ISSN 0265-5195

Forthwrite FIG UK

Special Issue June 2000

FIG UK Committee

Chair	**Chris Hainsworth,**	Microplex Ltd., 5a Riverfield Road, STAINES TW18 2EE 01784 457565 chris.hainsworth@dial.pipex.com
Secretary	**Doug Neale,**	58 Woodland Way, MORDEN SM4 4DS 020 8542 2747 dneale@w58wmorden.demon.co.uk
Editor	**Chris Jakeman,**	50 Grimshaw Road, PETERBOROUGH PE1 4ET 01733 753489 cjakeman@bigfoot.com
Treasurer	**Keith Matthews,**	20 Spindlebury, CULLOMPTON EX15 1SY 01884 34818
Webmaster	**Jenny Brien,**	Windy Hill, Drumkeen, BALLINAMALLARD, Co Fermanagh BT94 2HJ 02866 388 253 jennybrien@bmallard.swinternet.co.uk
Librarian	**Sylvia Hainsworth,**	Microplex Ltd., 5a Riverfield Road, STAINES 01784 457565 sylvia.hainsworth@dial.pipex.com

Membership enquiries, renewals and changes of address to Doug.
Technical enquiries and anything for publication to Chris Jakeman.
Borrowing requests for books, magazines and proceedings to Sylvia.

FIG UK Web Site

For indexes to our Library and Forthwrite and much more, see **`http://forth.org.uk`**

FIG UK Membership

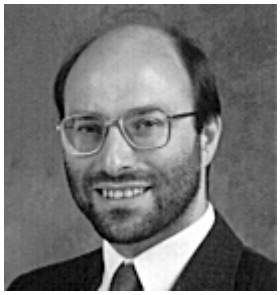
Payment entitles you to 6 issues of Forthwrite magazine and our membership services for that period (about a year). Fees are:

National and international	£12
International served by airmail	£22
Corporate	£36 (3 copies of each issue)

Forthwrite Deliveries

Your membership number appears on your envelope label. Please quote it in correspondence to us. Look out for the message "SUBS NOW DUE" on your sixth and last issue and please complete the renewal form enclosed.

Overseas members can opt to pay the higher price for airmail delivery.



Editorial

This is a special issue to publish in one place a series of 3 articles commissioned from John Tasgal.

The recent software explorations of the originator of Forth, Charles (Chuck) Moore, have attracted lots of attention in the Forth community. Chuck has given several talks and more information is available on the Ultra Technology web site published by Jeff Fox (www.ultratechnology.com) . A paper on implementing Machine Forth was presented by Reuben Thomas at the 1999 euroFORTH conference.

However, there are still details which are difficult to pin down and Chuck has published some code examples that warrant a detailed commentary. With help from Jeff Fox, John has endeavoured to fill this gap.

Until next time, keep on Forthing,

Chris Jakeman

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

An Introduction to Machine Forth

John Tasgal

There has been a lot of interest in Charles (Chuck) Moore's recent work on special processors and the languages that go with them (see issue June 1999 for an interview with Moore). These are Forths that deviate from the classical model to match the hardware more closely. The differences challenge our assumptions about standard Forth; could it become both simpler and better?

John Tasgal has researched both Machine Forth and Color Forth and expounds these differences. Alongside the Color Forth article (in the next issue) is a commentary on some of Chuck's published code, showing how complex code can be written with a simpler Forth.

Machine Forth (MF) is a development, principally by Charles Moore and Jeff Fox, of classical Forth. Its aim is to simplify both the design of stack chips and the Forth-style languages they use. It is a low-level Forth closely mapped to the underlying hardware.

There are two quite separate parts to this language:

The first part is a core which is the instruction set of a MISC (Minimal Instruction Set Computing) chip. The second part is an extension to the instruction set to allow word and dictionary building etc. .

As far as I know there is no standard and so I have chosen to use the Ultra Technology F21 chip as the 'reference' for the MF instruction set. This article describes the 'programming model' of a MISC chip. I only describe hardware where relevant.

Notation

`T(n)`	The n'th bit of register T
`T( n1 .. n0 )`	A bitfield in register T from bit `n1` down to bit `n0`

Phrases

To present the structural templates below in compact form these abbreviations are used for phrases, that is, a sequence of tokens :

`flag?`	A phrase which leaves a value in T(19..0) for use by `IF`
`carry?`	A phrase which leaves a value in T(20) for use by `-IF`
`<tt`	The phrase executed when `IF` is true
`<ff`	The phrase executed when `IF` is false

The MISC Chip

- There are 5 registers, 2 circular stacks, and a 5-bit opcode with 27 instructions decoded.
- All on-chip registers are 21 bits wide.
- The MSB, bit 20, is used for memory control for instructions which access external memory; as carry for the add instruction; and as an ordinary bit for the others.
- `IF` reads bits 19..0 and jumps if they are false. It does not pop the stack (unlike a classical `IF`). It is therefore called 'non-destructive'.
- `-IF` jumps if bit 20 is false. This too is non-destructive.
- `2/` on the F21 shifts all bits T(20..0) right. It can therefore be used either as `2/` or `U2/`, or for multiple-precision arithmetic.

The Registers

- **PC** The Program counter
- **A** The Address register for memory access
- **T** Top of data stack, the implied operand for arithmetic, logic and `IF` instructions
- **S** The 'subtop' register, the second on the data stack.
- **R** Top of return stack

The Circular Stacks

- **The Data Stack (S2 .. S11)** A 16-element circular stack below T and S
- **The Return Stack (R1 .. R10)** A 16-element circular stack below R

The Instruction Set

Control

- **`ELSE`** Unconditional jump

- **IF** Non-Destructive IF. Jump if T(19..0) is false (leaves stack untouched)
- **-IF** Non-destructive jump if-carry-false
- **CALL** A Subroutine call. Push PC+1 to R
- **RET** Return from Subroutine. Pop R to PC

A Register

- **A** **(-- A ; T = A)** Push A to T
- **@A** **(-- n0 ; T = ^A)** Fetch contents of memory at address A and push to T.
- **@A+** **(-- n0 ; T = ^A, A=A+1)** Fetch A and push to T. Increment A. ('Auto Post-Increment')
- **!A** **(n0 -- ; mem(A) = n0)** Pop T to memory at address A
- **!A+** **(n0 -- ; mem(A) = n0, A=A+1)** Pop T to memory at address A. Increment A
- **A!** **(a0 -- ; A = T)** Pop T to A

R Register and the Return Stack

- **POP** **(-- r0 ; r0 -R- ; T = R)** Pop R and push to T
- **PUSH** **(n0 -- ; -R- n0 ; R = T)** Pop T and push to R
- **@R+** **(-- n0 ; T = ^R, R=R+1)** Fetch from address in R, push to T. Increment R
- **!R+** **(n0 -- ; mem(R) = n0, R=R+1)** Pop T to memory at address R. Increment R

Data Stack Manipulation

- **DUP** **(n0 -- n0 n0)** Push T to T
- **DROP** **(n0 --)** Pop T
- **OVER** **(n1 n0 -- n1 n0 n1)** Push S to T

Arithmetic

- **+** **(n1 n0 -- n0' ; T = T + S)** Add S to T.

- **+*** **(n1 n0 -- n1 n0' ; T = T + S {T(0)=1})** If T(0) is true, add S to T non-destructively. A multiply step.
  (n1 n0 -- n1 n0 ; {T(0)=0}) If T(0) is false, do nothing.

Bitwise

- **COM** **(n0 -- n0' ; T = NOT(T))** Complement T. Invert each bit.
- **AND** **(n1 n0 -- n0' ; T = S AND T)** AND S to T
- **-OR** **(n1 n0 -- n0' ; T = S XOR T)** Exclusive OR S to T
- **2*** **(n0 -- n0' ; T = T * 2)** Shift Left one bit. Write 0 to T(0)
- **2/** **(n0 -- n0' ; T = T div 2)** Shift Right one bit. WriteT(20..1) to T(19..0). Write 0 to T(20).

Miscellaneous

- **#** **(-- n0 , | <number)** Fetch a number from PC+1 and push to T. Increment PC .
- **NOP** **( )** Do nothing for 1 cycle.

The Extensions

Very few words need to be added to an assembler based on the above instruction set to produce a working Forth system. The main categories are:

Definitions

- **:** Colon starts a new definition
- **;** Return. Does *not* end a definition
- **CREATE ... DOES** To allow new types
- **CODE ... ENDCODE** For machine code

Control Structures

These structures have the same meanings as Classical Forth but the flag/carry remain on the stack after execution.

▪ **flag? IF <tt THEN <ff**	If flag? is true execute <tt
▪ **carry? -IF <tt THEN <ff**	If carry? is true execute <tt
▪ **flag? IF <tt ELSE <ff THEN**	If flag? is true execute <tt, else execute <ff
▪ **carry? -IF <tt ELSE <ff THEN**	If carry? is true execute <tt, else execute <ff
▪ **(index) BEGIN ... NEXT**	A loop with an single index
▪ **BEGIN flag? WHILE <tt REPEAT**	While flag? is true execute <tt
▪ **BEGIN carry? -WHILE <tt REPEAT**	While carry is true execute <tt
▪ **BEGIN ... flag? UNTIL**	Loop until flag? is true
▪ **BEGIN ... carry? -UNTIL**	Loop until carry? is true

These allow words of various types to be defined; a dictionary to be built; and for the control of program flow. Numerous other words for arithmetic, logic, and Operating System functions can then be added to this extension.

Differences From Classical Forth

The Semicolon

This doesn't end a definition; it means simply 'return'. Definitions run into one another.

The Address Registers

This moves addressing from the the data stack to a register, either A or R.
Both registers also have auto-post-incrementing instructions.
This changes the style of Forth as pointer arithmetic becomes the method of choice over the use of `DO ... LOOP`'s with indexes.

Non-Destructive Conditionals

In Classical Forth, `IF` destroys the top of stack. However in Machine Forth `IF`, and therefore all the conditionals based on it, are non-destructive. This removes the need to use `DUP` when conditionals repeatedly test a flag.

But, it may lead to more use of `DROP` to remove a flag which would have been destroyed by a conventional conditional. This suggests that the behaviour of other words and if necessary the program structure itself should be adapted to optimise the use of non-destructive conditionals, rather than simply copying a program written using the destructive versions.

This is another example of a change in programming style.

Tail-Recursion Optimisation

In any definition the return action of the word before a semicolon, and of the semicolon itself, can always be compiled into a single return.

```
word1 ..... lastword ;
```

As nothing happens between **`lastword`** returning and '**`;`**' returning, the **`lastword`** return is superfluous.

A more elaborate example is the recursive call at the end of a `WHILE` loop. If we have a series of nested calls then the last instruction is in each case a return. At runtime this produces '**`; ; ; ; ;`**' viz. a sequence of returns.

The point is that when these calls unwind all that happens is that a sequence of returns are executed, one after the other. Nothing is done between them. The only necessary return is the first one pushed onto the return stack (and so the last to be executed).

Removing these superfluous returns is known as *tail-recursion optimisation*. Most Machine Forth compilers (and also Color Forth) contain a 'tail-recursion optimiser'.

Two syntaxes are currently in use to indicate that this optimisation is to be carried out:

- A special token '**`-;`**' (hyphen semicolon)
- A *smart semicolon,* which involves recognising the '**`lastword ;`**' pattern.

Tail-recursion optimisation is achieved through a compiler optimisation, and also by the syntax itself. The syntax is so designed that the programmer, simply by writing a semicolon after a recursive jump, causes the compiler to build a single return instead of nested returns. Therefore nested returns are eliminated *at the design stage* through a syntactical feature.

This is really a very unusual and elegant approach to this problem.

Next - The remaining two articles in this series appear in the next issue of Forthwrite, describing Charles Moore's newest Forth, Color Forth which builds on Machine Forth. This is followed by a detailed commentary on some of Chuck's Color Forth code to see how it is used in practice.

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

An Introduction to Color Forth

John Tasgal

Alongside 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 below.

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?`
`IF`	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.

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

The BMP Example

John Tasgal

Charles Moore has provided an example of Color Forth[1] in use, which describes the conversion of a video screen to a BMP file. Before looking at the code, here is some background information.

The Program

The aim is to format a video buffer and write it to another area of memory, the BMP buffer.

When this has been done, Color Forth is exited and, having recorded the start and length of that buffer, the memory is saved to disk using DOS. The screen is in VGA mode with a resolution of 640 columns and 480 rows. Each pixel is represented as a single byte, giving 256 colours.

The original implementation of Color Forth uses a 20-bit cell on the i21 processor. This code is for a PC implementation using a 32-bit cell.

BMP Format

A BMP (Window's Bit Map) file has three parts - a header, a palette, and the video data itself, as shown here.

Offset	Contents
0000h	Bitmap type ("BM" for Windows)
0002h	File size in bytes.
0006h	Reserved
000Ah	Bitmap Data Offset from beginning of file to the beginning of the bitmap data.
000Eh	Length of the Bitmap Info Header used to describe the bitmap colours etc (= 28h for Windows)
0012h	Horizontal width of bitmap in pixels.
0016h	Vertical height of bitmap in pixels.
001Ah	Number of planes in this bitmap.

[1] Dispelling the User Illusion
http://www.UltraTechnology.com/cm52299.htm

001Ch Bits Per Pixel
001Eh Compression Type. 0 = none; 1 = RLE8; 2 RLE4; 3 = Bitfields
0022h Size of bitmap data in bytes, rounded up to 4 byte boundary.
0026h Horizontal resolution in pixels/m.
002Ah Vertical resolution in pixels/m.
002Eh Number of colors used by this bitmap. For a 8bit/pixel bitmap this will be 256.
0032h Number of important colors
0036h The Palette of size = (#colours* 4) bytes. Each entry has 4 bytes: blue, green, red, filler. The filler is set to zero.
0436h Bitmap Data

The Algorithm

To minimise the amount of data to be stored, the extent of the image must be established.

First, the frame surrounding the image is filled with zeroes. Then, the rectangle defining the outer limits of the image is found by scanning the whole picture in four directions:

- top down to find the top edge;
- bottom up for the lower edge;
- left to right for the left edge;
- and right to left for the right edge.

This yields:

BUF a pointer variable to hold an address
H the height, and
W the width

That buffer is now formatted and written to the BMP buffer. Please refer to the 'User Illusion' text to see Charles Moore's description of this program.

Glossary

Variables:

BUF the address of the top left corner (and therefore the start of the image array in the video buffer)
H the height, and
W the width

Procedures:

```
ROW  ( stride ^row -- true | stride false )
```

Scan a row beginning at ^row.
Returns 0 for a blank line, non-zero otherwise.

ROWS (stride ^row --) Scan rows to find first non-zero row.
Store value in H.

COL (stride ^col -- true | stride false) Is this a blank column ?

COLS (stride ^col -- stride)
Scan columns looking for first non-blank.
Return width in W, and set BUF to first column.

N, (u advance --) Write a value to the location pointed to by BUF (the BMP buffer pointer).

2, (u --) Write a value to the BMP buffer, incrementing the BUF pointer by 2 bytes.

PACK (pel2 -- packpel2) Pack and invert pixels
(xbxa -- xxab) where a and b are nibble-sized pixels.

ROW ( ) Write a packed row to the BMP buffer

ROWS (^buf --) Write the video buffer to the data part of the BMP buffer

The Source Code[2]

Chuck uses a format of blocks organised in 12 lines of 20 characters. The code which follows uses a more conventional format.

```
FRAME                A comment
EMPTY ( -- )         Minimise the dictionary
Declare variables
VARIABLE BUF W H     Create buffer, width and height variables
```

Next define ROW, ROWS, COL and COLS. First is ROW :

1. set up the loop count
2. enter the begin ... next loop
3. fetch 4 pixels and increment A
4. if they're all blank then continue round the loop
5. else exit

---

[2]This is a fairly complete annotation but some parts of the code (marked with "??") defied analysis. This may be partly due to errors in the published HTML page which contains a few copying errors. Both John and I have spent some time trying (and failing) to decode the exact stack behaviour. If any reader can solve the puzzle, please write in. - Ed.

```
ROW     ( stride ^row -- true OR stride false )
                        Scan a row beginning at ^row
                        Return 0 for a blank line, non-zero otherwise
  A!    ( -- stride )           Read row address into A
  159_  ( -- stride i )         Push limit for loop to do 160 fetches of 4 bytes to
                                scan all 640 pixels across image.
                                [Original reads 169, not 159 - Ed.]
  BEGIN ( -- stride i )
    @+  ( -- stride i pel4 )    Fetch pel4, increment A. pel4 is shorthand for 4
                                byte-sized pixels packed into one 32-bit word.
    IF  ( -- stride i )         Test for a zero value indicating 4 blank pixels.
      +  ( -- stride+i )        If not found, leave a non-zero value on the stack
      ;  ( -- non-zero )        and return.
    THEN ( -- stride i )        If all pixels are blank, continue.
    DROP                        [?? Surely a mistake - Ed.]
  NEXT  ( -- stride i-1 )       Decrement loop count
  0_    ( -- stride 0 )         If it gets to here, the row is all zeroes,
;                               so push a zero and return
```

ROWS is a while-false loop .

1. while ROW returns false (a blank line)
2. decrement H

```
ROWS ( stride ^row -- )
Scans across to find first non-zero row.
Stores value in H.
  DUP BUF_ !  ( -- stride ^row )      Store current row in BUF
  ROW         ( -- true OR stride false ) Is this row non-blank ?
  IF          ( -- )            If a non-blank line
    DROP DROP                   [?? Surely a mistake - Ed.]
    ;         ( -- )            return (with BUF set to current row)
  THEN        ( -- stride )     If all pixels are blank, continue.
  DROP                          [?? Surely a mistake - Ed.]
  -1_H_+!     ( -- stride )     Decrement H
  DUP A +     ( -- stride ^row )      Point to next row[3]
  ROWS                          Jump back to first DUP (i.e. scan next row)
;
```

Identify the upper and lower edges by calling ROWS twice; first top down, then bottom up.

---

[3] This would work if A always held the row address at this point.

```
448 H !          Set H to max number of rows. As 480-32 =448, presumably these 32
                 rows are for the command line.

VGA 0            ( -- ^vga  0 )
OVER ROWS        ( -- ^vga ) Find first edge by scanning up.
-1280 SWAP       ( -- -1280 ^vga )
640 447 *                    This product is the max number of pixels in the image
+                ( -- -1280 ^vga+[640*447]  )
ROWS             ( -- )      Find second edge by scanning down.[4]

COL     ( ^col ^newcol -- true OR ^col false )
Do all rows in this column contain blank pixels?
  A!             ( -- ^col )              Make A point to the start of a column.
  H @ -1 +_      ( -- ^col #rows-1 )      Loop limit is number of rows-1 to scan.
  BEGIN          ( -- ^col i )
    @+           ( -- ^col i pel4 )       Fetch pel4 and increment A by 4
    FF_AND       ( -- ^col i pel )        Extract single pixel by clearing all but the
                                          lower 8 bits.
    IF           ( -- ^col i )            If pixel is not blank,
      +          ( -- non-zero )          leave a non-zero result
      ;          ( -- true )              and return.
    THEN         ( -- ^col i )            Else advance to the next row and continue
    DROP                                  [?? Surely a mistake - Ed.]
    A -644_ + A! ( -- ^col i )            Point A to next row -640 - 4. -4 is
                                          needed as @+ incremented A by 4
                                          (Original reads 544, not 644 - Ed.)
    NEXT         ( -- ^col i-1 )          Next row
  0_             ( -- ^col 0 )            If all rows are blank, push 0
;

COLS ( ^col -- )
Scan columns looking for first non-blank one.
Return width in W, and set BUF to first column
  DUP BUF_!      ( -- ^col )              Point BUF to start of column
  COL            ( -- true OR ^col false ) is this column blank?
  IF                                      If non-blank column,
    DROP
```

---

[4] We've now found H, the number of rows in the image, but we seem to have discarded the address where the first row of the image starts.

```
  ;           ( -- )                then return
  THEN        ( -- ^col )           Else continue
  DROP                              [?? Surely a mistake - Ed.]

  -1_ W_ +! ( -- stride )           Decrement W
  DUP A +   ( -- ^nextcol )         Point to start of next column
  COLS                              Jump back to first DUP i.e. scan next column
;
```

Identify the left and right edges by calling `COLS` twice: first left-to-right, then right-to-left

```
640 W !       ( -- )                Set W to max # of cols
BUF @ H @     ( -- BUF H )
640 * -1 +    ( -- BUF #pix )       #pix = (H*640)-1, i.e. no. of pixels
                                    containing the image after trimming blank rows
                                    from top and bottom.
OVER 639 +    ( -- BUF #pix buf' )  buf' = buf + 639
COLS          ( -- BUF #pix )       Scan left-to-right
2 + SWAP      ( -- #pix+2 BUF )
COLS          ( -- #pix+2 )         Scan right-to-left
DROP          ( -- )
W @ 1 + -2 AND ( -- W' )            Rounds W up to nearest even number so that
                                    2-byte reads and writes can be used.
W !           ( -- )                Store result in W
```

`BUF`, `H` and `W` now have their final values and we prepare to write to the buffer `BMP`.

```
BUF @             ( -- bufold )     Save old value of BUF
B71000 BUF !      ( -- bufold )     Change BUF
,                 ( -- )
4                 ( -- 4 )

N, ( n Advance -- )     Store a word of data at the location
                        pointed to by BUF. Advance the pointer BUF by
                        Advance bytes.
  SWAP        ( -- n2 Offset n1 )
  BUF @ !               Store the first word
  BUF_+!                Advance the pointer
                        [Original reads +1, not +! - Ed.]
;
```

```
2,    ( n -- )          Write a 16-bit word to the BUF buffer
  2_  ( -- n 2 )        2 = bytes to advance
  N,  ( -- )
;
```

Now writing to the `BMP` buffer can begin. First, the header:

```
BM                                  [Is this a misprint? - Ed.]
4D42 2,                             Store ASCII "BM" at offset  0000h
W @ H @     ( -- W H )
2 */        ( -- (W*H)/2 )          The bitmap size in bytes ..
16 4 * +    ( -- size1 )            Add 64 for the palette ..
54 +        ( -- size2 )            Add 54 for the header ..
,                                   and store at offset  0002h.
0 ,         0006h Reserved
118 ,       000Ah Bitmap Data Offset
40 ,        000Eh Length of the Bitmap Info Header
W @ ,       0012h Width
H @ ,       0016h Height
1 2,        001Ah Number of planes in this bitmap
4 2,        001Ch Bits Per Pixel
0 ,         001Eh Compression Type. 0 = none
W @ H @
2 */ ,      0022h  Size of bitmap data in bytes, rounded up to 4 byte boundary
0 ,         0026h  Horizontal resolution
0 ,         002Ah Vertical resolution
0 ,         002Eh  Number of colors used by this bitmap. For a 4bit/pixel bitmap
                   this should be 16 (?)
0 ,         0032h  Number of important colors
(?)         0036h  The Palette of size = (#colours* 4) bytes
ORGB        Probably moves pointer to the palette origin.
```

Next, write the Palette data. The fourth byte is the filler and is always zero. So only up to 3 bytes are ever specified.

```
  FFFFFF ,    FF00 ,     FF ,  FFFFF ,
  E00000 ,  E0C000 , FFFF00 , 808000 ,
  408080 ,  40F040 ,   40FC , E000C0 ,
  E00040 ,  C0FFFF , 404040 , FCFCFC ,
```

Finally, we have the video data section. Although the code has been written for 256 colours in memory, only 16 are used at present so the video data is packed before writing to the file. `PACK` takes two pixels of one byte each and packs them

in inverted order into one byte. For the algorithm below to work it assumes that the data is in two bytes with the pixel data in each of the lower nibbles; the data in the upper nibbles may be discarded.

i.e. `PACK ( xbxa -- xxab )`

Where `x` means don't care. The aim is to shift and swap, then do this `OR` as in:

```
xxa0
xx0b
xxab   after OR'ing
```

The data in the upper byte is ignored as, although the data is sent out a word at a time, it is overwritten by the next low byte to be sent. `'*/'` is shown black in the text - changed here to green.

```
PACK ( pel2 -- packpel2 ) Pack two pixels
     ( 0b0a -- xxab )     a and b are nibble-sized pixels
  DUP         ( -- 0b0a  0b0a )
  1_256_*/    ( -- 0b0a  000b ) shift n0 right 8 bits
  SWAP        ( -- 000b  0b0a )
  16_*        ( -- 000b  b0a0 ) shift n1 left 4 bits
  OR          ( -- xxab )        packed and inverted
;
```

`BMP` is written from bottom to top
Note: This is the second definition of `ROW` and `ROWS.`

```
ROW  ( -- )    Write a packed row to the BMP buffer
  W @ 2/       ( -- W/2 )        No. of bytes to write
  -1 +_        ( -- W/2-1 )
  BEGIN        ( -- i )
    @+         ( -- i pel4 )     Fetch 4 pixels
    PACK       ( -- i packpel2 ) Pack 2 pixels into the lowest byte
    1_N,       ( -- i )          Write the packed byte to BMP buffer
    A -2_+ A!  ( -- i )          Decrement A for next 2 pixels
  NEXT         ( -- i-1 )
;
```

[It is interesting to see that the code for `ROW, PACK` and `N,` work together to read data 2 bytes at a time and write it out byte by byte independently of the cell size of Color Forth (which seems to be 4 bytes) - Ed.]

```
ROWS ( ^buf -- )  Write the video buffer to the data part of the BMP
                  buffer.
  A!          ( -- )              Set  A  to point to the start of the data
  H @ -1 +_   ( -- i=H-1 )        Loop limit, once for each row
  BEGIN
    ROW       ( -- i )            Write a row
    A         ( -- i A )
    W @ -639 +_                   W-639  is computed at compile time
                                  (Original reads  @ - 639, not  @ -639 - Ed.)
    +         ( -- i A+(W-639) )  Address of next row
    A!
  NEXT        ( --i-1 )
;

ROWS ( -- )   Write the video data
```

Acknowledgements

I would like to thank Chris Jakeman for suggesting I write these articles.
Many thanks also to Jeff Fox for kindly reviewing an earlier draft and providing an excellent web site.

Editor's Note: We regret the problems found in decoding this example, which would surely be overcome with help from Chuck himself. We hope that they do not obscure the many techniques Chuck has introduced to simplify the compiler and make Forth more appropriate for his current work.

These include:

- Use of the special A register
- Optimisation using "**;**"
- Looping back to the start of a word by repeating its name
- Looping using `BEGIN NEXT`
- Colour syntax for brevity

FIG UK Services to Members

Magazine Forthwrite is our regular magazine, which has been in publication for more than 100 issues. Most of the contributions come from our own members and Chris Jakeman, the Editor, is always ready to assist new authors wishing to share their experiences of the Forth world.

Library Our library provides a service unmatched by any other FIG chapter. Not only are all the major books available, but also conference proceedings, back-issues of Forthwrite and also of the magazine of International FIG, Forth Dimensions. The price of a loan is simply the cost of postage out and back.

Web Site Jack Brien maintains our web site at http://forth.org.uk. He publishes details of FIG UK projects, a regularly-updated Forth News report, indexes to the Forthwrite magazine and the library as well as specialist contributions such as “Build Your Own Forth” and links to other sites. Don’t forget to check out the “FIG UK Hall of Fame”.

IRC Software for accessing Internet Relay Chat is free and easy to use. FIG UK members (and a few others too) get together on the #FIG UK channel every month. Check Forthwrite for details.

Members The members are our greatest asset. If you have a problem, don’t struggle in silence - someone will always be able to help. Do consider joining one of our joint projects. Undertaken by informal groups of members, these are very successful and an excellent way to gain both experience and good friends.

Beyond the UK FIG UK has links with International FIG, the German Forth-Gesellschaft and the Dutch Forth Users Group. Some of our members have multiple memberships and we report progress and special events. FIG UK has attracted a core of overseas members; please ask if you want an accelerated postal delivery for your Forthwrite.