ISSN 0265-5195

# Forthwrite FIGUK

Issue 107 June 2000

John Tasgal
0161 7739365
john@tcl.prestel.co.uk

# An Introduction to Color Forth

## John Tasgal

Following this Color Forth article is a commentary on some of Chuck's published code, showing how complex code can be written with a simpler Forth. John's introduction to Color Forth is necessarily incomplete as a definitive and comprehensive description will require Chuck's assistance.

Color Forth (CF) is an extremely original and interesting attempt to simplify both the structure and appearance of Forth. It inherits several features of Machine Forth, including the use of address registers. But Charles Moore has reverted in this, his latest Forth, to the destructive conditionals of Classical Forth.

Its two principal innovations are the use of colour to signify syntactic or semantic categories; and the simplification and reduction in the number of control structures.

The original source code was first shown on a monitor using a black background with coloured text. For obvious reasons of legibility I have changed the colour of the execution-mode tokens from white to black. There is also a special space character, a green space, which is shown here as a green underscore after the token
i.e. `'token_'`

The effect is to compile a literal: pop the top of stack; compile it's value; at run-time that value is pushed.

Note that in Color Forth there are no lines of source text: the code is interpreted token by token.

This article and its successor (the BMP example) are intended to be read alongside Charles Moore's description of Color Forth as given in the three references at the end.

### Notation and Glossary

| | |
|---|---|
| Ordinary text | Explanatory text - not source code |
| **`Source text`** | Source code (in a variety of colours) |

***Source Code Colour Key:***

| | |
|---|---|
| `( ) Text` | *Comments in blue* |
| `WORD` | Interpret mode in red (viz define this token as a new name in the dictionary) |
| `SWAP` | Compilation mode in green |
| `BUF` | Execution mode in black (white in the original video source) |
| `999` | Decimal numbers in grey |
| `FFFF` | *Hexadecimals in cyan* |
| `_` | Compile the number on the TOS (A green underscore which has the same meaning as a green space in the (video) source code) |

***Notation***

| | |
|---|---|
| `flag?` | A word or words which push a boolean value for use by IF. |
| `w0 w1 ..` | In the examples below these are assumed to be pre-defined application words. |

## *Basic Constructs*

Here is a list of elementary program structures. Each program or fragment is first shown on a single line, then explained in detail one token or one expression to the line.

```
1. WORD1 w0 w1 w2 ;  WORD1
Create a word and execute it.
```

The simplest CF program. Build a subroutine called `WORD1` then execute it.

### *Explanation:*

| | |
|---|---|
| `1.` | A comment |
| `WORD1` | Create `WORD1` |
| `w0 w1 w2` | Compile `w0`, `w1` and `w2` |
| `;` | Not compiled |
| `WORD1` | Executing `WORD1` causes `w0`, `w1` and `w2` to be executed. |

*Note that because of tail-recursion optimisation (see previous article) the `;` is not compiled.*

```
2. WORD1 w0 w1 w2 word1 ; WORD1
An infinite loop
```

***Explanation:***

| | |
|---|---|
| `WORD1` | Create `WORD1` |
| `w0 w1 w2` | Compile `w0`, `w1` and `w2` |
| `word1` | Compile a jump to `w0` |
| `;` | Not compiled |
| `WORD1` | Executing `WORD1` causes `w0`, `w1` and `w2` to be repeatedly executed. |

```
3. WORD1 flag? IF w0 w1 w2 ; THEN w3 w4 w5 ; WORD1
A Two-Branched Conditional
```

***Explanation:***

| | |
|---|---|
| `WORD1` | Create word `WORD1` |
| `flag?` | Compile `flag?` (which modifies the flag for use by `IF`) |
| `IF` | Compile the run time behaviour for `IF` |
| `w0 w1 w2` | Compile these words. Will be executed when `flag?` is true |
| `;` | Not compiled |
| `THEN` | |
| `w3 w4 w5` | Compile these words. Will be executed when `flag?` is false |
| `;` | |
| `WORD1` | If `flag?` is true execute `w0`, `w1` and `w2`, then return. Else execute `w3`, `w4` and `w5` and return. |

Note: For a single-branched test, just remove `w3 w4 w5`.

Note: Whereas with Machine Forth, the `flag?` was preserved, in Color Forth, Chuck has reverted to the classical `IF` which consumes the `flag?`.

```
4. WORD1 flag? IF w0 w1 w2 WORD1 ; THEN w3 w4 w5 ; WORD1
A While-True loop
```

***Explanation:***

| | |
|---|---|
| `WORD1` | Create word `WORD1` |
| `flag?` | Compile `flag?` |
| | Compile the run time behaviour for `IF` |
| `w0 w1 w2` | Compile these words. Will be executed when `flag?` is true |
| `WORD1` | Compile a jump to `flag?` |
| `;` | Not compiled |
| `THEN` | |

| | |
|---|---|
| `w3 w4 w5` | Compile these words. Will be executed when `flag?` is false |
| `;` | Not compiled |
| `WORD1` | While `flag?` is true, execute `w0`, `w1` and `w2` then repeat. Else execute `w3`, `w4` and `w5` and return. |

```
5. WORD1 flag? IF w0 w1 w2 ; THEN w3 w4 w5 WORD1 ; WORD1
A While-False loop
```

***Explanation:***

| | |
|---|---|
| `WORD1` | Create word `WORD1` |
| `flag?` | Compile `flag?` |
| `IF` | Compile the run time behaviour for `IF` |
| `w0 w1 w2` | Compile these words. Will be executed when `flag?` is true |
| `;` | Not compiled |
| `THEN` | |
| `w3 w4 w5` | Compile these words. Will be executed when `flag?` is false |
| `WORD1` | Compile a jump to `flag?` |
| `;` | Not compiled |
| `WORD1` | While `flag?` is false, execute `w3`, `w4` and `w5` and repeat. Else execute `w0`, `w1` and `w2` and return. |

```
6. WORD1 w1 WORD2 w2 WORD3 w3 ; WORD1 WORD2 WORD3
Multiple entry points
```

***Explanation:***

This is a feature not supported by ANS Forth.

| | |
|---|---|
| `WORD1` | Create `WORD1` |
| `w1` | Compile `w1` |
| `WORD2` | Create `WORD2` |
| `w2` | Compile `w2` |
| `WORD3` | Create `WORD3` |
| `w3` | Compile `w3` |
| `;` | Not compiled |
| `WORD1` | Execute `w1`, `w2` and `w3` then return |
| `WORD2` | Execute `w2` and `w3` then return |
| `WORD3` | Execute `w3` then return |

```
7. 255 FE +
Evaluate an expression containing literals
```

***Explanation:***

| | |
|---|---|
| `255 ( -- 255 )` | Push decimal `255` |
| `FE ( -- 255 254 )` | Push hex `FF` |

```
+    ( -- 509 )          Add them
```

```
8. WORD1 255_FE_+ ; WORD1
Compile an expression containing literals
```

***Explanation:***

[Note: The next stack pictures show the stack during *compilation*]

| | |
|---|---|
| `WORD1` | Compile `Word1` |
| `255 ( -- 255 )` | Push decimal `255` |
| `_ ( -- )` | Compile it |
| `FE ( -- 254 )` | Push hex `FE` |
| `_ ( -- )` | Compile it |
| `+` | Compile add |
| `;` | |
| `WORD1` | Execute `Word1` with the following run-time behaviour: |
| `( -- 255 )` | `255` pushed to stack |
| `( -- 255 254 )` | Hex `FE` pushed |
| `( -- 509 )` | Add |

```
9. 10_BEGIN w0 w1 w2 NEXT
Set up the index for a loop
```

***Explanation:***

This example pushes decimal `10` onto the stack at run-time, where it will be used as the index for the loop. [This is not as inconvenient as it seems because the A register is available for holding an intermediate value - Ed.]

`w0`, `w1` and `w2` are executed 10 times.

```
10. VARIABLE w0 w1 w2
Declare 3 variables called w1, w2 and w3
```

### *Some Idioms*

Here are some frequently-used instruction sequences:

```
1. BUF_A!
An efficient way to access a variable's address
```

Note: The variable's name is in black, followed by the green underscore.

***Explanation:***

| | | |
|---|---|---|
| `BUF` | *( -- ^buf )* | Push the address |
| `_` | *( -- )* | Compile the literal |
| `A!` | *( -- )* | Compile `A!` |

At run-time, the address is pushed, then `A!` stores it into the accumulator `A`.

*2.* `A 20_+ A!` *( A = A + 20 )*
`A -20_+ A!` *( A = A - 20 )*
*Read, modify and write A*

***Explanation:***

This is the sequence to modify `A` for pointer arithmetic where an increment-by-one isn't suitable. E.g. adding or subtracting the 'stride' of an array, or stepping through large fields in a record.
Note that there is no subtraction primitive. `'-'` must be defined as a high-level word.

*3.* `WORD1 @+ flag? IF w0 w1 w2 word1 ; THEN w3 w4 w5 ; ...`
*Process a stream using pointer arithmetic*

***Explanation:***

While `flag?` is true execute `w0`, `w1` and `w2` `then repeat`; else execute `w3`, `w4` and `w5` then return.
In each loop, initially fetch the contents pointed to by `A`, and increment `A`.

No `flag?` word is needed if it is intended to exit on `A`=zero (as false causes a jump to the exit sequence). So as a special case we can write:

```
WORD1 @+ IF w0 w1 w2 word1 ; THEN w3 w4 w5 ;  ...
```
or
```
WORD2 @+ DUP IF w0 w1 w2 word1 ; THEN DROP w3 w4 w5 ;  ...
```

The `WORD2` version has a `DUP` to allow the data value fetched to be used by the true block.
This is a fast and elegant way of scanning an array with an exit-on-zero.

```
4. WORD1 20_BEGIN @+ w0 w1 w2 NEXT ...
Process a stream using an index
```

***Explanation:***

Set the index to 20. In each loop, fetch the contents of A; increment A; process w0, w1 and w2 then decrement the index and repeat if not zero. WHILE loops are the method of choice as they don't require an index.

The next article is my annotation to Charles Moores's Color Forth program **BMP**, which converts a VGA screen to a BMP file.

***References***


1. Color Forth
   http://www.UltraTechnology.com/color4th.html
2. 1X Forth
   http://www.UltraTechnology.com/1xforth.htm
3. Dispelling the User Illusion
   http://www.UltraTechnology.com/cm52299.htm
   This includes the source code for the BMP example.


***Postscript***

In a recent message to the newsgroup (13th May) Jeff Fox reported that Chuck plans to publish Color Forth for PCs with Pentium processors. Some progress has been made in this but no release date could be given.