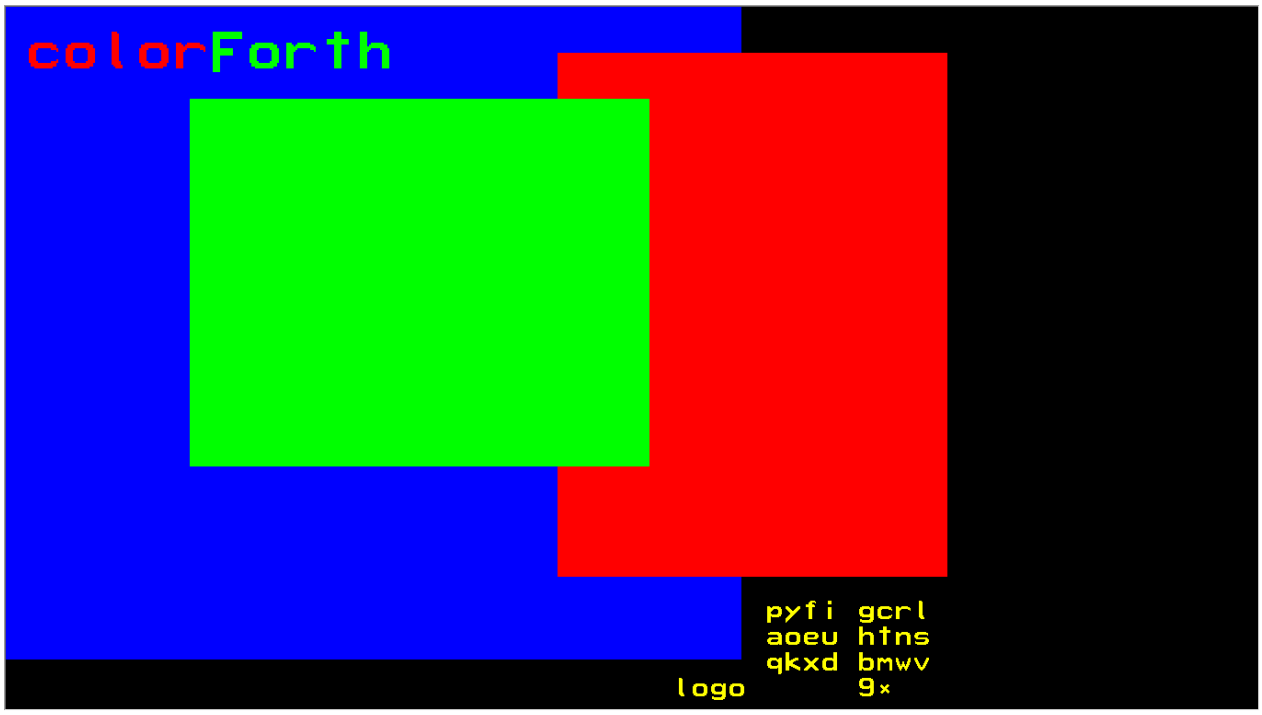
cf2019 colorForth



Contents

Summary 3
Keyboard 3
Colour 5
Token Colours 6
Actions, not Words 7
Using colorForth 7
The colorForth Editor 9
Useful Commands 12
History 13
Micro Forth 13
polyForth and chipForth 13
Slowing down for C 13
ANS Forth and Windows 13
colorForth 14

The Future 14
Philosophy 15
Keep it simple 15
The Maze Effect 15
Operating Systems and the Free Market Economy 16
Metadata and Files 17
The Library Trap 17
The Polarizing Effect 18
An anecdote : 18
colorForth Under the Hood 20
BIOS disk access 20
Video Display 20
Keyboard 20
Keypad 20
Appendix A Chuck Moore's colorForth Primer 21
Words 21
Numbers 21
Definitions 21
Loops 22
Conditions 22
Compiler 22
Program 23
Appendix B NASM Source Code 24
Appendix C colorForth Source Code 124
Appendix D "Coloring Forth" 168

Summary

colorForth is a dialect of the Forth programming language, both of which languages were invented by Charles H. "Chuck" Moore ; - Forth around 1968, and colorForth in the late 1990's.

Click here to skip straight to the "Using colorForth" section to start having fun!

colorForth uses numbered "blocks" to store and edit its source code, rather than files. Each block is 1024 bytes in size and is paired with "shadow block", on even and odd numbered blocks respectively.

The cf2019 system is created using the file cf2019.nasm assembled by the NASM Netwide ASSembler, together with colorForth blocks contained in the file cf2019Ref.img .

The resulting image file cf2019.img can be run by copying it onto a USB drive (e.g.using Rufus), or in a virtual machine such as Bochs, by running the Windows batch file ***go.bat***.

Note that running colorForth in Bochs (version 2.6) has some differences to running natively :

1. the Processor clock counter does not work
2. the hardware Random Number Generator is not emulated
3. the Programmable Interrupt Controller cannot be re-programmed. If you run the Interrupt demo on block 256, please do not try to save the system – power down Bochs instead.

The display of the source code uses a 12 x 24 x 16 bit colour, fixed-width font, in colours that indicate the function of each word, on a 1024 x 768 pixel black background.

Keyboard entry uses a 27 key subset of a normal keyboard, these key functions are described by the "keypad display" mnemonic, seen in the bottom right-hand corner of the display. Some function keys are also used to provide compatibility with conventional systems.

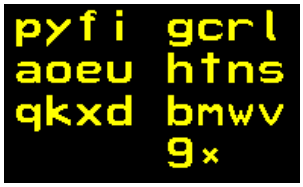


Figure 1 The keypad display

With the index fingers of each hand placed on the 'F' and 'J' keys, the keys used are that row and the rows above and below, plus the 'N', 'space' and 'AltGr' keys :

27 Keys
3 for each finger - up center down
3 for right thumb - up center right

colorForth is a concise language. The code to produce the startup logo screen pictured above is :

```
5× 5 for 2emit next ;
cf 25 dup at red $1 $3 $c $3 $a 5× green $14 $2 $1
$3 $3e 5× ;
logo show black screen 800 710 blue box 600 50 at 1
024 620 red box 200 100 at 700 500 green box text c
f keyboard ;
```

The three red words 5* , cf and logo are new words defined in terms of the other (green) words. Newly defined words can then be used as green words in later definitions. Literal numbers may be decimal or hexadecimal, the latter being marked by a $ symbol.

show sets the display task to execute the code following, using a cooperative multitasker. This allows dynamic displays which show the changing state of variables and hardware registers.

This is the documentation for the cf2019 distribution of colorForth that I maintain, distribute and publicise.

The source code and complete system (for Windows) is available from http://www.inventio.co.uk/cf2019

This is a snapshot of Work In Progress and may be subject to change in the future.

Feedback welcome : howerd@inventio.co.uk

Enjoy!

Howerd Oakford 2019 Jan 27

Colour

While the name “colorForth”, the coloured representation colorForth and the colourful appearance of the display all emphasise colour (spelled “color” in the USA), in fact the fundamental principles in colorForth go way beyond colour. Colour in this context is just one way of conveying meta-information about a computer program. For example, conventional Forth uses ‘:’ to indicate the definition of a new Forth word, colorForth uses the colour red together with starting the definition on a new line.

While conventional Forth can have coding style standards that usually specify that colon definitions start on a new line, this not required. In colorForth, red tokens (that start a new word definition) are displayed on a new line automatically. There are some special blue tokens that modify this default behaviour, and this can in any case be changed, if desired, in the NASM source code.

In the cf2019 distribution of colorForth, pressing the F4 function key toggles between colorForth mode and a more conventional Forth display. This is easy to do because the information and meta-information (information about the information) are stored in 32 bit tokens, and can be displayed in any desired way. The F4 function also makes it easier for people who are colour-blind to read the code.

To illustrate this, here is the code for displaying in “colour-blind” mode in a version of the editor written in colorForth. The first 9 lines define the colours and additional text to display for each change of token “colour” (i.e. state). The word state! is called immediately before each token is displayed, and compares the current and previous token colours and jumps to the correct action in the .old colour and .new colour tables defined by ‘jump’.

The same block in “colour-blind” mode looks like this:

If you are familiar with conventional Forth you will recognize the ‘:’ (red) as the start of a new word definition, the ‘(‘ and ‘)’ brackets to define (white) comments and the ‘[‘ and ‘]’ square brackets to wrap “immediate” (yellow) words.

The ‘mvar’ word represents a magenta variable, an interesting feature that is easy to implement in colorForth. When executed, a magenta variable returns the address of the next 32 bit cell in the source code block. If a value is stored into a magenta variable the source code is effectively changed, and due to the dynamic update of the display task the new value will be seen immediately on the screen.

Token Colours

The following colours and their meaning is described below, from file cf2019.nasm line 3832 :

```
actionColourTable:          ;    * = number
    dd colour_orange        ; 0    extension token, remove space from previous word, do not change colour
    dd colour_yellow        ; 1    yellow "immediate" word
    dd colour_yellow        ; 2  *  yellow "immediate" 32 bit number in the following pre-parsed cell
    dd colour_red           ; 3    red forth wordlist "colon" word
    dd colour_green         ; 4    green compiled word
    dd colour_green         ; 5  *  green compiled 32 bit number in the following pre-parsed cell
    dd colour_green         ; 6  *  green compiled 27 bit number in the high bits of the token
    dd colour_cyan          ; 7    cyan macro wordlist "colon" word
    dd colour_yellow        ; 8  *  yellow "immediate" 27 bit number in the high bits of the token
    dd colour_white         ; 9    white lower-case comment
    dd colour_white         ; A    first letter capital comment
    dd colour_white         ; B    white upper-case comment
    dd colour_magenta       ; C    magenta variable
    dd colour_silver        ; D
    dd colour_blue          ; E    editor formatting commands
    dd colour_black         ; F
```

Actions, not Words

I strongly recommend that you, dear reader, run cf2019 as a program on a suitable computer. There are two ways of doing this :

1. Copy the binary image file cf2019.img directly onto a USB drive, and boot the computer using this drive.
2. Run cf2019 in a bochs environment under Windows. Double click on the file **go.bat** in the cf2019 distribution to do this.

This is because “the map is not the territory” – both Forth and colorForth provide an interactive environment that is best experienced, rather than discussed.

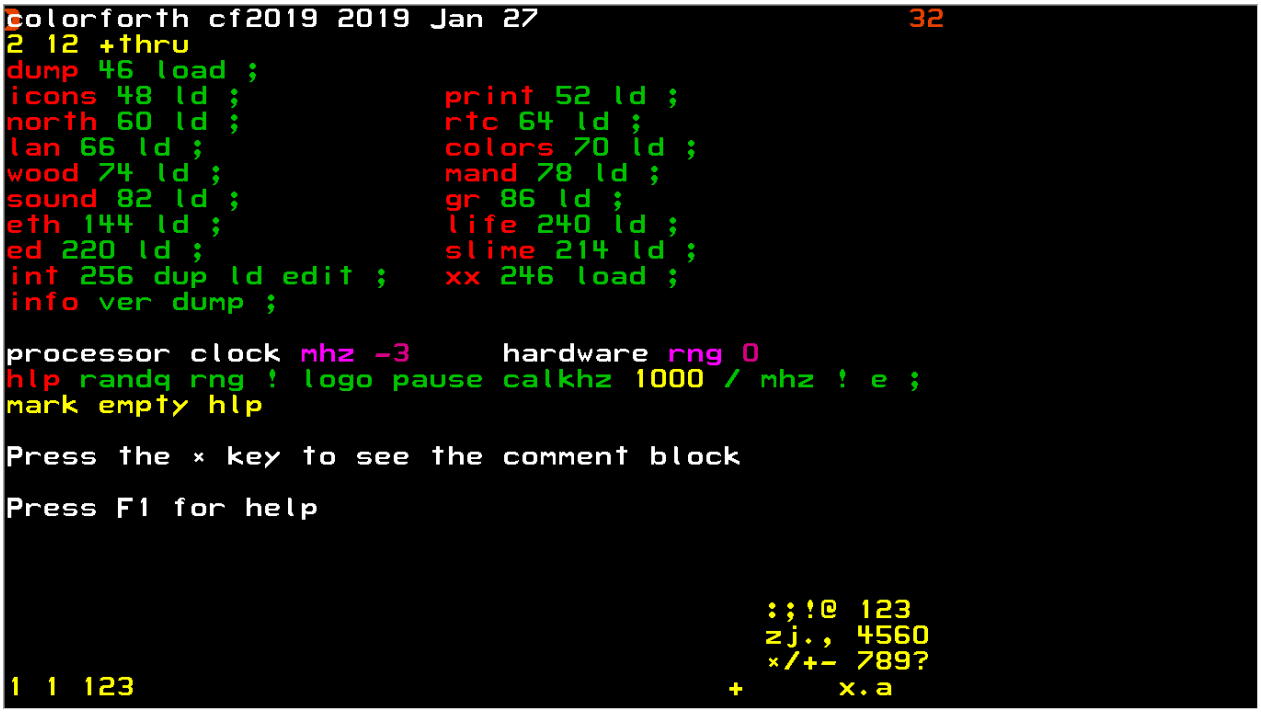
Using colorForth

Note : The space bar is used in colorForth as if it is the Enter key.

The system starts up in the Editor - press the space bar to exit, then the space bar again to enter numeric mode.

Enter a number and press the space bar again. Press the space bar again and enter the next number. After entering two numbers e.g. ‘1 1 ‘press the AltGr key to see the Alternative alpha keypad, and press ‘+’:

1 1 +



Above shows the state just before the final space bar. The ‘123’ represents the current word being typed, in this case ‘+’ . The two ‘1’s are on the stack, with the current word being typed, and the stack is shown in the bottom left of the screen. The orange ‘32’ in the top right is the current block number.

Pressing the space bar now will execute ‘+’ which adds the top two stack items, in this case ‘1’ and ‘1’, and will return ‘2’.

On a running cf2019 system :

Press F1 for the help screen. Press repeatedly to cycle round the main load block and its documentation shadow block and the Help screen.

Press F2 to toggle decimal and hexadecimal display of numbers.

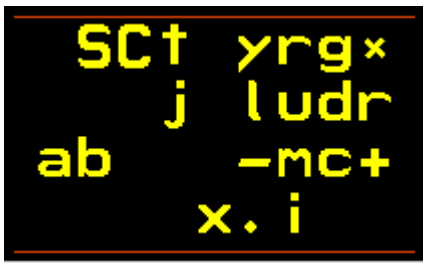
Press F3 to toggle display or hiding of blue token words. These words may be added or deleted like any other word in the editor. The following special blue words are detected and acted on by the editor display :

Blue word	**Function**
cr	one CR
,	one CR
-tab	move to the next 24 multiple column, disabling a CR for a red word
tab	move to the next 24 multiple column
br	two CRs
-cr	disable a CR for the next red or magenta word
cr+	one CR and indent 3 spaces
blue	no action
tab3	align to next 3 space column
.	add one space
..	add two spaces
...	add three spaces
....	add four spaces

Press F4 to toggle normal and “colour-blind” display mode, this also runs the editor.

The colorForth Editor

When the editor is run by typing ***e*** , ***256 edit*** , or by pressing F4, the keypad looks like this :



The mnemonics mean :

Mnemonic	**Function**	**Qwerty key (UK)**	**Qwertz key (German)**
S	White text CAPITALS	W	W
C	White text first letter only Capital	E	E
t	White text lower case	R	R
y	Yellow – immediate actions, not compiled	U	U
r	Red token – create a new word	I	I
g	Green – compiled token	O	O
*	Toggle main and shadow blocks	P	P
j	Jump to last edited block	F	F
l u d r	Cursor left up down right	J K L ;	J K L Ö
a	Silver (gray) token	Y	Y
b	Blue token	X	X
-	Decrease block number by 2	M	M
m	Magenta variable token	,	,
c	Cyan	.	.
+	increase block number by 2	/	-
x	Delete token	N	N
.	Exit the editor	Space bar	Space bar
i	Insert previously deleted token	AltGr	AltGr

Use the **+** and - keys to select the block to edit (also PgUp and PgDn), or press the space bar to exit the editor and type

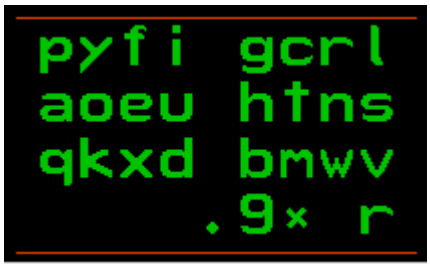
256 edit to edit block 256

Use the **l u d r** keys to move the cursor to the required location. The arrow keys and the Home and End keys can also be used to move the cursor.

Choose a colour, for example red to create a new word – the keypad now looks like this :

Press the required letters until you have finished entering the word, then press the space bar (‘9’).

The display then turns green to enter complied word :

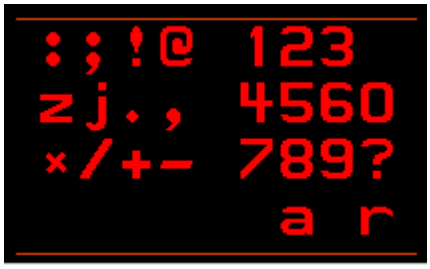


Press the required letters until you have finished entering the word, then press the space bar (‘9’).

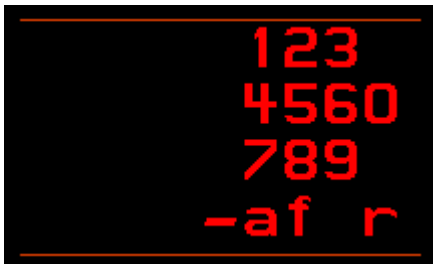
Repeat for the next word.

If you press the N key (‘.’) you will return to the main Editor keypad,so you can choose a different colour, move the cursor or select a new block to edit.

The AltGr key, labled * in the keypad toggles between the main alpha and Alternate alpha text entry mode :

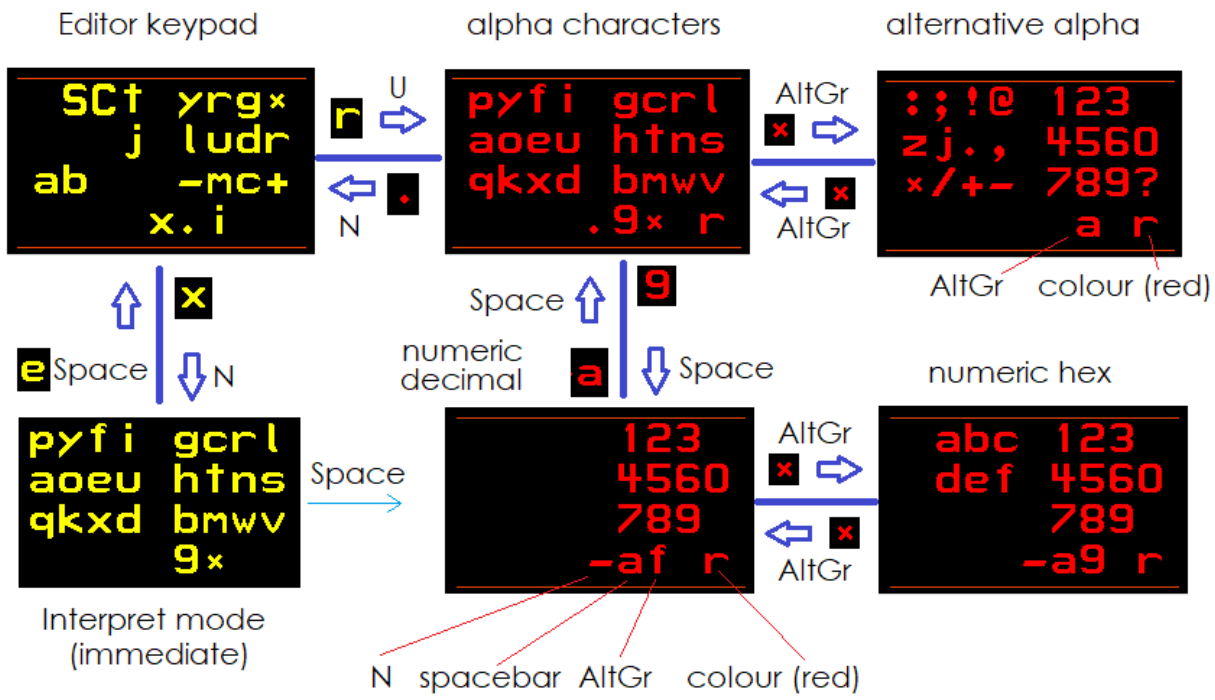


Press the required letters until you have finished entering the word, then press the AltGr key to return to alpha mode, then the space bar (‘9’) as described above.



Pressing the space bar in alpha mode will change to numeric mode, pressing the AltGr key toggles between decimal and hexadecimal display mode :

The following diagram show the different keypad mnemonics and the keys to press to change them :



The other colours, and editor / Interpret mode all have identical functionality.

Editor mode is shown by the two horizontal lines above and below the keypad.

When you have found and/or edited the block that you would like to run, leave the editor by pressing the space bar. You are now in Interpret mode (no more orange lines).

Type the command you would like followed by the space bar.

Useful Commands

Command	Action	
e	Run the Editor displaying the last block edited	
32 edit	Run the colorForth Editor displaying block 32	
xx	Run the colorForth Explorer	
ll	Load the current block displayed by the Editor	
vv	View the last block loaded by the command ld	
uu	Undo all changes to the current block	
ss	Save the current block to disk	
save	Save the entire system to disk. You can change the USB stick to save a backup.	
sa	Save and return to the Editor	
logo	Show the colorForth logo screen	
empty	Remove all compiled definitions since mark was called	
mark	Mark the current system state for empty	
$1000 dump	Dump the 16 32 bit cells starting at address $1000	
bye	Exit the system, discarding all edits since the last save, sa or ss	
hlp	Update the hardware system info and display the start block. This is currently block 32, and displays a list of Apps.	
life	Run the Conway's Game of Life demo. Press the space bar to exit (marked by a '.' In the keypad mnemonic), then type xx and press the space bar. Scroll to "Conways Game of Life" and press the e key to view the source code, then press the '*' key in the Editor to view the documentation shadow block.	

You can also use the cursor control keys in the editor or the arrow, Home and End keys to move the cursor immediately after a red word (in either Editor or Interpret mode), then press the Enter Key (on the QWERTY or QWERTZ keyboard, not the keypad) to execute that word.

History

Micro Forth

I discovered Forth, in the form of Micro Forth for the RCA CDP1802 processor chip, around 1979.

I was working for a small startup company in the UK, developing a Grain Moisture Meter, and was planning to use the RCA assembler. When I unpacked the newly arrived RCA COSMAC development system (an 8 bit CDP1802 processor clocked at 2 MHz, with 4K RAM) I noticed a single sheet of paper advertising MicroForth, and promising fast development times, small code size and fast run-time speed - all of these claims I later found out to be true.

Having convinced my boss that Micro Forth would be a good investment, I waited for some weeks for the 8-inch floppy disk to arrive by post from California, and followed the Quick Start Guide.

I typed **1 1 + .** and saw the result : **2** . I could talk to the computer, and it could reply, and it could even do maths. I was impressed, and at that moment my career as a computer programmer changed direction to the Forth side – I was hooked.

polyForth and chipForth

Having completed theMoisture Meter project , in the mid 1980's I looked for more Forth work, and got contracts working for COMSOL in the UK (a Forth software development house and supplier of Forth, Inc. products such as Micro Forth, polyForth and chipForth). It was through COMSOL that I got a job working on the Riyadh Airport HVAC system, foir AVCO in Huntsville Alabama.

Slowing down for C

Towards the end of the 1990's the demand for Forth had dimished, so I learned C. My first C contract was a six month project – when I discovered the details of the contract I was ***horrified*** – in Forth, I would normally have finished this sort of project in 6 weeks.

As I polished my C/C++ and later C# skills I found that it was not good etiquette to mention Forth at work, especially at interviews. In every big company, maybe 1 in 50 programmers would say something like "Oh yes, I used Forth back in the 80's – loved it!". And an equal number of people would look at me like I had escaped from a lunatic asylum, retro-computing museum or Area 51 UFO containment area. So I went into stealth mode – I kept "...one of the best-kept secrets in the computing world" to myself. It seems that software development departments encourage programmers to use their own tools, so I rarely had problems using Forth to develop "tools", even when the ultimate goal was to develop C or C++ programs.

This gave me an overall speed advantage of maybe 2 or 3 times compared to my colleagues – this resulted in long, relaxed contracts in C, interspersed with much shorter contracts in Forth.

ANS Forth and Windows

With Windows replacing MSDOS, I started using SwiftForth (Forth, Inc.'s product for Windows), MPE's VFX Forth and Win32Forth. Forth is a chameleon language – it adapts to its environment, in this case Windows.

I do not hate Windows, in fact I think Microsoft have produced products of a consistently high standard, at least since NT4. But Windows programming has become steadily more and more difficult. Using Forth I still have an advantage over my colleagues – my "secret weapon" is still loaded and ready for action.

colorForth

Around 2001 I downloaded Chuck Moores' public domain colorForth from his website and copied on to a 3.5 inch floppy disk. It was not easy to get working – I had to add a new, compatible floppy disk ISA board to make it work.

I was impressed, again, wrote the article : colorForth and the Art of the Impossible and presented it at EuroForth 2001. I also had the great good fortune to spend about 45 minutes with Chuck, looking at his colorForth CAD system, OKAD II.

I love working in colorForth – I think it must be something genetic, certainly it appears not to be curable.

I presented another paper at EuroForth 2003 "The colorForth Magenta Variable", and handed out floppy disks with the first distribution of my version of colorForth.

Time marches on, and one of my two PCs still with a floppy disk drive, died. I still have the other one, in the cellar, "just in case". But it became obvious that colorForth needed to be updated to run from a USB stick.

A decade or so later, I presented a paper "Crypto colorForth" at EuroForth 2017 (the video is here), and demonstrated colorForth running from a USB stick. I believe that security and complexity are incompatible in computer software, and that colorForth can be the basis of a very secure operating system (without using files).

Today I am launching colorForth cf2019 – there are a lot of changes, most of them for the better :

Byte addresses are used through out, all magic numbers have been replaced by equ's and the code is now better documented. It is far from perfect.

The user experience is more comfortable, most of the original apps now work again.

The Future

1. Make colorForth load and run from a FAT32 file system on a USB stick. I already have a FAT32 single sector bootloader. This will allow data transfer between the colorForth system and the rest of the world.
2. Add more drivers for Ethernet and WiFi hardware and mouse support.
3. Add an assembler/disassembler and the ability of colorForth to rebuild itself without NASM.
4. Add a secure data/metadata sharing and backup system (the You-Me Drive)

Philosophy

Keep it simple

From Chuck Moore's book Programming a Problem-oriented Language :

"The Basic Principle

• Keep it Simple

As the number of capabilities you add to a program increases, the complexity of the program increases exponentially. The problem of maintaining compatibility among these capabililties, to say nothing of some sort of internal consistency in the program, can easily get out of hand.

You can avoid this if you apply the Basic Principle.

You may be acquainted with an operating system that ignored the Basic Principle. It is very hard to apply. All the pressures, internal and external, conspire to add features to your program.

After all, it only takes a half-dozen instructions; so why not? The only opposing pressure is the Basic Principle, and if you ignore it, there is no opposing pressure."

The Maze Effect

My own experience of writing computer software is that it is like traversing a maze, rather than travelling a well mapped journey. It is often the case that you get to within one hedge-width of the goal, only to find that there is no way through – you must back-track, re-think and repeat . This equates to discarding already written software, which is often interpreted in a commercial software development environment as an expensive mistake, and so must be avoided.

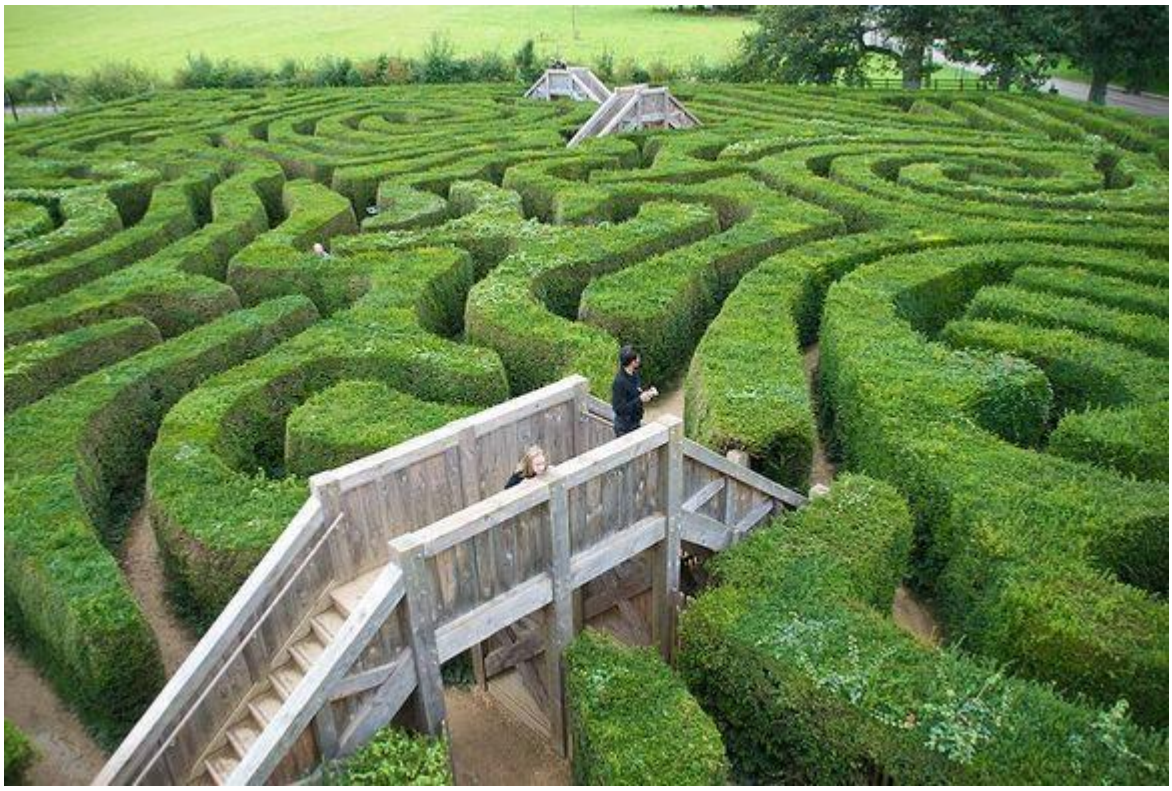


Figure 2 The Hedge Maze at Longleat, UK

The picture above shows a hedge maze – a good metaphor for real-world software development. You can also see here bridges that can be used to climb over hedges – the one in the foreground could represent the change from block-based to file-based source code. Clearly it is an enormous advantage if you want to go in that direction, but it can also prevent or hinder access to other goals. It is not clear in the above picture precisely what the goal is, and this often goes for software development.

It is very difficult to write a requirements specification for a system that has not yet been discovered.

Operating Systems and the Free Market Economy

In the developed world, commercial companies exist to make a profit – it is illegal to run a company that makes a loss – therefore there are two areas where “due diligence” must be applied :

1. Maximizing profit
2. Assessing risk

In the field of software development, maximising profit can be approximated to maximizing sales, if the development cost is (or can be made to be) a fixed amount.

Microsoft, for example achieved success initially by selling the MSDOS operating system. The “killer idea” was to sell a program to hardware manufacturers that provided a smooth interface to programs written by software developers. An entire eco-system evolved in which everybody gained :

1. hardware manufacturers sold more because their product could run more programs
2. software developers coul sell more software because it could then run or more hardware
3. Microsoft made a huge profit

When competitive Operating Systems (OS's) came along, it became necessary to "Lock in" users.

A number of techniques have been used :

1. Operating Systems must be complex, otherwise everyone can write their own
2. The interface must be incompatible with other OS's
3. Metadata must be held outside of the users' control

It is the last item that I wish to explain in more detail, and then confront :

Metadata and Files

What is a *file*, where does it exist and who owns it?

Take as an example a file in the MSDOS or Windows Command Shell environment. Typing CD shows you the current folder, typing DIR lists the files in that folder. This gives the illusion that your file exists in that folder – you can see, send it, open it, edit it etc. The file content, and the directory structure where it can be found may well exist on your computer – you can delete it after all, and its gone (or at least it has been moved somewhere else).

But the metadata (information about the file) can only be accessed through the Operating System – that is what the Operating System is for, after all.

Take another example :

I create a Word document using Windows and the Word app. I send in by TCP/IP to a colleague who opens it on an Apple computer using the MAC OS and Pages. The most important metadata of the file is its size. The TCP/IP programming interface requires an address of the file data and its length. The Windows file system provides these parameters to the TCP/IP program, the required amount of data is sent to the Apple computer, and its TCP/IP program passes the data to the MAC OS.

Note that the metadata (file size) is not attached to the file at any point. Certainly the TCP/IP program requires and supplies a length parameter, but it is not attached to the file.

Of course both Operating Systems provide a convenient user interface, but the users are locked-in.

Another protocol layer is required – for example SMB or SAMBA - and these layers can be made as complex and incompatible as is required to prevent another Operating System from competing easily.

There are many data exchange formats, for example XML and JSON – but these describe the data content of files and are not an alternative to files.

An important part of lock-in is never to attach important metadata such as the filesize to a standardised data structure. This is to prevent simple applications from using the "file" without being tied to a complex Operating System.

The Library Trap

Forth and colorForth provide an interactive environment, in which maze-traversal becomes an order of magnitude faster (and more enjoyable) than non-interactive, batch processed languages such as C/C++/C# - IMHO, YMMV.

Python also provides an interactive environment, with libraries - for certain classes of problems it is an extremely effective tool – I have used it for example to create AES128-GMAC signed packets for testing software. Protocols such as AES and GMAC are complex, maybe necessarily so for signing packets securely, and this certainly saved many weeks of work.

There are at least two problems with libraries :

1. As implemented in Python they come with a lot of baggage - an Operating System, installer programs - and of course Python interprets files - not exactly KISS.
2. One of the fundamental principles of both Forth and colorForth is that the user and computer develop a relationship – the user teaches the computer how to do new things by defining new words, and the computer tells the user what it thinks of these new words, by returning values and error messages. Libraries do not allow this interaction. Somebody else may have experienced this when the library was written, but now it is just code.

The Polarizing Effect

IMO Forth is a polarizing language because it can dramatically increase productivity in software development. This has several effects :

1. Non-Forth programmers feel uncomfortable when a Forth programmer produces a program in a half or a quarter of the time that they would take. Nobody likes to be made to feel like an idiot.
2. Forth programmers have a significant advantage, so they naturally love "...one of the best-kept secrets in the computing world."
3. Programming in Forth is fun – it is creative rather than mechanical work
4. Forth is different and can have a steep un-learning curve

So people either love Forth or hate Forth – there is wide distance between the two extremes.

An anecdote :

In the late 1990's I was working on a three month contract, programming an LCD drive in C.

There were four special constraints :

1. The interface was I2C and the power consumption was critical, so only the minimum information must be sent, that is required to update the LCD
2. There were two LCD driver chips, one for the left half of the display, the other for the right
3. The whole display is mounted upside down – the graphics mode x,y = (0,0) coordinate is in the bottom right hand corner.
4. The 8x8 character glyphs have to be rotated by 180 degrees.

The interface is similar to ANS Forth's TYPE and AT-XY, but in C.

To solve this problem I first added a simple talker program, written in C that connected the devices serial port to its I2C bus and memory - something like PEEK and POKE, I2Cread and I2Cwrite.

Then I used SwiftForth to create the other side of the talker interface and words to display characters and the memory map in the device that was used to store the current LCD data. I also a word to rotate the font data and save it as a C file.

Having developed the program in Forth on a PC, testing all the while on the actual ARM target device, I translated the Forth program to C. Since I knew that I would be doing this I did not use any fancy Forth features such as CREATE …. DOES> or EVALUATE , and I kept the Forth word names compatible with C – no “%&*+-. ><” etc. characters.

The timescale for this was roughly three weeks of fun Forth development, about a week to convert to C. Then I cleaned up the code a bit, improved the documentation and so on, and after about 6 weeks I presented the fnished code to my colleague – an experienced and competent C programmer. He was surprised that I had completed the work so quickly, but was embarrassed that I had shown that his original time estimate was so wrong. No one likes to feel that they are inferior to anybody else, or that their career based on C is maybe not the most productive. My contract was not extended.

colorForth Under the Hood

BIOS disk access

Originally using direct hardware access to Floppy disk controller hardware, now converted to use 16 bit BIOS calls from 32 bit protected mode. This means you can use a USB stick, or USB Floppy drive.

Video Display

The display setup uses VESA calls, 1024x768 16 bit colour mode, with some support for 800x600 16 bit colour. I did this was so that I could run colorForth on my Samsung NC10 netbook.

Keyboard

The keyboard keys are scanned directly from I/O ports, using a PAUSE in the wait loop. Luckily the BIOS handles a USB keyboard and emulates legacy hardware ports.

Keypad

colorForth does not use the keyboard in the usual way, but instead uses any type of 102 key keyboard to emulate a 27 key keypad, along the lines of a Dvorak keyboard.

Appendix A Chuck Moore's colorForth Primer

colorForth Primer
Chuck Moore

colorForth is a uniquely simple way of programming computers. It is particularly suited to the multi-computer chips of GreenArrays. How simple it is:

Words

colorForth uses words much as English does. (A *word* can be a subroutine, if that helps.) A *word* is a string of lower-case characters (from a set of 48) ending with space. The character @ is pronounced *fetch*and fetches a number from some address. Likewise, ! (*store*) stores a number. Some words:

- and or drop dup over push pop
- for next unext -if if then
- ; @ ! @+ !+ @b !b @p !p
- + . +* 2* 2/ - b! a!
- 12345 -1 144 0

If you type a word, the computer will perform some action. For example

- on

might turn on a light.

Numbers

Words that look like numbers are placed on a push-down stack (like a stack of dishes). @ also puts numbers on the stack. There they serve as arguments for later words:

- 1000 ms
- 3 !

Definitions

New words are defined in terms of old:

- toggle on 1000 ms off ;

The red word is defined by the following green words. When you type toggle, the light is turned on, the computer waits 1000 ms (milliseconds) then turns it off. Semicolon marks the end of this word (return from subroutine).

Other words:

- on 3 ! ;
- off 2 ! ;

Here a number is stored into a register to change an output.

Loops

Computers are good at repetition. Here's one way to define a loop:

- ms for 1ms next ;

The word for expects an argument and puts it into a counter. The word next returns to for that many times. The word 1ms waits 1 millisecond.

Conditions

Computers sometimes need to make decisions:

- abs -if - 1 + then ;

abs will return the absolute value of its argument. If it is negative, -if does a ones-complement and adds 1. If it is not negative (positive or 0) -if jumps to then and does nothing.

Compiler

colorForth compiles source code into machine instructions, which can then be executed. It uses color to indicate the function of a word:

- Yellow - a word to be executed
- Red - a word being defined
- Green - a word to be compiled as part of a definition
- White (or black) - a comment to be ignored

Color aids understanding, avoids syntax and simplifies the compiler.
The compiler reads words from text stored in memory. A special editor manages this text. colorForth code is exceptionally compact.

Program

A program in colorForth is a collection of simple words that describe a task. Although definitions can be long and complicated, that is not wise. A larger number of simpler words is easier to read, write, debug and document.

The computer begs fallible programmers: Keep It Simple, Stupid (KISS). colorForth helps.

Appendix D “Coloring Forth”

Coloring Forth

Because we must deal with the unknown, whose nature is by definition speculative and outside the flowing chain of language, whatever we make out of it will be no more than probability and no less than error.
– EDWARD SAID
Beginning, Intention and Method

Motivation

Core Words

Specials

bye

Quit ColorForth.

Stack No effect

ColorForth Source

```
bye
```

Assembler

```
Bye:
    push    0
    call    ExitProcess
```

Macros

```
macro2 dd offset semi
       dd offset cdup
       dd offset qdup
       dd offset cdrop
       dd offset then
       dd offset begin
```

; semi

Overview

Semicolon – terminates current definition.

Implementation

Implementation is non-trivial – it provides both tail-recursion support and some optimization. If the last compiled item was a `call` to a word, it's being replaced with a `jmp`. Otherwise, `ret` is compiled.

`List` variable contains address of last compiled word.

`H` stands for `HERE`

```
H       dd  0

list    dd  0,
0


semi:
    mov     edx, [H]
    sub     edx, 5
    cmp     [list], edx
    jnz     @f
    cmp     byte ptr [edx], 0e8h
    jnz     @f
    inc     byte ptr [edx]  ; jmp
    ret
@@: mov     byte ptr [5+edx], 0c3h ; ret
    inc     [H]
    ret
```

Sample

```
  Forth
x 1 2 + ;
y x x ;
```

Code

```
                        x:
008A07B5 8D 76 FC               lea         esi,[esi-4]
008A07B8 89 06                  mov         dword ptr [esi],eax
008A07BA B8 01 00 00 00         mov         eax,1
008A07BF 05 02 00 00 00         add         eax,2
008A07C4 C3                     ret
                        y:
008A07C5 E8 EB FF FF FF         call        x
008A07CA E9 E6 FF FF FF         jmp         x
```

```

```

This sample deserves some explanation.

EAX contains the topmost element of data stack. `ESI` points to the second element. In order to put the value **1** onto stack, we have to store current topmost element in memory and decrement stack pointer, and only then load the constant into `EAX`. This operation takes 2 Pentium commands, 5 bytes, XXX ticks . Not so bad.

As we can see, **2 +** is compiled into something shorter and better, we'll look at literal optimization later.

Now, for the word **X** the semicolon `;` has compiled `ret`, while for Y the second call to X has been replaced with a jump.

dup cdup

Implementation

As we can guess from the code below, `cdup` stands for "compile `dup`". It compiles 5 bytes onto the top of the dictionary, and we have seen these 5 bytes above. So, dup is implemented as a macro, which compiles code to push the topmost element from `EAX` onto in-memory stack.

```
cdup:
    mov     edx, [H]
    mov     dword ptr [edx], 89fc768dh
    mov     byte ptr [4+edx], 06
    add     [H], 5
    ret
```

Traditional Forth implementation could look like this.

```
hex

: dup 89fc768d , 06 c, ; immediate
```

?dup qdup

Implementation

```
qdup:
    mov     edx, [H]
    dec     edx
    cmp     [list], edx
    jnz     cdup
    cmp     byte ptr [edx], 0adh
    jnz     cdup
    mov     [H], edx
    ret
```

This code looks whether the last compiled instruction was `0adh`, which stands for

```
    lods    dword ptr [esi]
```

And this is nothing else than a `drop` – move second element into the top of stack register, and increment stack pointer. So, `?dup` works as `dup`, though if last compiled instruction was `drop`, it shifts `HERE` one byte back – actually, "uncompiles" the `drop`.

This trick is used for optimizing macros – immediate words – that compile code with stack notation `( -- n)`, for example, `0`, `A`, and `pop`, following words, which take one item off stack `( n -- )` – for example, `push` (traditional `>R`), `!` or `A!`.

```
0 ?dup c031 2, ;
a ?dup c28b 2, ;

pop ?dup 58 1, ;

! ?lit if ?lit if 5c7 2, swap a, , ; then 589 2, a, drop ; then a! 950489 3,
0 ,drop ;
push 50 1, drop ;
a! ?lit if ba 1, , ; then d08b 2, drop ;
```

The macro 0 puts zero onto stack. It compiles into

```
    xor     eax, eax
```

`A` puts the value of the address register onto stack. Corresponding Pentium code is

```
    mov     eax, edx
```

Let's consider an example

```
x1 here 2/ 2/ dup push a! 0 a ! pop @ ;
```

This code moves values 1 and 0 into the cell on the top of the dictionary.

```
here
2/
2/
push


drop
a!
0
dup

a
!


pop
@
;
```

```
008A07CF E8 8B 12 B6 FF       call        here (00401a5f)
008A07D4 D1 F8                sar         eax,1
008A07D6 D1 F8                sar         eax,1
008A07D8 8D 76 FC             lea         esi,[esi-4]
008A07DB 89 06                mov         dword ptr [esi],eax
008A07DD 50                   push        eax
008A07DE AD                   lods        dword ptr [esi]
008A07DF 8B D0                mov         edx,eax
008A07E1 B8 00 00 00 00       mov         eax,0
008A07E6 8D 76 FC             lea         esi,[esi-4]
008A07E9 89 06                mov         dword ptr [esi],eax
008A07EB 8B C2                mov         eax,edx
008A07ED 8B D0                mov         edx,eax
008A07EF AD                   lods        dword ptr [esi]
008A07F0 89 04 95 00 00 00 00 mov         dword ptr
[edx*4],eax
008A07F7 58                   pop         eax
```

Implicit `dups` and `drops` are highlighted with light blue. In future I will be replacing Pentium instructions with dup and drop macros. It's interesting to notice that dup is 5 bytes, white drop – only one.

This example, though a bit artificial, illustrates how address register and store-fetch pair works, as well as introduces "address as offset from 0 in cells" and "address as offset from 0 in bytes". I'll be calling these cell address and byte address.

.