**VFORTH Description**

VFORTH is a 32 bit Forth derivative. Its initial platform is a simple 32 bit virtual machine.

Like OCF (and like Chuck Moore’s 1970 FORTH version !), all words are executed. No exceptions. Some words, when executed, do compilation though.

Words have two forms, which are distinguished between using a bar prefix.

DUP Executes a DUP

DUP| Executes the code that compiles a DUP. Call to DUP or inline code.

RFORTH code is driven by either Color (like Color FORTH) or Syntax as follows. Color FORTHs colour scheme is kept, but this is not Color Forth. Code is compiled to ‘object code’.

Note that when producing the standard words (e.g. DUP) you cannot always use code that just compiles the primitive. The reason is because some words – R> >R and ; access or update the return stack and as such when called as a “word-ised” version will have the return call from that initial call in the way.

**Executable Words**

Directly executed words are in underline and/or Yellow.

In the example above DUP or DUP will execute DUP. A numeric constant will be pushed on the stack.

**Compiled Words**

Words that are compiled into the dictionary are in normal font and/or Green.

In the example above DUP or DUP will execute DUP| rather than DUP. This will normally either compile a call to DUP, or generate the inline code for DUP, but it also allows for words that are executed at Compile time. If (say) |IF exists but IF does not, then IF cannot be run directly, only in compile mode. If DUP exists but DUP| does not it compiles a call to DUP. If that does not exist it throws an error.

A numeric constant will cause code that generates that literal to be pushed on the stack. The dictionary is checked first. This will be presented to the system as |42 – compiles the code to push 42 on the stack – using the same syntax.

**Dictionary Entries**

Dictionary Entries are in bold and/or red. So **DROP** or DROP both define a word DROP. This does not put anything in code merely puts a reference in the dictionary (or overrides a currently existing one). So code like 8\* 2\* 4\* 2\* 2\* ; works. It will actually – record a reference to 8\*, compile the code for 2\* whatever that is, record a reference to 4\* and compile the code for 2\* twice, followed by the code for return. Compilation will be done by |2\* and |; respectively.

**Comments**

Comments are in italics and/or white. So *this Is a comment* and so is this. Comments by convention are *italicised like this* in editors (e.g. Kompozer or Blue Griffon) if the display system supports it. Obviously it has to support either bold/italic/underline or colour.

**CSS Styling**

<style type="text/css">  
em {

color: white;

}  
body {

font-size:x-large;  
font-family:consolas,tahoma,sans-serif;  
color: green;  
background-color:black;  
}  
strong {  
color: red;  
}  
u {  
text-decoration:none;  
color:yellow;  
}  
</style>

Code can be written in HTML editors that support italic bold underlined rendering as WYSIWYG (it will still work with <em>code</em> tags but it won’t be very readable. I have had success with both Kompozer and Blue Griffon.

The CSS on the right, when inserted at the top of a HTML file, makes it into a “Color FORTH” editor in ASCII.

A script will be created which takes the HTML and converts it to the internal format.

**Design**

Initially there will be a machine with about 40 primitives , non packed, on a 32 bit machine architecture with 4 byte words, lower end format (e.g. LSB first), byte addressed (e.g. a 386 structure).

Bootstrapping will be done via a very limited simple in built compiler which compiles the base primitive definitions, call and return and a skip call if zero primitive #0if

**FORTH primitives (bootstrap)**

@ ! c@ c! +! + - \* / and or xor

not 0= 0> 0< 0- 1+ 1- 2\* 2/ dup drop

swap rot over ; r> >r $hwio

*Note that 2\* and 2/ are logical shift lefts and rights, not arithmetic or division. So 2\* and 2/ shift zero bits on the appropriate end.*

**Bootstrapping language**

This is a vastly simplified FORTH.

// comments

: <definition> // New definition

alloc n // allocate nn (const) bytes after a variable.

if … then // conditional code, 1 level only.

self // recursive call/return.

\_\_main // Main program

; // closes any open then after

word / constant / $constant // words, constants, hex constants

wordsize // pushes word size on stack

%ntext // creates <space>text in colour n

*// (1 = definition,2 = compiled,3 = execute,7 = comment)*

The design of code should provide position independent code. The first three words, which are manually generated, are the variables HEAD (of dictionary), HERE (next free word of memory) and the offset to \_\_MAIN. These words are added by the bootstrap compiler.

**Virtual Machine**

2 stacks in same memory space as code.

00000000-7FFFFFFF Push literal on stack

C0000000-CFFFFFFF Relative call (from following instruction)

D0000000-DFFFFFFF Relative branch unconditional

E0000000-EFFFFFFF Relative branch if tos = 0 (dropped)

F0000000-F00000FF Primitive 0-255

**Hardware**

00000-3FFFF 256k bytes of RAM memory, 64k 32 bit words.

40000-7FFFF 256k bytes of RAM memory, 64k 32 bit words (non volatile)

Keyboard 0 $iohw <n> returns key pressed if any, 0 if no key press.

Video display <a> 1 $iohw sets write address in video RAM (32 x 16)

<d> 2 $iohw writes data to vRAM bits 0-7 char 8-10 colour BBCRGB