is 64KiB of memory available to the Forth virtual machine, of which only the first 16KiB can contain program instructions (or more accurately branch and call locations can only be in the first 16KiB of memory). The virtual places little restrictions on what goes where, however the eForth image maps things out in the following way:

Block	Region
0	Image header
0 - 15	Program Storage
16	User Data
17	Variable Stack
18 - 62	User data
63	Return Stack

The virtual machine uses the first six cells for special purposes, apart from the division between program/data and data only sections this is the only restriction that the *virtual machine* places on memory.

The first six locations are used for:

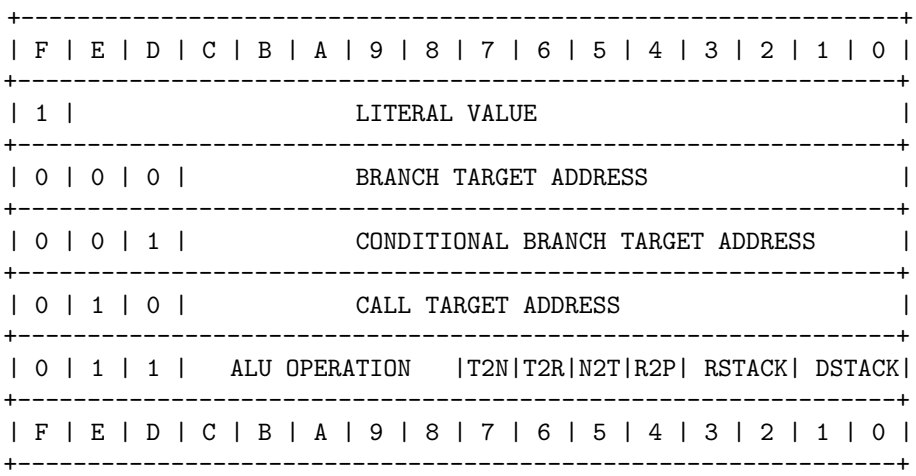
Address	Use
$0	Initial Program Counter (PC) value and reset vector
$2	Initial Top of Stack Register value
$4	Initial Return Stack Register value (grows downwards)
$6	Initial Variable Stack Register value (grows upwards)
$8	Instruction exception vector (trap handler)
$A	Virtual Machine memory in cells, if used, else $8000 assumed
$C	Virtual Machine memory in cells, if used, else $8000 assumed
$E	Virtual Machine Options, various uses
$10	Virtual Machine Reserved For Future Use
$12	Virtual Machine Reserved For Future Use

The eForth image maps things in the following way. The variable stack starts at the beginning of block 17 and grows upwards, the return stack starts at the end of block 63 and grows downward. The trap handler is set to a call a word called *-throw*, defined as ": -throw negate throw ;". The virtual machine image size is assumed to be the maximum allowable value of "$8000". The virtual machine memory size is specified in cells, not bytes, "$8000" is the maximum number of cells that a 16-bit value can address if the lowest bit is used to specify a byte.

Instruction Set Encoding

For a detailed look at how the instructions are encoded the source code is the definitive guide, available in the file forth.c, or consult the reference implementation in the Appendix.

A quick overview:




```
T   : Top of data stack
N   : Next on data stack
PC  : Program Counter

LITERAL VALUES : push a value onto the data stack
CONDITIONAL    : BRANCHS pop and test the T
CALLS          : PC+1 onto the return stack

T2N : Move T to N
T2R : Move T to top of return stack
N2T : Move the new value of T (or D) to N
R2P : Move top of return stack to PC

RSTACK and DSTACK are signed values (twos compliment) that are
the stack delta (the amount to increment or decrement the stack
by for their respective stacks: return and data)
```

ALU Operations

The ALU can be programmed to do the following operations on an ALU instruction, some operations trap on error (U/MOD, /MOD).

#	Mnemonic	Description
0	T	Top of Stack
1	N	Copy T to N
2	R	Top of return stack
3	T@	Load from address
4	NtoT	Store to address
5	T+N	Double cell addition
6	T*N	Double cell multiply
7	T&N	Bitwise AND
8	TorN	Bitwise OR
9	T^N	Bitwise XOR
10	~T	Bitwise Inversion
11	T--	Decrement

12	T=0	Equal to zero
13	T=N	Equality test
14	Nu<T	Unsigned comparison
15	N<T	Signed comparison
16	NrshiftT	Logical Right Shift
17	NlshiftT	Logical Left Shift
18	SP@	Depth of stack
19	RP@	R Stack Depth
20	SP!	Set Stack Depth
21	RP!	Set R Stack Depth
22	SAVE	Save Image
23	TX	Get byte
24	RX	Send byte
25	UM/MOD	um/mod
26	/MOD	/mod
27	BYE	Return

Encoding of Forth Words

Many Forth words can be encoded directly in the instruction set, some of the ALU operations have extra stack and register effects as well, which although would be difficult to achieve in hardware is easy enough to do in software.

Word	Mnemonic	T2N	T2R	N2T	R2P	RP	SP
dup	T	T2N					+1
over	N	T2N					+1
invert	~T						
um+	T+N						
+	T+N			N2T			-1
um*	T*N						
*	T*N			N2T			-1
swap	N	T2N					
nip	T						-1
drop	N						-1
exit	T				R2P	-1	
>r	N		T2R			1	-1
r>	R	T2N				-1	1
r@	R	T2N					1
@	T@						
!	NtoT						-1
rshift	NrshiftT						-1
lshift	NlshiftT						-1
=	T=N						-1
u<	Nu<T						-1
<	N<						-1
and	T&N						-1
xor	T^N						-1
or	T|N						-1
sp@	SP@	T2N					1
sp!	SP!						
1-	T--						
rp@	RP@	T2N					1
rp!	RP!						-1
0=	T=0						
nop	T						
(bye)	BYE						
rx?	RX	T2N					1
tx!	TX			N2T			-1
(save)	SAVE						-1

um/mod	UM/MOD	T2N					
/mod	/MOD	T2N					
/	/MOD						-1
mod	/MOD			N2T			-1
rdrop	T					-1	

Interaction

The outside world can be interacted with in two ways, with single character input and output, or by saving the current Forth image. The interaction is performed by three instructions.

eForth

The interpreter is based on eForth by C. H. Ting, with some modifications to the model.

eForth Memory model

The eForth model imposes extra semantics to certain areas of memory, along with the virtual machine.

Address	Block	Meaning
$0000	0	Initial Program Counter (PC)
$0002	0	Initial Top of Stack Register
$0004	0	Initial Return Stack Register
$0006	0	Initial Var. Stack Register
$0008	0	Instruction exception vector
$000A	0	VM Options bits
$000C	0	VM memory in cells
$000E	0	VM memory Reserved
$0010	0	VM memory Reserved
$0012-$001C	0	eForth Header
$001E-EOD	0-?	The dictionary
EOD-$3FFF	?-15	Compilation and Numeric Output
$4000	16	Interpreter variable storage
$4280	16	Pad area
$4400	17	Start of variable stack
$4800-$FBFF	18-63	Empty blocks for user data
$FC00-$FFFF	0	Return stack block

Error Codes

This is a list of Error codes, not all of which are used by the application.

Hex	Dec	Message
FFFF	-1	ABORT
FFFE	-2	ABORT"
FFFD	-3	stack overflow
FFFC	-4	stack underflow
FFFB	-5	return stack overflow
FFFA	-6	return stack underflow
FFF9	-7	do-loops nested too deeply during execution
FFF8	-8	dictionary overflow
FFF7	-9	invalid memory address
FFF6	-10	division by zero
FFF5	-11	result out of range

FFF4	-12	argument type mismatch
FFF3	-13	undefined word
FFF2	-14	interpreting a compile-only word
FFF1	-15	invalid FORGET
FFF0	-16	attempt to use zero-length string as a name
FFEF	-17	pictured numeric output string overflow
FFEE	-18	parsed string overflow
FFED	-19	definition name too long
FFEC	-20	write to a read-only location
FFEB	-21	unsupported operation
FFEA	-22	control structure mismatch
FFE9	-23	address alignment exception
FFE8	-24	invalid numeric argument
FFE7	-25	return stack imbalance
FFE6	-26	loop parameters unavailable
FFE5	-27	invalid recursion
FFE4	-28	user interrupt
FFE3	-29	compiler nesting
FFE2	-30	obsolescent feature
FFE1	-31	>BODY used on non-CREATEd definition
FFE0	-32	invalid name argument (e.g., TO xxx)
FFDF	-33	block read exception
FFDE	-34	block write exception
FFDD	-35	invalid block number
FFDC	-36	invalid file position
FFDB	-37	file I/O exception
FFDA	-38	non-existent file
FFD9	-39	unexpected end of file
FFD8	-40	invalid BASE for floating point conversion
FFD7	-41	loss of precision
FFD6	-42	floating-point divide by zero
FFD5	-43	floating-point result out of range
FFD4	-44	floating-point stack overflow
FFD3	-45	floating-point stack underflow
FFD2	-46	floating-point invalid argument
FFD1	-47	compilation word list deleted
FFD0	-48	invalid POSTPONE
FFCF	-49	search-order overflow
FFCE	-50	search-order underflow
FFCD	-51	compilation word list changed
FFCC	-52	control-flow stack overflow
FFCB	-53	exception stack overflow
FFCA	-54	floating-point underflow
FFC9	-55	floating-point unidentified fault
FFC8	-56	QUIT
FFC7	-57	exception in sending or receiving a character
FFC6	-58	[IF], [ELSE], or [THEN] exception

http://www.forth200x.org/throw-iors.html

Hex	Dec	Message
FFC5	-59	ALLOCATE
FFC4	-60	FREE
FFC3	-61	RESIZE
FFC2	-62	CLOSE-FILE
FFC1	-63	CREATE-FILE
FFC0	-64	DELETE-FILE
FFBF	-65	FILE-POSITION
FFBE	-66	FILE-SIZE

FFBD	-67	FILE-STATUS
FFBC	-68	FLUSH-FILE
FFBB	-69	OPEN-FILE
FFBA	-70	READ-FILE
FFB9	-71	READ-LINE
FFB8	-72	RENAME-FILE
FFB7	-73	REPOSITION-FILE
FFB6	-74	RESIZE-FILE
FFB5	-75	WRITE-FILE
FFB4	-76	WRITE-LINE

Virtual Machine Implementation in C

```
/* Embed Forth Virtual Machine, Richard James Howe, 2017-2018, MIT License */
#include "embed.h"
#include <assert.h>
#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

#define MAX(X, Y) ((X) < (Y) ? (Y) : (X))
typedef struct forth_t { uint16_t m[32768]; } forth_t;

void embed_die(const char *fmt, ...)
{
    va_list arg;
    va_start(arg, fmt);
    vfprintf(stderr, fmt, arg);
    va_end(arg);
    fputc('\n', stderr);
    exit(EXIT_FAILURE);
}

FILE *embed_fopen_or_die(const char *file, const char *mode)
{
    FILE *h = NULL;
    errno = 0;
    assert(file && mode);
    if(!(h = fopen(file, mode)))
        embed_die("file open %s (mode %s) failed: %s", file, mode, strerror(errno));
    return h;
}

forth_t *embed_new(void)
{
    forth_t *h = NULL;
    if(!(h = calloc(1, sizeof(*h))))
        embed_die("allocation (of %u) failed", (unsigned)sizeof(*h));
    return h;
}

static int save(forth_t *h, const char *name, const size_t start, const size_t length)
{
    assert(h && ((length - start) <= length));
    if(!name)
        return -1;
```

```
    FILE *out = embed_fopen_or_die(name, "wb");
    int r = 0;
    for(size_t i = start; i < length; i++)
        if(fputc(h->m[i]&255, out) < 0 || fputc(h->m[i]>>8, out) < 0)
            r = -1;
    fclose(out);
    return r;
}

static size_t embed_cells(forth_t const * const h) { assert(h); return h->m[5]; } /* count in cells, not bytes
forth_t *embed_copy(forth_t const * const h)      { assert(h); return memcpy(embed_new(), h, sizeof(*h)); }
int      embed_save(forth_t *h, const char *name) { return save(h, name, 0, embed_cells(h)); }
size_t   embed_length(forth_t const * const h)    { return embed_cells(h) * sizeof(h->m[0]); }
void     embed_free(forth_t *h)                   { assert(h); memset(h, 0, sizeof(*h)); free(h); }
char    *embed_get_core(forth_t *h)               { assert(h); return (char*)h->m; }

int embed_load(forth_t *h, const char *name)
{
    assert(h && name);
    FILE *input = embed_fopen_or_die(name, "rb");
    long r = 0, c1 = 0, c2 = 0;
    for(size_t i = 0; i < MAX(64, embed_cells(h)); i++) {
        if((c1 = fgetc(input)) < 0 || (c2 = fgetc(input)) < 0) {
            r = i;
            break;
        }
        h->m[i] = ((c1 & 0xffu)) | ((c2 & 0xffu) << 8u);
    }
    fclose(input);
    return r < 64 ? -1 : 0; /* minimum size checks, 128 bytes */
}

int embed_forth(forth_t *h, FILE *in, FILE *out, const char *block)
{
    assert(h && in && out);
    static const uint16_t delta[] = { 0, 1, -2, -1 };
    const uint16_t l = embed_cells(h);
    uint16_t * const m = h->m;
    uint16_t pc = m[0], t = m[1], rp = m[2], sp = m[3], r = 0;
    for(uint32_t d;;) {
        const uint16_t instruction = m[pc++];

        if(m[6] & 1) /* trace on */
            fprintf(m[6] & 2 ? out : stderr, "[ %4x %4x %4x %2x %2x ]\n", pc-1, instruction, t, m[2]-rp, sp-m[
        if((r = -!(sp < l && rp < l && pc < l)))
            goto finished;

        if(0x8000 & instruction) { /* literal */
            m[++sp] = t;
            t       = instruction & 0x7FFF;
        } else if ((0xE000 & instruction) == 0x6000) { /* ALU */
            uint16_t n = m[sp], T = t;
            pc = instruction & 0x10 ? m[rp] >> 1 : pc;

            switch((instruction >> 8u) & 0x1f) {
            case  1: T = n;                    break;
            case  2: T = m[rp];                break;
            case  3: T = m[(t>>1)%l];          break;
            case  4: m[(t>>1)%l] = n; T = m[--sp]; break;
            case  5: d = (uint32_t)t + n; T = d >> 16; m[sp] = d; n = d; break;
```

```
            case  6: d = (uint32_t)t * n; T = d >> 16; m[sp] = d; n = d; break;
            case  7: T &= n;                   break;
            case  8: T |= n;                   break;
            case  9: T ^= n;                   break;
            case 10: T = ~t;                   break;
            case 11: T--;                      break;
            case 12: T = -(t == 0);            break;
            case 13: T = -(t == n);            break;
            case 14: T = -(n < t);             break;
            case 15: T = -((int16_t)n < (int16_t)t); break;
            case 16: T = n >> t;               break;
            case 17: T = n << t;               break;
            case 18: T = sp << 1;              break;
            case 19: T = rp << 1;              break;
            case 20: sp = t >> 1;              break;
            case 21: rp = t >> 1; T = n;       break;
            case 22: T = save(h, block, n>>1, ((uint32_t)T+1)>>1); break;
            case 23: T = fputc(t, out);        break;
            case 24: T = fgetc(in); n = -1;    break; /* n = blocking status */
            case 25: if(t) { d = m[--sp]|((uint32_t)n<<16); T=d/t; t=d%t; n=t; } else { pc=4; T=10; } break;
            case 26: if(t) { T=(int16_t)n/t; t=(int16_t)n%t; n=t; } else { pc=4; T=10; } break;
            case 27: if(n) { m[sp] = 0; r = t; goto finished; } break;
            }
            sp += delta[ instruction       & 0x3];
            rp -= delta[(instruction >> 2) & 0x3];
            if(instruction & 0x20)
                T = n;
            if(instruction & 0x40)
                m[rp] = t;
            if(instruction & 0x80)
                m[sp] = t;
            t = T;
        } else if (0x4000 & instruction) { /* call */
            m[--rp] = pc << 1;
            pc      = instruction & 0x1FFF;
        } else if (0x2000 & instruction) { /* 0branch */
            pc = !t ? instruction & 0x1FFF : pc;
            t  = m[sp--];
        } else { /* branch */
            pc = instruction & 0x1FFF;
        }
    }
finished: m[0] = pc, m[1] = t, m[2] = rp, m[3] = sp;
    return (int16_t)r;
}

#ifdef USE_EMBED_MAIN
int main(int argc, char **argv)
{
    forth_t *h = embed_new();
    if(argc != 3 && argc != 4)
        embed_die("usage: %s in.blk out.blk [file.fth]", argv[0]);
    if(embed_load(h, argv[1]) < 0)
        embed_die("embed: load failed");
    FILE *in = argc == 3 ? stdin : embed_fopen_or_die(argv[3], "rb");
    if(embed_forth(h, in, stdout, argv[2]))
        embed_die("embed: run failed");
    return 0; /* exiting takes care of closing files, freeing memory */
}
#endif
```

ANSI Terminal Escape Sequence Word Set

The following word set implements ANSI escape codes.

```
variable <page>
variable <at-xy>

: page <page> @ execute ;   ( -- : page screen )
: at-xy <at-xy> @ execute ; ( x y -- : set cursor position )

: CSI $1b emit [char] [ emit ;
: 10u. base @ >r decimal 0 <# #s #> type r> base ! ; ( u -- )
: ansi swap CSI 10u. emit ; ( n c -- )
: (at-xy) CSI 10u. $3b emit 10u. [char] H emit ; ( x y -- )
: (page) 2 [char] J ansi 1 1 at-xy ; ( -- )
: sgr [char] m ansi ; ( -- )

1 constant red 2 constant green 4 constant blue

: color $1e + sgr ;

' (at-xy) <at-xy> !
' (page)  <page>  !
```

Usage is quite simple:

```
red color sgr  ( set terminal text color to red )
page           ( clear screen, set cursor to position 1,1 )
$8 $4 at-xy    ( set cursor to position column 8, row 4 )
0 sgr          ( reset terminal attributes, color goes back to normal )
```

Go online for more examples.

Sokoban

This is a game of Sokoban, to play, type:

```
cat sokoban.fth /dev/stdin | ./embed eforth.blk new.blk
```

On the command line. Four maps are provided, more can be found online at https://github.com/begoon/sokoban-maps, where the four maps were found.

```
\ Author:  Richard James Howe
\ License: MIT (excluding the maps)
0 <ok> !
only forth definitions hex
\ NB. 'blk' has the last block retrieved by 'block', not 'load' in this Forth

variable sokoban-wordlist
sokoban-wordlist +order definitions

\ @todo change character set
$20    constant maze
char X constant wall
char * constant boulder
char . constant off
char & constant on
```

```
char @ constant player
char ~ constant player+ ( player + off pad )
$10    constant l/b     ( lines   per block )
$40    constant c/b     ( columns per block )
     7 constant bell    ( bell character )

variable position  ( current player position )
variable moves     ( moves made by player )

( used to store rule being processed )
create rule 3 c, 0 c, 0 c, 0 c,

: n1+ swap 1+ swap ; ( n n -- n n )
: match              ( a a -- f )
  n1+ ( replace with umin of both counts? )
  count
  for aft
    count rot count rot <> if 2drop rdrop 0 exit then
  then next 2drop -1 ;

: beep bell emit ; ( -- )
: ?apply           ( a a a -- a, R: ? -- ?| )
  >r over swap match if drop r> rdrop exit then rdrop ;

: apply ( a -- a )
 $" @ "  $"  @"  ?apply
 $" @."  $"  ~"  ?apply
 $" @* " $"  @*" ?apply
 $" @*." $"  @&" ?apply
 $" @&." $"  ~&" ?apply
 $" @& " $"  ~*" ?apply
 $" ~ "  $" .@"  ?apply
 $" ~."  $" .~"  ?apply
 $" ~* " $" .@*" ?apply
 $" ~*." $" .@&" ?apply
 $" ~&." $" .~&" ?apply
 $" ~& " $" .~*" ?apply beep ;

: pack ( c0...cn b n -- )
  2dup swap c! for aft 1+ tuck c! then next drop ;

: locate ( b u c -- u f )
  >r
  begin
    ?dup
  while
    1- 2dup + c@ r@ = if nip rdrop -1 exit then
  repeat
  rdrop
  drop
  0 0 ;

: 2* 1 lshift ; ( u -- )
: relative swap c/b * + + ( $3ff and ) ; ( +x +y pos -- pos )
: +position position @ relative ; ( +x +y -- pos )
: double 2* swap 2* swap ;  ( u u -- u u )
: arena blk @ block b/buf ; ( -- b u )
: >arena arena drop + ;     ( pos -- a )
: fetch                     ( +x +y -- a a a )
  2dup   +position >arena >r
```

```
  double +position >arena r> swap
  position @ >arena -rot ;
: rule@ fetch c@ rot c@ rot c@ rot ; ( +x +y -- c c c )
: 3reverse -rot swap ;               ( 1 2 3 -- 3 2 1 )
: rule! rule@ 3reverse rule 3 pack ; ( +x +y -- )
: think 2dup rule! rule apply >r fetch r> ; ( +x +y --a a a a )
: count! count rot c! ;              ( a a -- )

\ 'act' could be made to be more elegant, but it works, it
\ handles rules of length 2 and length 3
: act ( a a a a -- )
  count swap >r 2 =
  if
     drop swap r> count! count!
  else
     3reverse r> count! count! count!
  then drop ;

: #boulders ( -- n )
   0 arena
   for aft
     dup c@ boulder = if n1+ then
     1+
   then next drop ;
: .boulders  ." BOLDERS: " #boulders u. cr ; ( -- )
: .moves    ." MOVES: " moves    @ u. cr ; ( -- )
: .help     ." WASD - MOVEMENT" cr ." H    - HELP" cr ; ( -- )
: .maze blk @ list ;                  ( -- )
: show ( page cr ) .maze .boulders .moves .help ; ( -- )
: solved? #boulders 0= ;               ( -- )
: finished? solved? if 1 throw then ; ( -- )
: instructions ;                      ( -- )
: where >r arena r> locate ;          ( c -- u f )
: player? player where 0= if drop player+ where else -1 then ;
: player! player? 0= throw position ! ; ( -- )
: start player! 0 moves ! ;           ( -- )
: .winner show cr ." SOLVED!" cr ;    ( -- )
: .quit cr ." Quitter!" cr ;          ( -- )
: finish 1 = if .winner exit then .quit ; ( n -- )
: rules think act player! ;           ( +x +y -- )
: +move 1 moves +! ;                  ( -- )
: ?ignore over <> if rdrop then ;     ( c1 c2 --, R: x -- | x )
: left  [char] a ?ignore -1  0 rules +move ; ( c -- c )
: right [char] d ?ignore  1  0 rules +move ; ( c -- c )
: up    [char] w ?ignore  0 -1 rules +move ; ( c -- c )
: down  [char] s ?ignore  0  1 rules +move ; ( c -- c )
: help  [char] h ?ignore instructions ; ( c -- c )
: end  [char] q ?ignore drop 2 throw ; ( c -- | c, R ? -- | ? )
: default drop ;  ( c -- )
: command up down left right help end default finished? ;
: maze! block drop ; ( k -- )
: input key ;        ( -- c )

sokoban-wordlist -order definitions
sokoban-wordlist +order

: sokoban ( k -- )
  maze! start
  begin
    show input ' command catch ?dup
```

```
  until finish ;

$18 maze!
only forth definitions

editor decimal z blk @
 1 i            XXXXX
 2 i            X   X
 3 i            X*  X
 4 i          XXX  *XXX
 5 i          X  *  * X
 6 i        XXX X XXX X     XXXXXX
 7 i        X   X XXX XXXXXXX  ..X
 8 i        X *  *             ..X
 9 i        XXXXX XXXX X@XXXX  ..X
10 i            X      XXX  XXXXXX
11 i            XXXXXXXX
12 i
dup 1+ l z
 1 i       XXXXXXXXXXXX
 2 i       X..  X     XXX
 3 i       X..  X *  *  X
 4 i       X..  X*XXXX  X
 5 i       X..    @ XX  X
 6 i       X..  X X  * XX
 7 i       XXXXXX XX* * X
 8 i         X *  * * * X
 9 i         X    X     X
10 i         XXXXXXXXXXXX
11 i
dup 1+ l z
 1 i               XXXXXXXX
 2 i               X     @X
 3 i               X *X* XX
 4 i               X *  *X
 5 i               XX* * X
 6 i       XXXXXXXXX * X XXX
 7 i       X....  XX *  *  X
 8 i       XX...    *  *   X
 9 i       X....  XXXXXXXXXX
10 i       XXXXXXXX
dup 1+ l z
 1 i                     XXXXXXXX
 2 i                     X  ....X
 3 i          XXXXXXXXXXX  ....X
 4 i          X    X  * *   ....X
 5 i          X ***X*  * X  ....X
 6 i          X  *     * X  ....X
 7 i          X ** X* * *XXXXXXXX
 8 i       XXXX  * X     X
 9 i       X   X XXXXXXXXX
10 i       X    *  XX
11 i       X **X** @X
12 i       X   X   XX
13 i       XXXXXXXXX
drop
q hex

.( Type '# sokoban' to play, where '#' is a block number ) cr
.( For example "$18 sokoban" ) cr
```

Follow the on screen instructions to play a game.

Conways Game of Life

A Conways Games Of Life in a few screens, adapted to this Forth:

```
0 <ok> !

\ Adapted from: http://wiki.c2.com/?ForthBlocks

only forth definitions hex
variable life-vocabulary
life-vocabulary +order definitions

   $40 constant c/b
   $10 constant l/b
c/b 1- constant c/b>
l/b 1- constant l/b>
    bl constant off
char * constant on

variable state-blk
variable life-blk
variable statep

: wrapy dup 0< if drop l/b> then dup l/b> > if drop 0 then ;
: wrapx dup 0< if drop c/b> then dup c/b> > if drop 0 then ;
: wrap  wrapy swap wrapx swap ;
: deceased? wrap c/b * + life-blk @ block + c@ off = ;
: living?  deceased? 0= ;
: (-1,-1) 2dup 1- swap 1- swap living? 1 and ;
: (0,-1)  >r 2dup 1- living? 1 and r> + ;
: (1,-1)  >r 2dup 1- swap 1+ swap living? 1 and r> + ;
: (-1,0)  >r 2dup swap 1- swap living? 1 and r> + ;
: (1,0)   >r 2dup swap 1+ swap living? 1 and r> + ;
: (-1,1)  >r 2dup 1+ swap 1- swap living? 1 and r> + ;
: (0,1)   >r 2dup 1+ living? 1 and r> + ;
: (1,1)   >r 1+ swap 1+ swap living? 1 and r> + ;
: mates (-1,-1) (0,-1) (1,-1) (-1,0) (1,0) (-1,1) (0,1) (1,1) ;
: born?  mates 3 = ;
: survives?  2dup living? -rot mates 2 = and ;
: lives?  2dup born? -rot survives? or ;        ( u u -- )
: newstate  state-blk @ block update statep ! ; ( -- )
: state!  statep @ c! 1 statep +! ;             ( c -- )
: alive  on state! ;                            ( -- )
: dead  off state! ;                            ( -- )
: cell?  2dup swap lives? if alive else dead then ; ( u u -- )
: rows   0 begin dup c/b < while cell? 1+ repeat drop ;
: iterate-block 0 begin dup l/b < while rows 1+ repeat drop ;
: generation  life-blk @ state-blk @ life-blk ! state-blk ! ;
: iterate  newstate iterate-block generation ;
: done?  key [char] q = ;                      ( -- f )
: prompt  cr ." q to quit" cr ;                ( -- )
: view (  page ) life-blk @ list prompt ;      ( -- )
: game  begin view iterate done? until ;       ( -- )

variable seed here seed !
: random seed 1 cells crc ?dup 0= if here then dup seed ! ;
```

```
: randomize ( k -- )
  block b/buf
  for aft
    random 1 and if on else off then over c! 1+
  then next
  drop ;

life-vocabulary -order definitions
life-vocabulary +order

: life life-blk ! state-blk ! game ;       ( k1 k2 -- )
: random-life $18 randomize $19 $18 life ; ( -- )

only forth definitions hex

editor $18 l z
3 i      ***
4 i      *
5 i       *
q
hex

.( Usage: $19 $18 life ) cr
.( Or 'random-life' ) cr
```

Run in the same way sokoban.fth.

Floating Point Arithmetic

Floating point implementation, and adapted version can be found in the unit tests file.

```
Vierte Dimension Vol.2, No.4 1986


    A FAST HIGH-LEVEL FLOATING POINT

        Robert F. Illyes

            ISYS
          PO Box 2516, Sta. A
          Champaign, IL 61820
          Phone: 217/359-6039

If binary normalization and rounding are used, a fast
single-precision FORTH floating point can be built with
accuracy adequate for many applications. The creation
of such high-level floating points has become of more
than academic interest with the release of the Novix
FORTH chip. The FORTH-83 floating point presented here
is accurate to 4.8 digits. Values may range from about
9E-4933 to about 5E4931. This floating point may be
used without fee provided the copyright notice and
associated information in the source are included.

FIXED VS. FLOATING POINT

FORTH programmers have traditionally favored fixed over
floating point. A fixed point application is harder to
write. The range of each value must be known and
```

considered carefully, if adequate accuracy is to be maintained and overflow avoided. But fixed point applications are much more compact and often much faster than floating point (in the absence of floating point hardware).

The debate of fixed vs. floating point is at least as old as the ENIAC, the first electronic digital computer. John von Neumann used fixed point on the ENIAC. He felt that the detailed understanding of expressions required by fixed point was desirable, and that fixed point was well worth the extra time (1).

But computers are no longer the scarce resource that they once were, and the extra programming time is often more costly than any advantages offered by fixed point. For getting the most out of the least hardware, however, fixed point will always be the technique of choice.

Fixed point arithmetic represents a real number as a ratio of integers, with an implicit divisor. This implicit divisor may be thought of as the representation of unity. If unity were represented by 300, for example, 2.5 would be represented by 750.

To multiply 2.5 by 3.5, with all values representing unity as ten, one would evaluate

$$\frac{25 * 35}{10}$$

The ten is called a scale factor, and the division by ten is called a scaling operation. This expression is obviously inefficient, requiring both a division and a multiplication to accomplish a multiplication.

Most scaling operations can, however, be eliminated by a little algebraic manipulation. In the case of the sum of two squares, for example, where A and B are fixed point integers and unity is represented by the integer U,

$$\frac{A * A}{U} + \frac{B * B}{U} \quad \text{-->} \quad \frac{(A * A)+(B * B)}{U}$$

In addition to the elimination of a division by U, the right expression is more accurate. Each division can introduce some error, and halving the number of divisions halves the worst-case error.

DECIMAL VS. BINARY NORMALIZATION

A floating point number consists of two values, an exponent and a mantissa. The mantissa may represent either an integer or a fraction. The exponent and the mantissa are related to each other in the same way as the value and power of ten in scientific notation are

related.

A mantissa is always kept as large as possible. This process is called normalization. The following illustrates the action of decimal normalization with an unsigned integer mantissa:

```
     Value       Stack representation
           4

     5 * 10         50000  0  --
           3
       * 10          7000  0  --
           3
       * 10         50000 -1  --
```

The smallest the mantissa can become is 6554. If a mantissa of 6553 is encountered, normalization requires that it be made 65530, and that the exponent be decremented. It follows that the worst-case error in representing a real number is half of 1 part in 6554, or 1 part in 13108. The error is halved because of the rounding of the real number to the nearest floating point value.

Had we been using binary normalization, the smallest mantissa would have been 32768, and the worst case error in representation would have been 1 part 65536, or 1/5 that of decimal normalization. LOG10(65536) is 4.8, the accuracy in decimal digits.

As with fixed point, scaling operations are required by floating point. With decimal normalization, this takes the form of division and multiplication by powers of ten. Unlike fixed point, no simple algebraic manipulation will partly eliminate the scale factors. Consequently there are twice as many multiplications and divisions as the floating point operators would seem to imply. Due to the presence of scaling in 73% of decimally normalized additions (2), the amount is actually somewhat over twice.

With binary normalization, by contrast, this extra multiplication effectively disappears. The scaling by a power of two can usually be handled with a single shift and some stack manipulation, all fast operations.

Though a decimally normalized floating point can be incredibly small (3), a binary normalized floating point has 1/5 the error and is about twice as fast.

It should be mentioned that the mantissa should be multiples of 2 bytes if the full speed advantage of binary normalization is to be available. Extra shifting and masking operations are necessary with odd byte counts when using the 2-byte arithmetic of FORTH.

NUMBER FORMAT AND ARITHMETIC

This floating point package uses an unsigned single

precision fraction with binary normalization, representing values from 1/2 to just under 1. The high bit of the fraction is always set.

The sign of the floating point number is carried in the high bit of the single precision exponent, The remaining 15 bits of the exponent represent a power of 2 in excess 4000 hex. The use of excess 4000 permits the calculation of the sign as an automatic outcome of exponent arithmetic in multiplication and division.

A zero floating point value is represented by both a zero fraction and a zero exponent. Any operation that produces a zero fraction also zeros the exponent.

The exponent is carried on top of the fraction , so the sign may be tested by the phrase DUP 0< and zero by the phrase DUP 0= .

The FORTH-83 Double Number Extension Word Set is required. Floating point values are used with the "2" words: 2CONSTANT, 2@, 2DUP, etc.

There is no checking for exponent overflow or underflow after arithmetic operation, nor is division by zero checked for. The rationale for this is the same as with FORTH integer arithmetic. By requiring that the user add any such tests, 1) all arithmetic isn't slowed by tests that are only sometimes needed and 2) the way in which errors are resolved may be determined by the user. The extremely large exponent range makes exponent overflow and underflow quite unlikely, of course.

All of the arithmetic is rounding. The failure to round is the equivalent of throwing a bit of accuracy away. The representational accuracy of 4.8 digits will be quickly lost without rounding arithmetic.

The following words behave like their integer namesakes:

```
F+  F-  F*  F/  F2*  F2/  FABS  FNEGATE  F<
```

Single precision integers may be floated by FLOAT, and produced from floating point by FIX and INT, which are rounding and truncating, respectively. DFLOAT floats a double precision integer.

NUMBER INPUT AND OUTPUT

Both E and F formats are supported. A few illustrations should suffice to show their usage. An underscore indicates the point at which the return key is pressed. PLACE determines the number of places to the right of the decimal point for output only.

```
12.34  F      F. _ 12.340
12.34  F      E. _ 1.234E1
.033 E -1002  E. _ 3.300E-1004
```

```
       4 PLACES

       2. F -3. F F/ F. _ -0.6667
       2. F -3. F F/ E. _ -6.6667E-1
```

F and E will correctly float any input string representing a signed double precision number. There may be as many as 9 digits to the right of the decimal point. Numbers input by E are accurate to over 4 digits. F is accurate to the full 4.8 digits if there are no digits to the right of the decimal point. Conversion is slightly less accurate with zeros to the right of the decimal point because a division by a power of ten must be added to the input conversion process.

F and E round the input value to the nearest floating point value. So a sixth digit will often allow a more accurately rounded conversion, even thought the result is only accurate to 4.8 digits. There is no advantage to including trailing zeros, however. In many floating points, this extra accuracy can only be achieved by the inconvenient procedure of entering the values as hexadecimal integers.

Only the leftmost 5 digits of the F. output are significant. F. displays values from 32767 to -32768, with up to 4 additional places to the right of the decimal point. The algorithm for F. avoids double rounding by using integer rather than floating point multiplication to scale by a power of ten. This gives as much accuracy as possible at the expense of a somewhat limited range. Since a higher limit on size would display digits that are not significant, this range limit does not seem at all undesirable.

Like E input, E. is accurate to somewhat over 4 digits. The principal source of inaccuracy is the function EXP, which calculates powers of 2.

The following extended fraction is used by EXP. It gives the square root of 2 to the x power. The result must be squared to get 2 to the x.

$$1 + \cfrac{2x}{34.668 - x - \cfrac{57828}{2001.18 + (2x)}}$$

In order to do E format I/O conversion, one must be able to evaluate the expressions

$$a = 10^{\frac{a}{\log_{10}(2)}} \quad \text{and} \quad b = 2^{b \cdot \log_{10}(2)}$$

These log expressions may be easily evaluated with great precision by applying a few fixed point techniques. First, a good rational approximation to

log10(2) is needed.

```
log10(2)     = .3010299957
4004 / 13301 = .3010299978
```

The following code will convert an integer power of ten, assumed to be on the stack, into a power of 2:

```
13301 4004 */MOD >R
FLOAT 4004 FLOAT F/ EXP
R> +
```

The first line divides the power of ten by log10(2) and pushes the quotient on the return stack. The quotient is the integer part of the power of two.

The second line finds the fractional part of the power of two by dividing the remainder by the divisor. This floating point fractional part is evaluated using EXP.

The third line adds the integer power of two into the exponent of the floating point value of the fractional part, completing the conversion.

The inverse process is used to convert a power of 2 to a power of ten.

FORTH-83 LIMITATIONS

Perhaps the most serious deficiency in the FORTH-83 with extensibility as its pre-eminent feature, it is surprisingly difficult to write standard code that will alter the processing of numeric input strings by the interpreter and compiler.

It is usually a simple matter to replace the system conversion word (usually called NUMBER) with a routine of ones choice. But there if no simple standard way of doing this. The interpreter, compiler and abort language are all interwoven, and may all have to be replaced if a standard solution is sought.

This floating point package assumes that double precision integers are generated if the numeric input string contains a period, and that a word PLACES can be written that will leave the number of digits to the right of the period. This does not seem to be guaranteed by FORTH-83, although it may be safely assumed on most systems that include double precision.

If you know how this part of your system works, you will probably want to eliminate the words E and F, and instead force floating point conversion of any input string containing a period. Double precision integers could still be generated by using a comma or other punctuation.

It is suggested that future FORTH standards include the word NUMBER, which is a vector to the current input numeric word.

It is also suggested that the Double Number Extension Wordset specification include a requirement that the interpreter and compiler be able to accept input strings specifying double precision values.

COMMENTS ON THE FOLLOWING CODE

The words ". and "- leave the ASCII values for period and minus, respectively. Replace these with whatever language you prefer for insertion of ASCII values.

The size of F+ can be considerably reduced at the expense of quite a lot of execution speed. Think twice before you simplify it.

The normalizing word NORM expects the stack value under the exponent to be a double precision signed integer. It leaves a normalized floating point number, rounding the double precision integer into the fraction.

ALIGN and RALIGN expect an integer shift count with an unsigned double precision number beneath. They leave double precision unsigned integer results. At least one shift is always performed. RALIGN rounds after alignment.

UM/ divides an unsigned double precision number by an unsigned single precision number, and rounds the single precision quotient.

ZERO forces a floating point value with a zero fraction to also have a zero exponent.

FSIGN applies the sign of the stack value under the exponent to the exponent. The double precision integer under an exponent is left unsigned.

FEXP evaluates a power of e. It is included because it is a trivial but useful application of EXP.

GET converts the next word in the input stream into a single precision signed integer.

REFERENCES

1. Von Neumann, J., John von Neumann Collected Works, vol. 5, p.113.

2. Knuth, D. K., The Art of Computer Programming, second edition, vol. 2, pp. 238,9.

3. Tracy, M., Zen Floating Point, 1984 FORML Conference Proceedings, pp. 33-35.

```
( FORTH-83 FLOATING POINT.
  ---------------------------------
  COPYRIGHT 1985 BY ROBERT F. ILLYES
```

```
    PO BOX 2516, STA. A
    CHAMPAIGN, IL 61820
    PHONE: 217/826-2734  )     HEX

: ZERO  OVER 0= IF DROP 0 THEN ;
: FNEGATE 8000 XOR ZERO ;
: FABS  7FFF AND ;
: NORM  >R 2DUP OR
    IF BEGIN DUP 0< NOT
       WHILE D2* R> 1- >R
       REPEAT SWAP 0< - ?DUP
       IF R> ELSE 8000 R> 1+ THEN
    ELSE R> DROP THEN ;

: F2*   1+ ZERO ;
: F*    ROT + 4000 - >R UM* R> NORM ;
: FSQ   2DUP F* ;

: F2/   1- ZERO ;
: UM/   DUP >R UM/MOD SWAP R>
    OVER 2* 1+ U< SWAP 0< OR - ;
: F/    ROT SWAP - 4000 + >R
    0 ROT ROT 2DUP U<
    IF   UM/ R> ZERO
    ELSE >R D2/ FABS R> UM/ R> 1+
    THEN ;

: ALIGN 20 MIN 0 DO D2/ LOOP ;
: RALIGN 1- ?DUP IF ALIGN THEN
    1 0 D+ D2/ ;
: FSIGN FABS OVER 0< IF >R DNEGATE R>
    8000 OR THEN ;

: F+    ROT 2DUP >R >R FABS SWAP FABS -
    DUP IF DUP 0<
        IF   ROT SWAP  NEGATE
             R> R> SWAP >R >R
        THEN 0 SWAP RALIGN
    THEN SWAP 0 R> R@ XOR 0<
    IF   R@ 0< IF 2SWAP THEN D-
         R> FSIGN ROT SWAP NORM
    ELSE D+ IF 1+ 2/ 8000 OR R> 1+
        ELSE R> THEN THEN ;

: F-    FNEGATE F+ ;
: F<    F- 0< SWAP DROP ;

( FLOATING POINT INPUT/OUTPUT ) DECIMAL

CREATE PL 3 , HERE  ,001 , ,   ,010 , ,
      ,100 , ,           1,000 , ,
    10,000 , ,         100,000 , ,
     1,000,000 , ,      10,000,000 , ,
   100,000,000 , ,   1,000,000,000 , ,

: TENS  2* 2* LITERAL + 2@ ;     HEX
: PLACES PL ! ;
: SHIFTS FABS 4010 - DUP 0< NOT
    ABORT" TOO BIG" NEGATE ;
: F#    >R PL @ TENS DROP UM* R> SHIFTS
```

```
    RALIGN PL @ ?DUP IF 0 DO # LOOP
    ". HOLD THEN #S ROT SIGN ;
: TUCK  SWAP OVER ;
: F.    TUCK <# F# #> TYPE SPACE ;
: DFLOAT 4020 FSIGN NORM ;
: F     DFLOAT POINT TENS DFLOAT F/ ;
: FCONSTANT F 2CONSTANT ;

: FLOAT DUP 0< DFLOAT ;
: -+    DROP SWAP 0< IF NEGATE THEN ;
: FIX   TUCK 0 SWAP SHIFTS RALIGN -+ ;
: INT   TUCK 0 SWAP SHIFTS  ALIGN -+ ;

1.      FCONSTANT ONE DECIMAL
34.6680 FCONSTANT X1
-57828. FCONSTANT X2
2001.18 FCONSTANT X3
1.4427  FCONSTANT X4

: EXP   2DUP INT DUP >R FLOAT F-
    F2* X2 2OVER FSQ X3 F+ F/
    2OVER F2/ F-     X1 F+ F/
    ONE F+ FSQ R> + ;
: FEXP  X4 F* EXP ;
: GET   BL WORD DUP 1+ C@ "- = TUCK -
    0 0 ROT CONVERT DROP -+ ;
: E     F GET >R R@ ABS 13301 4004 */MOD
    >R FLOAT 4004 FLOAT F/ EXP R> +
    R> 0< IF F/ ELSE F* THEN ;

: E.    TUCK FABS 16384 TUCK -
    4004 13301 */MOD >R
    FLOAT 4004 FLOAT F/ EXP F*
    2DUP ONE F<
    IF 10 FLOAT F* R> 1- >R THEN
    <# R@ ABS 0 #S R> SIGN 2DROP
    "E HOLD F# #>     TYPE SPACE ;
```

This code was found at: ftp://ftp.taygeta.com/pub/Forth/.