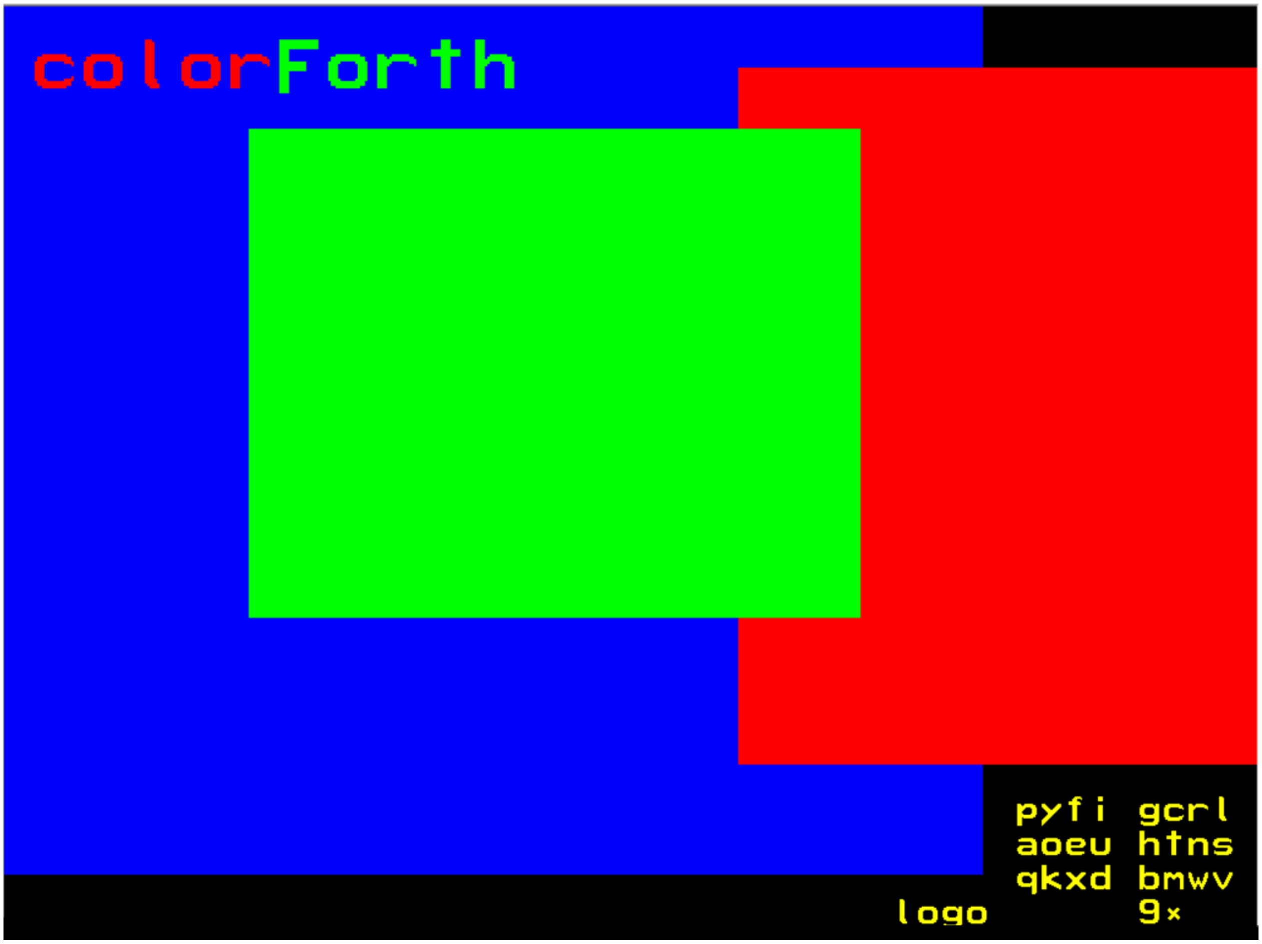
cf2023 colorForth



Contents

Summary 3
Keypad 3
Colour 5
Token Colours 6
Actions, not Words 7
Using colorForth 7
The colorForth Editor 9
colorForth and ASCII 12
Useful Commands 14
History 15

Micro Forth ... 15
polyForth and chipForth ... 15
Slowing down for C ... 15
ANS Forth and Windows ... 15
colorForth ... 16
The Future ... 16
Philosophy ... 17
Keep it simple ... 17
Chuck Moore from Fireside chat Nov 2020 ... 17
The Maze Effect ... 17
Operating Systems and the Free Market Economy ... 18
Metadata and Files ... 19
The Library Trap ... 19
The Polarizing Effect ... 20
An anecdote : ... 20
colorForth Under the Hood ... 22
BIOS disk access ... 22
Video Display ... 22
Keyboard ... 22
Keypad ... 22
Appendix A Chuck Moore's colorForth Primer ... 23
Words ... 23
Numbers ... 23
Definitions ... 23
Loops ... 24
Conditions ... 24
Compiler ... 24
Program ... 25
Appendix B NASM Source Code ... 26
Appendix C colorForth Source Code ... 149
Appendix D "Coloring Forth" ... 196

Summary

colorForth is a dialect of the Forth programming language, both 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 cf2023 system is created using the file cf2023.nasm assembled by the NASM Netwide ASSembler, together with colorForth blocks contained in the file cf2023Ref.img .

The resulting image file cf2023.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 16 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.

Keypad 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 cf2023 distribution of colorForth that I maintain, distribute and publicise.

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

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 2023 Apr 04

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 cf2023 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’.

```
Editor Display cblind 0                                 260
cb cblind @ 0 + drop ; state 16 statex 16
yellow $ffff00 color ;
+txt white $6d emit space ;
-txt white $6e emit space ;
+imm yellow $58 emit space ;
-imm yellow $59 emit space ;
+mvar yellow $9 emit $11 emit $5 emit $1 emit space
;
txts string $3010100 , $7060504 , $9090901 , $f0e0d
0c , ;
tx c-c $f and txts + 1@ $f and ;
.new state @ $f and jump nul +imm nul nul nul nul n
ul nul nul +txt nul nul +mvar nul nul nul ;
.old statex @ $f and jump nul -imm nul nul nul nul
nul nul nul -txt nul nul nul nul nul nul ;
state! n-x dup 0 + drop 0if drop ; then tx cb 0if d
rop ; then state @ swap dup state ! - drop if .old
.new state @ 0 + if dup statex ! then drop then ; n
load




                                               SCt  yrgx
                                              ⇤f⇥j  ←↑↓→
                                              ab    -mc+
                                        edit        x.i
```

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 cf2023.nasm line 4024 :

```
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 run cf2023 as a program on a suitable computer. There are two ways of doing this :

1. Copy the binary image file cf2023.img directly onto a USB drive, and boot the computer using this drive.
2. Run cf2023 in a bochs environment under Windows. Double click on the file **go.bat** in the cf2023 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 or thought about.

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 +

```
colorforth cf2022 2022 Apr 08                         64
processor clock mhz 4
dump x 5952 y 0 ld lblk 64
2 12 +thru
dump 78 load ;
icons 80 ld ;             serve 506 ld ;
north 92 ld ;             rtc 96 ld ;
lan 98 ld ;               colors 102 ld ;
wood 106 ld ;             mand 108 ld ;
sound 114 ld ;            gr 118 ld ;
eth 176 ld ;              life 272 ld ;
ed 252 ld ;               slime 246 ld ;
int 288 ld ;              xx 278 load ;
info ver dump ;           staks 504 ld ;

hardware rng 0
chm -- 0 mhz ! $1740 x ! 0 y ! 64 lblk ! $10000
ch n-- 64 block swap md5 dump ;
hlp randq rng ! logo pause calkhz
   onesec @ 1000 / mhz ! e ;
mark empty hlp

Press the x key to see the comment block
                                          :;!@ 123
Press F1                                  zj., 4560
                                          x/+- 789?
1 1 123                                 +     x.a
```

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 '64' 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 cf2023 system :

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

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, also runs the editor.

Press F6 to toggle between the current and last edited block.

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 arrows, 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
|-left-arrow	Find previous token name RSN	A	A
f	Find token name RSN	S	S
|-right-arrow	Find next token name RSN	D	D

- RSN = Real Soon Now…

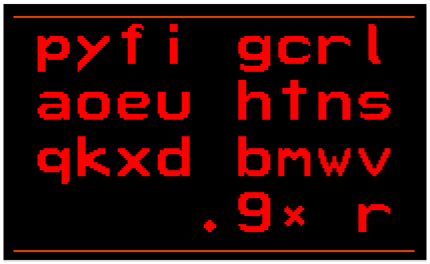
Use the **+** and - keys to select the block to edit (also PgUp and PgDn), or press the space bar to exit the editor and type

256 edit to edit block 256

Use the **l u d r** arrow 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.

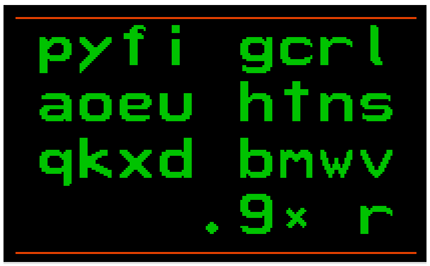
Pressing the keyboard Enter key executes the red word or locates the green word at the cursor.

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 a word to be compiled :

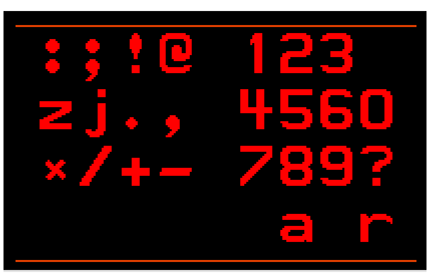


Press the required letters until you have finished entering the word, then press the space bar (‘9’).

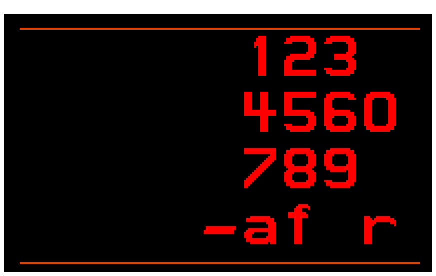
Repeat for the next word.

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.

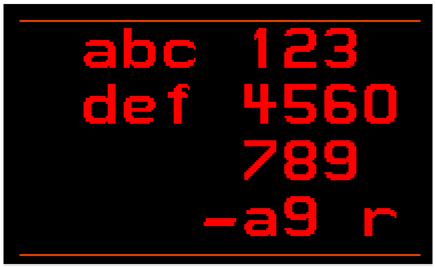
The AltGr key, labled * in the keypad toggles between the main alpha and Alternate alpha text entry mode :



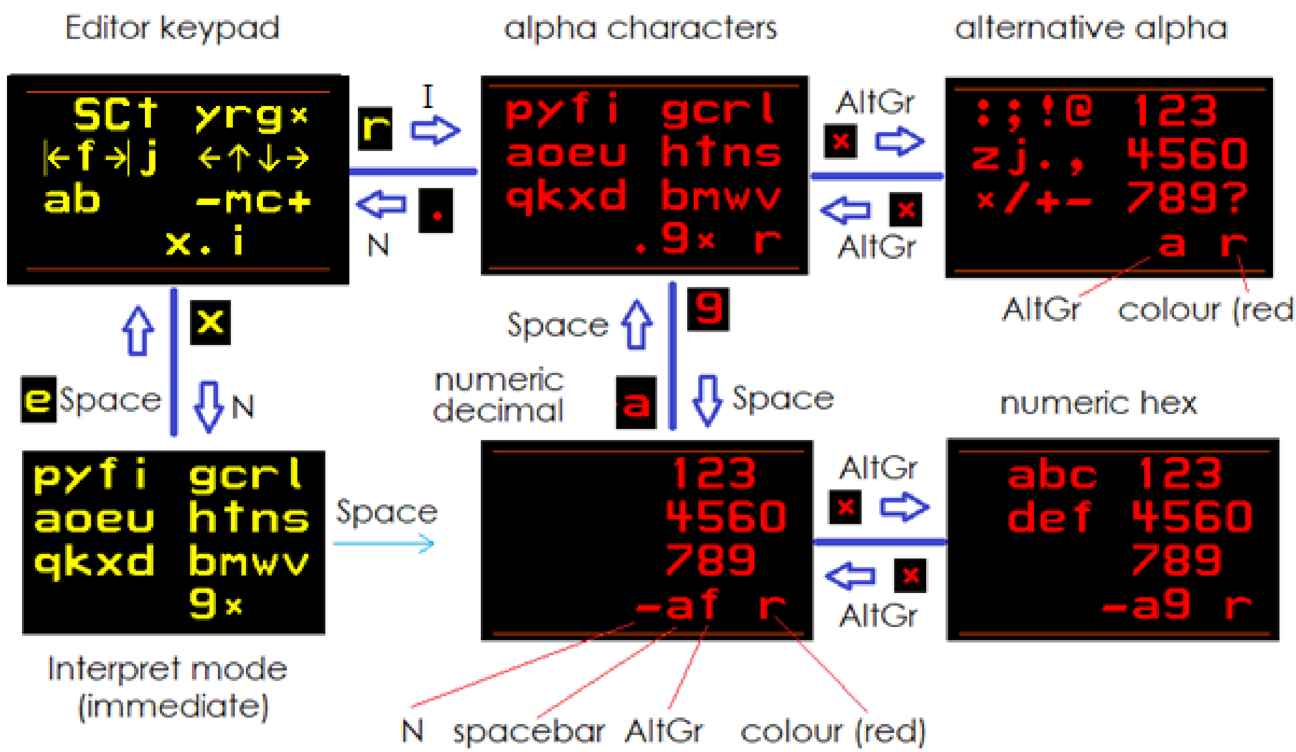
Press the required letters until you have finished entering the word, then press the AltGr key to return to alpha mode, then the space bar (‘9’) as described above.



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 similar functionality.

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

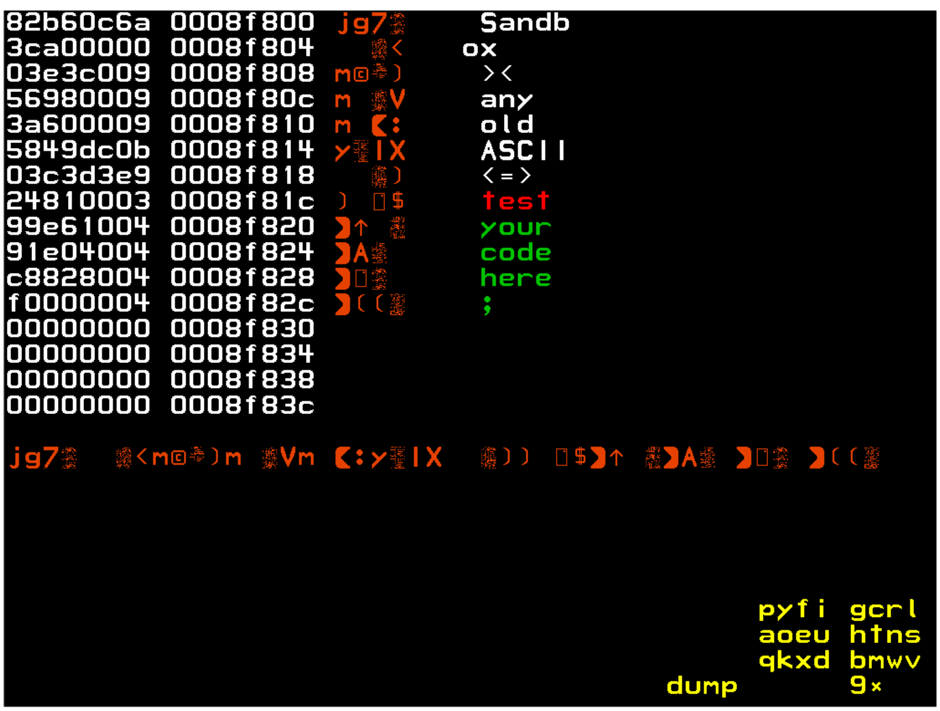
When you have found and/or edited the block that you would like to run, leave the editor by pressing the space bar. You are now in Interpret mode (no more horizontal lines).

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

colorForth and ASCII

The colorForth keypad has only 48 characters – this edition allows three ASCII (or UTF-8) characters to be encoded into a standard colorForth 32 bit token. This increases compatilbility with other Forth systems.

```
; ASCII / UTF8 support. If the first Shannon-Fano encoded letter is a 4 bit NULL,
; display the next 24 bits as three ASCII characters.
; $03e3c009 is displayed as '><'
```



cf2023.nasm ASCII Support code

This is the code to support decoding and display of ASCII characters.

```
        lowercase:   ; display a white text word in normal lower-case letters
            call white
        showSF_EDI_: ; ( -- ) \ display a Shanon-Fano encoded token pointed to by  edi  in the
current colour
            _DUP_
            mov _TOS_, [ ( edi * 4 ) - 0x04 ]   ; fetch the next token – drops through to
showShannonFano

        showShannonFano:    ; ( token -- ) \ display the Shannon-Fano encoded token on TOS
            ; ASCII / UTF8 support. If the first Shannon-Fano encoded letter is a 4 bit NULL,
            ; display the next 24 bits as three ASCII characters.
            mov _SCRATCH_, _TOS_            ; save the token value
            and _SCRATCH_, 0xF0000000
            cmp _SCRATCH_, 0x00000000
            jnz .forward
                ; display as three ASCII characters
                mov _SCRATCH_, _TOS_

                mov _TOS_, _SCRATCH_
                shr _TOS_, 20
                and _TOS_, 0x000000FF
                jz .null_terminator
                    _DUP_
                    call emit_

                mov _TOS_, _SCRATCH_
                shr _TOS_, 12
                and _TOS_, 0x000000FF
                jz .null_terminator
                    _DUP_
                    call emit_

                mov _TOS_, _SCRATCH_
                shr _TOS_, 4
                and _TOS_, 0x000000FF
                jz .null_terminator
                    _DUP_
                    call emit_

                ; arrive here if an ASCII character is an ASCII NULL, or if all three have been
emitted
                .null_terminator:
                call space_                 ; display a space character at the end of the word
                _DROP_
                ret

            .forward:

            ; display as Shannon-Fano encoded token name
            and _TOS_, byte -0x10   ; and _TOS_, 0xFFFFFFF0 ignore token colour when displaying
the letters

        lowercasePrimitive: ; ( token -- ) \ display the given Shanon-Fano encoded word in the
current colour
            call unpack
            jz lowercasePrimitiveEnd
            call emitSF_
            jmp lowercasePrimitive
        lowercasePrimitiveEnd:
            call space_
            _DROP_
            _DROP_
            ret
```

Useful Commands

Command	Action
e	Run the Editor displaying the last block edited
64 edit	Run the colorForth Editor displaying block 64
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 64, 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.
staks	Display the four task's stacks. 'U' means Unused, '.' means used. Checks that the stacks are not growing or shrinking…

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 CDP1802 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 the Moisture 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, for 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 Moore's 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 cf2023 – there are a lot of changes, most of them for the better :

The font is now in ASCII order, even though Shannon-Fano encoded names are still used internally.

Byte addresses are used throughout, 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-thMe 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."

Chuck Moore from Fireside chat Nov 2020

https://www.youtube.com/watch?v=81bkIqPpe0g 34:33

Sean: Why do you think Forth has failed to go mainstream?

Chuck: I used to think about that a great deal.

I think that what happens in the world is largely a matter of luck, unpredictable, unrelated to quality or cost - it's just a fad, and some fads stick, and some fads don't.

I don't think that there is any need for Forth to become a mainstream language.

I think it is a very excellent niche language, and a personal language.

It can be used to advantage by people like us, without requiring that the whole world give us permission.

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.

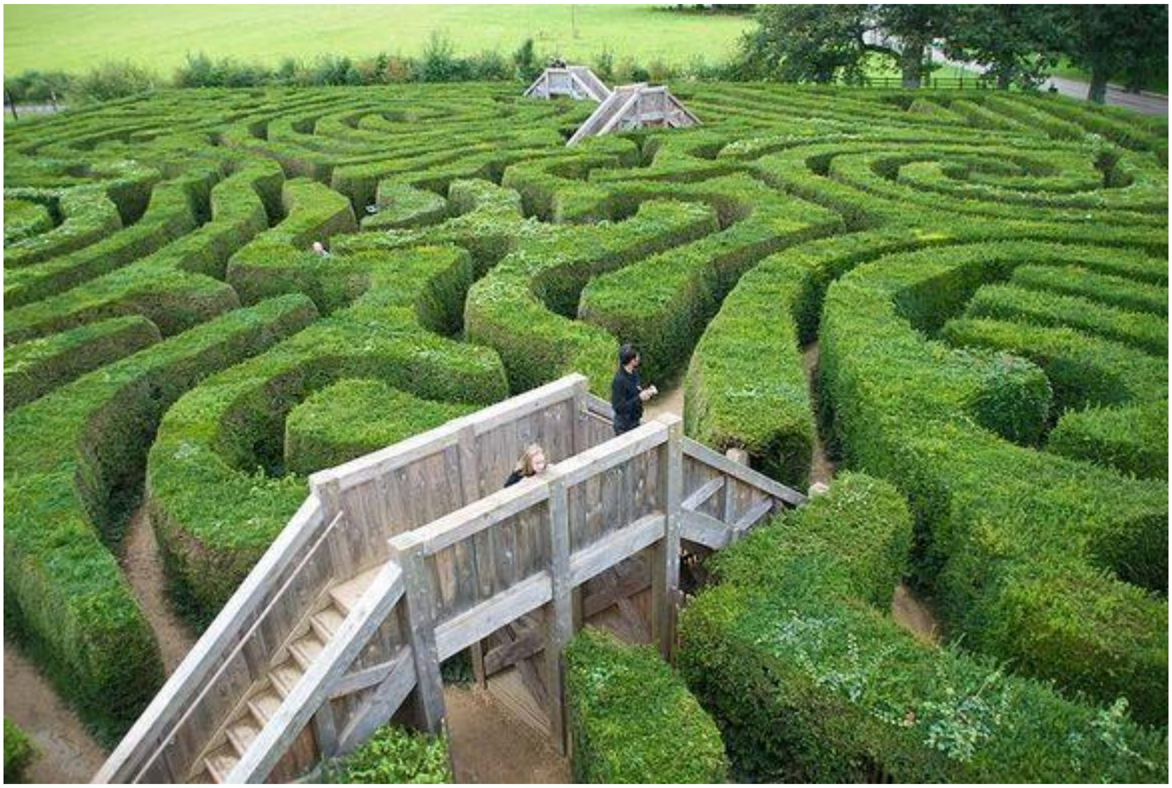


Figure 2 The Hedge Maze at Longleat, UK

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 could sell more software because it could then run on 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 it, 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 it 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 driver 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 (instead of top left).
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 B NASM Source Code

```
; cf2023.nasm 2023 Apr 04  MD5 "roomed-zebra"
; Fixing Magenta Variables
; Added "locate" - pressing the Enter key executes the red word or locates the green word at the cursor
; "chm" ( check MD5 ) in colorForth shows "ff" "la:rrr" "nos" "td"
; colorForth for 80x86 PC for NASM , with 1024x768 and 800x600 graphics options
; Adapted by Howerd Oakford from code by :
; Chuck Moore : inventor, MASM
; Mark Slicker : ported to GNU Assembler
; Peter Appelman : ported to NASM with qwerty keyboard
; Josh Grams : multitasker
; John Comeau : BIOS boot from ClusterFix
; Marco Nicola : 2drop and 2dup bug fix
; Miroslav Popov : keyboard instead of keypad typos
; and others... Thanks to all!!!
; Feedback welcome : howerd@inventio.co.uk www.inventio.co.uk

; %define NOT_BOCHS ; Bochs cannot handle resetting of the PIT chips, so we disable this to allow
operation in Bochs

; CPU 386 ; Assemble instructions for the 386 instruction set

%define FORCE_800x600_VESA  0       ; true to force 800 x 600 x 16 bits for testing in  bochs

%define START_BLOCK_NUMBER      64  ; must be an even number. Note: if you change this you must shift the
blocks in cf2022Ref.img accordingly!

%define SIZE_OF_FONT_IN_BLOCKS  12
%define OFFSET_OF_FONT          ( ( START_BLOCK_NUMBER - SIZE_OF_FONT_IN_BLOCKS ) * 0x400 )
%define LAST_BLOCK_NUMBER       511 ; must be an odd number

%define SECTORS_TO_LOAD ( ( LAST_BLOCK_NUMBER + 1 ) * 2 )   ; number of 512 byte sectors

%define BITS_PER_PIXEL  16  ; MUST BE 16 !!! display pixel sizes, colour depth = 16 bit ( 2 bytes )

; for the maximum supported screen : 1024 x 76go8 pixels :
%define MAX_SCREEN_WIDTH ( 1024 )   ; maximum screen width in pixels
%define MAX_SCREEN_HEIGHT ( 768 )   ; maximum screen height in pixels

%define BYTES_PER_PIXEL ( BITS_PER_PIXEL / 8 )

PIXEL_SHIFT equ 1           ; how many bits to shift to scale by BYTES_PER_PIXEL

; Memory Map
; start   length
; 0x100000 ....    RAM
; 0xC0000 0xFFFFF BIOS video ROM - its not RAM!
; 0xB8000 0x08000 BIOS video RAM
; 0x10000 0xA8000 cf2022.img file is copied here
; 0x0F000 0x01000 BIOS shadow RAM - its OK to use this if we do not call the video BIOS
; 0x0A000 0x05000 BIOS video RAM - do not use until we have changed video mode
; 0x07c00 0x00200 BPB Boot sector after loading by BIOS
; 0x07c0b <----- di points here, the BPB ( + offset )  and variables ( - offset ) are accessed via [di]
; 0x07b8c 0x00080 variables referenced via [di], followed by BPB variables referenced via [di]
; 0x07800         Stacks, size = 0x0200 each , growing downwards
; 0x02000 0x06800 SECTOR_BUFFER
; 0x00000 0x02000 BIOS RAM

%define SECTOR_BUFFER        0x00002000                     ; buffer for disk reads and writes
%define SECTOR_BUFFER_SIZE   0x4800                         ; 18 K bytes, 36 x 512 byte sectors
%define INTERRUPT_VECTORS    ( SECTOR_BUFFER - 0x0400 )      ; the IDT register points to these interrupt
vectors
%define VESA_BUFFER          ( INTERRUPT_VECTORS - 0x0400 ) ; for the VESA mode information
%define DAP_BUFFER           ( VESA_BUFFER   - 0x0020 )     ; 0x1BE0 for the Int 0x13 Disk Address Packet
(DAP)
%define DISK_INFO            ( DAP_BUFFER    - 0x0020 )     ; for the Int 0x13 AH=08h get info
%define IDT_AND_PIC_SETTINGS ( DISK_INFO     - 0x0040 )     ; bytes 0x00 - 0x05 SIDT value, 0x06 PIC1 IMR
, 0x07 PIC2 IMR values saved at startup
```

```
%define V_REGS           ( IDT_AND_PIC_SETTINGS - 0x0020 ) ; test only - registers before and after thunk
call
%define MD5_OUTPUT_BUFFER    ( V_REGS - 0x0020 )            ; the MD5 hash result

%define TRASH_BUFFER         ( (508 * 0x0400) + 0x10000 )   ; Block 508, saves words deleted while editing

%define PIC_BIOS_IDT_SETTINGS   ( IDT_AND_PIC_SETTINGS )   ; bytes 0x00 - 0x05 SIDT value, 0x06 PIC1 IMR ,
0x07 PIC2 IMR values saved at startup
%define PIC_BIOS_IMR_SETTINGS   ( IDT_AND_PIC_SETTINGS + 6 ) ; bytes 0x00 - 0x05 SIDT value, 0x06 PIC1 IMR
, 0x07 PIC2 IMR

%define PIC_NEW_IDT_SETTINGS    ( IDT_AND_PIC_SETTINGS + 0x10 )   ; bytes 0x00 - 0x05 SIDT value, 0x08 new
PIC1 IMR , 0x09 new PIC2 IMR
%define PIC_NEW_IMR_SETTINGS    ( IDT_AND_PIC_SETTINGS + 0x16 )   ; bytes 0x00 - 0x05 SIDT value, 0x08 new
PIC1 IMR , 0x09 new PIC2 IMR

%define IDT_AND_PIC_SETTINGS_PAD ( IDT_AND_PIC_SETTINGS + 0x20 )

%define vesa_BytesPerScanLine   ( VESA_BUFFER + 0x0E )  ; screen width ( number of horizontal pixels )
%define vesa_XResolution        ( VESA_BUFFER + 0x12 )  ; screen width ( number of horizontal pixels )
%define vesa_YResolution        ( VESA_BUFFER + 0x14 )  ; screen height ( number of vertical pixels )
%define vesa_BitsPerPixel       ( VESA_BUFFER + 0x19 )  ; bits per pixel
%define vesa_SavedMode          ( VESA_BUFFER + 0x1E )  ; "Reserved" - we save the VESA mode here
%define vesa_PhysBasePtr        ( VESA_BUFFER + 0x28 )  ; address of linear frame buffer

%define BOOTOFFSET      0x7C00

%assign RELOC_BIT 16                            ; the relocation address must be a power of 2
%assign RELOCATED 1 << RELOC_BIT                ; 0x10000

; ******************************************************************************
; Data and Return stack allocation, four pairs of data and return stacks
; Note : the return stack must be in the lowest 64K byte segment, for the BIOS calls to work
; ******************************************************************************
%define DATA_STACK_SIZE_0   $0400   ;
%define DATA_STACK_SIZE_1   $0500   ; must be > $400 for colorForth "Life" program to work
%define DATA_STACK_SIZE_2   $0100   ;
%define DATA_STACK_SIZE_3   $0100   ;
%define DATA_STACK_SIZE_GAP $0100   ; leave space under the last data stack to check for underflow

%define RETURN_STACK_SIZE   $0100   ;

; return stacks
%define RETURN_STACK_0      ( $7800 )        ; top of stack memory area
%define RETURN_STACK_1      ( RETURN_STACK_0 - RETURN_STACK_SIZE )
%define RETURN_STACK_2      ( RETURN_STACK_1 - RETURN_STACK_SIZE )
%define RETURN_STACK_3      ( RETURN_STACK_2 - RETURN_STACK_SIZE )

; data stacks
%define DATA_STACK_0        ( RETURN_STACK_3 - RETURN_STACK_SIZE )
%define DATA_STACK_1        ( DATA_STACK_0   - DATA_STACK_SIZE_0 ) ; BIG data stack for the show task
%define DATA_STACK_2        ( DATA_STACK_1   - DATA_STACK_SIZE_1 )
%define DATA_STACK_3        ( DATA_STACK_2   - DATA_STACK_SIZE_2 )
%define STACK_MEMORY_START  ( DATA_STACK_3   - DATA_STACK_SIZE_3 - DATA_STACK_SIZE_GAP )

; four pairs of stacks, one for each task
%define TOTAL_STACK_SIZE    ( RETURN_STACK_0 - STACK_MEMORY_START )

%define STACK_ANALYSIS_BUFFER    ( STACK_MEMORY_START - 0x200 )

; ******************************************************************************
; ******************************************************************************

%define _TOS_  eax
%define _TOS_x_  ax
%define _TOS_l_  al

%define _SCRATCH_  ebx
%define _SCRATCH_x_  bx
%define _SCRATCH_l_  bl
```

```
%define _MOV_TOS_LIT_  (0xB8)   ; the opcode for mov eax, 32_bit_literal  (in next 32 bit cell)

%macro _DUP_  0                 ; Top Of Stack is in the  _TOS_  register
    lea esi, [ esi - 4 ]        ; pre-decrement the stack pointer, does not set the flags
    ; sub esi, byte 0x04        ; pre-decrement the stack pointer, also sets the flags
    mov [ esi ], _TOS_          ; copy the Top Of Stack ( TOS ) register to Second On Stack ( on the real
stack )
%endmacro

%macro _SWAP_  0
    xchg _TOS_, [ esi ]
%endmacro

%macro _OVER_  0
    sub esi, byte 0x04          ; pre-decrement the stack pointer
    mov [ esi ], _TOS_          ; copy the Top Of Stack ( TOS ) register to Second On Stack ( on the real
stack )
    mov _TOS_, [ esi + 4 ]
%endmacro

%macro _DROP_ 0
    lodsd  ; loads a 32 bit dword from [ds:esi] into  eax  then increments  esi  by 4
%endmacro
; Note : stosd ; stores  eax  into the location pointed to by  edi  then increments  edi  by 4
; Note also : eax is used as  _TOS_  ( Top Of Stack )

; Define the location of the wordlists in RAM
%define START_OF_RAM    0x00468000
%define WORDLIST_SIZE      0x2000
%define ForthNames      START_OF_RAM                ; copied to RAM here from ROM ( i.e. boot program )
version
%define ForthLocates    ( ForthNames     + WORDLIST_SIZE ) ; copied to RAM here from ROM ( i.e. boot
program ) version
%define ForthJumpTable  ( ForthLocates   + WORDLIST_SIZE ) ; copied to RAM here from ROM ( i.e. boot
program ) version
%define MacroNames      ( ForthJumpTable + WORDLIST_SIZE ) ; copied to RAM here from ROM ( i.e. boot
program ) version
%define MacroJumpTable  ( MacroNames     + WORDLIST_SIZE ) ; copied to RAM here from ROM ( i.e. boot
program ) version
%define MacroLocates    ( MacroJumpTable + WORDLIST_SIZE ) ; copied to RAM here from ROM ( i.e. boot
program ) version
; some more room...
%define H0              0x00480000 ;  ( MacroLocates + WORDLIST_SIZE )   ; initial value of the dictionary
pointer

%define SECTOR  512     ; bytes per floppy sector
%define HEADS   2       ; heads on 1.44M floppy drive
%define SECTORS 18      ; floppy sectors per track
%define CYLINDER (SECTOR * SECTORS * HEADS)
%define CELL 4          ; bytes per cell
%define DEBUGGER 0xe1   ; port to hardware debugger?

; int 0x13 Disk Address Packet (DAP) pointed to by  si  :
%define o_Int13_DAP_size           ( 0x00 ) ; 2  0x0010
%define o_Int13_DAP_num_sectors    ( 0x02 ) ; 2  0x0001
%define o_Int13_DAP_address        ( 0x04 ) ; 2  0x2000
%define o_Int13_DAP_segment        ( 0x06 ) ; 2  0x0000
%define o_Int13_DAP_LBA_64_lo      ( 0x08 ) ; 4  0x00000028
%define o_Int13_DAP_LBA_64_hi      ( 0x0C ) ; 4  0x00000000
; extended DAP values
%define o_Int13_DAP_readwrite      ( 0x10 ) ; 2  0x0000
%define o_Int13_DAP_saved_DX       ( 0x12 ) ; 2  0x0000
%define o_Int13_DAP_returned_AX    ( 0x14 ) ; 2  0xHH00 see AH Return Code below
%define o_Int13_DAP_returned_carry_flag ( 0x16 ) ; 2  0x0000
%define o_Int13_DAP_saved_CHS_CX   ( 0x18 ) ; 2  0x0000
%define o_Int13_DAP_saved_CHS_DX   ( 0x1A ) ; 2  0x0000

%macro LOAD_RELATIVE_ADDRESS  1
    mov _TOS_, ( ( ( %1 - $$ ) + RELOCATED ) )
```

```
%endmacro

; emit the given following character
%macro EMIT_IMM 1
;    push esi
    _DUP_
    mov _TOS_, %1
    call emit_
;    pop esi
%endmacro

; ****************************************************************************
; Registers used
; ****************************************************************************
; _TOS_ is the top stack item ( eax --> ebx )
; esp the call ... ret  return stack pointer
; edi  dictionary pointer ( H --> : HERE ( -- a )   H @ ; )
; esi is the stack pointer, also needed by lods and movs
; e.g. lodsd  loads a 32 bit dword from [ds:esi] into _TOS_, increments  esi  by 4
; ebx  scratch register
; ecx  counter and scratch register
; edx  run-time pointer (?), "a register" used by  a!  , otherwise scratch register
; ebp  variable pointer register
; "ds" = selector 0x10 ==> 0x0000:0000
; "es" = selector 0x10 ==> 0x0000:0000
; "ss" = selector 0x10 ==> 0x0000:0000

; colours RGB in 16 bits
colour_background   equ 0x0000
colour_yellow       equ 0xFFE0
colour_black        equ 0x0000
colour_red          equ 0xF800
colour_green        equ 0x0600
colour_cyan         equ 0x07FF
colour_white        equ 0xFFFF
colour_light_blue   equ 0x841F
colour_silver       equ 0xC618
colour_magenta      equ 0xF81F
colour_magentaData  equ 0xD010
colour_blue         equ 0x001F
colour_orange       equ 0xE200
colour_dark_yellow  equ 0xFFE0
colour_dark_green   equ 0x07C0
colour_PacMan       equ 0xE200
colour_blockNumber  equ 0xE200

[BITS 16]                           ; Real Mode code (16 bit)

org RELOCATED

start:
codeStart:
    jmp  main_16bit    ; 0x03 bytes |  EB 58 90  00 Jump to boot code
    times 3 - ($ - $$) nop        ; fill with 1 or 0 no-ops to address 3
    ; BIOS boot parameter table = 0x25 bytes
    db 'cf2022 0'      ; 03 Eight byte OEM name
    dw 0x0200          ; 11 Number of Bytes Per Sector
    db 0x08            ; 13 Number of Sectors Per Cluster
    dw 0x05E0          ; 14 Number of Reserved Sectors until the FAT
    db 0x02            ; 16 Number of Copies of FAT : always = 2
    dw 0x0000          ; 17 Maximum number of Root Directory Entries
    dw 0x0000          ; 19 Not used for FAT32
    db 0xF8            ; 21 Media type F0 = 1.44M 3.5 inch floppy disk, F8 = hard disk changes 2022 Mar14
    dw 0x0000          ; 22 Sectors Per FAT for FAT12 and FAT16 - not used for FAT32
    dw 0x003F          ; 24 Sectors per Track
    dw 0x00FF          ; 26 Number of heads
    dd 0x00000038      ; 28 Hidden sectors preceding the partition that contains this FAT volume
    dd 0x007477C8      ; 32
    dd 0x00001D10      ; 36 Sectors Per FAT for FAT32
    dw 0x0000          ; 40
```

```
    dw 0x0000          ; 42
    dd 0x00000002      ; 44 Start of all directories, including root.
    dw 0x0001          ; 48
    dw 0x0006          ; 50 Offset in sectors from this sector to the backup BPB sector
;    times 12 db 0      ; 0x0C bytes |  00 00 00 00 00 00 00 00 00 00 00 00  52
;    db 0x00            ; 64
;    db 0x00            ; 65
;    db 0x29            ; 66 Extended Boot Signature
;    dd 0x44444444      ; 67 serial number
;    db 'colorForth '   ; 71 Eleven byte Volume Label
;    db 'cFblocks'      ; 82 Eight byte File System name

; ******************************************************************************
; ******************************************************************************

align 8, nop    ; has to be aligned to 8 for GDT
    ; Note : we are NOT using null descriptor as GDT descriptor, see: http://wiki.osdev.org/GDT_Tutorial
    ; "The null descriptor which is never referenced by the processor. Certain emulators, like Bochs, will
complain about limit exceptions if you do not have one present.
    ; Some use this descriptor to store a pointer to the GDT itself (to use with the LGDT instruction).
    ; The null descriptor is 8 bytes wide and the pointer is 6 bytes wide so it might just be the perfect
place for this."

gdt:                                ; the GDT descriptor
    dw gdt_end - gdt - 1            ; GDT limit
    dw gdt0 + BOOTOFFSET            ; pointer to start of table, low 16 bits
    dw 0 , 0                        ; the high bits of the longword pointer to gdt

gdt0:                               ; null descriptor
    dw 0    ; 0,1 limit 15:0
        dw 0    ; 2,3 base  15:0
        db 0    ; 4   base  23:16
        db 0    ; 5   type
        db 0    ; 6   limit 19:16, flags
        db 0    ; 7   base  31:24
code32p_SELECTOR_0x08 equ $ - gdt0
; bytes   1 0     3 2     5 4     7 6
    dw 0xFFFF, 0x0000, 0x9A00, 0x00CF   ; 32-bit protected-mode code, limit 0xFFFFF
data32p_SELECTOR_0x10 equ $ - gdt0
    dw 0xFFFF, 0x0000, 0x9200, 0x00CF   ; 32-bit protected-mode data, limit 0xFFFFF
code16r_SELECTOR_0x18 equ $ - gdt0
    dw 0xFFFF, 0x0000, 0x9A00, 0x0000   ; 16-bit real-mode code, limit 0xFFFFF
data16r_SELECTOR_0x20 equ $ - gdt0
    dw 0xFFFF, 0x0000, 0x9200, 0x0000   ; 16-bit real-mode data, limit 0xFFFFF
gdt_end:

; ******************************************************************************
; ******************************************************************************

; align to 4 so we can access variables from high-level Forth
align 4, nop

data_area:   ; data area begins here

bootsector:                         ; LBA of boot sector
    dd 0

; save disk information, cylinder, sector, head and drive from BIOS call
driveinfo_Drive_DX:                 ; use low byte to store boot Drive into from BIOS DL
    dw 0

driveinfo_CX:         ; [7:6] [15:8][7] logical last index of cylinders = number_of - 1 (because index
starts with 0)
                      ; [5:0][7] logical last index of sectors per track = number_of (because index starts
with 1)
    dw 0

; cylinders, sectors, heads of boot drive
; low word: high byte is head
; high word: cylinder and sector: C76543210 C98S543210
```

```
driveinfo_Cylinder:
    db 0
driveinfo_Head:
    db 0
driveinfo_SectorsPertrack:
    dw 0

align 4, nop

destination:
    dd RELOCATED

dispPtr:
    dd 0x00000140

v_bytesPerLine:
    dd 0x00

v_scanCode:
    dd 0x00

align 4

; ******************************************************************************
; the main program called from initial 16 bit mode
; ******************************************************************************

main_16bit:

    cli                         ; clear interrupts
                                ; turns out we don't need interrupts at all, even when using BIOS routines
                                ; but we need to turn them off after disk calls because BIOS leaves them
on

    push si                     ; need to transfer SI to unused register BX later

; note: cannot touch DX or BP registers until we've checked for partition boot
; (SI could be used as well as BP but we use SI for relocation)

 ;see mbrboot.nasm
                                ; Note : relocate the bootblock before we do anything else
    pop bx                      ; we cannot use the current stack after changing SS or SP
                                ; ... because mbrboot.nasm places stack at 0x7c00, in SECTOR_BUFFER
                                ; and we cannot use BP because its default segment is SS
    xor ax, ax
    mov ds, ax
    mov es, ax

    mov si, BOOTOFFSET
    mov di, SECTOR_BUFFER
    mov sp, di
    mov cx, 0x100
    rep movsw                   ; note that this instruction doesn't change AX , it moves DS:SI to ES:DI
and increments SI and DI

    mov ss, ax                  ; stack segment also zero
    mov ah, 0xb8                ; video RAM
    mov gs, ax                  ; store in unused segment register

    lgdt [gdt - $$ + BOOTOFFSET]

    call SetupUnrealMode     ; gs and ss must be initialized before going to Unreal Mode

; ******************************************************************************
; Enable the A20 address line, otherwise all odd 1 MByte pages are disabled
; Using the "PS/2 Controller" or 8042 "Keyboard controller"
; ******************************************************************************
    ; from  http://wiki.osdev.org/%228042%22_PS/2_Controller#Step_1:_Initialise_USB_Controllers
    ; Write a command to the on-board 8042 "Keyboard controller" port 0x64 :
    ; 0x20     Read "byte 0" from internal RAM     Controller Configuration Byte
```

```
    ; 0x21 to 0x3F    Read "byte N" from internal RAM (where 'N' is the command byte & 0x1F)
    ; 0x60    Write next byte to "byte 0" of internal RAM (Controller Configuration Byte)
    ; 0x61 to 0x7F    Write next byte to "byte N" of internal RAM (where 'N' is the command byte & 0x1F)
    ; 0xA7    Disable second PS/2 port
    ; 0xA8    Enable second PS/2 port
    ; 0xA9    Test second PS/2 port
    ;     0x00 test passed
    ;     0x01 clock line stuck low
    ;     0x02 clock line stuck high
    ;     0x03 data line stuck low
    ;     0x04 data line stuck high
    ; 0xAA    Test PS/2 Controller
    ;     0x55 test passed
    ;     0xFC test failed
    ; 0xAB    Test first PS/2 port
    ;     0x00 test passed
    ;     0x01 clock line stuck low
    ;     0x02 clock line stuck high
    ;     0x03 data line stuck low
    ;     0x04 data line stuck high
    ; 0xAC    Diagnostic dump (real all bytes of internal RAM)    Unknown
    ; 0xAD    Disable first PS/2 port     None
    ; 0xAE    Enable first PS/2 port  None
    ; 0xC0    Read controller input port  Unknown (none of these bits have a standard/defined purpose)
    ; 0xC1    Copy bits 0 to 3 of input port to status bits 4 to 7    None
    ; 0xC2    Copy bits 4 to 7 of input port to status bits 4 to 7    None
    ; 0xD0    Read Controller Output Port     Controller Output Port (see below)
    ; 0xD1    Write next byte to Keyboard Controller Output Port Note: Check if output buffer is empty
first
    ; 0xD2    Write next byte to first PS/2 port output buffer
    ; 0xD3    Write next byte to second PS/2 port output buffer
    ; 0xD4    Write next byte to second PS/2 port input buffer
    ; 0xF0 to 0xFF  Pulse output line low for 6 ms.
    ;     Bits 0 to 3 are used as a mask (0 = pulse line, 1 = do not pulse line) and correspond to 4
different output lines.
    ;     Bit 0 is the "reset" line, active low.
    mov al, 0xD1    ; 0xD1 = Write next byte to Keyboard Controller Output Port
    out 0x64, al    ; On-board controller Command Write
.back:
    in al, 0x64
    and al, 0x02
    jnz .back
    mov al, 0x4B
    out 0x60, al

; ******************************************************************************
; Get disk drive parameters from the BIOS
; ******************************************************************************

    mov di, (data_area - $$ + BOOTOFFSET)   ; setup the data index pointer
    xor eax, eax
    bts eax, 16                     ; in case NOT booted from partition: sector 1, head 0, cylinder 0
    or dh, dh                       ; booted from partition?
    jz .forward3
    mov eax, [ bx + 8 ]             ; SI (now BX) contains pointer to partition record
    mov [ byte di + (bootsector - data_area) ], eax     ; offset 8 was LBA of first absolute sector
    mov eax, [bx]                   ; CHS of first sector in partition
.forward3:
    mov al, dl                      ; bootdrive into AL
    mov [ word di + ( driveinfo_Drive_DX - data_area) ], eax    ; save the Drive info from BIOS
    mov ah, 8                       ; get drive parameters
    push es                         ; this operation messes with ES
    push di                         ; and DI
    mov di, DISK_INFO               ; point di at the table returned by this software interrupt
    int 0x13
    jc $                            ; stop here on error

    call ReSetupUnrealMode
    pop di
    pop es
```

```
; *****************************************************************************
; load the bootdisk into both low and high RAM
; *****************************************************************************

    mov [ byte di + ( driveinfo_Cylinder - data_area) ], dx            ; heads in high byte
    and cl, 0x3F                    ; we don't care about two high bits of cylinder count
    mov [ byte di + ( driveinfo_SectorsPertrack - data_area) ], cx     ; cylinders and sectors/track
    mov dx, [ byte di + ( driveinfo_Drive_DX - data_area) ]         ; restore dl Drive value from BIOS, dh
= 0
;    mov dl, 0x00    ; try this 2022 Mar 14
    mov cx, [ di + ( driveinfo_CX - data_area) ]              ; restore cl value, ch = 0
    mov si, SECTORS_TO_LOAD

    mov bx, SECTOR_BUFFER            ; relocate the sector we are running from
    call relocate

    mov bx, BOOTOFFSET              ; we will fix this below by adding 0x200
                                    ; remember the sector is 1-based, head and cylinder both 0-based

.nextsector:
    inc cl
    dec si
    jz setVideoMode    ; success, so setup the video now...

.bootload:
    mov ax, 0x201                   ; read 1 sector
    add bh, 0x02                    ; into next available slot in RAM
    jnz .forward
    sub bh, 0x02                    ; at 0x10000 we go back to 0xfe00
.forward:
    int 0x13
    call ReSetupUnrealMode
    jc $                            ; stop here on error
    call relocate
    mov al, cl
    and al, 0x3F                    ; low 6 bits
    cmp al, [ byte di + ( driveinfo_SectorsPertrack - data_area) ]
    jnz .nextsector
    inc dh                          ; next head
    cmp dh, [ byte di + ( driveinfo_Head - data_area) ]
    jna .forward2                   ; not JNZ, the head index is 1 less than head count
    xor dh, dh
    inc ch                          ; next cylinder
    jnz .forward2
    add cl, 0x40                    ; bit 8 of cylinder count
.forward2:
    and cl, 0xC0                    ; clear sector count, low 6 bits of cl
    jmp short .nextsector

; *****************************************************************************
; *****************************************************************************
; Start here after loading the program
; *****************************************************************************
; *****************************************************************************

; From : VESA BIOS EXTENSION (VBE) Core Functions Standard Version: 3.0 Date: September 16, 1998
; Mandatory information for all VBE revisions
; dw ModeAttributes         ; 0x00 mode attributes
; db WinAAttributes         ; 0x02 window A attributes
; db WinBAttributes         ; 0x03 window B attributes
; dw WinGranularity         ; 0x04 window granularity
; dw WinSize                ; 0x06 window size
; dw WinASegment            ; 0x08 window A start segment
; dw WinBSegment            ; 0x0A window B start segment
; dd WinFuncPtr             ; 0x0C real mode pointer to window function
; dw BytesPerScanLine       ; 0x10 bytes per scan line              <--------------
; Mandatory information for VBE 1.2 and above
; dw XResolution            ; 0x12 horizontal resolution in pixels   <-------------- scrnw
; dw YResolution            ; 0x14 vertical resolution in pixels     <-------------- scrnh
```

```
; db XCharSize              ; 0x16 character cell width in pixels
; db YCharSize              ; 0x17 character cell height in pixels
; db NumberOfPlanes         ; 0x18 number of memory planes
; db BitsPerPixel           ; 0x19 bits per pixel               <-------------- bpp
; db NumberOfBanks          ; 0x1A number of banks
; db MemoryModel            ; 0x1B memory model type
; db BankSize               ; 0x1C bank size in KB
; db NumberOfImagePages     ; 0x1D number of images
; db Reserved               ; 0x1E reserved for page function       <-------------- mode (we copy it
here)
; Direct Color fields (required for direct/6 and YUV/7 memory models)
; db RedMaskSize            ; 0x1F size of direct color red mask in bits
; db RedFieldPosition       ; 0x20 bit position of lsb of red mask
; db GreenMaskSize          ; 0x21 size of direct color green mask in bits
; db GreenFieldPosition     ; 0x22 bit position of lsb of green mask
; db BlueMaskSize           ; 0x23 size of direct color blue mask in bits
; db BlueFieldPosition      ; 0x24 bit position of lsb of blue mask
; db RsvdMaskSize           ; 0x25 size of direct color reserved mask in bits
; db RsvdFieldPosition      ; 0x26 bit position of lsb of reserved mask
; db DirectColorModeInfo    ; 0x27 direct color mode attributes
; Mandatory information for VBE 2.0 and above
; dd PhysBasePtr            ; 0x28 physical address for flat memory frame buffer  <-------------- vframe
; dd Reserved               ; 0x2C Reserved - always set to 0
; dw Reserved               ; 0x30 Reserved - always set to 0
; Mandatory information for VBE 3.0 and above
; dw LinBytesPerScanLine    ; 0x32 bytes per scan line for linear modes
; db BnkNumberOfImagePages  ; 0x34 number of images for banked modes
; db LinNumberOfImagePages  ; 0x35 number of images for linear modes
; db LinRedMaskSize         ; 0x36 size of direct color red mask (linear modes)
; db LinRedFieldPosition    ; 0x37 bit position of lsb of red mask (linear modes)
; db LinGreenMaskSize       ; 0x38 size of direct color green mask  (linear modes)
; db LinGreenFieldPosition  ; 0x39 bit position of lsb of green mask (linear modes)
; db LinBlueMaskSize        ; 0x3A size of direct color blue mask  (linear modes)
; db LinBlueFieldPosition   ; 0x3B bit position of lsb of blue mask (linear modes)
; db LinRsvdMaskSize        ; 0x3C size of direct color reserved mask (linear modes)
; db LinRsvdFieldPosition   ; 0x3D bit position of lsb of reserved mask (linear modes)
; dd MaxPixelClock          ; 0x3E maximum pixel clock (in Hz) for graphics mode
; times 189 db 0            ; 0x42 remainder of ModeInfoBlock
; End                       ; 0xFF

scanVESA:   ; ( w+h+b -- )  in ax
    mov bx, ax
    push di                         ; save di
    mov cx, ( 0x4117 - 1 )          ; start scanning from the expected VESA mode 0x4117 ( the -1 is
because of the  inc cx  below )
.back:
    inc cl                          ; increment just the bottom byte, we test 0x41xx
    cmp cl, 0x16                    ; scanned from 0x4117 to 0x4116, not found, so show error
    jz .failure
    mov di, VESA_BUFFER             ; buffer for the VESA mode information block
    mov ax, 0x4F01                  ; INT 0x10, AX=0x4F01, CX=mode Get Mode Info
    int 0x10
    cmp al, 0x4F                    ; success code = 0x4F
    jne .back                       ; try the next VESA mode
    mov ax, [di + 0x12]             ; width
    add ax, [di + 0x14]             ; height
    add al, [di + 0x19]             ; bits per pixel
;    adc ah, 0                       ; should not be necessary for the expected result, 0x400+0x300+0x10
    cmp ax, bx                      ; width + height + bits per pixel
    je .success
    jne .back                       ; try the next VESA mode
.failure:                           ; VESA mode not found, so continue
    pop di                          ; restore di
    mov ax, 0                       ; return flag false
    add ax, 0                       ; set the zero flag
    ret
.success:
    mov [ di + ( vesa_SavedMode - VESA_BUFFER ) ], cx ; save the VESA mode in the VESA_BUFFER at offset
0x1E "Reserved"
    mov ax, 1                       ; return flag true
```

```
    add ax, 0                       ; set the zero flag
    pop di                          ; restore di
    ret

setVESA:    ; we found a valid VESA mode

    push ds                         ; clear all flags including Interrupt using DS, known to be zero
    popf                            ; this is necessary to clear T flag also, end register display

    call greet      ; show greeting message

    mov bx, cx
    mov ax, 0x4F02  ; INT 0x10, AX=0x4F02, BX=mode, ES:DI=CRTCInfoBlock Set Video Mode
    int 0x10

    jmp main_32bit

setVideoMode:
%if ( FORCE_800x600_VESA == 0 )     ; test the 800x600 mode in bochs, which supports 1024x768
    mov ax, ( 1024 + 768 + BITS_PER_PIXEL ) ; try the highest resolution first
    call scanVESA                   ; if VESA mode is found, jump to setVESA
    jnz setVESA                     ; success - we found the requested VESA mode
%endif
    mov ax, ( 800 + 600 + BITS_PER_PIXEL ) ; then try a lower resolution
    call scanVESA                   ; if VESA mode is found, jump to setVESA
    jnz setVESA                     ; success - we found the requested VESA mode

;    mov ax, 640 + 480 + BITS_PER_PIXEL  ; then try an even lower resolution
;    call scanVESA                   ; if VESA mode is found, jump to setVESA
;    jnz setVESA                     ; success - we found the requested VESA mode
    jmp showVESAerror               ; we have tried all VESA modes without success, so report an error

; ******************************************************************************
; ******************************************************************************

relocate:                   ; copy 512 bytes from  [bx]  to FS:[destination]
    pusha
    mov cx, 0x200 / 2
    mov si, bx
    mov ebx, [ byte di + ( destination - data_area) ]
.back:
    lodsw               ; load the 16 bit value pointed to by SI into  ax
    mov [fs:ebx], ax    ; Note : the  fs:  uses the 32 bit FS value setup in Unreal Mode to move the data
outside of the 1 Mbyte Real Mode address range
    add ebx, byte +2
    loop .back

    mov [ byte di + ( destination - data_area) ], ebx
    popa
    ret

    ; not used because it is very slow :
; now set up for trap displaying registers on screen during bootup
;    push cs
;    push showstate - $$ + BOOTOFFSET
;    pop dword [word +4]

; ******************************************************************************
; ******************************************************************************
;1. MasterBoot Record - MBR   at Sector     0 (decimal 0)        MBR
; Partition at offset 1BE
;   BootSignature               0
;   Start Head|Sector|Cylinder   1   1     0
;   Partition Type              B  DOS 7.1+
;   End   Head|Sector|Cylinder  FE  3F    3E5
;   BPBsectorNumber                  00  \ was 3F
;   Size of partition (decimal)  16035777 sectors,  8210317824 bytes,  8017889 Ki bytes,  7830 Mi bytes,
8 Gi bytes
; Partition at offset 1CE
;   BootSignature               0
```

```
;   Start Head|Sector|Cylinder   0   0     0
;   Partition Type               0  Empty partition
;   End   Head|Sector|Cylinder   0   0     0
;   BPBsectorNumber                 0
;   Size of partition (decimal)         0 sectors,           0 bytes,  0 Ki bytes,  0 Mi bytes,

; pretend to be a Master Boot Record so that the BIOS will load us
times ( 0x000001BE - ( $ - $$ ) ) db 0x77
    db 0x80, 0x01, 0x01, 0x00, 0x0B, 0xFE, 0xFF, 0xE5, 0x00, 0x00, 0x00, 0x00, 0xC1, 0xAF, 0xF4, 0x00 ;
0x1BE DOS partition 0  working on PC
    db 00, 00, 00, 00, 00, 00, 00, 00   ; 0x1CE first 8 bytes of empty partition 1

SetupUnrealMode:
    ; set the FS segment in "unreal" mode, must be done before the Trap Flag is set in EFLAGS register
    mov eax, cr0
    or al, 1    ; set the "protected mode enable" bit => "unreal mode"
    mov cr0, eax
    push word data32p_SELECTOR_0x10 ; set the FS segment
    pop fs
    dec al      ; clear the "protected mode enable" bit
    mov cr0, eax
    push ds                         ; now set FS to 0
    pop fs

ReSetupUnrealMode:
    push cs                         ; for iret
    pushf                           ; for iret
    pusha
    mov bp, sp
    mov ax, [bp + 16]               ; get flags
;    or ah, 0x01                    ; set Trap Flag, bit 8 in the EFLAGS register ; debug only - very
slow!
    and ah, ~0x02                   ; reset interrupt flag
    xchg ax, [ bp + 20 ]            ; swap flags with return address
    mov [ bp + 16 ], ax             ; return address at top of stack after popa
    popa
    iret

; ******************************************************************************
; ******************************************************************************

times 512 - 2 - ($ - $$) nop        ; fill with no-ops to 55AA at end of boot sector
    db 0x55 , 0xAA  ; boot sector terminating bytes

; ******************************************************************************
; End of Boot Sector
; ******************************************************************************

; ******************************************************************************
; Show the user a null terminated string - writes directly into video RAM
; ******************************************************************************

displayString:

    ; restore the pointer to screen memory into di
    mov di, (data_area - $$ + BOOTOFFSET)
    mov ax, [ di + ( dispPtr - data_area) ]
    mov di, ax

    push es             ; save es
    mov ax, 0xb800      ; video RAM segment
    mov es, ax

backhere2:
    lodsb               ; loads a byte from [ds:si] into al, then increments  si
    cmp al, 0
    jz forward1         ; If al = 0 then leave the loop
    mov ah, 0x0D        ; text colour, magenta on black background
    stosw               ; stores  ax  into  [es:di]  then increments  di
    jmp backhere2
```

```
forward1:
    ; save the pointer to screen memory from di
    mov ax, di
    mov di, (data_area - $$ + BOOTOFFSET)
    mov [ di + ( dispPtr - data_area) ], ax
    pop es          ; restore es
    ret

; display a string then Wait for a key press
displayStringW:

    pusha
    call displayString

    xor  ax, ax     ; wait for and get a key press ( AX = 0 )
    int  0x16       ; BIOS interrupt Read a Key From the Keyboard
    popa
    ret

; msg_greeting2:
;     db ' Press any key : ' , 0x00

msg_VESAerror:
    db 'No valid VESA mode found! ' , 0x02, 0x00
;    db ' No VESA mode ' , 0x02, 0x00

[BITS 16]                           ; Real Mode code (16 bit)

showVESAerror:
    call greet
    push si
    mov word [ di + ( dispPtr - data_area) ] , 0x000001E0    ; line 3 0x50 x 2 x 3 = 0x1E0
    mov si, ( msg_VESAerror - $$ + BOOTOFFSET )  ; string to display
    call displayStringW
    pop si
    ret

greet:     ; jump here to show 16 bit version text
    push si
    mov word [ di + ( dispPtr - data_area) ] , 0x00000140    ; line 2 0x50 x 2 x 2 = 0x140
    mov si, ( version - $$ + BOOTOFFSET )  ; string to display
    call displayString
;    mov si, ( msg_greeting2 - $$ + BOOTOFFSET )  ; string to display
;    call displayStringW
    pop si
    ret

; ******************************************************************************
; the main program in 32 bit ( protected ) mode
; ******************************************************************************

main_32bit:

    call setProtectedModeAPI        ; called from 16 bit code, returns in 32 bit code

[BITS 32]                           ; Protected Mode code (32 bit) - assemble for 32 bit mode from now on

    mov esp, RETURN_STACK_0         ; setup the return stack pointer
    mov esi, ( DATA_STACK_0 + 4 )   ; setup our data stack pointer

    call save_BIOS_idt_and_pic      ; to be restored later, when making BIOS calls
    call init_default_PIC_IMRs      ; set the default values and copy the BIOS Interrupt Vectors to our
new table
    _DUP_
    mov _TOS_, INTERRUPT_VECTORS
    call lidt_                      ; Load the new Interrupt Descriptor Table

    jmp dword warm

; ******************************************************************************
```

```
; calculate Cylinder, Head and Sector from zero-based sector number
; see http://teaching.idallen.com/dat2343/00f/calculating_cylinder.htm
; Note : uses pushad to copy registers onto the ESP stack, stores the
; calculated values onto the stack at the correct offsets, then restores the
; stack back to the registers.
; ******************************************************************************

sector_chs:  ; ( sector -- eax ) calculate CHS from a sector number in eax,
    ; returns with DX = HHDD, CX = CCSS where HH=head, DD=drive, CC=cylinder, SS=sector
    ; Note that the input sector number is zero based, and that the high 16 bits of EAX must be 0
    pushad  ; Pushes all general purpose registers onto the stack in the following order:
        ; EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The value of ESP is the value before the actual push of
ESP
        ;  7    6    5    4    3    2    1    0   offset in cells from ESP
    mov ebp, esp    ; copy the original ESP stack pointer to EBP so we can access items on the stack
easily

    ; save the register values in the DAP buffer for use later, via ESI
    mov esi, DAP_BUFFER

    add eax, [ bootsector - $$ + BOOTOFFSET]
    push eax                            ; save it while we calculate heads*sectors-per-track
    mov al, [ driveinfo_Head - $$ + BOOTOFFSET]      ; index of highest-numbered head
    inc al                              ; 1-base the number to make count of heads
    mul byte [ driveinfo_SectorsPertrack - $$ + BOOTOFFSET]     ; sectors per track
    mov ebx, eax
    pop eax
    xor edx, edx                        ; clear high 32 bits
    div ebx                             ; leaves cylinder number in eax, remainder in edx
    mov ecx, eax                        ; store cylinder number in another register
    mov eax, edx                        ; get remainder into AX
    mov bl, [ driveinfo_SectorsPertrack - $$ + BOOTOFFSET]      ; number of sectors per track
    div bl                              ; head number into AX, remainder into DX
    mov bl, al                          ; result must be one byte, so store it in BL
    rol ecx, 8                          ; high 2 bits of cylinder number into high 2 bits of CL
    shl cl, 6                           ; makes room for sector number
    or cl, ah                           ; merge cylinder number with sector number
    inc cl                              ; one-base sector number
    mov [ ebp + ( 6 * 4 ) ], ecx        ; store the result in ECX position on esp stack
    mov word [ esi + o_Int13_DAP_saved_CHS_CX ], cx  ; also save the calculated CX value
    mov cx, [ driveinfo_Drive_DX - $$ + BOOTOFFSET]    ; drive number in low 8 bits
    mov ch, bl                          ; place head number in high bits
;    mov cl, 0x80
    mov [ ebp + ( 5 * 4 ) ], ecx    ; store the result in EDX position on esp stack
    mov word [ esi + o_Int13_DAP_saved_CHS_DX ], cx  ; also save the calculated DX value
    popad                           ; restore registers from esp stack
    ret

; ******************************************************************************
; enter Protected Mode (32 bit) and Real Mode (16 bit)
; from http://ringzero.free.fr/os/protected%20mode/Pm/PM1.ASM
; ******************************************************************************

[BITS 16]   ; Real Mode code (16 bit)

enterProtectedMode:                 ; must come from a 'call' , can not be inlined
    pop ax
    push code32p_SELECTOR_0x08
    push ax
    retf

setProtectedModeAPI:                ; set protected mode from 'Real' mode. Called from 16 bit code,
returns to 32 bit code
    pushad                          ; save all registers as doublewords
    mov eax, cr0
    or al, 1
    mov cr0, eax                    ; set the Protected Mode bit in the Control Register
    xor eax, eax                    ; clear high bits of eax
    call enterProtectedMode
```

```
[BITS 32]                          ; Protected Mode code (32 bit)

    mov eax, data32p_SELECTOR_0x10  ; Protected Mode data segment
    mov es, ax
    mov ds, ax
    mov ss, ax                     ; this makes stack segment 32 bits
    popad
    o16 ret

enter16bitProtectedMode:           ; 32 bit code. Must come from a 'call' , can not be inlined
    pop eax                        ; return address
    push dword code16r_SELECTOR_0x18    ; select 16-bit Protected Mode AKA 'Real' Mode
    push eax
    retf

setRealModeAPI:                    ; set 'Real' mode from protected mode.
                                   ; Called from 32 bit code, returns to 16 bit code
                                   ; assumed that protected-mode stack is based at 0
                                   ; and that bits 16 through 19 will not change during time in realmode
    pushad                         ; save 32-bit values of registers
    mov ecx, esp                   ; do all possible 32-bit ops before going to 16 bits
    mov edx, cr0
    call enter16bitProtectedMode

[BITS 16]                          ; Real Mode code (16 bit)

    mov ax, data16r_SELECTOR_0x20
    mov ds, ax
    mov es, ax
    mov ss, ax                     ; here the stack becomes 16 bits based at 0, and SP used not ESP
                                   ; *** consider stack to be invalid from here until we reach real mode
***
    xor cx, cx                     ; clear low 16 bits
    shr ecx, 4                     ; move high 4 bits into cl
    dec dl                         ; leave protected mode, only works if we KNOW bit 0 is set
    mov cr0, edx
    call enterRealMode
    xor ax, ax
    mov ds, ax
    mov es, ax
    mov ss, cx
    ; note we don't need to set SP to 8xxx if ESP is b8xxx, since
    ; the b000 is now in SS, and the b of b8xxx is ignored in real mode
    popad
    o32 ret

enterRealMode:                     ; 16 bit code. Must come from a 'call' , can not be inlined
    pop ax
    push fs                        ; real-mode code segment
    push ax
    retf

[BITS 32]                          ; Protected Mode code (32 bit)

; ****************************************************************************
; ****************************************************************************

;%include "JCreadwrite.nasm"
; JCreadwrite.nasm 2012 Oct 23   read and write the disk using 16 bit BIOS calls
; BIOS read and write routines for colorForth

[BITS 32]                          ; Protected Mode code (32 bit)

bios_read:  ; ( a c -- a' c' )   \  read cylinder c into address a , leave next address and cylinder
                                   ; c is cylinder, we will use 1.44Mb floppy's idea of cylinder
regardless
                                   ; a is byte address
                                   ; leave updated c and a on stack as c' and a'
                                   ; a cylinder is 36 tracks of 512 bytes each, 0x4800 bytes, 0x1200
cells (words)
```

```
    cli                             ; disable interrupts
    pushad                          ; push all registers ( except esp ) and flags onto the stack
    mov ebp, esp                    ; copy of stack pointer for use below ( * ), points to registers
copied by pushad , above

    mov ecx, HEADS * SECTORS        ; sectors per track (both heads)
    mul cl                          ; sector number goes into AX
                                    ; note that resultant sector number is zero-based going into
sector_chs!
                                    ; set up loop to read one floppy cylinder's worth

    push eax                        ; absolute sector number to start
.back:
    push ecx
    call sector_chs                 ; convert to Cylinder-Head-Sector in CX-DX
    call .readsector

    mov ebx, [ ebp + ( 1 * 4 ) ]    ; ( * ) get ESI stored on stack, via stack pointer saved in ebp
    mov edi, [ebx]                  ; destination index address for movsd
    mov ecx, ( 512 >> 2 )           ; number of 32-bit words to move, 512 bytes
    mov esi, SECTOR_BUFFER          ; source index for movsd
    rep  movsd                      ; copy ecx 32 bit words from ds:esi to es:edi
    mov [ebx], edi
    pop ecx
    pop eax
    inc eax
    push eax
    loop .back
    pop eax
    inc dword [ebp + 7 * 4]         ; for updated cylinder number after return
    popad
    ret

.readsector:                        ; no need to save registers because we take care of them in calling
routine
    call setRealModeAPI
[BITS 16]                           ; Real Mode code (16 bit)
    mov bx, SECTOR_BUFFER
    mov ax, 0x0201                  ; read 1 sector
    int 0x13
    cli                             ; BIOS might have left interrupts enabled
    call setProtectedModeAPI        ; called from 16 bit code, returns to 32 bit code
[BITS 32]                           ; Protected Mode code (32 bit)
    ret

bios_write:     ; ( a c -- a' c' )   \  write cylinder c from address a , leave next address and cylinder
    cli                             ; disable interrupts
    pushad
    mov ebp, esp
                                    ; eax contains cylinder to start, the 'c' parameter
    mov ecx, HEADS * SECTORS        ; sectors per track (both heads)
    mul cl                          ; absolute sector number goes into AX

    mov ebx, [ebp + ( 1 * 4 ) ]     ; stored ESI on stack
    mov esi, [ebx]                  ; word address, 'a' parameter
;        shl esi, 2                 ; change word address into byte address
                                    ; set up loop to write one floppy cylinder's worth
    push eax                        ; absolute sector number to start

.back:
    push ecx
                                    ; load sector data into buffer
                                    ; DO NOT take advantage of knowing ECX only has byte value
    mov ecx, 128 ; ( 512 >> 2 )     ; number of 32-bit words to move
    mov edi, SECTOR_BUFFER
    rep  movsd                      ; copy ecx 32 bit words from ds:esi to es:edi
    call sector_chs                 ; convert to Cylinder-Head-Sector in CX-DX
    call .writesector
    pop ecx
```

```
    pop eax
    inc eax
    push eax
    loop .back
    pop eax
    inc dword [ ebp + ( 7 * 4 ) ]   ; for updated cylinder after return (EAX)
    mov ebx, [ ebp + ( 1 * 4 ) ]    ; stored ESI on stack
    mov [ebx], esi                  ; updated address
    popad
    ret

.writesector:                       ; no need to save registers because we take care of them in calling
routine
    call setRealModeAPI
[BITS 16]                           ; Real Mode code (16 bit)
    mov bx, SECTOR_BUFFER
    mov ax, 0x0301                  ; write 1 sector
    int 0x13
    cli                             ; BIOS might have left interrupts enabled
    call setProtectedModeAPI        ; called from 16 bit code, returns to 32 bit code
[BITS 32]                           ; Protected Mode code (32 bit)
    ret

 times (0x400 - ($ - $$)) nop

; ****************************************************************************
; ****************************************************************************
; After Two Sectors
; ****************************************************************************
; ****************************************************************************

version:
    db 'cf2023 1v0 2023Apr04 Chuck Moore' , 0x00    ; 0x20 + 1 bytes
    db ' Howerd Oakford inventio.co.uk' ,  0x00     ; 0x1E + 1 bytes, total 0x40

nul:                ; do nothing...
    ret

; ****************************************************************************
; Co-operative multi-tasker with comments from code by Josh Grams
; ****************************************************************************

; This version of colorforth has three tasks; main (the quit loop),
; draw (user defined), and serve (also user defined).  Each has two
; grows-down stacks.  A suffix of 's' indicates the return stack, 'd'
; indicates the data stack.  Thus 'draws' and 'drawd' are the tops of
; the return and data stacks, respectively, for the draw task.

; When we switch tasks, we need to switch stacks as well.  We do this
; by pushing eax (cached top-of-stack) onto the data stack, pushing
; the data stack pointer onto the return stack, and then saving the
; return stack pointer into the save slot for the task.

; 'me' points to the save slot for the current task
me:
    dd main
x_screenTask:
    dd nul
x_serverTask:
    dd nul
x_serverTask2:
    dd nul

pause_:
    _DUP_
    push esi
    mov _TOS_, [ me ]               ; points to main at startup
    mov [_TOS_], esp
    add _TOS_, byte 0x04
    jmp _TOS_
```

```
resume:
    pop _TOS_
    mov esp, [_TOS_]
    mov [ me ], _TOS_
    pop esi
    _DROP_
    ret

; these are the save slots - each is followed by code to resume the
; next task - the last one jumps 'round to the first.
round:
    call resume
main:                               ; main task
    dd 0                            ; new stack location
    call resume
draw:                               ; screen draw task
    dd 0                            ; new stack location
    call resume
serv1:                               ; server task
    dd 0                            ; new stack location
    call resume
serv2:                              ; server task 2
    dd 0                            ; new stack location
    jmp short round                 ; loop forever between 3 stacks

activate:   ; ( a -- )    \ activate the draw task to execute colorForth code at the given address
    mov edx, DATA_STACK_1 - 4
    mov [edx], ecx
    mov ecx, RETURN_STACK_1 - 4
    pop dword [ecx]
    lea ecx, [ ecx - 0x04 ]
    mov [ecx], edx
    mov dword [ draw ], ecx
    _DROP_
    ret

show:   ; ( -- )    \ set the screen task to execute the code following  show
    pop dword [ x_screenTask ]      ; copy the return address of the calling word into the screenTask
variable
    _DUP_
    xor _TOS_, _TOS_
    call activate
.back:
    call graphAction                ; perform a graphical update
    call [ x_screenTask ]           ; execute the code that called show, saved on entry
    call switch                     ; copy the screen image to the VESA buffer
    xor _TOS_, _TOS_
    call pause_
    inc _TOS_
    jmp short .back

initshow:                           ; called by warm
    call show
    ; <--- this address ( on the return stack from the preceding call ) goes into  x_screenTask
    ret                             ; makes this a no-op "show"

freeze:
    pop dword [ x_screenTask ]
    _DUP_
    xor _TOS_, _TOS_
    call activate
.back:
    ; call graphAction                ; perform a graphical update
    call [ x_screenTask ]           ; execute the code that called show, saved on entry
    ; call switch                     ; copy the screen image to the VESA buffer
    xor _TOS_, _TOS_
    call pause_
    inc _TOS_
    jmp short .back
```

```
; ****************************************************************************
; ; Server task 1
; ****************************************************************************

activate1:  ; ( a -- )    \ activate the server task to execute colorForth code at the given address
    mov edx, DATA_STACK_2 - 4
    mov [edx], ecx
    mov ecx, RETURN_STACK_2 - 4
    pop dword [ecx]
    lea ecx, [ ecx - 0x04 ]
    mov [ecx], edx
    mov [ serv1 ], ecx
    _DROP_
    ret

serv1_:
    pop dword [ x_serverTask ]
    call activate1
.back:
    ; call graphAction                ; perform a graphical update
    call [ x_serverTask ]           ; execute the code that called show, saved on entry
    ; call switch                     ; copy the screen image to the VESA buffer
    xor _TOS_, _TOS_
    call pause_
    inc _TOS_
    jmp short .back

initserv1_:
    call serv1_
    ret

; ****************************************************************************
; Server task 2
; ****************************************************************************

activate2:  ; ( a -- )    \ activate the server task to execute colorForth code at the given address
    mov edx, DATA_STACK_3 - 4
    mov [edx], ecx
    mov ecx, RETURN_STACK_3 - 4
    pop dword [ecx]
    lea ecx, [ ecx - 0x04 ]
    mov [ecx], edx
    mov [ serv2 ], ecx
    _DROP_
    ret

serv2_:
    pop dword [ x_serverTask2 ]
    call activate2
.back:
    ; call graphAction              ; perform a graphical update
    call [ x_serverTask2 ]          ; execute the code that called show, saved on entry
    ; call switch                   ; copy the screen image to the VESA buffer
    xor _TOS_, _TOS_
    call pause_
    inc _TOS_
    jmp short .back

initserv2_:
    call serv2_
    ret

; ****************************************************************************
; ****************************************************************************

c_:     ; ( -- )   \ clear the data stack for keyboard task
    mov esi, ( DATA_STACK_0 + 4 )
    ret
```

```
; ******************************************************************************
; ******************************************************************************

mark:
    mov ecx, [ v_MacroWordCount]
    mov [ mark_MacroWordCount], ecx
    mov ecx, [ v_ForthWordCount ]
    mov [ mark_v_ForthWordCount], ecx
    mov ecx, [ v_BlueWordCount ]
    mov [ mark_v_BlueWordCount], ecx
    mov ecx, [ v_H ]
    mov [ mark_H ], ecx
    ret

empty_:
    cli                         ; disable interrupts
    call initserv1_             ; we must set the server tasks to their Nop loop
    call initserv2_             ; because the code that they might be running will soon be gone...
    mov ecx, [ mark_H ]
    mov [ v_H ], ecx
    mov ecx, [ mark_v_BlueWordCount]
    mov [ v_BlueWordCount ], ecx
    mov ecx, [ mark_v_ForthWordCount]
    mov [ v_ForthWordCount ], ecx
    mov ecx, [ mark_MacroWordCount]
    mov [ v_MacroWordCount ], ecx
    ret

; ******************************************************************************
; ******************************************************************************

mfind:  ; ( sf -- )   \ ecx = index ; find the Shannon-Fano word sf in the Macro wordlist, return its
index in ecx
    mov ecx, [ v_MacroWordCount ]   ; count of Macro wordlist words
    push edi
    lea edi, [ ( ecx * 4 ) + MacroNames - 4 ]
    jmp short ffind

find_:   ; ( sf -- )   \ ecx = index ; find the Shannon-Fano word sf in the Forth wordlist, return its
index in ecx
    mov ecx, [ v_ForthWordCount ]   ; count of Forth wordlist words
    push edi
    lea edi, [ ( ecx * 4 ) + ForthNames - 4 ]   ; set edi to the top of the Forth name table
ffind:
    std                     ; scan backwards
    repne scasd             ; find the 32 bit Shanon-Fano encoded name, compare eax with doubleword at
es:edi and set status flags.
    cld                     ; reset the direction flag
    pop edi
    ret

; ******************************************************************************
; ******************************************************************************

abort_:
    jmp dword [ x_abort ]

; ******************************************************************************
; compile drop - inline compilation of a single byte 'lodsd' :
;   loads a 32 bit dword from [ds:esi] into  eax  then increments  esi  by 4
; ******************************************************************************

cdrop:
    mov edx, [ v_H ]                ; HERE into edx
    mov [ v_lastAddress ], edx      ; save HERE into v_lastAddress
    mov byte [edx], 0xAD            ; 0xAD is the opcode for 'lodsd' = _DROP_
    inc dword [ v_H ]               ; increment HERE
    ret

; ******************************************************************************
```

```
; adup , dup and qdup
; ******************************************************************************

;  1147                                      _DUP_
;  1148 000004E7 8D76FC              <1>  lea esi, [ esi - 4 ]
;  1150 000004EA 8906                <1>  mov [ esi ], _TOS_
adup:
    _DUP_   ; runtime action of the  _DUP_  macro_
    ret

cdup:   ; compile the action of the  _DUP_  macro_
    mov edx, [ v_H ]
; this code sets the flags
;    mov dword [edx], 0x8904EE83     ; assemble the instruction sequence for  DUP  "sub esi, byte 0x04" ,
"mov [esi], eax"
; this code does not set the flags
    mov dword [edx], 0x89FC768D     ; assemble the instruction sequence for  DUP  "lea esi, [ esi - 4 ]" ,
"mov [esi], eax"
    mov byte [ edx + 4 ], 0x06      ;  "8d 76 fc" , "89 06" ( the first 4 are expressed in little endian
format above )
    add dword [ v_H ], byte 0x05    ; update HERE by the 5 bytes we have just compiled
    ret

; qdup is a Macro word that optimises what would be a compiled sequence of  DROP  DUP  into doing nothing
qdup:   ; compile a DUP unless we have just compiled a  DROP  , in which case just delete the  DROP.
    mov edx, [ v_H ]                ; look at HERE
    dec edx                         ; step back one byte
    cmp dword [ v_lastAddress ], edx ; test if we have just compiled one byte in the current word
    jnz cdup                        ; if we have just compiled one byte, compile in the runtime code for
DUP  after this one byte
    cmp byte [edx], 0xAD            ; test if we have just compiled a DROP, 0xAD is the opcode for 'lodsd'
= _DROP_
    jnz cdup                        ; if we have not just compiled a DROP, compile in the runtime code for
DUP  after this one byte
    mov [ v_H ], edx                ; update HERE to remove the DROP that was just compiled
    ret


; ******************************************************************************
; ******************************************************************************

select_define:      ; select how to define the word
    pop dword [ adefine ]
    ret

macro_:     ; select the Macro wordlist
    call select_define
macrod:
    push _TOS_
    mov ecx, [ v_MacroWordCount]                ; save the word count into ecx
    inc dword [ v_MacroWordCount]               ; increment the word count for the next time around

    ; save the source address in the locate table
    shl edi, 2                                  ; convert to an address in bytes
    mov dword [ ( MacroLocates ) + ( ecx * 4 ) ], edi
    shr edi, 2

    lea ecx, [ ( ecx * 4 ) + MacroNames ]       ; ecx contains the address to store the new token name in
the wordlist
    mov _TOS_, ( MacroJumpTable - MacroNames )  ; _TOS_ contains the relative address of the jump table
    jmp short add_word_to_wordlist

forth_:      ; select the Forth wordlist
    call select_define
forthd:
    push _TOS_
    mov ecx, [ v_ForthWordCount ]               ; save the word count into ecx
    inc dword [ v_ForthWordCount ]              ; increment the word count for the next time around

    ; save the source address in the locate table
```

```
    shl edi, 2                                  ; convert to an address in bytes
    mov dword [ ( ForthLocates ) + ( ecx * 4 ) ], edi
    shr edi, 2                                  ; convert back to an address in 32 bit cells

    lea ecx, [ ( ecx * 4 ) + ForthNames ]       ; ecx contains the address to store the new token name in
the wordlist
    mov _TOS_, ( ForthJumpTable - ForthNames )  ; _TOS_ contains the relative address of the jump table
    ; Note : falls through to add_word_to_wordlist

add_word_to_wordlist:
    mov edx, [ ( edi * 4 ) - 0x04 ]     ; edi points to the source token in the block, edx contains the
token Shannon-Fano token name to compile
    and edx, dword 0xFFFFFFF0           ; 'and' out the token colour. Tokens in a wordlist of token names
have a 'colour' of 0
    mov [ecx], edx                      ; ecx contains the address to store the new token name in the
wordlist
    mov edx, [ v_H ]
    mov [ecx+_TOS_], edx
    lea edx, [ecx+_TOS_]
    shr edx, 0x02
    mov [ v_lastToken ], edx
    pop _TOS_
    mov [ v_lastAddress ], esp
    mov dword [ lit ], adup
    ret

; ****************************************************************************
; ****************************************************************************

alit:
    mov dword [ lit ], adup

literal:
    call qdup                       ; compile the runtime action for  DUP , unless we have just compiled a
DROP , in which case do nothing
    mov edx, [ v_lastAddress ]      ; select the wordlist to add the literal to
    mov [ v_lastAddress_copy ], edx
    mov edx, [ v_H ]
    mov [ v_lastAddress ], edx
    mov byte [edx], _MOV_TOS_LIT_   ; the opcode for mov eax, 32_bit_literal  (in next 32 bit cell)
    mov [ edx + 0x01 ], _TOS_       ; the literal value follows in the next 4 bytes in the dictionary
    add dword [ v_H ], byte 0x05    ; move the dictionary pointer forward 5 bytes
    ret

; ****************************************************************************
; ****************************************************************************

cnum:
    call [ lit ]
    mov _TOS_, [ ( edi * 4 ) + 0x00 ]
    inc edi
    jmp short cshrt

cshort:
    call [ lit]
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]
    sar _TOS_, 0x05
cshrt:
    call literal
    _DROP_
    ret

%if 1
; new improved code
; ****************************************************************************
; Magenta Variables run time code
; ecx contains the token number of the new Magenta Variable in the wordlist
;    This is found by searching the rodlist for the variable's name
; ****************************************************************************
```

```
m_var_forth_action:             ; code field for a magenta variable in the Forth wordist - return the
variable's address
    _DUP_                       ; runtime code to duplicate the TOS
    mov [ v_test1 ], ecx
    mov _TOS_, ecx              ; the offset in 32 bit cells of the Name of the current word being
executed
    shl _TOS_, 2                ; the offset in bytes of the Name of the current word being defined
    add _TOS_, ForthLocates     ; the address in bytes of the Name of the current word being defined
    mov _TOS_, [ _TOS_ ]        ; fetch the address in 32 bit cells of the data
;    shl _TOS_, 2                ; convert to an address in bytes
    ret

m_var_macro_action:             ; code field for a magenta variable in the Macro wordist - compile a
literal that returns the variable's address
    call [ lit ]
    mov [ v_test2 ], ecx
    mov _TOS_, ecx              ; the offset in 32 bit cells of the Name of the current word being defined
    shl _TOS_, 2                ; the offset in bytes of the Name of the current word being defined
    add _TOS_, MacroLocates     ; the address in bytes of the Name of the current word being defined
    mov _TOS_, [ _TOS_ ]        ; fetch the address in 32 bit cells of the data
    ; shl _TOS_, 2                ; convert to an address in bytes
    jmp short cshrt

; ******************************************************************************
; Create a Magenta Variable
; ecx contains the token number of the new Magenta Variable in the wordlist
;    i.e. where the variable name is stored, set by forthd or macrod
; ******************************************************************************

m_variable:                     ; create a magenta variable

    call forthd                 ; select the Forth wordlist
    mov dword [ ( ForthJumpTable - ForthNames ) + ecx ], m_var_forth_action
    shl edi, 2                  ; convert to an address in bytes
    mov dword [ ( ForthLocates   - ForthNames ) + ecx ], edi
    shr edi, 2                  ; convert back to an address in 32 bit cells

    call macrod                 ; select the Macro wordlist
    mov dword [ ( MacroJumpTable - MacroNames ) + ecx ], m_var_macro_action
    shl edi, 2                  ; convert to an address in bytes
    mov dword [ ( MacroLocates   - MacroNames ) + ecx ], edi
    shr edi, 2                  ; convert back to an address in 32 bit cells

    inc edi
    ret

%else

; original code

; ******************************************************************************
; magenta variables
; ecx contains the address to store the new token name in the wordlist
;   set by forthd or macrod
; ******************************************************************************

m_var_forth_action:           ; code field for a magenta variable in the Forth wordist
    _DUP_               ; runtime code to duplicate the TOS
    mov _TOS_, [ 4 + ForthNames + ( ecx * 4 ) ]
    shl _TOS_, 2
    ret

m_variable:     ; define a magenta variable
    call forthd         ; select the Forth wordlist
    mov dword [ ForthJumpTable - ForthNames + ecx ], m_var_forth_action
    inc dword [ v_ForthWordCount ]
    mov [ ecx + 4 ], edi
    call macrod         ; select the Macro wordlist
    mov dword [ MacroJumpTable - MacroNames + ecx ], m_var_macro_action
    inc dword [ v_MacroWordCount ]
```

```
    mov [ ecx + 4 ], edi
    inc edi
    ret

m_var_macro_action:           ; code field for a magenta variable in the Macro wordist
    call [ lit ]
    mov _TOS_, [ 4 + MacroNames + ( ecx * 4 ) ]
    shl _TOS_, 2
    jmp short cshrt

%endif

; ******************************************************************************
; ******************************************************************************

ex1:
    xor edi, edi
.back:
    dec dword [ v_words ]
    jz ex2
    _DROP_
    jmp short .back

execute_lit:    ; ( -- )
    mov dword [ lit ], alit
    _DUP_
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]
execute_:    ; ( name -- )
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
ex2:
    call find_
    jnz abort_
    _DROP_
    jmp dword [ ( ecx * 4 ) + ForthJumpTable ]

; ******************************************************************************
; ******************************************************************************

qcompile:
    call [ lit ]
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
    call mfind
    jnz .forward
    _DROP_
    jmp dword [ ( ecx * 4 ) + MacroJumpTable ]
.forward:
    call find_
    mov _TOS_, [ ( ecx * 4 ) + ForthJumpTable ]

qcom1:
    jnz abort_
call_:
    mov edx, [ v_H ]
    mov [ v_lastAddress ], edx
    mov byte [edx], 0xE8        ; 0xE8 is the opcode for 'call immediate'
    add edx, byte 0x05
    sub _TOS_, edx
    mov [ edx - 0x04 ], _TOS_
    mov [ v_H ], edx
    _DROP_
 ret

; ******************************************************************************
; ******************************************************************************

compile:
    call [ lit]
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
```

```
    call mfind
    mov _TOS_, [ ( ecx * 4 ) + MacroJumpTable ]
    jmp short qcom1

; ****************************************************************************
; ****************************************************************************

short_:
    mov dword [ lit], alit
    _DUP_
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]
    sar _TOS_, 0x05
    ret

; ****************************************************************************
; ****************************************************************************

num:
    mov dword [ lit], alit
    _DUP_
    mov _TOS_, [ ( edi * 4 ) + 0x00 ]
    inc edi
    ret

; ****************************************************************************
; ****************************************************************************

comma_:          ; 4 byte  ,
    mov ecx, 0x04
dcomma:     ; c,  performed n times ( n in ecx )
    mov edx, [ v_H ]
    mov [edx], _TOS_
    mov _TOS_, [ esi ]
    lea edx, [ ecx + edx ]
    lea esi, [ esi + 0x04 ]
    mov [ v_H ], edx
    ret

comma1_:     ; 1 byte  c,
    mov ecx, 0x01
    jmp short dcomma

comma2_:     ; 2 byte  w,
    mov ecx, 0x02
    jmp short dcomma

comma3_:     ; 3 byte  c, c, c,
    mov ecx, 0x03
    jmp short dcomma

; ****************************************************************************
; ****************************************************************************

semicolon:
    mov edx, [ v_H ]
    sub edx, byte 0x05
    cmp [ v_lastAddress ], edx
    jnz .forward
    cmp byte [edx], 0xE8            ; 0xE8 is the opcode for 'call immediate'
    jnz .forward
    inc byte [edx]
    ret
.forward:
    mov byte [ edx + 0x05 ], 0xC3   ; 0xC3 is the opcode for 'ret'
    inc dword [ v_H ]
    ret

; ****************************************************************************
; ****************************************************************************
```

```
then:
    mov [ v_lastAddress ], esp
    mov edx, [ v_H ]
    sub edx, _TOS_
    mov [ _TOS_ - 0x01 ], dl
    _DROP_
    ret

begin_:
    mov [ v_lastAddress ], esp
here_:
    _DUP_
    mov _TOS_, [ v_H ]
    ret

; ****************************************************************************
; ****************************************************************************

qlit:  ; ?lit
    mov edx, [ v_H ]
    lea edx, [ edx - 0x05 ]
    cmp [ v_lastAddress ], edx
    jnz .forward
    cmp byte [edx], _MOV_TOS_LIT_   ; the opcode for mov eax, 32_bit_literal  (in next 32 bit cell)
    jnz .forward
    _DUP_
    mov _TOS_, [ v_lastAddress_copy ]
    mov [ v_lastAddress ], _TOS_
    mov _TOS_, [ edx + 0x01 ]
    cmp dword [ edx - 5 ], 0x89FC768D   ; assemble code 8D 76 FC 89 rr => lea esi, [ esi - 0x04 ] ;  mov [
esi ], register
    ; like dup but with the register value still to follow in the next byte
    jz .forward2
    mov [ v_H ], edx
    jmp dword cdrop
.forward2:
    add dword [ v_H ], byte -0x0A
    ret
.forward:
    xor edx, edx
    ret

less:
    cmp [ esi ], _TOS_
    js .forward
    xor ecx, ecx
.forward:
    ret

qignore:
    test dword [ ( edi * 4 ) - 0x04 ], 0xFFFFFFF0
    jnz .forward
    pop edi
    pop edi
.forward:
    ret

jump:
    pop edx
    add edx, _TOS_
    lea edx, [ edx + ( _TOS_ * 4 ) + 0x05 ]
    add edx, [ edx - 0x04 ]
    _DROP_
    jmp edx

; convert block start address to cell address, add the RELOCATED colorForth system base
blockToCellAddress:  ; ( blk -- a' )   \ add the RELOCATED offset and convert to cell address
    add _TOS_, [ v_offset ]   ; add the RELOCATED block number offset
    shl _TOS_, 0x08           ; convert to cell address
    ret
```

```
cellAddressToBlock:   ; ( a -- blk )  \ convert cell address to block number and subtract the RELOCATED
block number offset
    shr _TOS_, 0x08           ;  convert cell address to block number
    sub _TOS_, [ v_offset ]   ; subtract the block number of block 0
    ret

_load_:   ; ( blk -- )    \ load the given block number
    call blockToCellAddress   ; add the RELOCATED block number offset and convert to cell address
    push edi
    mov edi, _TOS_
    _DROP_
interpret:
    mov edx, [ ( edi * 4 ) + 0x00 ]
    inc edi
    and edx, byte 0x0F
    call [ ( edx * 4 ) + tokenActions ]
    jmp short interpret

    align 4, db 0   ; fill the gap with 0's

; : r@ qdup $8B 1, $C7 1, ;    \ mov _TOS_, edi  also db 0x89, 0xF8
; : nload r@ $0100 / #2 + load ;
; : +load ( n -- ) r@ $0100 / + load ;
nload:  ; ( -- )    \ load the next source block following the one currently being loaded
    call cblk_
    add _TOS_, 0x02
    jmp _load_

plusLoad:   ; ( n -- )    \ load the n'th source block following the one currently being loaded
    mov _SCRATCH_, _TOS_    ; save the required offset
    _DROP_
    call cblk_
    add _TOS_, _SCRATCH_
    jmp _load_

; : THRU ( f l -- )   1+ SWAP DO  I LOAD  LOOP ;
thru_:  ; ( first last -- )   \ load from the first to the last block
    add _TOS_, 0x02
    mov _SCRATCH_, _TOS_
    _DROP_                          ; TOS = first, SCRATCH = last
    mov ecx, _SCRATCH_
    sub ecx, _TOS_                  ; ecx = count
    jz .end                         ; exit if count is zero
    jc .end                         ; exit if count is negative
    shr ecx, 1                      ; divide by 2, as we skip 2 blocks each time round the loop
.back:
    _DUP_
    _DUP_   ; just to be safe...
    push ecx
    push _SCRATCH_
    call _load_
    pop _SCRATCH_
    pop ecx
    _DROP_  ; just to be safe...
    add _TOS_, 0x02
    loop .back
.end:
    _DROP_
    ret

v_temp:
    dd 0

plusThru_:  ; ( first+ last+ -- )   \ load from the first to the last block relative to the current block
being loaded
    call cblk_
    mov [ v_temp ], _TOS_
    _DROP_
    mov _SCRATCH_, [ v_temp ]
```

```
    add [ esi ], _SCRATCH_          ; add current block to second on stack
    add _TOS_, _SCRATCH_            ; add current block to top of stack
    call thru_
    ret

cblk_:   ; ( -- n )  \ return the currently compiling block number - only valid while compiling
    _DUP_
    mov _TOS_, edi            ; edi  contains the cell address in the block currently being compiled
    call cellAddressToBlock   ; convert to block number relative to block 0
    ret

rblk_:   ; ( -- n )  \ return the block number offset of the RELOCATED address
    _DUP_
    mov _TOS_, ( RELOCATED >> ( 2 + 8 ) )
    ret

ablk_:   ; ( a -- n )  \ convert byte address to block number
    shr _TOS_, 0x02
    call cellAddressToBlock
    ret

erase_:   ; ( a n -- )  \ erase n bytes starting at address a
    mov ecx, eax
    _DROP_
    push edi
    mov edi, eax
    xor eax, eax
    rep stosb
    pop edi
    _DROP_
    ret

v_curs_to_source:   ; ( n -- a32 )   \ return the cell address of the current cursor position in the
current block being edited
    mov _SCRATCH_, _TOS_
    mov _TOS_, [ v_blk ]            ; get the currently edited block number
    call blockToCellAddress
    add _TOS_, _SCRATCH_            ; add the cursor position (cell address) in the block
    ret

nth_to_token:   ; ( n -- tok )   \ return the token at the n'th cursor position in the current block being
edited
    call v_curs_to_source
    shl _TOS_, 0x02                 ; convert cell address to byte address
    mov _TOS_, [ _TOS_ ]            ; fetch the token
    ret

v_curs_to_token:   ; ( -- tok )   \ return the token at the current cursor position in the current block
being edited
    _DUP_
    mov _TOS_, [ v_blk ]            ; get the currently edited block number
    call nth_to_token
    ret

; : ?f $C021 2, ;
;qf:
;    db 0x21, 0xC0   ; and _TOS_, _TOS_
;    ret

; ******************************************************************************
; ******************************************************************************

top_:   ; ( -- )   \ set the cursor to the left margin horizontally and 3 pixels down from the top
vertically
    mov ecx, [ v_leftMargin ]
    shl ecx, 0x10
    add ecx, byte 0x03
    mov [ v_gr_xy ], ecx
    ; mov [ xycr], ecx
    ret
```

```
qcr: ; ( -- )   \ ?cr  do a  CR  if the cursor has gone past the right margin
    mov cx, [ v_gr_x ]
    cmp cx, [ v_rightMargin ]
    js cr_forward
cr_:  ; ( -- )
    mov ecx, [ v_leftMargin ]
    shl ecx, 0x10
    mov cx, [ v_gr_xy ]
    add cx, [ v_iconh ]
    mov [ v_gr_xy ], ecx
cr_forward:
    ret

green:  ; ( -- )
    _DUP_
    mov _TOS_, colour_green
    jmp set_color_

yellow:  ; ( -- )
    _DUP_
    mov _TOS_, colour_yellow
    jmp set_color_

; red:  ; ( -- )    ; see redWord:
;    _DUP_
;    mov _TOS_, colour_red
;    jmp set_color_

white:  ; ( -- )
    _DUP_
    mov _TOS_, colour_white
set_color_:  ; ( rgb16 -- )
    mov [ v_foregroundColour ], _TOS_
    _DROP_
    ret

rgb:    ; ( rgb32 -- rgb16 )    ; convert from 32 bit ( 8:8:8:8 _RGB ) colour to 16 bit ( 5:6:5 RGB )
colour value
    ror _TOS_, 8
    shr ax, 2
    ror _TOS_, 6
    shr al, 3
    rol _TOS_, ( 6 + 5 )
    and _TOS_, 0x0000FFFF
    ret

bye_:  ; ( -- )   \ exit colorForth
    call setRealModeAPI
[BITS 16]                           ; Real Mode code (16 bit)
    int 0x19    ; reboot the computer
    ; should never get past this point.... but in case we do...
    cli                             ; BIOS might have left interrupts enabled
    call setProtectedModeAPI        ; called from 16 bit code, returns to 32 bit code
[BITS 32]                           ; Protected Mode code (32 bit)
    ret

%if 0
pci:
    mov edx, 0x0CF8
    out dx, _TOS_
    lea edx, [ edx + 0x04 ]
    in _TOS_, dx
    ret

device:
    times ( 0x93a - ( $ - $$ ) ) nop  ; fill with nops to find_display ???

find_display:                       ; called by warm
    mov _TOS_, 0x3000000            ; PCI class code 3 = display controller
```

```
    call device                     ; returns header address
    lea _TOS_, [ _TOS_ + 0x10 ]     ; point to Base Address #0 (BAR0)
    mov cl, 0x06
.next:
    _DUP_
    call pci
    and al, 0xFB
    xor al, 0x08
    jz .forward
    _DROP_
    lea _TOS_, [ _TOS_ + 0x04 ]
    loop .next
    lea _TOS_, [ _TOS_ - 0x18 ]
    _DUP_
    call pci
    and al, 0xF0
.forward:
    mov [ v_frameBuffer ], _TOS_     ; set framebuffer address
    _DROP_
    ret

fifo:
    _DROP_
    ret

%endif

graphAction:
    ret

; ******************************************************************************
; ******************************************************************************
; grapics mode dependent code
; ******************************************************************************
; ******************************************************************************

; ******************************************************************************
; 1024x768 display
; ******************************************************************************

scrnw1 equ 1024              ; screen width in pixels
scrnh1 equ 768               ; screen height in pixels
iconw1 equ ( 16 + 4 )        ; icon width
iconh1 equ ( 24 + 4 )        ; icon height for 768 pixel high screen

keypadY1 equ 4               ; location of keyboard display vertically in lines from the bottom

initIconSize1:
    mov dword [ v_iconw ], iconw1
    mov dword [ v_nine_iconw ], ( iconw1 * 9 )
    mov dword [ v_twentytwo_iconw ], ( iconw1 * ( 13 + 9 ) )
    mov dword [ v_10000_iconw ], ( iconw1 * 0x10000 )
    mov dword [ v_iconh ], iconh1
    mov dword [ v_keypadY_iconh ], keypadY1 * iconh1
    ret

switch1:     ; copy our created image to the real display buffer
    push esi
    push edi
    mov  esi, dword [ vframe ]      ; vframe  points to where we create our image
    mov  edi, [ vesa_PhysBasePtr ]  ; VESA frame buffer, saved by VESA BIOS call, the address in RAM that
is displayed by the hardware
    mov  ecx, ( ( scrnw1 * scrnh1 ) / 4 ) * BYTES_PER_PIXEL   ; the / 4 is because we are moving doubles =
4 bytes each
    rep  movsd                      ; copy ecx 32 bit words from ds:esi to es:edi
    pop  edi
    pop  esi
    ret

clip1:
```

```
    mov  edi, [ v_gr_xy ]
    mov  ecx, edi
    test cx, cx
    jns  .forward
    xor  ecx, ecx
.forward:
    and  ecx, 0x0000FFFF
    mov  [ v_yc ], ecx
    imul ecx, ( scrnw1 * BYTES_PER_PIXEL )
    sar  edi, 16
    jns  .forward2
    xor  edi, edi
.forward2:
    mov  [ v_xc ], edi
    lea  edi, [ edi * BYTES_PER_PIXEL + ecx ]
    add  edi, [ vframe ]
    ret

bit16:                               ; write a 16 x 24 glyph to the graphic screen
    lodsw                            ; load the 16 bit value pointed to by SI into  ax
    xchg al, ah                      ; eax_TOS_
.back:
    shl ax, 0x01                     ; eax_TOS_
    jnc  .forward
    mov  [ edi ], dx                 ;
    jmp .forward2
.forward:
    ror edx, 0x10                    ; use the background colour, in the high 16 bits
;    mov [ edi ], dx                 ;
    ror edx, 0x10                    ; return to the foreground colour, in the low 16 bits
.forward2:
    add edi, byte BYTES_PER_PIXEL
    loop .back
    ret

; write the background after the glyph
bit16Background:          ; number of pixels to write in  ecx ,  screen address in  edi , colours in edx
    ror edx, 0x10           ; use the background colour, in the high 16 bits
.back:
;    mov [ edi ], dx        ;
    add edi, byte BYTES_PER_PIXEL
    loop .back
    ror edx, 0x10           ; return to the foreground colour, in the low 16 bits
    ret

bit32:                          ; write a 32 x 48 double size glyph to the graphic screen
    lodsw                   ; load the 16 bit value pointed to by SI into  ax
    xchg al, ah             ; eax_TOS_
    mov ecx, 0x10
.back:
    shl  _TOS_, 1           ; eax_TOS_
    jnc  .forward
    mov  [ edi ], dx
    mov  [ edi + BYTES_PER_PIXEL ], dx

    cmp byte [ displayMode ], 0
    jnz .width2
    mov  [ edi + ( scrnw1 * BYTES_PER_PIXEL ) ], dx
    mov  [ edi + ( scrnw1 * BYTES_PER_PIXEL ) + BYTES_PER_PIXEL ], dx
    jmp .widthEnd
.width2:
    mov  [ edi + ( scrnw2 * BYTES_PER_PIXEL ) ], dx
    mov  [ edi + ( scrnw2 * BYTES_PER_PIXEL ) + BYTES_PER_PIXEL ], dx
.widthEnd:
.forward:
    add edi, byte ( BYTES_PER_PIXEL * 2 )
    loop .back
    ret
```

```
; Table that maps the three levels of Shannon-Fano codes to ASCII, followed by a copy in Capitalised or
larger form
; The original colorForth font has "capital numbers" - larger bold versions, all offset by 0x30 from the
normal ones.
; The new font maps the "larger forms" to offset 0x80, i.e. $30 --> $B0, two forms of '0'.
ShannonFano:
;       000000001111111122222222222222222222222222222222                         <- levels
;       0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01234567 <- index
    db 0x00     ; a space in the cf font
    ;  0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF01234567 <- index
    db "rtoeanismcylgfwdvpbhxuq0123456789j-k.z/;'!+@*,? RTOEANISMCYLGFWDVPBHXUQ" ; ASCII equivalents
     ; 89ABCDEF01
    db 0xB0,0xB1,0xB2,0xB3,0xB4,0xB5,0xB6,0xB7,0xB8,0xB9; larger "capital"forms of "0123456789"
    ;   23456
    db "J_K.Z"  ; normal capitals
    ; 789ABCDEF
    db 0xAF,0xBB,0xA7,0xA1,0xAB,0xC0,0xAA,0xAC,0xBF; larger "capital" forms of "/;'!+@*,?"
    ; 0123456789ABCDEF0123456789ABCDEF
    db 0x10,0x11,0x12,0x13,0x23,0x14,0x15,0x16,0x5B,0x5D,0x17,0x18,0x19,0x28,0x29,0x24 ; $60
....#...[]...()$

emit1:      ; ( c -- )   \ display a single width and height character
    call qcr
    push esi
    push edi
    push edx
    imul _TOS_, byte 16*24/8
    mov esi, [ v_font ]
    add esi, _TOS_
    call clip1
    mov edx, [ v_foregroundColour ]
    mov ecx, 0x18   ; 24 lines
.back:
    push ecx
    mov ecx, 0x10
    call bit16
    mov ecx, 0x04
    push edi
    call bit16Background
    pop edi
    pop ecx
    add edi, ( scrnw1 - 16 ) * BYTES_PER_PIXEL  ; address of the next line of the glyph
    loop .back      ; next horizontal line

    mov ecx, 0x04   ; 4 background lines
.back2:
    push ecx
    mov ecx, 0x10
    call bit16Background
    mov ecx, 0x04
    push edi
    call bit16Background
    pop edi
    pop ecx
    add edi, ( scrnw1 - 16 ) * BYTES_PER_PIXEL  ; address of the next line of the glyph
    loop .back2      ; next horizontal line

    pop edx
    pop edi
    pop esi
    _DROP_
space1:
    add dword [ v_gr_xy ], iconw1 * 0x10000 ; 22 horizontal pixels
    ret

two_emit1:  ; double width and height character
    push esi
    push edi
    push edx
    imul _TOS_, byte 16*24/8
```

```
    mov esi, [ v_font ]
    add esi, _TOS_
    call clip1
    mov edx, [ v_foregroundColour ]
    mov ecx, 0x18   ; 24 lines
.back:
    push ecx
    call bit32
    add edi, (2*scrnw1-16*2)*BYTES_PER_PIXEL
    pop ecx
    loop .back
    pop edx
    pop edi
    pop esi
    add dword [ v_gr_xy ], iconw1 * 2 * 0x10000 ; 44 horizontal pixels
    _DROP_
    ret

setupText__1:   ; setup for full screen text window display
    call white
    mov dword [ v_leftMargin ], 0x03
    mov dword [ v_rightMargin ], ( scrnw1 - iconw1 )
    jmp dword top_

box1: ; ( width height -- )
    call clip1
    cmp _TOS_, scrnh1+1
    js .forward
    mov _TOS_, scrnh1
.forward:
    mov ecx, _TOS_
    sub ecx, [ v_yc ]
    jng .forward3
    cmp dword [esi], scrnw1+1
    js .forward2
    mov dword [esi], scrnw1
.forward2:
    mov _TOS_, [ v_xc ]
    sub [esi], _TOS_
    jng .forward3
    mov edx, scrnw1
    sub edx, [esi]
    shl edx, PIXEL_SHIFT
    mov _TOS_, [ v_foregroundColour ]
.back:
    push ecx
    mov ecx, [esi]
    rep stosw   ; stosw depends on BYTES_PER_PIXEL, either stosw or stosd
    add edi, edx
    pop ecx
    loop .back
.forward3:
    _DROP_
    _DROP_
    ret

wash1:   ; ( colour -- )   \ fill the full screeen with the given colour
    call set_color_
    _DUP_

    xor _TOS_, _TOS_     ; x,y = 0,0 top left corner
    mov [ v_gr_xy ], _TOS_

    mov _TOS_, scrnw1
    _DUP_
    mov _TOS_, scrnh1
    jmp dword box_

; ****************************************************************************
; 800x600 screen
```

```
; ****************************************************************************

scrnw2 equ 800               ; screen width in pixels
scrnh2 equ 600               ; screen height in pixels
iconw2 equ ( 16 + 1 )        ; icon width
iconh2 equ ( 24 - 1 )        ; icon height for NC10 600 pixel high screen

keypadY2 equ 4               ; location of keyboard display vertically in lines from the bottom

initIconSize2:
    mov dword [ v_iconw ], iconw2
    mov dword [ v_nine_iconw ], ( iconw2 * 9 )
    mov dword [ v_twentytwo_iconw ], ( iconw2 * ( 13 + 9 ) )
    mov dword [ v_10000_iconw ], ( iconw2 * 0x10000 )
    mov dword [ v_iconh ], iconh2
    mov dword [ v_keypadY_iconh ], keypadY2 * iconh2
    ret

switch2:     ; copy our created image to the real display buffer
    push esi
    push edi
    mov  esi, dword [ vframe ]      ; vframe  points to where we create our image
    mov  edi, [ vesa_PhysBasePtr ]  ; VESA frame buffer, saved by VESA BIOS call, the address in RAM that
is displayed by the hardware
    mov  ecx, ( ( scrnw2 * scrnh2 ) / 4 ) * BYTES_PER_PIXEL   ; the / 4 is because we are moving doubles =
4 bytes each
    rep  movsd                      ; copy ecx 32 bit words from ds:esi to es:edi
    pop  edi
    pop  esi
    ret

clip2:
    mov  edi, [ v_gr_xy ]
    mov  ecx, edi
    test cx, cx
    jns  .forward
    xor  ecx, ecx
.forward:
    and  ecx, 0x0000FFFF
    mov  [ v_yc ], ecx
    imul ecx, ( scrnw2 * BYTES_PER_PIXEL )
    sar  edi, 16
    jns  .forward2
    xor  edi, edi
.forward2:
    mov  [ v_xc ], edi
    lea  edi, [ edi * BYTES_PER_PIXEL + ecx ]
    add  edi, [ vframe ]
    ret

emit2:      ; ( c -- )   \ display a single width and height character
    call qcr
    push esi
    push edi
    push edx
    imul _TOS_, byte 16*24/8
    mov esi, [ v_font ]
    add esi, _TOS_
    call clip2
    mov edx, [ v_foregroundColour ]
    mov ecx, 0x18   ; 24 lines
.back:
    push ecx
    mov ecx, 0x10
    call bit16
    mov ecx, 0x04
    push edi
    call bit16Background
    pop edi
    pop ecx
```

```
    add edi, ( scrnw2 - 16 ) * BYTES_PER_PIXEL  ; address of the next line of the glyph
    loop .back      ; next horizontal line

    mov ecx, 0x04   ; 4 background lines
.back2:
    push ecx
    mov ecx, 0x10
    call bit16Background
    mov ecx, 0x04
    push edi
    call bit16Background
    pop edi
    pop ecx
    add edi, ( scrnw2 - 16 ) * BYTES_PER_PIXEL  ; address of the next line of the glyph
    loop .back2      ; next horizontal line

    pop edx
    pop edi
    pop esi
    _DROP_
space2:
    add dword [ v_gr_xy ], iconw2 * 0x10000 ; 22 horizontal pixels
    ret

two_emit2:  ; double width and height character
    push esi
    push edi
    push edx
    imul _TOS_, byte 16*24/8
    mov esi, [ v_font ]
    add esi, _TOS_
    call clip2
    mov edx, [ v_foregroundColour ]
    mov ecx, 0x18   ; 24 lines
.back:
    push ecx
    call bit32
    add edi, (2*scrnw2-16*2)*BYTES_PER_PIXEL
    pop ecx
    loop .back
    pop edx
    pop edi
    pop esi
    add dword [ v_gr_xy ], iconw2 * 2 * 0x10000 ; 44 horizontal pixels
    _DROP_
    ret

setupText__2:   ; setup for full screen text window display
    call white
    mov dword [ v_leftMargin ], 0x03
    mov dword [ v_rightMargin ], ( scrnw2 - iconw2 )
    jmp dword top_

box2: ; ( width height -- )
    call clip2
    cmp _TOS_, scrnh2+1
    js .forward
    mov _TOS_, scrnh2
.forward:
    mov ecx, _TOS_
    sub ecx, [ v_yc ]
    jng .forward3
    cmp dword [esi], scrnw2+1
    js .forward2
    mov dword [esi], scrnw2
.forward2:
    mov _TOS_, [ v_xc ]
    sub [esi], _TOS_
    jng .forward3
    mov edx, scrnw2
```

```
    sub edx, [esi]
    shl edx, PIXEL_SHIFT
    mov _TOS_, [ v_foregroundColour ]
.back:
    push ecx
    mov ecx, [esi]
    rep stosw   ; stosw depends on BYTES_PER_PIXEL, either stosw or stosd
    add edi, edx
    pop ecx
    loop .back
.forward3:
    _DROP_
    _DROP_
    ret

wash2:    ; ( colour -- )   \ fill the full screeen with the given colour
    call set_color_
    _DUP_

    xor _TOS_, _TOS_     ; x,y = 0,0 top left corner
    mov [ v_gr_xy ], _TOS_

    mov _TOS_, scrnw2
    _DUP_
    mov _TOS_, scrnh2
    jmp dword box_

; ******************************************************************************
; select which display mode code to use
; ******************************************************************************

displayMode:
    dd 1    ; 0 = 1024x768x16, 1 = 800x600x16

initIconSize: ; sets up the size of an icon (glyph) according to the 800x600 or 1024x768 display size
    cmp byte [ displayMode ], 0
    jz initIconSize1
    jmp initIconSize2

switch:
    cmp byte [ displayMode ], 0
    jz switch1
    jmp switch2

clip:
    cmp byte [ displayMode ], 0
    jz clip1
    jmp clip2

emitSF_:
    mov al, [ ShannonFano + _TOS_ ]
emit_:      ; ( c -- )   display byte c on the screen
    cmp byte [ displayMode ], 0
    jz emit1
    jmp emit2

space_:
    cmp byte [ displayMode ], 0
    jz space1
    jmp space2

type_:      ; ( a n -- )   display n bytes at address a on the screen
    mov ecx, _TOS_
    _DROP_
    mov _SCRATCH_, _TOS_
    .back:
        pusha
        _DUP_
        mov al, [ _SCRATCH_ ]
        and _TOS_, 0x000000FF
```

```
        call emit_
        popa
        inc _SCRATCH_
    loop .back
    _DROP_
    ret

; double size versions of emit, 32 x 48 pixels per glyph
two_emit_SF:
    mov al, [ ShannonFano + _TOS_ ]
two_emit:
    cmp byte [ displayMode ], 0
    jz two_emit1
    jmp two_emit2

setupText_:     ; setup for full screen text window display
    cmp byte [ displayMode ], 0
    jz setupText__1
    jmp setupText__2

line_:  ; ( startX length -- )   \ draw a horizontal line in the current colour, from startX relative to
current clip window, of given length in pixels
    cmp byte [ displayMode ], 0
    jnz .forward
    call clip1
    jmp .common
.forward:
    call clip2
.common:
    mov ecx, [esi]
    shl ecx, PIXEL_SHIFT
    sub edi, ecx
    mov ecx, _TOS_
    mov _TOS_, [ v_foregroundColour ]
    rep stosw   ;
    inc dword [ v_gr_xy ]
    _DROP_
    _DROP_
    ret

box_:
    cmp byte [ displayMode ], 0
    jz box1
    jmp box2

page_:    ; ( -- )    \ fill the full screen with the current background colour
    _DUP_
    mov _TOS_, colour_background   ;
    jmp wash_

screen_:    ; ( -- )   \ fill the full screen with the current foreground colour
    _DUP_
    mov _TOS_, [ v_foregroundColour ]  ;     ; select the foreground colour in the low 16 bits
;    jmp wash_                      ; fall through to wash1

wash_:  ; ( colour -- )   \ fill the full screeen with the given colour
    mov [ v_washColour ], _TOS_
    cmp byte [ displayMode ], 0
    jz wash1
    jmp wash2

; *****************************************************************************
; *****************************************************************************
; *****************************************************************************

setCyan:    ; ( -- )
    _DUP_
    mov _TOS_, colour_cyan
    jmp dword set_color_
```

```
setMagenta:    ; ( -- )
    _DUP_
    mov _TOS_, colour_magenta
    jmp dword set_color_

setMagentaData:    ; ( -- )
    _DUP_
    mov _TOS_, colour_magentaData
    jmp dword set_color_

setBlue:    ; ( -- )
    _DUP_
    mov _TOS_, colour_blue
    jmp dword set_color_

setRed:    ; ( -- )
    _DUP_
    mov _TOS_, colour_red
    jmp dword set_color_

setGreen:    ; ( -- )
    _DUP_
    mov _TOS_, colour_green
    jmp dword set_color_

setSilver:    ; ( -- )         \ AKA grAy
    _DUP_
    mov _TOS_, colour_silver
    jmp dword set_color_

history:
    times 11 db 0

echo_:
    push esi
    mov ecx, 11-1
    lea edi, [ history ]
    lea esi, [ edi + 1 ]
    rep movsb
    pop esi
    mov byte [ history+11-1 ], al
    _DROP_
    ret

right:
    _DUP_
    mov ecx, 11
    lea edi, [history]
    xor _TOS_, _TOS_
    rep stosb
    _DROP_
    ret

down:
    _DUP_
    xor edx, edx
    mov ecx, [ v_iconh ]
    div ecx
    mov _TOS_, edx
    sub edx, [ v_iconh ]
    add edx, ( 3 * 0x10000 )+ 0x8000 + 3
    mov [ v_gr_xy ], edx
; zero:
    test _TOS_, _TOS_
    mov _TOS_, 0
    jnz .dw
    inc _TOS_
.dw:
    ret
```

```
lm:     ; ( leftMargin -- )
    mov [ v_leftMargin ], _TOS_
    _DROP_
    ret

rm:     ; ( rightMargin -- )
    mov [ v_rightMargin ], _TOS_
    _DROP_
    ret

_at:    ; ( y x -- )
    mov word [ v_gr_y ], ax
    _DROP_
    mov word [ v_gr_x ], ax
    _DROP_
    ret

plus_at:    ; ( y x -- )
    add word [ v_gr_y ], ax
    _DROP_
    add word [ v_gr_x ], ax
    _DROP_
    ret

storew_:    ; ( w a -- ) \  ; : !w a! $00028966 3, drop ;
    db 0x8B, 0xD0           ; mov edx,eax       a! $D08B 2,   ( ?lit not true )
    db 0x66, 0x89, 0x02     ; mov [edx],ax      $00028966 3,
    _DROP_                  ; lodsd
    ret                     ; ret

storeu_:    ; ( u a -- ) \  ; : !l a! $0289 2, drop ; forth
    db 0x8B, 0xD0           ; mov edx,eax       a! $D08B 2,   ( ?lit not true )
    db 0x89, 0x02           ; mov [edx],eax     $0289 2,
    _DROP_                  ; lodsd
    ret                     ; ret

uplus_: ; ( u u -- u )   \ : u+ ?lit if $0681 2, , ; then $00044601 3, drop ;
    db 0x01, 0x46, 0x04     ; add [esi+0x4],eax   $00044601 3,   ( ?lit not true )
    _DROP_                  ; lodsd
    ret                     ; ret

%if 1
; the various pieces of code used by a! and +! in colorForth blocks 22 and 24
 plusStore:  ; ( n a -- )
    ;  : a! ?lit if $BA 1, , ; then $D08B 2, drop ;
     mov dword edx, 0x12345678                   ; db 0xBA, 0x78, 0x56, 0x34, 0x12
     mov edx, _TOS_                                ; db 0x8B, 0xD0 == db 0x89, 0xC2
    ;  : +! ?lit if ?lit if $0581 2, swap a, , ; then $0501 2, a, drop ; then a! $0201 2, drop ;
     add [ dword 0x12345678 ], _TOS_               ; db 0x01, 0x05, 0x78, 0x56, 0x34, 0x12
     add dword [ dword 0x12345678 ], 0x98765432 ; db 0x81, 0x05, 0x78, 0x56, 0x34, 0x12, 0x32, 0x54, 0x76,
0x98
     add [ edx ], _TOS_                            ; db 0x01, 0x02
     ret
%endif

octant:
    _DUP_
    mov _TOS_, 0x43
    mov edx, [ esi + 0x04 ]
    test edx, edx
    jns .forward
    neg edx
    mov [ esi + 0x04 ], edx
    xor al, 0x01
.forward:
    cmp edx, [ esi ]
    jns .forward2
    xor al, 0x04
.forward2:
    ret
```

```
hicon:
   db 0x30, 0x31, 0x32, 0x33
   db 0x34, 0x35, 0x36, 0x37
   db 0x38, 0x39, 0x61, 0x62
   db 0x63, 0x64, 0x65, 0x66
;  db 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F
;  db 0x20, 0x21, 0x05, 0x13, 0x0A, 0x10, 0x04, 0x0E

edig1:
    _DUP_
digit:
    push ecx
    mov al, [ _TOS_ + hicon ]
    call emit_
    pop ecx
    ret

odig:
    rol _TOS_, 0x04
    _DUP_
    and _TOS_, byte 0x0F
    ret

h_dot_n:
    mov edx, _TOS_
    neg _TOS_
    lea ecx, [ ( _TOS_ * 4 ) + 0x20 ]
    _DROP_
    rol _TOS_, cl
    mov ecx, edx
    jmp short h_dot_one

dotHex8_: ; ( u -- )   \ display a hexadecimal number with leading zeros, 8 .hex
    mov ecx, 0x08
h_dot_one:
    call odig
    call digit
    loop h_dot_one
    _DROP_
    ret

dotHex2_: ; ( c -- )   \ display a hexadecimal number with leading zeros, 2 .hex
    shl _TOS_, 24
    mov ecx, 0x02
    call h_dot_one
    ret

dotHex4_: ; ( w -- )   \ display a hexadecimal number with leading zeros, 4 .hex
    shl _TOS_, 16
    mov ecx, 0x04
    call h_dot_one
    ret

dotHex:     ; ( u -- )   \ display a hexadecimal number
    EMIT_IMM('$')
    mov ecx, 0x07
.back:
    call odig
    jnz .forward
    _DROP_
    loop .back
    inc ecx
.back2:
    call odig
.back3:
    call digit
    loop .back2
    call space_
    _DROP_
```

```
    ret
.forward:
    inc ecx
    jmp short .back3

qdot:   ; ( u -- )   \ display a decimal or hexadecimal number, depending on base
    cmp dword [ base ], byte 10
    jnz dotHex
dotDecimal:                         ; display a decimal number
;    EMIT_IMM('#')
    mov edx, _TOS_
    test edx, edx
    jns .forward
    neg edx                         ; negate the value and display a minus sign if required
    EMIT_IMM('-')
.forward:
    mov ecx, 0x08
.back:
    mov _TOS_, edx
    xor edx, edx
    div dword [ ecx * 4 + tens ]
    test _TOS_, _TOS_
    jnz .forward2
    dec ecx
    jns .back
    jmp short .forward3
.back2:
    mov _TOS_, edx
    xor edx, edx
    div dword [ ecx * 4 + tens ]
.forward2:
    call edig1
    dec ecx
    jns .back2
.forward3:
    mov _TOS_, edx
    call edig1
    call space_
    _DROP_
    ret

eight:  ; display eight characters for one long line in a keypad mnemonic, with a space between the groups
of four
    add edi, byte 0x0C
    call four
    call space_
    sub edi, byte 0x10
four:       ; display four characters for one line in a keypad mnemonic
    mov ecx, 0x04
four1:      ; set ecx to the required number of characters to display
    push ecx
    _DUP_
    xor _TOS_, _TOS_
    mov al, [edi+0x04]
    inc edi
    call emit_
;    call emitSF_    ; Note : The characters returned by a keypad are Shannon-Fano encoded
    pop ecx
    loop four1
    ret

displayTheStack:  ; display the stack
    mov edi, ( DATA_STACK_0 - 4 )   ; save empty stack pointer, plus one ( stack grows downwards )
.back:
    mov edx, [ main ]        ; copy the current stack pointer
    cmp [edx], edi
    jnc .forward            ; test for empty stack, meaning done
    _DUP_
    mov _TOS_, [edi]          ; fetch the value of the current stack item
    sub edi, byte 0x04      ;
```

```
    call qdot               ; display one stack item
    jmp short .back         ; next stack item
.forward:
    ret

yShift equ 3

displayBlockNumber:    ;  ( -- )     ; in the top right corner of the screen
    _DUP_
    mov _TOS_, [ v_foregroundColour ]
    _DUP_
    mov _TOS_, [ vesa_XResolution ]   ; was this  : mov _TOS_, ( scrnw )
    and _TOS_, 0xFFFF
    sub _TOS_, [ v_nine_iconw ]
    mov _SCRATCH_, _TOS_                ; save for later
    mov [ v_leftMargin ], _TOS_
    mov [ word v_gr_y ], ax
    add _TOS_, [ v_nine_iconw ]
    mov [ v_rightMargin ], _TOS_
    mov _TOS_, _SCRATCH_
    shl _TOS_, 16
    add _TOS_, yShift
    mov [ v_gr_xy ], _TOS_
    _DUP_
    mov _TOS_, [ v_washColour ]  ; so we do not see the number yet, just measure its width
;    mov _TOS_, colour_blockNumber
;    shr _TOS_, 16                 ; select the background colour in the high 16 bits
    call set_color_
    _DUP_
    mov _TOS_, [ v_blk ]
    call qdot
    mov _SCRATCH_, [ v_gr_xy ]            ; current x,y coordinate, x in high 16 bits
    shr _SCRATCH_, 16
    sub _SCRATCH_, [ v_leftMargin ]   ; _SCRATCH_ is now the width of number string, in pixels
    sub _SCRATCH_, [ v_iconw ]        ; correction...
    shl _SCRATCH_, 16
    mov _TOS_, [ vesa_XResolution ]   ; screen width in pixels
    ; and _TOS_, 0xFFFF   ; not needed because of the  shl  below
    shl _TOS_, 16
    add _TOS_, yShift
    sub _TOS_, _SCRATCH_
    mov [ v_gr_xy ], _TOS_

    _DUP_
    mov _TOS_, colour_blockNumber
    ror _TOS_, 16
    call set_color_
    _DUP_
    mov _TOS_, [ v_iconw ]
    add _TOS_, _TOS_
    _DUP_
    mov _TOS_, [ v_iconh ]
    call box_
    mov [ v_gr_xy ], _TOS_

    mov _TOS_, colour_blockNumber
    _DUP_
    call set_color_
    _DUP_
    mov _TOS_, [ v_blk ]
;    mov _TOS_, [ v_numberOfMagentas ]

    call qdot
    _DROP_
    mov [ v_foregroundColour ], _TOS_
    _DROP_
ret

; ****************************************************************************
; keyboard displays
```

```
; ****************************************************************************

showEditBox:    ; v_at set up for start coordinate of box, width and height on stack
    sub dword [ v_gr_xy ], 0x000C0004   ; move the start position left  and up   by 0xXXXXYYYY
    mov dword _SCRATCH_, [ v_foregroundColour ]
    mov dword [ v_foregroundColour ], colour_orange
    mov ecx, 2
.loop:
    push ecx
    _DUP_
    mov _TOS_, 0            ; SOS = x start position in pixels, relative to current clip "window"
    _DUP_
    mov _TOS_, [ v_iconw ]
    shl _TOS_, 3            ; multiply by 8
    add _TOS_, [ v_iconw ]  ; multiply by 9
    add _TOS_, [ v_iconw ]  ; multiply by 10
    ; TOS = length of horizontal line in pixels
    call line_
    mov ecx, [ v_iconh ]
    shl ecx, 2              ; multiply by 4
    add ecx, 4              ; draw the lower line below the text
    add dword [ v_gr_xy ], ecx ; move the start position down by 4 character heights
    pop ecx
    loop .loop

    mov dword [ v_foregroundColour ], _SCRATCH_
    ret

; keypa (d)
displayTheKeypad_:   ; the Keypad is the mnemonic at the bottom right of the display, showing the actions
of each of the 27 keys used
    call setupText_
    mov edi, [ dword currentKeypadIcons ]
    _DUP_
    mov _TOS_, [ keypad_colour ]
    call set_color_
    mov _TOS_, [ vesa_XResolution ] ; was this  : mov _TOS_, ( scrnw )
    and _TOS_, 0xFFFF
    sub _TOS_, [ v_nine_iconw ]
    sub _TOS_, 16
    mov [ v_leftMargin ], _TOS_     ; x coordinate of left margin of keypad display
    mov edx, _TOS_                  ;
    add edx, [ v_nine_iconw ]       ; x coordinate of right margin of keypad display
    mov [ v_rightMargin ], edx
    shl _TOS_, 0x10
    mov edx, [ vesa_YResolution ]   ; was this  : mov _TOS_, ( scrnw )
    and edx, 0x0000FFFF
    push _SCRATCH_
    mov _SCRATCH_, [ v_keypadY_iconh ]
    add _SCRATCH_, 10
    sub edx, _SCRATCH_              ; ( ( keypadY * iconh ) + 10 )
    add _TOS_, edx

    mov [ v_gr_xy ], _TOS_

    test byte [ v_quitMode ], 0xFF
    jz .forward
    pusha
    call showEditBox
    popa
    mov [ v_gr_xy ], _TOS_
.forward:

    pop _SCRATCH_
    call eight
    call eight
    call eight
    call cr_
;    add dword [ v_gr_xy ],  ( 4 * iconw * 0x10000 )        ; shift horizontal pixels to the right
    mov _SCRATCH_, [ v_iconw ]
```

```
    shl _SCRATCH_, ( 2 + 16 ) ; ( 4 * iconw * 0x10000 )  ; shift horizontal pixels to the right
    add dword [ v_gr_xy ], _SCRATCH_
    mov edi, [ shiftAction ]
    add edi, byte 0x0C
    mov ecx, 0x03
    call four1

    call space_
    _DUP_
    mov _TOS_, [ v_hintChar ]
    call emit_

    mov dword [ v_leftMargin ], 0x03
    mov word [ v_gr_x ], 0x03
    call displayTheStack
    mov _TOS_, [ vesa_XResolution ]   ; was this  : mov _TOS_, ( scrnw )
    and _TOS_, 0xFFFF
    sub _TOS_, [ v_twentytwo_iconw ]
    add _TOS_, 3
    mov word [ v_gr_x ], ax
    lea edi, [ ( history - 4 )]     ; the text entered so far
    mov ecx, 0x0B
    jmp dword four1

; ******************************************************************************
; Tables of keys to return when each of the 24 main keypad positions are pressed
; Note : The keypad key lists below use Shannon-Fano encoded characters
; ******************************************************************************
alphaKeypad:        ; the 'alpha' character keypad keys, the start screen for key entry
    db 'gcrl' ;
    db 'htns' ;
    db 'bmwv' ;
    db 'pyfi' ;
    db 'aoeu' ;
    db 'qkxd' ;

graphicsKeypad:     ; the 'graphics' character keypad icons (Note: not numbers, just characters)
    db '123 ' ;     Note : these are Capital (larger) numbers
    db '4560' ;
    db '789?' ;
    db ':;!@' ;
    db 'zj.,' ;
    db '*/+-' ;

decimalKeypad:      ; the decimal number entry keypad icons
    db '123 ' ;
    db '4560' ;
    db '789 ' ;
    db '    ' ;
    db '    ' ;
    db '    ' ;

hexadecimalKeypad:  ; the hexadecimal number entry keypad icons
    db '123 ' ;
    db '4560' ;
    db '789 ' ;
    db ' abc' ;
    db ' def' ;
    db '    ' ;

; ******************************************************************************
; get keyboard keys
; ******************************************************************************

letter:
    cmp al, 0x04  ; ignore 0 to 3, NOP, N, spacebar, AltGr
    js .forward
    mov edx, [ currentKeypadIcons ]
    mov al, [ _TOS_ + edx ]
.forward:
```

```
    ret

key_map_table:   ; map 8042 scan type 1 keycode to colorForth character values
    db 16, 17, 18, 19,  0,  0,  4,  5 ; 0x10 - 0x17
    db  6,  7,  0,  0,  0,  0, 20, 21 ; 0x18 - 0x1F
    db 22, 23,  0,  0,  8,  9, 10, 11 ; 0x20 - 0x27
    db  0,  0,  0,  0, 24, 25, 26, 27 ; 0x28 - 0x2F
    db  0,  1, 12, 13, 14, 15,  0,  0 ; 0x30 - 0x37 N
    db  3,  2                         ; 0x38 - 0x39 alt space

; ToDo: add a timeout to the loop
WaitToReceiveKey:      ; Wait until there is byte to receive from the keyboard controller
.back:
    in al, 0x64     ; On-board controller status read
    test al, 1      ; OBF (Output Buffer Full)
    jnz .forward    ; exit when bit 0 = 1 the On-board controller has a new character for us
    xor _TOS_, _TOS_
    call pause_    ; not ready yet, so let the other task(s) have a turn
    jmp  .back      ; jump back and try again
.forward:
;    call pause_    ; not ready yet, so let the other task(s) have a turn
    ret

v_lineOffsetTablePtr:
    dd 0 ; times 16 dd 0

lineOffsetZero:
    mov dword [ v_lineOffset ], 0x00
    ret

lineOffsetPlus:
    add dword [ v_lineOffset ], 0x0C
    ret

lineOffsetMinus:
    sub dword [ v_lineOffset ], 0x0C
    jns .forward
    call lineOffsetZero
.forward:
    ret

; ******************************************************************************
; F1 Help screens
; ******************************************************************************

help0:  ; save  v_blk , display the first help screen
    _DUP_
    cmp dword [ v_blk ], LAST_BLOCK_NUMBER     ; we are displaying the first Help screen
    je .forward
    mov _TOS_, [ v_blk ]
    mov [ v_saved_v_blk ], _TOS_
.forward:
    mov dword [ v_blk ], LAST_BLOCK_NUMBER
    _DROP_
    ret

help1:  ; display the second help screen
    mov dword [ v_blk ], ( START_BLOCK_NUMBER + 1 )
    ret

help2:  ; display the second third screen
    mov dword [ v_blk ], ( START_BLOCK_NUMBER )
    ret

help3:  ; restore the original screen being edited
    _DUP_
    mov _TOS_, [ v_saved_v_blk ]
    mov [ v_blk ], _TOS_
    _DROP_
    ret
```

```
HelpTable:
    dd help0
    dd help1
    dd help2
    dd help3

help:
    _DUP_
    mov _TOS_, [ v_help_counter ]
    and _TOS_, 0x03
    call dword [ ( _TOS_ * 4 ) + HelpTable ]
    _DROP_
    inc byte [ v_help_counter ]
    ret

; ****************************************************************************
; Editor
; ****************************************************************************

e_plus:
    call colourBlindModeToggle
    jmp abort_e

abort_e:
    ; call abort
    call c_
abort_e2:
    mov esp, RETURN_STACK_0
    call e_
    ret

EnterKeyAction:   ; ( -- )     \ action when the keyboard enter key is pressed
    ; mov byte [ v_quitMode ], 0x00     ; turn off the edit mode orange lines around the keypad
    ; _DUP_
    mov _TOS_, [ v_cad ]
    sub _TOS_, 1                ; step to before the token before the cursor
    shl _TOS_, 2                ; convert cell address to byte address
    mov _TOS_, [ _TOS_ ]
    mov _SCRATCH_, _TOS_
    and _SCRATCH_, 0x0F         ; check the token type = 3 == red

    cmp _SCRATCH_, 0x03
    je  .redEnterAction

    cmp _SCRATCH_, 0x0C         ; check the token type = 12 == magenta.  NOT WORKING YET ToDo: fix this
    je  .magentaEnterAction

    cmp _SCRATCH_, 0x04
    je  .greenEnterAction       ; check the token type = 4 == green

    jmp .skip

.greenEnterAction:
    ; and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
    call fnd_
    call loc_
    jmp .skip

.redEnterAction:
    ; and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
    call execute_
    jmp .skip

.magentaEnterAction:    ; ToDo: make this work!!!
    ; and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
    _DUP_
    mov _TOS_, [ v_cad ]
    _DUP_
```

```
    jmp .skip

    mov _TOS_, [ v_cad ]
    sub _TOS_, 2                ; step to before two tokens before the cursor
    shl _TOS_, 2                ; convert cell address to byte address
    ; mov _TOS_, [ _TOS_ ]
    ; call execute_
    _DUP_
    jmp .skip

.skip:
    _DROP_
    ret

%define FirstFkey  (59)     ; F1 = 59

FkeyTable:  ; ( c -- a )   \ function key action table
;    dd nul               ; 57
;    dd nul               ; 58
    dd help              ; 59 F1
    dd toggleBase0       ; 60 F2 decimal/hex number display
    dd seeb              ; 61 F3 show/hide blue words
    dd e_plus            ; 62 F4 editor
    dd tog_show_ASCII    ; 63 F5 show/hide the ASCII keyboard entry field at the cursor
    dd otherBlock        ; 64 F6 display the previously edited block
    dd locate            ; 65 F7 locate the definition of the word under the cursor
    dd nul               ; 66 F8
    dd toggleBase        ; 67 F9
    dd c_                ; 68 F10
    dd nul               ; 69 Num Lock
    dd nul               ; 70
    dd cursorHome        ; 71 Home
    dd cursorUp          ; 72 Up arrow
    dd nextBlock         ; 73 PgUp
    dd nul               ; 74 -
    dd cursorLeft        ; 75 Left arrow
    dd otherBlock        ; 76    display the previously edited block
    dd cursorRight       ; 77 Right arrow
    dd nul               ; 78 +
    dd cursorEnd         ; 79 End
    dd cursorDown        ; 80 Down arrow
    dd previousBlock     ; 81 PgDn
    dd destack           ; 82 Insert
    dd deleteAction      ; 83 Delete
    dd nul               ; 84
    dd nul               ; 85
    dd nul               ; 86
    dd toggleBase0       ; 87 F11
    dd nul               ; 88 F12
    dd EnterKeyAction    ; 89 really 121 Enter
    dd abort_e           ; 90 really 123 Escape

processFkey:    ; ( n -- )   \ process the given function key code
;    cmp _TOS_, 121
;    jne .forward1
;    sub _TOS_, ( 121 - 89 )
;.forward1:
    sub _TOS_, FirstFkey   ; convert Fn key value to index from 0
    and _TOS_, 0x1F
    call dword [ ( _TOS_ * 4 ) + FkeyTable ]
;    _DROP_
    ; call e_
    ret

get_key_:    ; ( -- c )   \ waits for and returns a character from the keyboard, assumes Scan Code Set 1,
set up by the BIOS
    _DUP_
    xor _TOS_, _TOS_
.back:
    ; check if the key is a function key
```

```
    cmp _TOS_, FirstFkey  ; F1 key
    js .forward4
    cmp _TOS_, FirstFkey + 32  ; Fxx key + 1
    jns .forward4
    call processFkey
.forward4:
    _DROP_
    call get_qwerty_key_
;    call WaitToReceiveKey  ; Wait until there is a byte to receive from the keyboard controller
;    in   al, 0x60       ; read the key value from the Keyboard data port
    mov al, [ v_scanCode ]
;    test al, 0xF0       ; we are only interested in certain key codes (?)
;    jz   .back
    cmp  al, 0x3A       ; exclude keycodes greater than 0x39,  cmp  is like  sub  but only affects the
flags
    jnc  .back
    mov  al, [ key_map_table - 0x10 + EAX ] ; convert to the colorForth value using the 'key_map_table'
table
    ret

; ****************************************************************************
; get qwerty keys
; ****************************************************************************

align 4, db 0   ; fill the gap with 0's

; times 0x40 db 0x00,

qwerty_key_map_table:
;         0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
    db 0x0B, 0x18, 0x02, 0x19, 0x03, 0x1A, 0x04, 0x1B, 0x05, 0x1C, 0x06, 0x1D, 0x07, 0x1E, 0x08, 0x1F ;
0x00
    db 0x09, 0x20, 0x0A, 0x21, 0x1e, 0x05, 0x30, 0x13, 0x2E, 0x0A, 0x20, 0x10, 0x12, 0x04, 0x21, 0x0E ;
0x10
    db 0x22, 0x0D, 0x23, 0x14, 0x17, 0x07, 0x24, 0x22, 0x25, 0x24, 0x26, 0x0C, 0x32, 0x09, 0x31, 0x06 ;
0x20
    db 0x18, 0x03, 0x19, 0x12, 0x10, 0x17, 0x13, 0x01, 0x1F, 0x08, 0x14, 0x02, 0x16, 0x16, 0x2F, 0x11 ;
0x30
    db 0x11, 0x0F, 0x2D, 0x15, 0x15, 0x0B, 0x2C, 0x26, 0x0C, 0x23, 0x34, 0x25, 0x35, 0x27, 0x27, 0x28 ;
0x40
    db 0x28, 0x29, 0x82, 0x2A, 0x8D, 0x2B, 0x83, 0x2C, 0x89, 0x2D, 0x33, 0x2E, 0xB5, 0x2F, 0x39, 0x80 ;
0x50
    db 0x1C, 0x81, 0x0E, 0x82, 0x01, 0x83, 0x3B, 0x84, 0x29, 0x30
    ; test only
; times 0x40 db 0x00,

get_qwerty_key_:           ; get a qwerty key character
    _DUP_
.back:
    call WaitToReceiveKey
    in al, 0x60

    cmp _TOS_, 0x1C     ; the Enter key scan code
    jne .forward1
    ; add _TOS_, ( 89 - 0x1C ) ; convert the code for the Enter key to 89
    mov _TOS_, 89
.forward1:

    cmp _TOS_, 0x81     ; the Escape key scan code
    jne .forward2
    add _TOS_, ( 90 - 0x81 ) ; convert the code for the Escape key to 90
.forward2:

;    cmp _TOS_, 0x03     ; the Left Alt key scan code
;    jne .forward3
;    add _TOS_, 0x02 ; convert the code for the Left Alt key to a space key
; .forward3:

    mov [ v_scanCode ], al
    mov ecx, _TOS_             ; copy keycode into cl
```

```
    and cl, 0x7F                ; filter out key-up bit 7
    cmp cl, 0x2A                ; g?
    jz .got_c_or_g
    cmp cl, 0x36                ; c?
    jnz .not_c_or_g
.got_c_or_g:
    and al, 0x80                ; extract key-up bit
    xor al, 0x80                ; complement it
    mov [ v_qwerty_key ], _TOS_
    jmp short .back
.not_c_or_g:
    or al, al                   ; check if key-up
    js .back                    ; if so, try again to get keydown event
    and al, 0x7F                ; filter out key-up bit
    or _TOS_, [ v_qwerty_key ]
    mov edx, qwerty_key_map_table
    mov ecx, 0x35
.back2:
    cmp [edx], al
    jz .forward
    add edx, byte 0x02
    loop .back2
    xor _TOS_, _TOS_
    ret
.forward:
    mov al, [edx+0x01]
    sub edx, qwerty_key_map_table
    shr edx, 1
    mov [ v_digin ], edx
    cmp _TOS_, 59  ; F1 key
;    jnz .forward4
;    ; jmp dword [ _TOS_ * 4 + qwertyActionTable - 0x200 ]
;    xor dword [ current], ((setBase_decimal - $$) ^ (setBase_hex - $$))
;    call toggleBase
;.forward4:
    ret

; ****************************************************************************
; keypad jump tables
; actions for the three editor state change keys : N spacebar AltGr
; ****************************************************************************

graph0:
    dd nul0, nul0, nul0, alph0
    db '  a ' ; _ _ a _ '  a ' ;

graph1:
    dd word0, exit_, lj, alph
    db 'x.a ' ;

alpha0:
    dd nul0, nul0, number, star0
    db ' 9* ' ;

alpha1:
    dd word0, exit_, lj, graph
    db 'x.* ' ;

numb0:  ; the number keypad before the '-' key has been pressed ???
    dd nul0, minusSign, alphn, toggleBase
    db '-af ' ; 0x23, 0x05, 0x0E, 0x00       ; - a f _ '-af ' ;

numb1:  ; the number keypad after the '-' key has been pressed ???
    dd number0, minusSign, endn, toggleBase
    db '-af ' ; 0x15, 0x25, 0x00, 0x00       ; x . _ _ 'x.  ' ;

; ****************************************************************************
; Shannon-Fano compression
; ****************************************************************************
```

```
bits_:
    db 0x1C

lj0:
    mov cl, [ bits_ ]
    add cl, 0x04
    shl dword [ esi ],cl
    ret

lj:
    call lj0
    _DROP_
    ret

full:
    call lj0
    inc dword [ v_words ]
    mov byte [ bits_ ], 0x1C
    sub [ bits_ ], ch
    mov _TOS_, edx
    _DUP_
    ret

pack0:
    add _TOS_, byte  0x50
    mov cl, 0x07
    jmp short pack1

pack_:
    cmp al, 0x10
    jnc pack0
    mov cl, 0x04
    test al, 0x08
    jz pack1
    inc ecx
    xor al, 0x18
pack1:
    mov edx, _TOS_
    mov ch,cl
.back:
    cmp [ bits_ ], cl
    jnc .forward
    shr al,1
    jc full
    dec cl
    jmp short .back
.forward:
    shl dword [ esi ],cl
    xor [ esi ], _TOS_
    sub [ bits_ ], cl
    ret

exit_:      ; exit to the quit loop
    call right
    mov _TOS_, [ v_words ]
    lea esi, [ esi + (_TOS_ * 4 ) ]
    _DROP_
    jmp quit_

word_:
    call right
    mov dword [ v_words ], 0x01
    mov dword [ chars ], 0x01
    _DUP_
    mov dword [ esi ], 0x00
    mov byte [ bits_ ], 0x1C
word1:
    call letter
    jns .forward
    mov edx, [ shiftAction ]
```

```
    jmp dword [edx+_TOS_*4]
.forward:
    test al,al
    jz word0
    _DUP_
    call echo_
    mov al, [ _TOS_ + ASCII_to_SF_table ]
    call pack_
    inc dword [ chars ]
word0:
    _DROP_
    call get_key_
    jmp short word1

; ****************************************************************************
; number display
; ****************************************************************************

digitTable:                         ; convert a keypad key value to a number
    times 0x30 db 0x00                                       ;
    db 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09   ; '0123456789' ; 0x30 to 0x39
    times 0x27 db 0x00
    db 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F                           ; 'abcdef'
    times 0x34 db 0x00

ASCII_to_SF_table:                  ; to convert ASCII value to ShannonFano number
   ;  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,  0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
   db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ;
0x00
   db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 ;
0x10
   db 0x00, 0x2A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x59,  0x00, 0x00, 0x2D, 0x2B, 0x2E, 0x23, 0x25, 0x27 ;
0x20
   db 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,  0x20, 0x21, 0x29, 0x28, 0x00, 0x00, 0x00, 0x2F ;
0x30
   db 0x2C, 0x35, 0x43, 0x3A, 0x40, 0x34, 0x3E, 0x3D,  0x44, 0x37, 0x52, 0x54, 0x3C, 0x39, 0x36, 0x33 ;
0x40
   db 0x42, 0x47, 0x31, 0x38, 0x32, 0x46, 0x41, 0x3F,  0x45, 0x3B, 0x56, 0x00, 0x00, 0x00, 0x00, 0x53 ;
0x50
   db 0x00, 0x05, 0x13, 0x0A, 0x10, 0x04, 0x0E, 0x0D,  0x14, 0x07, 0x22, 0x24, 0x0C, 0x09, 0x06, 0x03 ;
0x60
   db 0x12, 0x17, 0x01, 0x08, 0x02, 0x16, 0x11, 0x0F,  0x15, 0x0B, 0x26, 0x00, 0x00, 0x00, 0x00, 0x00 ;
0x70

v_sign:                             ; set to 0xXX when the '-'key is pressed on the keypad
    db 0x00

minusSign:
    ; not byte [ v_sign ]
    mov byte [ v_sign ], '-'
    jmp short number2

number0:
    _DROP_
    jmp short number3

number:
    call [ setCurrentBase ]
    mov byte [ v_sign ] , 0x00
    xor _TOS_, _TOS_
number3:
    call get_key_
    call letter
    jns .forward
    mov edx, [ shiftAction ]
    jmp dword [edx+_TOS_*4]
.forward:
    test al,al
    jz number0
    mov al, [ _TOS_ + digitTable ]
```

```
    test byte [ v_sign ], '-'
    jz .forward2
    neg _TOS_
.forward2:
    mov edx, [ esi ]
    imul edx, [ base ]
    add edx, _TOS_
    mov [ esi ], edx
number2:
    _DROP_
    mov dword [ shiftAction ], numb1
    jmp short number3

endn:
    _DROP_
    call [ anumber]
    jmp quit_

setBase_decimal:                     ; set the system base to decimal
    mov dword [ base ], 0x0A
    mov dword [ shiftAction ], numb0
    mov dword [ currentKeypadIcons ], ( decimalKeypad - 4 )
    ret

setBase_hex:                         ; set the system base to hexadecimal
    mov dword [ base ], 0x10
    mov dword [ shiftAction ], numb0
    mov dword [ currentKeypadIcons ], ( hexadecimalKeypad - 4 )
    ret

toggleBase0:
    ; the 'xor's below change the content of 'setCurrentBase_base' and the keypad icon
    xor dword [ setCurrentBase], ((setBase_decimal - $$) ^ (setBase_hex - $$))
    xor byte [ numb0 + 18 ], ( 0x39 ^ 0x66 ) ;  0x39 = '9' , 0x66 = 'f' toggle '9' and 'f' on keypad
display line
    call [ setCurrentBase ]
    ret

toggleBase:
    call toggleBase0
    jmp dword number0

; ****************************************************************************
; text entry
; ****************************************************************************

exitn_:
    _DROP_
    _DROP_
    jmp quit_

nul0:
    _DROP_
    jmp short quit2

clearHintChar:
    push _TOS_
    xor _TOS_, _TOS_
    mov byte [ v_hintChar ], 0x00   ; clear the hint character
    pop _TOS_
    ret

quit_:                               ; get a word from keypad and interpret it
    mov dword [ shiftAction ], alpha0
    lea edi, [ alphaKeypad - 4]
quit1:
    mov [ dword currentKeypadIcons ], edi
quit2:
    test dword [ x_qwerty ], 0xFFFFFFFF
    jz .forward
```

```
    jmp dword [ x_qwerty ]          ; jump to the address in x_qwerty if it is non-zero
.forward:
    call get_key_                   ; calls pause_ while waiting for a character
    cmp al, 0x04                    ;
    jns .forward2
    mov edx, [ shiftAction ]
    jmp dword [ edx + _TOS_ * 4 ]   ; alpha0 jump table element
.forward2:
    add dword [ shiftAction ], byte +0x14
    call word_
    call [ aword ]
    jmp short quit_                ; endless loop

alphn:
 _DROP_

alph0:
    mov dword [ shiftAction ], alpha0
    lea edi, [ alphaKeypad - 4 ]
    jmp short Xstar0

star0:
    mov dword [ shiftAction ], graph0
    lea edi, [ ( graphicsKeypad - 4 ) ]
    Xstar0:
    _DROP_
    jmp short quit1

alph:
    mov dword [ shiftAction ], alpha1
    lea edi, [ alphaKeypad - 4]
    jmp short Xgraph

graph:
    mov dword [ shiftAction ], graph1
    lea edi, [ ( graphicsKeypad - 4 ) ]
    Xgraph:
    mov [ currentKeypadIcons ], edi
    jmp dword word0

; Note: defining drawTheCursor as a sub-routine and calling it produces a strange bug :
; moving left 24 times using the left arrow key, from the end of the block, crashes the editor.
; I suspect that the use of the stack to store (and later replace) deleted tokens gets confused
; if a call to drawTheCursor happens occasionally...
; This code should be re-worked. It is just too delicate...
; drawTheCursor:
;     mov [ v_cad ], edi
;     push _SCRATCH_
;     mov _SCRATCH_, [ v_10000_iconw ]
;     sub dword [ v_gr_xy ], _SCRATCH_   ; move one icon's worth of horizontal pixels to the left
;     _DUP_
;     mov _SCRATCH_, [ v_foregroundColour ]   ; save the current colour
;     mov _TOS_, colour_PacMan        ; for the "PacMan" cursor
;     call set_color_
;     mov _TOS_, 0x04                 ; display the "PacMan" cursor
;     mov cx, [ v_gr_x ]
;     cmp cx, [ v_rightMargin ]
;     js .forward5
;     ; the cursor is too far to the right on the screen
;     call emit_
;     mov [ v_10000_iconw ], _SCRATCH_
;     sub dword [ v_gr_xy ], _SCRATCH_  ; move one icon's worth of horizontal pixels to the left
;     jmp .forward6
; .forward5:
;     ; the cursor can be drawn
;     call emit_
; .forward6:
;     call doShowASCII                ; optionally show the ASCII entry field
;     mov dword [ v_foregroundColour ], _SCRATCH_     ; restore the current colour
;     pop _SCRATCH_
```

```
;     ret

; ******************************************************************************
; Shannon-Fano decompression and display
; ******************************************************************************

unpack:     ; ( token -- token' nextCharacter )
    _DUP_   ; copy TOS to our data stack SOS
    test _TOS_, _TOS_
    js .forward
    shl dword [ esi ], 0x04
    rol _TOS_, 0x04
    and _TOS_, byte 0x07
    ret
.forward:
    shl _TOS_,1
    js .forward2
    shl dword [ esi ], 0x05
    rol _TOS_, 0x04
    and _TOS_, byte 0x07
    xor al, 0x08
    ret
.forward2:
    shl dword [ esi ], 0x07
    rol _TOS_, 0x06
    and _TOS_, byte 0x3F
    sub al, 0x10
    ret

; show the PacMan-like cursor
show_cursor:  ; ( a cursor -- a' )  edi  contains pointer to current address to display
    _DUP_
    inc dword [ esi ]
    cmp [ v_curs ], edi
    jnz .forward                ; address to display = cursor address?
    mov [ v_curs ], _TOS_       ; yes,
.forward:
    cmp _TOS_, [ v_curs ]       ; time to draw the cursor?
    jz .forward2
    jns .forward4               ; time to draw the cursor?
    mov [ v_pcad ], edi         ; no, so exit
.forward4:
    _DROP_
    ret                         ; exit here

.forward2:
    ; call drawTheCursor ; Note: do not do this!!! See notes for drawTheCursor:
    mov [ v_cad ], edi
    push _SCRATCH_
    mov _SCRATCH_, [ v_10000_iconw ]
    sub dword [ v_gr_xy ], _SCRATCH_  ; move the graphic position one icon's worth of horizontal pixels to
the left
    _DUP_
    mov _SCRATCH_, [ v_foregroundColour ]   ; save the current colour
    mov _TOS_, colour_PacMan
    call set_color_
    mov _TOS_, 0x04   ; display the "PacMan" cursor
    mov cx, [ v_gr_x ]
    cmp cx, [ v_rightMargin ]
    js .forward5
    call emit_
    mov _SCRATCH_, [ v_10000_iconw ]
    sub dword [ v_gr_xy ], _SCRATCH_  ; move one icon's worth of horizontal pixels to the left
    jmp .forward6
.forward5:
    call emit_
.forward6:
    mov dword [ v_foregroundColour ], _SCRATCH_     ; restore the current colour
    pop _SCRATCH_
    ret
```

```
    ret

; ****************************************************************************
; Conventional Forth display (does not require colours) - colour-blind mode
; ****************************************************************************

currentState:   ; the current token colour
    dd 0

lastState:      ; the last token colour
    dd 0

txt0:
    call white
    EMIT_IMM('(')
    call space_
    ret

txt1:
    call white
    EMIT_IMM(')')
    call space_
    ret

imm0:
    call yellow
    EMIT_IMM('[')
    call space_
    ret

imm1:
    call yellow
    EMIT_IMM(']')
    call space_
    ret

mvar0:
    call yellow
    EMIT_IMM('[')
    call space_
    EMIT_IMM('m')
    EMIT_IMM('v')
    EMIT_IMM('a')
    EMIT_IMM('r')
    call space_
    ret

mvar1:
    call yellow
    EMIT_IMM(']')
    call space_
    ret

; unfortunately we need to display the ':' after the CR, so must do this in  redWord , not here
; colon0:
;     call red
;     EMIT_IMM(':')
;     call space_
;     ret
;
;     dd nul, imm0, nul, colon0, nul, nul, nul, nul, nul, txt0, nul, nul, mvar0, nul, nul, nul

txts:
    db 0, 1, 1, 3, 4, 5, 6, 7, 1, 9, 9, 9, 12, 13, 14, 15

tx:     ; ( c -- c )   \ return the value in the given offset in  txts
    and _TOS_, 0xFF
    mov _TOS_, [ _TOS_ + txts ]
    and _TOS_, 0xFF
    ret
```

```
newActions:
    dd nul, imm0, nul, nul, nul, nul, nul, nul, nul, txt0, nul, nul, mvar0, nul, nul, nul

dotNew:     ; ( state -- )
    call [ ( _TOS_ * 4 ) + newActions ]
    ret

oldActions:
    dd nul, imm1, nul, nul, nul, nul, nul, nul, nul, txt1, nul, nul, mvar1, nul, nul, nul

dotOld:     ; ( state -- )
    call [ ( _TOS_ * 4 ) + oldActions ]
    ret

colourBlindAction:  ; ( state -- state )    \ perform the required action on change of state
    push _SCRATCH_
    _DUP_
    call tx
    cmp _TOS_, 0x00
    jz .end                             ; no action on extension tokens, value 0
    mov _SCRATCH_, [ currentState ]
    mov [ currentState ], _TOS_
    cmp _SCRATCH_, [ currentState ]     ; compare the new state on TOS to the last one saved in
currentState
    jz .end                             ; exit if there has been no change of state
    _DUP_
    mov _TOS_, _SCRATCH_
    call dotOld                         ;
    mov _TOS_, [ currentState ]
    call dotNew
    _DROP_
    cmp byte [ currentState ], 0x0000
    jz .end
    mov _SCRATCH_, [ currentState ]
    mov [ lastState ], _SCRATCH_
 .end:
    _DROP_
    pop _SCRATCH_
    ret

; \ Block 70
; ( Colourblind Editor Display )
; #1 MagentaV currentState $01 MagentaV lastState
; : +txt white $6D emit space ;
; : -txt white $6E emit space ;
; : +imm yellow $58 emit space ;
; : -imm yellow $59 emit space ;
; : +mvar yellow $09 emit $11 emit $05 emit $01 emit space ;
; : txts string $03010100 , $07060504 , $09090901 , $0F0E0D0C , ( ; )
; : tx ( c-c ) $0F and txts + 1@ $0F and ;
; : .new currentState @ $0F and jump nul +imm nul nul nul nul nul nul nul +txt nul nul +mvar nul nul nul ;
; : .old lastState @ $0F and jump nul -imm nul nul nul nul nul nul nul -txt nul nul nul nul nul nul ;
; here
; : cb ( n-n ) #0 + 0if ; then tx
;    currentState @ swap dup currentState ! - drop if .old .new
;    currentState @ #0 + if dup lastState ! then then ;
; : cbs ( -- here ) #0 + $00 + cblind ! ;

; colourBlind:    ; ( state -- state )    \ vectored colorForth to display colourBlind extra characters (
e.g. ':' for red words )
;    call dword [ x_colourBlind ]
;    ret

; ******************************************************************************
; Show an ASCII editable entry field at the cursor
; ******************************************************************************

ShowASCIIAction:    ; ( -- )
;    call white
```

```
    call space_
    EMIT_IMM('U')
    EMIT_IMM('U')
    EMIT_IMM('U')
    EMIT_IMM('U')
    call space_
    ret
    ret

; ******************************************************************************
; ******************************************************************************

lowercase:   ; display a white text word in normal lower-case letters
    call white
showSF_EDI_:  ; ( -- )   \ display a Shanon-Fano encoded token pointed to by  edi  in the current colour
    _DUP_
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]   ; fetch the next token - drops through to showShannonFano

showShannonFano:    ; ( token -- ) \ display the Shannon-Fano encoded token on TOS
    ; ASCII / UTF8 support. If the first Shannon-Fano encoded letter is a 4 bit NULL,
    ; display the next 24 bits as three ASCII characters.
    mov _SCRATCH_, _TOS_           ; save the token value
    and _SCRATCH_, 0xF0000000
    cmp _SCRATCH_, 0x00000000
    jnz .forward
        ; display as three ASCII characters
        mov _SCRATCH_, _TOS_

        mov _TOS_, _SCRATCH_
        shr _TOS_, 20
        and _TOS_, 0x000000FF
        jz .null_terminator
            _DUP_
            call emit_

        mov _TOS_, _SCRATCH_
        shr _TOS_, 12
        and _TOS_, 0x000000FF
        jz .null_terminator
            _DUP_
            call emit_

        mov _TOS_, _SCRATCH_
        shr _TOS_, 4
        and _TOS_, 0x000000FF
        jz .null_terminator
            _DUP_
            call emit_

        ; arrive here if an ASCII character is an ASCII NULL, or if all three have been emitted
        .null_terminator:
        call space_                 ; display a space character at the end of the word
        _DROP_
        ret

    .forward:

    ; display as Shannon-Fano encoded token name
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )  ignore the token colour
when displaying the letters
lowercasePrimitive:  ; ( token -- ) \ display the given Shanon-Fano encoded word in the current colour
    call unpack
    jz lowercasePrimitiveEnd
    call emitSF_
    jmp lowercasePrimitive
lowercasePrimitiveEnd:
    call space_
    _DROP_
    _DROP_
    ret
```

```
typeNumber32tok:     ; ( token -- ) \ display the given Shanon-Fano encoded word as a number in the
current colour
    _DROP_ ; call dotHex8_
    mov dword [ lastTokenWasLiteral ], 0xFFFFFFF
    ret

typeNumber32:     ; ( token -- ) \ display the given Shanon-Fano encoded word as a hex number in the
current colour
    call dotHex8_
    mov dword [ lastTokenWasLiteral ], 0x00000000
    ret

typeNumber27:     ; ( token -- ) \ display the given Shanon-Fano encoded word as a 27 bit hex number in
the current colour
    shr _TOS_, 5
    call dotHex
    ret

lastTokenWasLiteral:
    dd 0x00

lastShannonFanoToken:
    dd 0x00

magentaPrimitive:   ; ( token -- )
    call showShannonFano
    mov dword [ lastTokenWasLiteral ], 0xFFFFFFF
    ret

displayOneShannonFanoActions:   ;    * = number
    dd showShannonFano          ; 0     extension token, remove space from previous word, do not change
the colour
    dd showShannonFano          ; 1     yellow "immediate" word
    dd typeNumber32tok          ; 2  *  yellow "immediate" 32 bit number in the following pre-parsed cell
    dd showShannonFano          ; 3     red forth wordlist "colon" word
    dd showShannonFano          ; 4     green compiled word
    dd typeNumber32tok          ; 5  *  green compiled 32 bit number in the following pre-parsed cell
    dd typeNumber27             ; 6  *  green compiled 27 bit number in the high bits of the token
    dd showShannonFano          ; 7     cyan macro wordlist "colon" word
    dd typeNumber27             ; 8  *  yellow "immediate" 27 bit number in the high bits of the token
    dd showShannonFano          ; 9     white lower-case comment
    dd camelcasePrimitive       ; A     first letter capital comment
    dd uppercasePrimitive       ; B     white upper-case comment
    dd magentaPrimitive         ; C     magenta variable
    dd showShannonFano          ; D
    dd showShannonFano          ; E     editor formatting commands
    dd showShannonFano          ; F

times 0x20 db 0x55
testme:
    dd 0x75240CFF ; 0xFF, 0x0C, 0x24, 0x75
    dd 0x123456
    ret
times 0x20 db 0x77

leave_:     ; terminate a  for ... next  loop
    mov dword [ esp + 4 ], 0x01
    ret

dotsf_:  ; ( token -- )   \ display the given Shannon-Fano encoded word in the token's colour
    push edi
    mov edx , _TOS_
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
    _DUP_
    mov edi, [ lastTokenWasLiteral ]
    test edi, 0x00000000
    jz .forward3
    mov edx, 0
.forward3:
```

```
    and edx, byte 0x0F
    jnz .forward     ; do not change the colour if this is an extension token
        ; this is an extension token
        mov edx, [ lastShannonFanoToken ]
        ; if the colour is Camelcase 0x0A, make it lowercase 0x09
        ; e.g. Interrupt would be shown as InterrUpt if the exension token is displayed with an initial
Capital
        mov _SCRATCH_, edx
        and _SCRATCH_, 0x0F     ; just the colour
        sub _SCRATCH_, 0x0A
        jne .foward4
            and edx, 0xFFFFFFF0     ; remove the colour
            or edx, 0x00000009      ; make it lowercase
        .foward4:
        mov _SCRATCH_, [ v_10000_iconw ]
        sub dword [ v_gr_xy ], _SCRATCH_   ; move iconw horizontal pixels back, to remove the space at the
end of the last word
    jmp .forward2
    .forward:
        ; this is not an extension token
        mov [ lastShannonFanoToken ], edx
    .forward2:
    push _TOS_
    mov _TOS_, [ ( edx * 4 ) + actionColourTable ]
    call set_color_
    pop _TOS_
    call [ ( edx * 4 ) + displayOneShannonFanoActions ]
    pop edi
    ret

redWord:     ; display a red word
    mov cx, [ v_gr_x ]
    cmp cx, [ v_leftMargin ]
    jz .forward     ; do not do a  cr  if we are already at the left margin
    mov cl, [ v_not_cr ]
    cmp cl, 0
    jnz .forward    ; do not do a  cr  if it has been disabled by a blue  -cr  token
    call cr_
.forward:
    mov byte [ v_not_cr ], 0
    call setRed

    cmp byte [ v_colourBlindMode ], 0x00
    jz .forward2
    test byte [ v_blk ], 0x01   ; do not display colour-blind characters in odd numbered shadow blocks
    jnz .forward2
    EMIT_IMM(':')   ; emit a ':' if in colour-blind mode
    call space_
.forward2:
    jmp showSF_EDI_

greenWord:     ; display a green word
    call setGreen
    jmp showSF_EDI_

cyanWord:     ; display a cyan word
    call setCyan
    jmp showSF_EDI_

yellowWord:     ; display a yellow word
    call yellow
    jmp showSF_EDI_

; Note : Camelcase tokens do not support ASCII output
camelcase:    ; display a white word with the first letter Capitalised
    call white
    _DUP_
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
camelcasePrimitive:
```

```
    call unpack
    add al, 0x30                    ; make the first character upper case
    call emitSF_                    ; display it
camelcasePrimitive_2:               ; display the rest of the word
    call unpack
    jz lowercasePrimitiveEnd
    call emitSF_
    jmp camelcasePrimitive_2

; Note : UPPERCASE tokens do not support ASCII output
uppercase:   ; display a white word with all letters CAPITALISED
    call white
    _DUP_
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
uppercasePrimitive:
    call unpack
    jz lowercasePrimitiveEnd
    add al, 0x30
    call emitSF_
    jmp uppercasePrimitive

extension:  ; display an extension token, do not change the colour
    mov _SCRATCH_, [ v_10000_iconw ]
    sub dword [ v_gr_xy ], _SCRATCH_   ; move iconw horizontal pixels back, to remove the space at the end
of the last word
    test dword [ ( edi * 4 ) - 0x04 ], 0xFFFFFFF0
    jnz showSF_EDI_
    dec edi
    mov [ v_lcad ], edi
    call space_
    call show_cursor                ; show the PacMan-like cursor
    pop edx                         ; EXIT from calling word
    _DROP_                          ; the ret below will return to the word that called  extension
    ret                             ; so it looks like it never happened

greenShortNumber:    ; display the green compiled 27 bit number in the high bits of the token
    mov edx, [ ( edi * 4 ) - 0x04 ]
    sar edx, 0x05
    jmp short greenNumber1

magentaVariable:    ; display a magenta variable using the 32 bit number in the following pre-parsed cell
    mov dword [ x_numberDisplay ], dotDecimal
    cmp dword [ base ], byte 0x0A   ; check the current BASE value ( 10 or 16 for decimal or hex)
    jz .forward
    mov dword [ x_numberDisplay ], dotHex
.forward:
    call setMagenta
    call showSF_EDI_              ; display the name of the variable
    mov edx, [ ( edi * 4 ) + 0x00 ]    ; load the value of the variable from the pre-parsed source
    inc edi                 ; step over the variable value in the pre-parsed source
    call setMagentaData
    jmp short displayNumber

greenNumber:                ; display the value of a hexadecimal/decimal number in green
    mov edx, [ ( edi * 4 ) + 0x00 ]    ; load the value of the variable from the pre-parsed source
    inc edi                 ; step over the variable value in the pre-parsed source
greenNumber1:
    call green
    jmp short displayNumber

yellowShortNumber:
    mov edx, [ ( edi * 4 ) - 0x04 ] ; load the value of the number from the current token in the pre-
parsed source
    sar edx, 0x05                   ; remove the token colour bits
    jmp short yellowNumber1

yellowNumber:     ; ( -- ) display a number word, constant value following in the pre-parsed source
    mov edx, [ ( edi * 4 ) + 0x00 ]    ; load the value of the number from the pre-parsed source
    inc edi                 ; step over the number value in the pre-parsed source
```

```
yellowNumber1:    ; ( -- ) display a yellow number word
    call yellow
displayNumber: ; ( rgb -- )   display the number in edx  with the given colour, using the base implied in
x_numberDisplay
    _DUP_
    mov _TOS_, edx
    ; jmp qdot
    jmp dword [ x_numberDisplay ]

; *****************************************************************************
; Blue words - formatting the editor display
; *****************************************************************************

get_x:  ; ( -- c )  \ return the current x character position
    push edx
    _DUP_
    xor _TOS_, _TOS_
    mov ax, word [ v_gr_x ]
    xor edx, edx                    ; clear high 32 bits of dividend
    div dword [ v_iconw ]           ; EDX:EAX divided by the icon width , EAX now contains the current
character position, EDX the remainder
    pop edx
    ret

set_x:  ; ( c -- )  \ set the current x character position
    push edx
    xor edx, edx
    mul dword [ v_iconw ]
    mov word [ v_gr_x ], ax
    pop edx
    _DROP_
    ret

tab_n:    ; ( n -- )    \ align to the next n character column
    mov ecx, _TOS_
    pusha
    call get_x
    xor edx, edx                    ; clear high 32 bits of dividend
    mov _SCRATCH_, ecx
    div _SCRATCH_
    mul _SCRATCH_
    add _TOS_, ecx
    call set_x
    popa
    _DROP_
    ret

tab3:       ; ( -- )
   _DUP_
   mov _TOS_, 0x03
   call tab_n
   ret

tab4:       ; ( -- )
   _DUP_
   mov _TOS_, 0x04
   call tab_n
   ret

tab5:       ; ( -- )
   _DUP_
   mov _TOS_, 0x05
   call tab_n
   ret

tab6:       ; ( -- )
   _DUP_
   mov _TOS_, 0x06
   call tab_n
   ret
```

```
tab7:       ; ( -- )
   _DUP_
   mov _TOS_, 0x07
   call tab_n
   ret

tab8:       ; ( -- )
   _DUP_
   mov _TOS_, 0x08
   call tab_n
   ret

%define TAB_SIZE  24

tab:        ; ( -- )
   _DUP_
   mov _TOS_, TAB_SIZE
   call tab_n
   ret

; tab:    ; ( -- )    \ align to the next n character column
;     pusha
;     call get_x
;     xor edx, edx                    ; clear high 32 bits of dividend
;     mov _SCRATCH_, TAB_SIZE
;     div _SCRATCH_
;     mul _SCRATCH_
;     add _TOS_, TAB_SIZE
;     call set_x
;     popa
;     ret

; tab3:
;     pusha
;     call get_x
;     xor edx, edx                    ; clear high 32 bits of dividend
;     mov _SCRATCH_, 0x03
;     div _SCRATCH_
;     mul _SCRATCH_
;     add _TOS_, 0x03
;     call set_x
;     popa
;     ret
;
; tab6:
;     pusha
;     call get_x
;     xor edx, edx                    ; clear high 32 bits of dividend
;     mov _SCRATCH_, 0x06
;     div _SCRATCH_
;     mul _SCRATCH_
;     add _TOS_, 0x06
;     call set_x
;     popa
;     ret
;
; tab7:
;     pusha
;     call get_x
;     xor edx, edx                    ; clear high 32 bits of dividend
;     mov _SCRATCH_, 0x07
;     div _SCRATCH_
;     mul _SCRATCH_
;     add _TOS_, 0x07
;     call set_x
;     popa
;     ret
;
; tab8:
```

```
;     pusha
;     call get_x
;     xor edx, edx                    ; clear high 32 bits of dividend
;     mov _SCRATCH_, 0x08
;     div _SCRATCH_
;     mul _SCRATCH_
;     add _TOS_, 0x08
;     call set_x
;     popa
;     ret

br:          ; ( -- )
    call cr_
    call cr_
    ret

not_cr:      ; ( -- )
    not byte [ v_not_cr ]
    ret

not_tab:     ; ( -- )
    call not_cr
    call tab
    ret

cr_plus:     ; ( -- )
    call cr_
    call space_
    call space_
    call space_
    ret

four_spaces:        ; ( -- )
    call space_
three_spaces:       ; ( -- )
    call space_
two_spaces:         ; ( -- )
    call space_
    call space_
    ret

BlueNames:    ; name   routine      comment
    dd 0x9080000E  ; cr     cr_          move to the next line
    dd 0xE64B8C0E  ; -tab   not_tab      prevent the next CR
    dd 0x25C6000E  ; tab    tab          align to next TAB_SIZE space column
    dd 0xC620000E  ; br     br           cr cr
    dd 0xE721000E  ; -cr    not_cr       prevent the next CR
    dd 0x90FB000E  ; cr+    cr_plus      cr and 3 spaces
    dd 0x25C7AC0E  ; tab3   tab3         align to next 3 space column
    dd 0x25C7B00E  ; tab4   tab4         align to next 4 space column
    dd 0x25C7B40E  ; tab5   tab5         align to next 5 space column
    dd 0x25C7B80E  ; tab6   tab6         align to next 6 space column
    dd 0x25C7BC0E  ; tab7   tab7         align to next 7 space column
    dd 0x25C7C00E  ; tab8   tab8         align to next 8 space column
    dd 0xEA00000E  ; .      space_
    dd 0xEBD4000E  ; ..     two_spaces
    dd 0xEBD7A80E  ; ...    three_spaces
    dd 0xEBD7AF5E  ; ....   four_spaces

BlueJumpTableROM:  ; name_SF     name    comment
    dd cr_          ; 0x9080000E  cr      move to the next line
    dd not_tab      ; 0xE64B8C0E  -tab    prevent the next CR
    dd tab          ; 0x25C6000E  tab     align to next TAB_SIZE space column
    dd br           ; 0xC620000E  br      cr cr
    dd not_cr       ; 0xE721000E  -cr     prevent the next CR
    dd cr_plus      ; 0x90FB000E  cr+     cr and 3 spaces
    dd tab3         ; 0x25C7AC0E  tab3    align to next 3 space column
    dd tab4         ; 0x25C7B00E  tab4    align to next 4 space column
    dd tab5         ; 0x25C7B40E  tab5    align to next 5 space column
    dd tab6         ; 0x25C7B80E  tab6    align to next 6 space column
```

```
    dd tab7         ; 0x25C7BC0E  tab7    align to next 7 space column
    dd tab8         ; 0x25C7C00E  tab8    align to next 8 space column
    dd space_       ; 0xEA00000E  .       space_
    dd two_spaces   ; 0xEBD4000E  ..      two_spaces
    dd three_spaces ; 0xEBD7A80E  ...     three_spaces
    dd four_spaces  ; 0xEBD7AF5E  ....    four_spaces
BlueJumpTableROM_end:

find_Blue_word_:   ; ( sf -- index )   \ ecx = index ; find the Shannon-Fano word sf in the Blue wordlist,
return its index in ecx
    push edi
    mov ecx, 16   ; count of Blue wordlist words
    lea edi, [ ( ecx * 4 ) + BlueNames - 4 ]   ; set edi to the top of the Blue name table
    std                    ; scan backwards
    repne scasd            ; find the 32 bit Shanon-Fano encoded name, compare eax with doubleword at
es:edi and set status flags.
    cld                    ; reset the direction flag
    mov _TOS_, ecx
    pop edi
    ret

blueWord:   ; ( -- )    \ format the editor display screen using certain blue tokens. ToDo: This should ba
a table...
    _DUP_
    mov al, [ v_seeb ]
    cmp al, 0
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]   ; fetch the next token - then calls showShannonFano
    jz .forward
    call setBlue
    _DUP_
    call showShannonFano
.forward:
    mov _TOS_, [ ( edi * 4 ) - 0x04 ]
    call find_Blue_word_             ; returns the index TOS
    call [ ( _TOS_ * 4 ) + BlueJumpTableROM ]
    _DROP_
    ret

; silverWord:   ; ( -- )    ; ToDo: find out what this is for (GreenArrays gray token?)
;     mov edx, [ ( edi * 4 ) - 0x04 ] ; load the value of the action from the current token in the pre-
parsed source
;     sar edx, 0x05                   ; remove the token colour bits
;     _DUP_
;     mov _TOS_, colour_white
;     cmp dword [ x_numberDisplay ], dotDecimal
;     jz .forward
;     mov _TOS_, colour_silver
; .forward:
;    jmp short displayNumber
;    ret

silverWord:     ; display a silver word
    call setSilver
    jmp showSF_EDI_

displayShannonFanoActions:  ;    * = number
    dd extension            ; 0     extension token, remove space from previous word, do not change the
colour
    dd yellowWord           ; 1     yellow "immediate" word
    dd yellowNumber         ; 2  *  yellow "immediate" 32 bit number in the following pre-parsed cell
    dd redWord              ; 3     red forth wordlist "colon" word
    dd greenWord            ; 4     green compiled word
    dd greenNumber          ; 5  *  green compiled 32 bit number in the following pre-parsed cell
    dd greenShortNumber     ; 6  *  green compiled 27 bit number in the high bits of the token
    dd cyanWord             ; 7     cyan macro wordlist "colon" word
    dd yellowShortNumber    ; 8  *  yellow "immediate" 27 bit number in the high bits of the token
    dd lowercase            ; 9     white lower-case comment
    dd camelcase            ; A     first letter capital comment
    dd uppercase            ; B     white upper-case comment
    dd magentaVariable      ; C     magenta variable
```

```
    dd silverWord           ; D
    dd blueWord             ; E     editor formatting commands
    dd nul                  ; F

v_lineOffset:
    dd 1    ; the top line of the display

doColourBlind:  ; ( state -- )   \ add conventional Forth punctuation based on the new and last states
    cmp byte [ v_colourBlindMode ], 0x00
    jz .forward3
    test byte [ v_blk ], 0x01       ; do not display colour-blind characters in odd numbered shadow blocks
    jnz .forward3
    call dword colourBlindAction    ; pass the new state to colourBlind so that extra characters can be
added to the display
    .forward3:
    _DROP_
    ret

doShowASCII:    ;
    cmp byte [ v_show_ASCII ], 0x00
    jz .forward4

    call dword ShowASCIIAction    ; pass the new state to colourBlind so that extra characters can be
added to the display
    .forward4:
    ret

plusList:   ; ( -- ) display the current colorForth block
    _DUP_
    xor _TOS_, _TOS_                        ; set Top Of Stack to 0
    mov [ currentState ], _TOS_             ; initialise the colour-blind state machine
    mov [ lastState ], _TOS_                ; initialise the colour-blind state machine
    mov [ v_display_token_number ], _TOS_   ; initialise the displayed token nunber
    _DROP_

    call setupText_                 ; setup the clip window for this display
    _DUP_
    mov _TOS_, [ v_lcad ]
    mov [ v_cad ], _TOS_
    mov _TOS_, [ v_blk ]                    ; get the current block number to be edited
    call blockToCellAddress                 ; add the RELOCATED block number offset and convert to cell
address
    mov edi, _TOS_
    xor _TOS_, _TOS_
    add edi, [ v_lineOffset ]
    mov [ v_pcad ], edi
.back:
    mov edx, dword [ ( edi * 4 ) + 0x00 ]   ; edi is the display pointer and is a cell address
    call show_cursor                        ; show the PacMan-like cursor
    inc edi

    inc word [ v_display_token_number ]     ; count up the number of tokens displayed
    cmp word [ v_display_token_number ], 0xE0   ; save the last 128 bytes for version information
    jne .forward4
        call setBlue
        EMIT_IMM('<')   ; warm the user that we have hit the limit of the block display "<<"
        EMIT_IMM('<')
        ret     ; we have displayed enough tokens now
.forward4:

    ; adjust the number base according to bit 5 of the token value, only used by number display words
    ; this section of code displays numbers in the base that they were defined as, not according to the
current base
    mov dword [ x_numberDisplay ], dotDecimal   ; set the display base to decimal
    test dl, 0x10
    jz .forward2
    mov dword [ x_numberDisplay ], dotHex       ; set the display base to hexadecimal (overwrites
dotDecimal that was just set)
.forward2:
```

```
    and edx, byte 0x0F
    _DUP_
    mov _TOS_, edx
    call doColourBlind
    call [ ( edx * 4 ) + displayShannonFanoActions ]
    jmp short .back

refresh:                            ; refresh the editor display
    call show                       ; set the screen task to execute the code following (in a repeating
loop) :
    call page_                      ; clear the screen
    call displayBlockNumber         ; display the current block number on the screen, top right corner
    call plusList                   ; list the contents of the block
    _DUP_
    mov _TOS_, 0x0F
    call doColourBlind             ; display the final colour-blind punctuation, set up for next call of
plusList
    jmp dword displayTheKeypad_

align 4, db 0   ; fill the gap with 0's

actionColourTable:          ;    * = number
    dd colour_orange        ; 0     extension token, remove space from previous word, do not change the
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 cf2022
    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

vector:
    dd 0  ; pointer to call table for keypad ( see keypd )

action:
    db 1

align 4, db 0   ; fill the gap with 0's

cursorLeft:     ; ( -- )
    dec dword [ v_curs ]
    jns .forward
        inc dword [ v_curs ]
    .forward:
    ret

limitToEndOfBlock:
    call countTokens
    cmp _TOS_, dword [ v_curs ]
    jns .forward
        mov dword [ v_curs ], _TOS_
    .forward:
    _DROP_
    ret

cursorRight:
    inc dword [ v_curs ]
    call limitToEndOfBlock
    ret

countAllTokens:     ; ( -- x ) \ counts red and magenta tokens and all tokens in the current block
```

```
    _DUP_
    xor _TOS_, _TOS_
    mov dword [ v_numberOfMagentas ], _TOS_
    mov dword [ v_numberOfRedAndMagentas ], _TOS_   ; count up Red and Magenta tokens
    mov dword [ v_numberOfTokens ], _TOS_           ; count all tokens
    mov dword [ v_numberOfBigConstants ], _TOS_     ; count of 32 bit literal tokens

    mov ecx, 0x00100      ; 256 x 4 byte cells = 1 block
.loop:

    _DUP_
    mov _TOS_, [ v_numberOfTokens ]
    call nth_to_token
    mov _SCRATCH_, _TOS_
    _DROP_
    cmp _SCRATCH_, 0x00
    je .forward         ; exit if the token value is 0, means end of block

    inc dword [ v_numberOfTokens ]

    and _SCRATCH_, 0x0F     ; look at the token type

    cmp _SCRATCH_, 0x03     ; red token
    jne .forwardRed
        inc dword [ v_numberOfRedAndMagentas ]
    .forwardRed:

    cmp _SCRATCH_, 0x0C     ; magenta token
    jne .forwardMagenta
        inc dword [ v_numberOfRedAndMagentas ]
        inc dword [ v_numberOfMagentas ]    ; correction for magenta variables
        inc dword [ v_numberOfTokens ]      ; step over the Magenta variable data cell
    .forwardMagenta:

    cmp _SCRATCH_, 0x02     ; yellow 32 bit literal
    jne .forwardBig
        inc dword [ v_numberOfBigConstants ]    ; correction for literal constants
        inc dword [ v_numberOfTokens ]          ; step over the data cell
    .forwardBig:

    cmp _SCRATCH_, 0x05     ; green 32 bit literal
    jne .forwardBig2
        inc dword [ v_numberOfBigConstants ]    ; correction for literal constants
        inc dword [ v_numberOfTokens ]          ; step over the data cell
    .forwardBig2:

    loop .loop
.forward:               ; found the end of the block
;    mov _TOS_, dword [ v_numberOfRedAndMagentas ]
    ret

countRedAndMagentaTokens:     ; ( -- n ) \ counts red and magenta tokens in the current block
    call countAllTokens
    mov _TOS_, dword [ v_numberOfRedAndMagentas ]
    ret

countTokens:     ; ( -- n ) \ counts all tokens up to the end of the current block
    call countAllTokens
    mov _TOS_, dword [ v_numberOfTokens ]
    sub _TOS_, dword [ v_numberOfMagentas ]
    sub _TOS_, dword [ v_numberOfBigConstants ]
    and _TOS_, 0x00003FF   ; limit the maximum numer of tokens, just in case
    ret

; ******************************************************************************

cursorDownToNth:     ; ( -- ) \ step down to after the v_cursLine'th red or magenta token
    _DUP_
    xor _TOS_, _TOS_
    mov dword [ v_numberOfMagentas ], _TOS_
```

```
    mov dword [ v_curs ], _TOS_
    mov dword [ v_numberOfBigConstants ], _TOS_

    mov dword _TOS_, [ v_cursLine ]
    mov dword [ v_curs_number_down ], _TOS_

    mov ecx, 0x00100      ; 256 x 4 byte cells = 1 block
.loop:

    cmp dword [ v_curs_number_down ], 0x00  ; test for zero
    je .forward     ; jump to the end if  v_curs_number_down  reaches zero

    _DUP_
    mov _TOS_, [ v_curs ]
    call nth_to_token
    mov _SCRATCH_, _TOS_
    _DROP_
    cmp _SCRATCH_, 0x00
    je .endOfBlock         ; exit if the token value is 0, means end of block

    inc dword [ v_curs ]

    and _SCRATCH_, 0x0F     ; look at the token type

    cmp _SCRATCH_, 0x03     ; red token
    jne .forwardRed
        dec dword [ v_curs_number_down ]
    .forwardRed:

 ;   cmp _SCRATCH_, 0x0E     ; blue token
 ;   jne .forwardBlue
 ;       dec dword [ v_curs_number_down ]
 ;   .forwardBlue:

    cmp _SCRATCH_, 0x0C     ; magenta token
    jne .forwardMagenta
        dec dword [ v_curs_number_down ]
        inc dword [ v_numberOfMagentas ]    ; correction for magenta variables
        inc dword [ v_curs ]                ; step over the Magenta variable data cell
    .forwardMagenta:

    cmp _SCRATCH_, 0x02     ; yellow 32 bit literal
    je .forwardBig

    cmp _SCRATCH_, 0x05     ; green 32 bit literal
    jne .forwardBig2
    .forwardBig:
        inc dword [ v_numberOfBigConstants ]    ; correction for literal constants
        inc dword [ v_curs ]                    ; step over the data cell
    .forwardBig2:

    loop .loop
.forward:               ; found the right number of red or magenta tokens, so exit
    mov _SCRATCH_, dword [ v_numberOfMagentas ]
    add _SCRATCH_, dword [ v_numberOfBigConstants ]
    sub dword [ v_curs ], _SCRATCH_  ; the correction for magenta variables
.endOfBlock:
    call limitToEndOfBlock
    _DROP_
    ret

cursorUp:     ; ( -- ) \ step down to after the next red token, or after 0x16 steps, or until the end of
the block
    dec dword [ v_cursLine ]
    jnz .forward
    mov dword [ v_cursLine ], 0x00
.forward:
;    mov dword [ v_cursLine ], 0x03
    call cursorDownToNth
    ret
```

```
cursorDown:     ; ( -- ) \ step down to after the next red token, or after 0x16 steps, or until the end of
the block
    inc dword [ v_cursLine ]
    call countRedAndMagentaTokens
    inc dword _TOS_     ; add one so that we can go past the last token to the end of the block
    cmp dword [ v_cursLine ], _TOS_
    js .forward
    mov dword [ v_cursLine ], _TOS_
.forward:
    _DROP_
 ;   mov dword [ v_cursLine ], 0x02
    call cursorDownToNth
    ret

cursorEnd:     ; ( -- )
    call countRedAndMagentaTokens
    inc dword _TOS_     ; add one so that we can go past the last token to the end of the block
    mov dword [ v_cursLine ], _TOS_
    _DROP_
    call cursorDownToNth
    call limitToEndOfBlock
    ret

cursorHome:     ; ( -- )
    xor _SCRATCH_, _SCRATCH_
    mov dword [ v_numberOfMagentas ], _SCRATCH_
    mov dword [ v_curs ], _SCRATCH_                ; the graphics cursor for drawing the block
    mov dword [ v_lineOffset ], _SCRATCH_          ; the cursor position to start drawing the block
    mov dword [ v_lineOffsetTablePtr ], _SCRATCH_   ; a pointer to the cursor for each line in the display
    mov dword [ v_numberOfMagentas ], _SCRATCH_     ; count of Magenta variables displayed so far in the
edited block
    mov dword [ v_cursLine ], _SCRATCH_
    ret

nextBlock:     ; ( -- )
    add dword [ v_blk ], byte 0x02
    call lineOffsetZero
    ret

previousBlock:
    cmp dword [ v_blk ], byte ( START_BLOCK_NUMBER + 2 )
    js .forward
    sub dword [ v_blk ], byte 0x02
.forward:
    call lineOffsetZero
    ret

otherBlock:
    call swap_with_other_
    ret

tog_show_ASCII:
    not byte [ v_show_ASCII ]
    ret

shadow:     ; alternate between source and shadow blocks
    xor dword [ v_blk ], byte 0x01
    ret

insert0:    ; ( ... -- )
    mov ecx, [ v_lcad ]
    add ecx, [ v_words ]
    xor ecx, [ v_lcad ]
    and ecx, 0xFFFFFF00
    jz insert1
    mov ecx, [ v_words ]
.back:
    _DROP_
    loop .back
```

```
    ret

insert1:
    push esi
    mov esi, [ v_lcad ]
    mov ecx, esi
    dec esi
    mov edi, esi
    add edi, [ v_words ]
    shl edi, 0x02
    sub ecx, [ v_cad ]
    js .forward
    shl esi, 0x02
    std
    rep  movsd                  ; copy ecx 32 bit words from ds:esi to es:edi
    cld
.forward:
    pop esi
    shr edi, 0x02
    inc edi
    mov [ v_curs ], edi
    mov ecx, [ v_words ]
.back:
    dec edi
    mov [ ( edi * 4 ) + 0x00 ], _TOS_
    _DROP_
    loop .back
    ret

insert:
    call insert0
    mov cl, [ action ]
    xor [ edi * 4 + 0x00 ],cl
    cmp cl, 0x03                    ; if we are a red token
    jnz .forward
    mov byte [ action ], 0x04       ; switch to green
    mov dword [ keypad_colour ], colour_green
    mov word [ v_hintChar ], 'g'    ; mark the green keypad with a 'g'
    .forward:
    ret

_word1:
    pop dword [ aword ]
    mov dword [ aword ], ex1
    ret

_word:
    mov dword [ aword ], _word1
    jmp dword quit_

tokenAction_1:
    _DUP_
    mov _TOS_, 0x01
    cmp byte [ action ], 0x04
    jz .forward2
    mov al, 0x03
.forward2:
    cmp dword [ base ], byte 0x0A
    jz .forward
    xor al, 0x10
.forward:
    _SWAP_
    mov dword [ v_words ], 0x02
    jmp short insert

tokenAction:
    test byte [ action ], 0x0A
    jnz .forward
    mov edx, _TOS_
    and edx, 0xFC000000
```

```
    jz .forward2
    cmp edx, 0xFC000000
    jnz tokenAction_1
.forward2:
    shl _TOS_, 0x05
    xor al, 0x02
    cmp byte [ action ], 0x04
    jz .forwardBack
    xor al, 0x0B
.forwardBack:
    cmp dword [ base ], byte 0x0A
    jz .forward4
    xor al, 0x10
.forward4:
    mov dword [ v_words ], 0x01
    jmp insert
.forward:
    cmp byte [ action ],  0x09
    jnz .forward3
    mov edx, _TOS_
    shl edx, 0x05
    sar edx, 0x05
    cmp edx, _TOS_
    jz .forward5
.forward3:
    _DROP_
    ret
.forward5:
    shl _TOS_, 0x05
    xor al, 0x06
    jmp short .forwardBack

enstack: ; ( ... n -- )   ; ctrlY action, delete the token at the cursor and put it into the trash buffer
    _DUP_
    mov _TOS_, [ v_cad ]
    sub _TOS_, [ v_pcad ]
    jz .forward
    mov ecx, _TOS_
    xchg _TOS_, edx
    push esi
    mov esi, [ v_cad ]
    lea esi, [ (esi * 4) - 0x04 ]
    mov edi, [ v_trash ]    ; setup EDI to point to the current trash buffer address
.back:
    std
    lodsd ; _DROP_          ; loads EAX with the value pointed to by EDI = [ v_trash ]
    cld
    stosd                   ; stores EAX into the location pointed to by EDI = [ v_trash ] and increments
EDI
    loop .back
    xchg _TOS_, edx         ;
    stosd                   ; stores EAX into the location pointed to by EDI and increments EDI
    mov [ v_trash], edi     ; update the current trash buffer address
    pop esi
.forward:
    _DROP_
    ret

deleteAction:
    call enstack
    mov edi, [ v_pcad ]
    mov ecx, [ v_lcad ]
    sub ecx, edi
    shl edi, 0x02
    push esi
    mov esi, [ v_cad ]
    shl esi, 0x02
    rep  movsd                  ; copy ecx 32 bit words from ds:esi to es:edi
    pop esi
    jmp dword cursorLeft
```

```
act0:
    call enstack
    jmp dword cursorLeft

yellowAction:
    mov al, 0x01
    jmp short actt

redAction:  ; red : start creating a new definition
    mov al, 0x03
    jmp short actt

greenAction:   ; green, start compiling an existing definition
    mov al, 0x04
    jmp short actt

textAction:
    mov al, 0x09
    jmp short actt

CapitalAction:
    mov al, 0x0A
    jmp short actt

capitalS_Action:
    mov al, 0x0B
    jmp short actt

grayAction:
    mov al, 0x0D
    jmp short actt

blueAction:
    mov al, 0x0E
    jmp short actt

cyanAction:
    mov al, 0x07

actt:   ; ( action -- )
    mov [ action ], al
    mov dword [ aword ], insert
    mov _TOS_, [ ( _TOS_ * 4 ) + actionColourTable ]
actn:
    mov [ keypad_colour ], _TOS_
    pop _TOS_
    _DROP_
    jmp dword quit_

magentaAction:   ; magenta variable action
    mov byte [ action ], 0x0C
    mov _TOS_, colour_magenta
    mov dword [ aword ], .forward
    jmp short actn
    .forward:
    _DUP_
    xor _TOS_, _TOS_
    inc dword [ v_words ]
    jmp dword insert

editorExit:       ; ( -- )   \ leave the editor
    pop _TOS_
    _DROP_
    mov dword [ aword ], ex1
    mov dword [ anumber ], nul
    mov byte [ alpha0 + ( 4 * 4 ) ], 0x00
    mov dword [ alpha0 + 4 ], nul0
    mov dword [ keypad_colour ], colour_yellow
    mov byte [ v_quitMode ], 0x00
```

```
    mov byte [ v_hintChar ], 0x00   ; no hint chararacter
    jmp dword quit_

destack:                    ; ctrlZ action, insert the next token from the trash buffer
    mov edx, [ v_trash ]
    cmp edx, TRASH_BUFFER   ; do not insert if we have emptied the trash buffer
    jnz .forward
    ret
.forward:
    sub edx, byte 0x08
    mov ecx, [edx+0x04]
    mov [ v_words ], ecx
.back:
    _DUP_
    mov _TOS_, [edx]
    sub edx, byte 0x04
    loop .back
    add edx, byte 0x04
    mov [ v_trash ], edx
    jmp dword insert0

; ******************************************************************************
; Locate
; ******************************************************************************

locate:
    mov ecx, [ v_blk ]
    xchg ecx, [ v_locatedBlock ]
    mov [ v_blk ], ecx

    mov ecx, [ v_curs ]
    xchg ecx, [ v_locatedCurs ]
    mov [ v_curs ], ecx

    ret

; ******************************************************************************
; Editor keypad and action table
; ******************************************************************************

editorActionTable:
    dd nul           , deleteAction    , editorExit    , destack      ;
    dd yellowAction  , redAction       , greenAction   , shadow       ; y r g *
    dd cursorLeft    , cursorUp        , cursorDown    , cursorRight  ; l u d r
    dd previousBlock , magentaAction   , cyanAction    , nextBlock    ; - m c +
    dd nul           , capitalS_Action , CapitalAction , textAction   ; _ S C t
    dd nul           , locate          , nul           , otherBlock   ; _ L _ j
ekbd0:
    dd grayAction    , blueAction      , nul           , act0         ; a b _ _
    db 'x'           , '.'             , 'i'           , 0x00         ; four characters to display on the
bottom line of the keyboard

editorKeyTableHintChars:    ; display the current edit colour and mode in the bottom right hand corner of
the keyboard
    db '    '   ;
    db 'yrg '   ; y r g _
    db '    '   ; l u d r
    db ' mc+'   ; - m c +
    db ' SCt'   ; _ S C t
    db ' l  '   ; _ _ _ j
    db 'ab  '   ; a b _ _

; Editor keypad display
; _ S C t  y r g *
; < l > j  l u d r
; a b _ k  - m c +
;     x . i

editorKeypad:                   ; the main editor keyboard icons
    db 'yrg*'                   ; yellow, red, green, shadow
```

```
    db 0x10, 0x11, 0x12, 0x13   ; 'ludr' arrow glyphs (left, up, down, right)
    db '-mc+'                    ; previous block, magenta, cyan, next block
    db ' SCt'                    ; ALL CAPITALS, Capital start, lower case text
    db 0x14,'l', 0x17, 'j'      ; left glyph, "locate", right glyph, j = other block
    db 'ab  '                    ; grAy, blue

set_e_main:
    mov dword [ shiftAction ], ekbd0
    mov dword [ currentKeypadIcons ], ( editorKeypad - 4 )
    mov dword [ keypad_colour ], colour_yellow
    ret

edit0:
    _DROP_
    jmp short edit2

save_edit_state:   ; ( n -- )   \ save edit block n
    push ecx
    mov ecx, [ v_blk ]
    mov [ v_otherBlock ], ecx       ; save the current edit block to the "other" block variable
    mov ecx, [ v_curs ]
    mov [ v_otherCursor ], ecx      ; save the current edit block cursor to the "other" cursor variable
    pop ecx
    mov [ v_blk ], _TOS_            ; set the new edit block
    _DROP_                          ; discard n
    ret

swap_with_other_:
    push ecx
    mov ecx, [ v_blk ]
    xchg ecx, [ v_otherBlock ]
    mov [ v_blk ], ecx
    mov ecx, [ v_curs ]
    xchg [ v_otherCursor ], ecx      ; save the current edit block cursor to the "other" cursor variable
    mov [ v_curs ], ecx
    pop ecx
    ret

edit_:   ; ( n -- )   \ edit block n
    call save_edit_state
e_:
    mov byte [ v_quitMode ], 0xFF
    call refresh
plus_e:
    mov dword [ anumber ], tokenAction
    mov byte [ alpha0+4*4 ], 0x25
    mov dword [ alpha0 + 4 ], edit0
edit2:
    call set_e_main
    .back:
    call clearHintChar
    call get_key_
    push _TOS_
    mov al, [ editorKeyTableHintChars + _TOS_ ]
    mov [ v_hintChar ], _TOS_
    pop _TOS_
    call [ ( _TOS_ * 4 ) + editorActionTable ]
    _DROP_
    jmp short .back

convertAddress:     ; ( a32 -- )    set up the block at the given 32 bit cell address, including the
cursor position
    mov _SCRATCH_, _TOS_
    and _SCRATCH_, 0x00FF
    mov [ v_curs ], _SCRATCH_           ; cell offset in block
    call cellAddressToBlock
    mov [ v_blk ], _TOS_
    _DROP_
    ret
```

```
editAddress_:    ; ( a32 -- )    edit the block at the given 32 bit byte address, including the cursor
position
    mov _SCRATCH_, _TOS_
    shr _SCRATCH_, 2
    and _SCRATCH_, 0x00FF
    mov [ v_curs ], _SCRATCH_          ; cell offset in block
    sub _TOS_, RELOCATED               ; subtract the addess of block 0
    shr _TOS_, 10                      ;
    call edit_
    ret

keypd_:    ; display the keypad vectors and display characters at the address on top of the return stack
    pop edx                            ; keypd_ is followed by call table then keymap
    mov [ vector ], edx                ; edx points to the next colorForth word to be executed
    add edx, ( 28 * 5 )                ; 28 keys, 5 bytes per compiled call
    mov [ currentKeypadIcons ], edx
    sub edx, byte +16
    mov [ shiftAction ], edx
.back:
    call get_key_                      ; calls pause_ while waiting for a character
    mov edx, [ vector ]
    add edx, _TOS_
    lea edx, [ ( _TOS_ * 4 ) + edx + 0x05 ]
    add edx, [ edx - 0x04 ]
    _DROP_
keypd1:
    call edx
    jmp short keypd_.back

; ****************************************************************************
; QWERTY support
; ****************************************************************************

qwertyKeyboard:
    dd 0
    dd 0
    dd 0
    dd 0x01040f17       ; 'qwer'
    dd 0
    dd 0

qwertToggleBase:
    xor dword [ setCurrentBase ], ((setBase_decimal - $$) ^ (setBase_hex - $$))
    xor byte [ ( numb0 + 12 ) ], 0x2F
qwertToggleBase1:
;    call [ setCurrentBase ]
;    mov dword [ qwertyKeyboard ], 0x00            ; '' => decimal
;    cmp dword [ base ], byte +0x10
;    jnz .forward
;    mov dword [ qwertyKeyboard ], 0x00150414      ; 'hex'
; .forward:
;    mov dword [ currentKeypadIcons ], keypd1
;    mov dword [ shiftAction ], qwertyKeyboard
    ret

qwertyAction4:
    call qwertToggleBase
    jmp qwertyAction3

qwertyActionTable:
    dd endn, endn, exitn_, qwertyAction3, qwertyAction4

qwertFunction1:
    call right
    db 0xC7
    add _TOS_, ( qwertyKeyboard + 4 )
    push es
    push ss
    or [_TOS_], _TOS_
    call qwertToggleBase1
```

```
    mov byte [ v_sign ], 0x00
    mov _TOS_, [ v_digin ]
qwertyAction5:
    call get_qwerty_key_
    jz .forward4
    jmp dword [ _TOS_ * 4 + qwertyActionTable - 0x200 ]
.forward4:
    test _TOS_, _TOS_
    jng qwertyAction3
    cmp al, 0x23
    jz .forward3
    mov _TOS_, [ v_digin ]
    cmp _TOS_, [ base ]
    jns .forward2
    test byte [ v_sign ], 0xFF
    jz .forward
    neg _TOS_
.forward:
    mov edx, [ esi ]
    imul edx, [ base]
    add edx, _TOS_
    mov [ esi ], edx
.forward2:
    jmp short qwertyAction3
.forward3:
    xor [ v_sign ], _TOS_
    neg dword [ esi ]

qwertyAction3:
    _DROP_
    jmp short qwertyAction5

qwertToggleBaseTable2:
    dd lj, lj, exit_

qwertyFunction2:
    mov dword [ ( qwertyKeyboard + 4 ) ], 0x02150402  ; 'text'
    call right
    mov dword [ v_words ], 0x01
    mov dword [ chars], 0x01
    _DUP_
    mov dword [ esi ], 0x00
    mov byte [ bits_ ], 0x1C
.back:
    jz .forward
    cmp _TOS_, 0x83
    jns .forward
    jmp dword [ _TOS_*4 + qwertToggleBaseTable2 - 0x200 ]
.forward:
    test _TOS_, _TOS_
    jng .forward2
    cmp _TOS_, 0x30
    jns .forward2
    _DUP_
    call echo_
    call pack_
    inc dword [ chars]
.forward2:
    _DROP_
    call get_qwerty_key_
    jmp short .back

qwertyAction2:
    call qwertToggleBase
    jmp dword nul0

qwertyAction1:
    jmp dword [ alpha0 + 4 ]

qwertyTable1:
```

```
    dd nul0
    dd nul0
    dd nul0
    dd qwertyAction1
    dd qwertyAction2

qwertyDoAction:
    mov dword [ ( qwertyKeyboard + 4 ) ], 0x00   ; clear the 'text' string
    mov dword [ shiftAction ], qwertyKeyboard
    mov dword [ currentKeypadIcons ], keypd1

.back2:
    call get_qwerty_key_
    jz .forward
    jmp dword [ ( _TOS_ * 4 ) + qwertyTable1 - 0x0200 ]

.forward:
    cmp al, 0x30
    jnz .back
    mov dword [ ( qwertyKeyboard + 4 ) ], 0x02150402  ; 'text'
    _DROP_
    jmp short .back2
.back:
    test _TOS_, _TOS_
    jng .forward3
    test dword [ ( qwertyKeyboard + 4 ) ], 0xFFFFFFFF
    jnz .forward2
    cmp byte [ v_digin ], 0x0A
    js qwertFunction1
.forward2:
    cmp _TOS_, 0x30
    jns .forward3
    call qwertyFunction2
    call [ aword ]
    _DUP_
.forward3:
    _DROP_
    jmp dword quit_

qwert:      ; selects QWERTY keyboard entry
    mov dword [ x_qwerty ], qwertyDoAction
    ret

; ****************************************************************************

abort_action:
    cmp edi, ( RELOCATED / 4 )  ; if we are compiling a block, show the location of the error
    ; edi is a cell address, so divide by 4
    jc .forward
    _DUP_
    mov _TOS_, [ v_blk ]
    mov [ v_otherBlock ], _TOS_ ; save the last block to be edited
    mov _TOS_, edi
    call convertAddress
.forward:
    mov esp, RETURN_STACK_0
    cmp esi, ( DATA_STACK_0 + 4 )
    jc .forward2
    mov esi, ( DATA_STACK_0 + 4 )
.forward2:
    mov dword [ tokenActions + ( 3 * 4 ) ], forthd
    mov dword [ tokenActions + ( 4 * 4 ) ], qcompile
    mov dword [ tokenActions + ( 5 * 4 ) ], cnum
    mov dword [ tokenActions + ( 6 * 4 ) ], cshort
    mov _TOS_, 0x3F     ; '?' character to follow the display of the unknown word
    call echo_
;    jmp abort_e2
    jmp dword quit_

; ****************************************************************************
```

```
rquery: ; r?
    _DUP_
    mov _TOS_, RETURN_STACK_0
    sub _TOS_, esp
    shr _TOS_,1
    shr _TOS_,1
    ret

boot:
    ; see http://wiki.osdev.org/PS2_Keyboard#CPU_Reset
    mov al, 0xFE
    out 0x64, al
    jmp short $         ; we should never get here, because the processor will be rebooted... stop here
just in case

wipe:  ; ( -- )    \ wipe the currently edited block
    _DUP_
    mov _TOS_, [ v_blk ]
    mov ecx, 0x40
wipe2:
    push edi
    call blockToCellAddress     ; add the RELOCATED block number offset and convert to cell address
    shl _TOS_, 2                ; convert to byte address
    mov edi, _TOS_
    xor _TOS_, _TOS_
    rep stosd  ; stores  eax  into the location pointed to by  edi  then increments  edi  by 4, does this
ecx  times
    pop edi
    _DROP_
    ret

wipes:  ; ( startblock# #blocks -- )    \ wipes #blocks starting from block startblock#   ( was erase )
    mov ecx, _TOS_
    shl ecx, 0x06               ;  convert blocks to cells, multiply by 64
    _DROP_
    jmp wipe2

copy_:   ; ( blk -- )    \ copy the given block (and shadow) to the currently displayed block (and shadow)
    cmp _TOS_, byte 0x0C        ; below block 12 is machine code
    jc abort_
    push edi
    push esi
    push ecx
    call blockToCellAddress     ; source block
    shl _TOS_, 0x02             ; convert cell address to byte address
    mov esi, _TOS_
    mov _TOS_, [ v_blk ]
    call blockToCellAddress     ; destination block
    shl _TOS_, 0x02             ; convert cell address to byte address
    mov edi, _TOS_
    mov ecx, 0x0200
    rep  movsd                  ; copy ecx 32 bit words from ds:esi to es:edi
    pop ecx
    pop esi
    pop edi
    _DROP_
    ret

debug:
    mov dword [ v_gr_xy ], 0x302B5
    _DUP_
    mov _TOS_, [ main ]
    push dword [_TOS_]
    call dotHex
    _DUP_
    pop _TOS_
    call dotHex
    _DUP_
    mov _TOS_, [ draw ]
```

```
    call dotHex
    _DUP_
    mov _TOS_, esi
    jmp dword dotHex

; ******************************************************************************

tic0:
    dec dword [ v_words ]
    jz .forward
    _DROP_
    jmp short tic0
.forward:
    ret

tic_:   ; ( -- a )   \ return the byte address of the next word entered
    call _word          ; allow user to enter the word to  search for
    call tic0           ; remove the entered word from the stack
    call find_          ; find the word in the dictionary, return its index in  ecx
    jnz abort_
    mov _TOS_, [ ( ecx * 4 ) + ForthJumpTable ]     ; return the word's address from the jump table
    ret

itick:
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
    call find_
    mov _TOS_, [ ( ecx * 4 ) + ForthJumpTable ]
    ret

; ******************************************************************************

plusList_words:   ; ( -- ) display the current colorForth block
    _DUP_
    xor _TOS_, _TOS_
    mov [ currentState ], _TOS_
    mov [ lastState ], _TOS_
    mov [ v_curs ], _TOS_           ; set the cursor to top left (0, 0) ToDo: Note : does not work!
    _DROP_

    call setupText_                 ; setup the clip window for this display
    _DUP_
    mov _TOS_, [ v_lcad ]
    mov [ v_cad ], _TOS_
    mov _TOS_, [ v_blk ]            ; get the current block number to be edited
    call blockToCellAddress         ; add the RELOCATED block number offset and convert to cell address
    mov edi, _TOS_
    xor _TOS_, _TOS_
    add edi, [ v_lineOffset ]
    mov [ v_pcad ], edi
.back:
    mov edx, dword [ ( edi * 4 ) + 0x00 ]   ; edi is the display pointer and is a cell address
    call show_cursor                        ; show the PacMan-like cursor
    call space_
    inc edi
    and edx, byte 0x0F
    _DUP_
    mov _TOS_, 0x09
    ; call doColourBlind
    call [ ( edx * 4 ) + displayShannonFanoActions ]
    jmp short .back

refresh_words:                      ; refresh the editor display to show all Forth words
    call show                       ; set the screen task to execute the code following :
    call page_                      ; clear the screen
    ; call displayBlockNumber       ; display the current block number on the screen
    call plusList_words             ; list the contents of the block
    _DUP_
    ; mov _TOS_, 0x0F
    ; call doColourBlind             ; display the final colour-blind punctuation, set up for next call of
plusList
```

```
    jmp dword displayTheKeypad_

words_:     ; \ show the ForthNames array as if it is a sequence of blocks
    _DUP_
    mov _TOS_, ForthNames
    call a2blk_
    call save_edit_state
    call refresh_words
    ret

; ****************************************************************************

; Int 0x13 AH Return Code error type
; 0x00 Success
; 0x01 Invalid Command
; 0x02 Cannot Find Address Mark
; 0x03 Attempted Write On Write Protected Disk
; 0x04 Sector Not Found
; 0x05 Reset Failed
; 0x06 Disk change line 'active'
; 0x07 Drive parameter activity failed
; 0x08 DMA overrun
; 0x09 Attempt to DMA over 64kb boundary
; 0x0A Bad sector detected
; 0x0B Bad cylinder (track) detected
; 0x0C Media type not found
; 0x0D Invalid number of sectors
; 0x0E Control data address mark detected
; 0x0F DMA out of range
; 0x10 CRC/ECC data error
; 0x11 ECC corrected data error
; 0x20 Controller failure
; 0x40 Seek failure
; 0x80 Drive timed out, assumed not ready
; 0xAA Drive not ready
; 0xBB Undefined error
; 0xCC Write fault
; 0xE0 Status error
; 0xFF Sense operation failed

; ****************************************************************************
; 16 bit BIOS disk read/write from 32 bit
; ****************************************************************************

; set the required parameters into the DAP buffer for the LBA BIOS extended read/write calls.
; Also set up the extra DAP buffer values for use by the CHS BIOS calls, if the LBA call fails.
; This is to avoid returning from 16 bit mode to calculate the values.
setupDAP_:   ; ( sector n cmd -- )   \ setup the DAP for the given LBA sector number

    push edi

    xor ecx, ecx
    mov edi, (data_area - $$ + BOOTOFFSET)   ; setup the data index pointer
    mov cx, [ word di + ( driveinfo_Drive_DX - data_area ) ]  ; restore the boot drive into dl
    mov edi, DAP_BUFFER
    mov word [ edi + o_Int13_DAP_saved_DX ], cx     ; setup DX value returned by the BIOS

    mov word [ edi + o_Int13_DAP_readwrite ], ax    ; set the read/write cmd value, 0x0000 or 0x0001
    _DROP_

    ; limit the number of sectors to the size of the SECTOR_BUFFER
    cmp _TOS_, ( SECTOR_BUFFER_SIZE / 0x0200 )
    js .forward
        mov _TOS_, ( SECTOR_BUFFER_SIZE / 0x0200 )
    .forward:
    mov word [ edi + o_Int13_DAP_num_sectors ], ax
    _DROP_

    mov dword [ edi + o_Int13_DAP_LBA_64_lo ], eax
    push eax    ; save for later
```

```
    xor eax, eax
    mov dword [ edi + o_Int13_DAP_LBA_64_hi ], eax

    ; buffer within low 16 bits of address space
    mov word [ edi + o_Int13_DAP_segment ], ax
    mov ax, ( SECTOR_BUFFER )
    mov word [ edi + o_Int13_DAP_address ], ax

    ; set the configuration buffer values from the registers
    mov eax, 0x0010
    mov word [ edi + o_Int13_DAP_size ], ax  ; setup DAP buffer size

; setup values for CHS BIOS disk calls

    pop eax                         ; restore the start sector number
    add eax, [ bootsector - $$ + BOOTOFFSET]    ; add the bootsector from the drive parameter table

    push eax                        ; save it while we calculate heads*sectors-per-track
    mov al, [ driveinfo_Head - $$ + BOOTOFFSET]      ; index of highest-numbered head
    inc al                          ; 1-base the number to make count of heads
    mul byte [ driveinfo_SectorsPertrack - $$ + BOOTOFFSET]     ; sectors per track
    mov ebx, eax
    pop eax
    xor edx, edx                    ; clear high 32 bits
    div ebx                         ; leaves cylinder number in eax, remainder in edx
    mov ecx, eax                    ; store cylinder number in another register
    mov eax, edx                    ; get remainder into AX
    mov bl, [ driveinfo_SectorsPertrack - $$ + BOOTOFFSET]      ; number of sectors per track
    div bl                          ; head number into AX, remainder into DX
    mov bl, al                      ; result must be one byte, so store it in BL
    rol ecx, 8                      ; high 2 bits of cylinder number into high 2 bits of CL
    shl cl, 6                       ; makes room for sector number
    or cl, ah                       ; merge cylinder number with sector number
    inc cl                          ; one-base sector number
    mov word [ edi + o_Int13_DAP_saved_CHS_CX ], cx  ; also save the calculated CX value
    mov cx, [ driveinfo_Drive_DX - $$ + BOOTOFFSET]    ; drive number in low 8 bits
    mov ch, bl                      ; place head number in high bits
    mov word [ edi + o_Int13_DAP_saved_CHS_DX ], cx  ; also save the calculated DX value

    pop edi
    _DROP_

    ret

; ****************************************************************************
; BIOS read/write 512 byte LBA sectors
; ****************************************************************************

BIOS_ReadWrite_Sector_LBA: ; ( -- )   \ try to read or write using the extended disk BIOS calls,
; \ if that fails, try the CHS BIOS call. Parameters are in the DAP buffer.

   pushf   ; save the processor flags, especially interrupt enable

%ifdef NOT_BOCHS
    call restore_BIOS_idt_and_pic   ;
%endif

    _DUP_
    xor _TOS_, _TOS_
    call lidt_                      ; Load the BIOS Interrupt Descriptor Table

    call setRealModeAPI
[BITS 16]                           ; Real Mode code (16 bit)
    mov si, DAP_BUFFER
    mov byte ah, [ si + o_Int13_DAP_readwrite ]  ; 0x00 for read, 0x01 for write
    or  ah, 0x42                    ; BIOS extended read/write
    mov al, 0x00
    mov dx, [ si + o_Int13_DAP_saved_DX ]
    int 0x13
```

```
    cli                             ; BIOS might have left interrupts enabled

    mov word [ si + o_Int13_DAP_returned_AX ], ax  ; save the value in AX that the BIOS call returned
    jnc .forward
        mov si, DAP_BUFFER

        mov byte ah, [ si + o_Int13_DAP_readwrite ]  ; 0x00 for read, 0x01 for write
        or  ah, 0x02                ; CHS BIOS mode, read  al  sectors, set above
        mov al, byte [ si + o_Int13_DAP_num_sectors ]   ; restore the number of sectors saved by setupDAP_

        mov word cx, [ si + o_Int13_DAP_saved_CHS_CX ]  ; restore the CX value calculated by sector_chs
        mov word dx, [ si + o_Int13_DAP_saved_CHS_DX ]  ; restore the DX value calculated by sector_chs
        mov word bx, [ si + o_Int13_DAP_address ]       ; restore the address saved by setupDAP_
        int 0x13
        cli                         ; BIOS might have left interrupts enabled

        mov si, DAP_BUFFER
        mov word [ si + o_Int13_DAP_returned_AX ], ax  ; the BIOS call returned AX
        mov ax, 0x0001
        jc .forward2
           mov ax, 0x0000
        .forward:
        mov [ si + o_Int13_DAP_returned_carry_flag ], ax  ; the BIOS call returned carry flag
    .forward2:
    mov [ si + o_Int13_DAP_returned_carry_flag ], ax  ; the BIOS call returned carry flag

    call setProtectedModeAPI        ; called from 16 bit code, returns to 32 bit code
[BITS 32]                           ; Protected Mode code (32 bit)

%ifdef NOT_BOCHS
    call restore_new_idt_and_pic
%endif

    _DUP_
    mov _TOS_, INTERRUPT_VECTORS
    call lidt_                      ; Load the new Interrupt Descriptor Table

    popf   ; restore the processor flags, especially interrupt enable

    ret

Read_Sector_LBA:    ; ( sector n -- )   "rlba"   GetFlag returns 0 for success
    _DUP_
    mov eax, 0x0000                 ; read command
    call setupDAP_                  ; setup up the DAP table using 3 items from the stack ( start n cmd --
)
    cli                             ; disable interrupts
    pushad                          ; Pushes all general purpose registers onto the stack
    call BIOS_ReadWrite_Sector_LBA
    popad                           ; restore the registers pushed by  pushad
    ret

Write_Sector_LBA:   ; ( sector n -- )   "wlba"
    _DUP_
    mov eax, 0x0001                 ; write command
    call setupDAP_                  ; setup up the DAP table using 3 items from the stack ( start n cmd --
)
    cli                             ; disable interrupts
    pushad                          ; Pushes all general purpose registers onto the stack
    call BIOS_ReadWrite_Sector_LBA
    popad                           ; restore the registers pushed by  pushad
    ret

ReadSectors:    ; ( a sector n -- a' )  \ read  n  sectors from  sector  into address  a
    call Read_Sector_LBA            ; reads  n  sectors starting from  sector  into the SECTOR_BUFFER

    push esi                        ; esi is changed by rep movsw

    mov esi, DAP_BUFFER
    xor ecx, ecx
```

```
    mov word cx, [ si + o_Int13_DAP_num_sectors ]   ; restore the number of sectors saved by setupDAP_
    mov ebx, ecx                    ; save number of sectors for later
    mov esi, SECTOR_BUFFER          ; source address
    mov edi, eax                    ; destination address
    shl ecx, 0x07                   ; 512 bytes in cells = 2 ** 7
    rep movsd                       ; does not change AX , it moves DS:SI to ES:DI and increments SI and
DI

    ; ( a -- a' )
    mov ecx, ebx
    shl ecx, 0x09                   ; 512 bytes in bytes = 2 ** 9
    add eax, ecx                    ; increment the address that is TOS

    pop esi
    ; ( a -- a' sector' )
    _DUP_
    push esi
    mov esi, DAP_BUFFER             ; esi is changed by rep movsd above
    xor ecx, ecx
    mov word cx, [ si + o_Int13_DAP_LBA_64_lo ]   ; restore the start sector
    pop esi
    mov eax, ecx
    add eax, ebx

;    call GetFlag
    ret

WriteSectors:    ; ( a sector n -- a' ) \ write  n  sectors starting at  sector  from address  a

    push ecx
    push edx

    mov edx, [ esi + 4 ]            ; save  a  from stack in  edx

    push esi                        ; esi is also changed by rep movsw
    mov esi, DAP_BUFFER
    xor ecx, ecx
    mov word cx, [ si + o_Int13_DAP_num_sectors ]   ; restore the number of sectors saved by setupDAP_
    mov ebx, ecx                    ; save number of sectors for later

    shl ecx, 0x07                   ; 512 bytes in cells = 2 ** 7

    mov esi, edx                    ; source address
    mov edi, SECTOR_BUFFER          ; destination address
    rep movsd                       ; does not change AX , it moves DS:SI to ES:DI and increments SI and
DI

    pop esi

    push ebx
    call Write_Sector_LBA    ; writes  n  sectors starting from  sector  from the SECTOR_BUFFER
    pop ebx
;    push esi                        ; esi is also changed by rep movsw

    ; ( a -- a' )
    mov ecx, ebx
    shl ecx, 0x09                   ; 512 bytes in bytes = 2 ** 9
    add eax, ecx                    ; increment the address that is TOS

;    pop esi
    ; ( a -- a' sector' )
    _DUP_
    push esi
    mov esi, DAP_BUFFER             ; esi is changed by rep movsd above
    xor ecx, ecx
    mov word cx, [ si + o_Int13_DAP_LBA_64_lo ]   ; restore the start sector
    pop esi
    mov eax, ecx
    add eax, ebx
```

```
    pop edx
    pop ecx

    ret

SaveAll_:    ; ( -- ) "sss"
    pushf   ; save the processor flags, especially interrupt enable
    cli

    _DUP_
    xor eax, eax
    call block_
    _DUP_
    xor eax, eax
    mov ecx, 0x21   ; 32 x 16 Kbytes= 512 + 32 Kbytes
    .back:
    _DUP_
    mov eax, 0x20   ; 32 x 512 byte sectors = 16 Kbytes
    call WriteSectors   ; ( a sector n -- a' ) \ write  n  sectors starting at  sector  from address  a
    loop .back
    _DROP_
    _DROP_

;    ; repeat the first group of sectors, to flush the save
;    _DUP_
;    xor eax, eax
;    call block_
;    _DUP_
;    xor eax, eax
;    mov ecx, 0x01   ; 1 x 16 Kbytes= 162 Kbytes
;    .back2:
;    _DUP_
;    mov eax, 0x20   ; 32 x 512 byte sectors = 16 Kbytes
;    call WriteSectors   ; ( a sector n -- a' ) \ write  n  sectors starting at  sector  from address  a
;    loop .back2
;    _DROP_
;    _DROP_

;    ; repeat the last sector, to flush the save
;    _DUP_
;    ; address
;    mov eax, LAST_BLOCK_NUMBER
;    call block_
;    _DUP_
;    ; sector number
;    mov eax, ( LAST_BLOCK_NUMBER * 2 )  ; 01 x 512 byte sectors = 512 bytes, just write one sector
;    _DUP_
;    ; number of 512 byte sectors to write
;    mov eax, 0x01   ; 01 x 512 byte sectors = 512 bytes, just write one sector
;    call WriteSectors   ; ( a sector n -- a' ) \ write  n  sectors starting at  sector  from address  a
;    _DROP_
;    _DROP_

    popf   ; restore the processor flags, especially interrupt enable
ret

GetFlag: ; ( -- error | 0 )   0 for success, else the error type ( eax == 0x100 is Invalid Command )
    _DUP_
    xor eax, eax
    push edi
    mov edi, DAP_BUFFER
    mov ax, [ edi + o_Int13_DAP_returned_carry_flag ]  ; the BIOS call returned carry flag
    add ax, 0
    jz .forward
        mov ax, [ edi + o_Int13_DAP_returned_AX ]  ; the BIOS call returned error value in ax
    .forward:
    pop edi
    ret

%if 0
```

```
BIOS_Read_Sector_CHS:
    call setRealModeAPI
[BITS 16]                           ; Real Mode code (16 bit)
    mov si, DAP_BUFFER
    mov al, byte [ si + o_Int13_DAP_num_sectors ]   ; setup the number of sectors saved by setupDAP_
;     and al, 0x0F        ; limit to 16 sectors
    mov ah, 0x02        ; CHT BIOS mode, read  al  sectors, set above
    mov word cx, [ si + o_Int13_DAP_saved_CHS_CX ]  ; setup the CX value calculated by sector_chs
    mov word dx, [ si + o_Int13_DAP_saved_CHS_DX ]  ; setup the DX value calculated by sector_chs
    mov word bx, [ si + o_Int13_DAP_address ]       ; setup the address saved by setupDAP_
    int 0x13
    cli                             ; BIOS might have left interrupts enabled

    mov si, DAP_BUFFER
    mov word [ si + o_Int13_DAP_returned_AX ], ax  ; the BIOS call returned AX
    mov ax, 0x0001
    jc .forward
       mov ax, 0x0000
    .forward:
    mov [ si + o_Int13_DAP_returned_carry_flag ], ax  ; the BIOS call returned carry flag

    call setProtectedModeAPI        ; called from 16 bit code, returns to 32 bit code
[BITS 32]                           ; Protected Mode code (32 bit)
    ret

; rchs:
Read_Sector_CHS:    ; ( sector n -- f )   "rchs"  returns 0 for success
    call setupDAP_              ; ( start n -- ) store the sector number into the Disk Address Packet
    cli                         ; disable interrupts
    pushad                      ; Pushes all general purpose registers onto the stack
    call BIOS_Read_Sector_CHS
    popad                       ; restore the registers pushed by  pushad
;     _DROP_
    jmp GetFlag

; wcht:
Write_Sector_CHS:   ; ( sector -- )   "wcht"
    call setupDAP_              ; store the sector number into the Disk Address Packet
    cli                         ; disable interrupts
    pushad                      ; Pushes all general purpose registers onto the stack
    call BIOS_Read_Sector_CHS
    popad                       ; restore the registers pushed by  pushad
    ret

%endif

; ******************************************************************************
; ******************************************************************************

%if 0
[BITS 16]                           ; Real Mode code (16 bit)
storeBefore:    ; ( -- )   \ store registers to the V_REGS array
    mov word [ V_REGS + 0x00 ], ax
    mov word [ V_REGS + 0x04 ], bx
    mov word [ V_REGS + 0x08 ], cx
    mov word [ V_REGS + 0x0C ], dx
    mov word [ V_REGS + 0x10 ], si
    mov word [ V_REGS + 0x14 ], di
    mov word [ V_REGS + 0x18 ], bp
    push ax                         ; save eax
    pushfd                          ; push the 32 bit eflags register onto the stack
    pop ax                          ; and pop it off into eax
    mov word [ V_REGS + 0x1C ], ax  ; eflags
    pop ax
    mov word [ V_REGS + 0x1E ], ax  ; eflags top 16 bits
    pop ax                          ; restore eax
    ret

storeAfter:     ; ( -- )   \ store registers to the V_REGS array
    mov word [ V_REGS + 0x20 ], ax
```

```
    mov word [ V_REGS + 0x24 ], bx
    mov word [ V_REGS + 0x28 ], cx
    mov word [ V_REGS + 0x2C ], dx
    mov word [ V_REGS + 0x30 ], si
    mov word [ V_REGS + 0x34 ], di
    mov word [ V_REGS + 0x38 ], bp
    push ax                         ; save eax
    pushfd                          ; push the 32 bit eflags register onto the stack
    pop ax                          ; and pop it off into eax
    mov word [ V_REGS + 0x3C ], ax  ; eflags
    pop ax
    mov word [ V_REGS + 0x3E ], ax  ; eflags top 16 bits
    pop ax                          ; restore eax
    ret
[BITS 32]                           ; Protected Mode code (32 bit)

BIOS_thunk:     ; ( -- )   \ call the BIOS - registers will have previously been setup
    call setRealModeAPI
[BITS 16]                           ; Real Mode code (16 bit)
    push ax
    push es                         ; this operation messes with ES
    push di                         ; and DI
    call storeBefore
    int 0x13
    jc $                            ; stop here on error
    call storeAfter
    pop di
    pop es
    pop ax
    cli                             ; BIOS might have left interrupts enabled
    call setProtectedModeAPI        ; called from 16 bit code, returns to 32 bit code
[BITS 32]                           ; Protected Mode code (32 bit)
    ret

%endif

%if 0
th_:         ; ( ax bx cx dx si di es -- w )   \ th ( thunk to BIOS Int 0x13 )
             ; eax = 0x DH DL AH AL , returns in same order
    cli                             ; disable interrupts
    pushad   ; Pushes all general purpose registers onto the stack in the following order:
             ; EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The value of ESP is the value before the actual push
of ESP
             ;  7    6    5    4    3    2    1    0   offset in cells from ESP

;    call setupDAP_

    push edi
    mov di, (data_area - $$ + BOOTOFFSET)   ; setup the data index pointer
    mov dx, [ byte di + ( driveinfo_Drive_DX - data_area) ]  ; restore the boot drive from dx (and head? )
;    mov dl, 0x80
    mov ebx, SECTOR_BUFFER
    mov eax, ( 0x0200 + ( ( SECTOR_BUFFER_SIZE / 512 ) & 0xFF ) ) ; read n sectors to fill the buffer
    mov ecx, 0x0201                  ; cylinder | sector

    call BIOS_thunk

    pop edi
    popad   ; restore the stack values pushed by  pushad
    ret
%endif
%if 0
XXXrsect_:     ; ( sector -- ax )   \
    pushad  ; Pushes all general purpose registers onto the stack
    push edi

;    call sector_chs                  ; store th sector number into the Disk Address Packet

    mov di, (data_area - $$ + BOOTOFFSET)   ; setup the data index pointer
    mov dx, [ byte di + ( driveinfo_Drive_DX - data_area) ]  ; restore the boot drive from dx (and head? )
```

```
;    mov dl, 0x80

    cli                             ; disable interrupts
;    mov esi, DAP_BUFFER
;    _DUP_
    mov eax, 0x0201     ; BIOS read, one sector
    mov bx, SECTOR_BUFFER
    call BIOS_thunk

    pop edi
    popad   ; restore the stack values pushed by  pushad
    ret
%endif

; ****************************************************************************
; ****************************************************************************

%define FORTH_INITIAL_WORD_COUNT  ( ( ForthJumpTableROM_end - ForthJumpTableROM ) / 4 )     ; in cells
%define MACRO_INITIAL_WORD_COUNT  ( ( MacroJumpTableROM_end - MacroJumpTableROM ) / 4 )     ; in cells
%define BLUE_INITIAL_WORD_COUNT   ( ( BlueJumpTableROM_end  - BlueJumpTableROM )  / 4 )     ; in cells

warm:   ; warm start
    mov _SCRATCH_, STACK_MEMORY_START ; start of stack memory area
    mov ecx, ( TOTAL_STACK_SIZE >> 2 )  ; number of 32 bit cells to fill with the pattern
.back:
    mov dword [ _SCRATCH_ ], 0x55555555             ; fill with this pattern
    add _SCRATCH_, 0x04
    loop .back

    xor ecx, ecx                    ; assumed by initshow to have been previously zeroed
;    call initshow                  ; sets up do-nothing "show" task
    call refresh                    ; starts the editor display task
    call initserv1_                 ; sets up do-nothing "serv1" task
    call initserv2_                 ; sets up do-nothing "serv2" task ToDo: fix the serv2 task...

    mov dword [ v_ForthWordCount ], FORTH_INITIAL_WORD_COUNT ; initial #words
    mov dword [ v_MacroWordCount ], MACRO_INITIAL_WORD_COUNT ; initial #macros
    mov dword [ v_BlueWordCount ],  BLUE_INITIAL_WORD_COUNT  ; initial #blues

    mov dword [ v_trash ], TRASH_BUFFER

    push esi

;Forth wordlist
    lea esi, [ ForthNamesROM ]
    mov edi, ForthNames
    mov ecx, [ v_ForthWordCount ]
    rep  movsd                  ; copy ecx 32 bit words from ds:esi to es:edi
    lea esi, [ ForthJumpTableROM ]
    mov edi, ForthJumpTable
    mov ecx, [ v_ForthWordCount ]
    rep  movsd                  ; copy ecx 32 bit words from ds:esi to es:edi
; Macro wordlist
    lea esi, [ MacroNamesROM ]
    mov edi, MacroNames
    mov ecx, [ v_MacroWordCount ]
    rep  movsd                  ; copy ecx 32 bit words from ds:esi to es:edi
    lea esi, [ MacroJumpTableROM ]
    mov edi, MacroJumpTable
    mov ecx, [ v_MacroWordCount ]
    rep  movsd                  ; copy ecx 32 bit words from ds:esi to es:edi

    pop esi

    mov dword [ v_H ], H0
    mov dword [ x_qwerty ], 0x00        ; select non-qwerty mode
    mov dword [ v_offset ], ( RELOCATED >> ( 2 + 8 ) ) ; 0x10000 >> 2 >> 8, offset of RELOCATED block 0 as
1024 byte block number

; Historical note. This bug took about 15 hours to find and fix...
```

```
; Below is code to track down a bug : Block 64 offset 0x7C contained 0x800
; The code with the two test functions was re-compiled using the cf2022Ref.img file
; Then block 64 was manually fixed ": rtc 94 ld ;"
; The source blocks were saved with "sa"
; cf2022 was restarted without recompilation
; Looking at blocks 506 and 507 showed that the bug occurred between the two copy functions
;
;    ; OK at this point
;    mov _SCRATCH_, [ v_blk ]
;    mov dword [ v_blk ], 506   ; block 506 shows corruption
;    mov _TOS_, 64
;    call copy_
;    mov [ v_blk ], _SCRATCH_

    ; setup  v_bytesPerLine
    mov _TOS_, [ vesa_XResolution ]
    and _TOS_, 0xFFFF
    imul _TOS_, BYTES_PER_PIXEL
    mov [ v_bytesPerLine ], _TOS_
    ; was :     mov [ v_bytesPerLine + RELOCATED ], _TOS_   <--- BUG!!!

;    ; NOT OK at this point
;    mov _SCRATCH_, [ v_blk ]
;    mov dword [ v_blk ], 507   ; block 506 shows corruption???
;    mov _TOS_, 64
;    call copy_
;    mov [ v_blk ], _SCRATCH_

    ; set up  fov
    mov _TOS_, [ vesa_YResolution ]
    and _TOS_, 0x0000FFFF
    mov _SCRATCH_, _TOS_
    shl _SCRATCH_, 1
    shr _TOS_, 1
    add _TOS_, _SCRATCH_
    imul _TOS_, 10
    mov [ v_fov ], _TOS_

    ; select which code to use, depending on the display mode
    mov byte [ displayMode ], 0
    cmp word [ vesa_XResolution ], scrnw1
    jz .forward
    mov byte [ displayMode ], 1
.forward:

; *****************************************************************************
; miscellaneous setup
; *****************************************************************************

    call randInit_      ; initialise the Marsaglia Pseudo Random Number Generator
    call initIconSize   ; sets up the size of an icon (glyph) according to the 800x600 or 1024x768 display
size
    call cursorHome     ; setup the initial cursor location
    call c_             ; clear the stack

; *****************************************************************************
; erase the DAP buffer, for the Int 0x13 Disk Address Packet (DAP)
; *****************************************************************************
    _DUP_
    mov _TOS_, SECTOR_BUFFER
    _DUP_
    mov _TOS_, SECTOR_BUFFER_SIZE
    call erase_

; *****************************************************************************
; load the colorForth source starting at the first colorForth source block
; *****************************************************************************
    _DUP_
    mov _TOS_, START_BLOCK_NUMBER
    _DUP_                        ; not sure why we need this...
```

```
    call _load_
    jmp dword quit_

; ****************************************************************************
; ****************************************************************************

pad_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_pad
    ret

v_srch:     ; variables to search for a token name
    dd 0xC4B80000   ; token name "pad"
    dd 0            ; token name extension (optional)
    dd 0            ; current found address
    dd 0            ; last found address
    dd ( START_BLOCK_NUMBER * 1024 )        ; start searching from here
    dd ( ( LAST_BLOCK_NUMBER + 1 ) * 1024 ) ; end the search here

vsrch_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_srch
    ret

srch_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_srch
    ret

; align 4, db 0  ; variables must be on dword boundary so that "dump" can show them correctly
;
; hsvv:    ; the start address of the pre-assembled high level Forth words
;     dd 0
;     times 0x28 db 0

xy_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_gr_xy
    ret

x_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_x
    ret

y_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_y
    ret

z_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_z
    ret

lblk_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_lblk
    ret

fov_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_fov
    ret

tokenActions_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS tokenActions
    ret
```

```
last_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS last
    ret

version_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS version
    ret

vframe_:     ; ( -- a )  \ return the video frame address, where we create the image to be displayed
    _DUP_
    mov _TOS_, [ vframe ]
    ret

vars_:
    _DUP_
    LOAD_RELATIVE_ADDRESS vars
    ret

base_:
    _DUP_
    LOAD_RELATIVE_ADDRESS base
    ret

hex_:
    mov byte [ base ], 16
    ret

decimal_:
    mov byte [ base ], 10
    ret

block_: ; ( block -- address )   \ : block ( n -- n ) $400 * ; block number to byte address of block
    shl _TOS_, 0x0A
    add _TOS_, RELOCATED
    ret

a2blk_: ; ( address -- block )   \ byte address of block to block number
    sub _TOS_, RELOCATED
    shr _TOS_, 0x0A
    ret

scrnw_:    ; ( -- n )   screen width ( number of horizontal pixels )
    _DUP_
    xor _TOS_, _TOS_
    mov word ax, [ vesa_XResolution ]
    ret

scrnh_:    ; ( -- n )    screen height ( number of vertical pixels )
    _DUP_
    xor _TOS_, _TOS_
    mov word ax, [ vesa_YResolution ]     ; v_scrnh
    ret

bpp_:    ; ( -- n )    bits per pixel
    _DUP_
    xor _TOS_, _TOS_
    mov byte al, [ vesa_BitsPerPixel ]     ; v_bitsPerPixel
    ret

iconw_:    ; ( -- n )    icon width ( number of pixels between characters, fixed font width )
    _DUP_
    mov _TOS_, [ v_iconw ]
    ret

iconh_:    ; ( -- n )    icon height ( number of pixels between lines )
    _DUP_
    mov _TOS_, [ v_iconh ]
    ret
```

```
counter_:   ; ( -- n )    roughly 1 ms counter
    _DUP_
    RDTSC   ; Read Time-Stamp Counter  https://c9x.me/x86/html/file_module_x86_id_278.html
    mov ecx, 1000
    idiv ecx
    ret

; : drop ( lodsd, flags unchanged, why sp is in ESI )
; : a! ?lit if $BA 1, , ; then $D08B 2, drop ;
; : p@ ( a-n ) qdup a! $EC 1, ;
; : p! ( na- ) a! $EE 1, drop ;
; ( Real Time Clock )
; : rtc@ ( t-c ) $70 p! $71 p@ ;
; : rtc! ( ct- ) $70 p! $71 p! ;
; : hi ( -- ) #10 rtc@ $80 and drop 0if hi ; then ;
; : lo ( -- ) #10 rtc@ $80 and drop if lo ; then ;
; : calkhz ( -- ) hi lo counter hi lo counter swap -
;    dup onesec ! #1 rshift #250 + #500 / dup khz ! ;
; : ms ( n- ) khz @ * counter + begin pause dup counter
;    invert + drop -if drop ; then end drop ;


p70_fetch:  ;  ( reg -- c )
    mov edx, 0x70   ; db 0xBA  dd 0x70
    IN AL, DX       ; db 0xEC
    ret

p70_store:  ;  ( c reg -- )
    mov edx, 0x70   ; db 0xBA  dd 0x70
    OUT DX, AL      ; db 0xEE
    _DROP_
    _DROP_
    ret

p71_fetch:  ;  ( reg -- c )
    mov edx, 0x71   ; db 0xBA  dd 0x71
    IN AL, DX       ; db 0xEC
    ret

p71_store:  ;  ( c reg -- )
    mov edx, 0x71   ; db 0xBA  dd 0x71
    OUT DX, AL      ; db 0xEE
    _DROP_
    _DROP_
    ret

rtc_fetch_:  ;  ( reg -- c )
    mov edx, 0x70   ; db 0xBA  dd 0x70
    OUT DX, AL      ; db 0xEE
    mov edx, 0x71   ; db 0xBA  dd 0x71
    IN AL, DX       ; db 0xEC
    ret

rtc_store_:  ;  ( c reg -- )
    mov edx, 0x70   ; db 0xBA  dd 0x70
    OUT DX, AL      ; db 0xEE
    mov edx, 0x71   ; db 0xBA  dd 0x71
    _DROP_
    OUT DX, AL      ; db 0xEE
    _DROP_
    ret

rtc_hi:     ; ( -- )   wait for the RTC second pulse to go high
    _DUP_
    .back:
    mov _TOS_, 10   ; Update in progress" flag (bit 7 of Status Register A).
    call rtc_fetch_
    and al, 0x80
    jz .back
```

```
    _DROP_
    ret

rtc_lo:     ; ( -- )   wait for the RTC second pulse to go low
    _DUP_
    .back:
    mov _TOS_, 10   ; Update in progress" flag (bit 7 of Status Register A).
    call rtc_fetch_
    and al, 0x80
    jnz .back
    _DROP_
    ret

get_proc_clk:   ; ( -- d )  get the processor clock counter
    _DUP_
    RDTSC                           ; Read Time-Stamp Counter
https://c9x.me/x86/html/file_module_x86_id_278.html
    _DUP_
    mov _TOS_, edx                  ; put the high cell in TOS
    ret

calck_:         ; ( -- )   calibrate the ms counter clock
    call rtc_hi                     ; wait for the RTC second pulse to go high
    call rtc_lo                     ; wait for the RTC second pulse to go low
    call get_proc_clk
    call d_negate_                  ; so the d_plus_ later subtracts this value
    call rtc_hi                     ; wait for the RTC second pulse to go high
    call rtc_lo                     ; wait for the RTC second pulse to go low
    call get_proc_clk
    call d_plus_                    ; double number "subtract"
    mov [ v_onesec ], _TOS_
    mov _SCRATCH_, [ esi ]
    mov [ v_onesec + 4 ], _SCRATCH_ ; put the result in onesec
    ret

ms_:        ; ( n -- )   delay n milli seconds
    _DROP_
    ret

onesec_:    ; ( -- a )    return the address of the onesec variable
    _DUP_
    LOAD_RELATIVE_ADDRESS v_onesec ;
    ret

khz_:       ; ( -- a )    return the address of the khz variable
    _DUP_
    LOAD_RELATIVE_ADDRESS v_khz ;
    ret

font_:    ; ( -- n )    return the address of the font pointer
    _DUP_
    LOAD_RELATIVE_ADDRESS v_font ; font16x24
    ret

last:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_lastToken
    ret

blk_:       ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_blk
    ret

seeb:       ; ( -- )    \ toggle the display of blue words in the editor
    not byte [ v_seeb ]
    ret

colourBlindModeToggle:   ; ( -- )    \ toggle the editor display colorForth / ANS style
    not byte [ v_colourBlindMode ]
```

```
    ret

curs:       ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_curs
    ret

; analyse stack usage
; the stack areas are initialised to all 'U's at power up
; areas of 8 bytes that re not all 'U's are marked by a byte in a 512 byte buffer
; analyse_eight_bytes:    ; ( a -- a' )   zero flag is true if all 'U's
;     xor edx, edx
;     cmp dword [ _TOS_ ], 0x55555555
;     jz .forward
;         inc edx     ; not all 'U's
;     .forward:
;     inc dword _TOS_     ; next address
;     cmp dword [ _TOS_ ], 0x55555555
;     jz .forward2
;         inc edx     ; not all 'U's
;     .forward2:
;     inc dword _TOS_     ; next address
;     add edx, 0
;     ret

analyse_stacks:
    mov _TOS_, STACK_MEMORY_START
    mov _SCRATCH_, STACK_ANALYSIS_BUFFER
    mov ecx, ( TOTAL_STACK_SIZE / 8 ) ;  0x200
    .back:
        ; call analyse_eight_bytes
        xor edx, edx
        cmp dword [ _TOS_ ], 0x55555555
        jz .forward
            inc edx     ; not all 'U's
        .forward:
        add _TOS_, 4    ; next address
        cmp dword [ _TOS_ ], 0x55555555
        jz .forward2
            inc edx     ; not all 'U's
        .forward2:
        add _TOS_, 4    ; next address
        add edx, 0

        jz .forward3
            mov byte [ _SCRATCH_], 0x2E
            jmp .forward4
        .forward3:
            mov byte [ _SCRATCH_], 0x55
        .forward4:
        inc _SCRATCH_   ; next address in the results buffer
    loop .back
    ret

stacks_:    ; ( -- a n )
    call analyse_stacks
    _DUP_
    mov _TOS_, STACK_ANALYSIS_BUFFER
    ret
    _DUP_
    mov _TOS_, STACK_MEMORY_START
    _DUP_
    mov _TOS_, TOTAL_STACK_SIZE
    ret

%if 0
stacks_:   ;  ( -- a )   \ return the address of the stack memory information ( see v_stack_info for
details )
;RETURN_STACK_SIZE
;DATA_STACK_SIZE
```

```
;STACK_MEMORY_START       ; bottom of stack memory
;TOTAL_STACK_SIZE
    _DUP_
    mov _TOS_, RETURN_STACK_0 - 0x3C           ; top of task 0 return stack
    _DUP_
    mov _TOS_, DATA_STACK_0   - 0x3C           ; top of task 0 data stack
    _DUP_
    mov _TOS_, RETURN_STACK_1 - 0x3C           ; top of task 1 return stack
    _DUP_
    mov _TOS_, DATA_STACK_1   - 0x3C           ; top of task 1 data stack
    _DUP_
    mov _TOS_, RETURN_STACK_2 - 0x3C           ; top of task 2 return stack
;    _DUP_
;    mov _TOS_, DATA_STACK_2   - 0x3C           ; top of task 2 data stack
;    LOAD_RELATIVE_ADDRESS v_stack_info
    ret
%endif

ekt:   ; ( -- a ) ; editor key table - variable containing vectors for editor keys beginning with null
    ; and the shift keys.  Then follows right hand top, middle, bottom rows,
    ; and left hand top, middle, bottom rows. (from ColorForth2.0a.doc)
    _DUP_
    LOAD_RELATIVE_ADDRESS editorActionTable
    ret

vword_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_words
    ret

;vregs_:   ; ( -- a )
;    _DUP_
;    mov _TOS_, V_REGS
;    ret

ivec_:   ; ( -- a )
    _DUP_
    mov _TOS_, INTERRUPT_VECTORS
    ret

pic_:   ; ( -- a )
    _DUP_
    mov _TOS_, IDT_AND_PIC_SETTINGS
    ret

%if 0

From : https://pdos.csail.mit.edu/6.828/2014/readings/hardware/8259A.pdf

The following registers can be read via OCW3 (IRR and ISR or OCW1 [IMR]).

Interrupt Request Register (IRR):
8-bit register which contains the levels requesting an interrupt to be acknowledged.
The highest request level is reset from the IRR when an interrupt is acknowledged. (Not affected by IMR.)

In-Service Register (ISR):
8-bit register which contains the priority levels that are being serviced.
The ISR is updated when an End of Interrupt Command is issued.

Interrupt Mask Register:
8-bit register which contains the interrupt request lines which are masked.
    The IRR can be read when, prior to the RD pulse, a Read Register Command is issued with OCW3 (RR = 1,
RIS = 0.)
    The ISR can be read, when, prior to the RD pulse, a Read Register Command is issued with OCW3 (RR = 1,
RIS = 1).
There is no need to write an OCW3 before every status read operation,
   as long as the status read corresponds with the previous one; i.e., the 8259A 'remembers' whether
   the IRR or ISR has been previously selected by the OCW3.
   This is not true when poll is used.
   After initialization the 8259A is set to IRR.
```

```
For reading the IMR, no OCW3 is needed.
The output data bus will contain the IMR whenever RD is active and A0 = 1 (OCW1).
Polling overrides status read when P = 1, RR = 1 in OCW3.

From : https://en.wikibooks.org/wiki/X86_Assembly/Programmable_Interrupt_Controller

Remapping
Another common task, often performed during the initialization of an operating system, is remapping the
PICs.
That is, changing their internal vector offsets, thereby altering the interrupt numbers they send.
The initial vector offset of PIC1 is 8, so it raises interrupt numbers 8 to 15.
Unfortunately, some of the low 32 interrupts are used by the CPU for exceptions
(divide-by-zero, page fault, etc.), causing a conflict between hardware and software interrupts.
The usual solution to this is remapping the PIC1 to start at 32, and often the PIC2 right after it at 40.
This requires a complete restart of the PICs, but is not actually too difficult, requiring just eight
'out's.

mov al, 0x11
out 0x20, al                    ; restart PIC1
out 0xA0, al                    ; restart PIC2

mov al, 0x20
out 0x21, al                    ; PIC1 now starts at 32
mov al, 0x28
out 0xA1, al                    ; PIC2 now starts at 40

mov al, 3
out 0x21, al                    ; setup cascading
mov al, 0x02
out 0xA1, al

mov al, 0x01
out 0x21, al
out 0xA1, al                    ;done!

From: cf2019 Forth block 244
: p!   pc! ;  \ 8 bit port store
: pic1! $21 p! ;
: pic2! $A1 p! ;

: !pic cli
( init )       $11 dup $20 p! $A0 p!
( irq )        $20 pic1! $28 pic2!
( master )     #4 pic1!
( slave )      #2 pic2!
( 8086 mode )  #1 dup pic1! pic2!
( mask irqs )  $FF pic2! $FA pic1! ;

Re-factored :
: !pic cli
\ PIC1
( init )     $11 $20 p!
( irq )      $20 $21 p!
( master )   $04 $21 p!
( 8086 mode) $01 $21 p!
( mask irqs) $FA $21 p!
\ PIC2
( init )     $11 $A0 p!
( irq )      $28 $A1 p!
( slave )    $02 $A1 p!
( 8086 mode) $01 $A1 p!
( mask irqs) $FF $A1 p!
;

%endif

dap_:   ; ( -- a )
    _DUP_
    mov _TOS_, DAP_BUFFER
```

```
    ret

sect_:   ; ( -- a )
    _DUP_
    mov _TOS_, SECTOR_BUFFER
    ret

digin:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_digin
    ret

actc:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS actionColourTable
    ret

tickh:   ; ( -- a )    HERE variable address
    _DUP_
    LOAD_RELATIVE_ADDRESS v_H
    ret

md5buf_:   ; ( -- a )    \ the address of the 16 byte MD5 output buffer
    _DUP_
    mov _TOS_, MD5_OUTPUT_BUFFER
    ret

; ******************************************************************************
; wordlist addresses and lengths
; ******************************************************************************

maca_:    ; ( -- a )  \ the address of the Macro wordlist
    _DUP_
    mov _TOS_,  MacroNames
    ret

macn_:   ; ( -- a )   the number of words in the Macro wordlist
    _DUP_
    LOAD_RELATIVE_ADDRESS v_MacroWordCount
    ret

macl_:    ; ( -- a )  \ the address of the Macro Locates list
    _DUP_
    mov _TOS_,  MacroLocates
    ret

mact_:    ; ( -- a )  \ the address of the Macro Jump Table
    _DUP_
    mov _TOS_,  MacroJumpTable
    ret

ftha_:    ; ( -- a )  \ the address of the Forth wordlist
    _DUP_
    mov _TOS_,  ForthNames
    ret

fthn_:   ; ( -- a )   the number of words in the Forth wordlist
    _DUP_
    LOAD_RELATIVE_ADDRESS v_ForthWordCount
    ret

fthl_:    ; ( -- a )  \ the address of the Forth Locates list
    _DUP_
    mov _TOS_,  ForthLocates
    ret

ftht_:    ; ( -- a )  \ the address of the Forth Jump Table
    _DUP_
    mov _TOS_,  ForthJumpTable
    ret
```

```
; the blue wordlist is not extensible, at the moment
blua_:    ; ( -- a )  \ the address of the Blue wordlist
    _DUP_
    mov _TOS_,  BlueNames
    ret

blun_:   ; ( -- a )   the number of words in the Blue wordlist
    _DUP_
    LOAD_RELATIVE_ADDRESS v_BlueWordCount
    ret

; ******************************************************************************
; loc , see and fnd
; ******************************************************************************

loc_:   ; ( token# -- a )
    mov dword _TOS_, [ ( _TOS_ * 4 ) + ForthLocates ]
    ; ret
    cmp _TOS_, 0x0000000
    jz .forward                 ; do nothing is the locate field is 0
    jmp editAddress_
    .forward:
    ret

see_:   ; ( token# -- a )
    mov dword _TOS_, [ ( _TOS_ * 4 ) + ForthJumpTable ]
    ; call dmp_
    ret

fnd_:       ; ( sf -- token# )
    and _TOS_, byte -0x10 ; (saves 2 bytes compared to 'and _TOS_, 0xFFFFFFF0' )
    call find_
    mov _TOS_, 0x00000000
    jnz .skip
    mov _TOS_, ecx
.skip:
    ret

dmp_:
    mov _TOS_, [ ( _TOS_ * 4 ) + ForthNames ]
    ret

; ******************************************************************************
; ******************************************************************************

offset_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_offset
    ret

vesa:   ; ( -- a )
    _DUP_
    mov _TOS_, VESA_BUFFER
    ret

vesamode_:   ; ( -- u )
    _DUP_
    xor _TOS_, _TOS_
    mov word ax, [ vesa_SavedMode ]     ; the saved  VESA video mode value
    ret

fetchDX_:   ; ( -- c )
    _DUP_
    xor _TOS_, _TOS_
    push edi
    mov edi, DAP_BUFFER
    mov _TOS_l_, [ edi + o_Int13_DAP_saved_DX ]     ; setup DX value returned by the BIOS
    pop edi
    ret
```

```
trash_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_trash
    ret

buffer_:   ; ( -- a )
    _DUP_
    mov _TOS_, SECTOR_BUFFER ;0x25300
    ret

cad:   ; ( -- a )   \ the address of the cursor as an offset from the start of the currently displayed
block
    _DUP_
    LOAD_RELATIVE_ADDRESS v_cad
    ret

pcad:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS v_pcad
    ret

; hsvv_:   ; ( -- a )
;     _DUP_
;     LOAD_RELATIVE_ADDRESS hsvv
;     ret

displ:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS displayShannonFanoActions
    ret

cBlindAddr_:   ; ( -- a )
    _DUP_
    LOAD_RELATIVE_ADDRESS x_colourBlind
    ret

; ******************************************************************************
; memory operators
; ******************************************************************************

cFetch_:      ; ( a -- c ) \ c@
    xor _SCRATCH_, _SCRATCH_
    mov byte _SCRATCH_l_, [ _TOS_ ] ;
    mov _TOS_, _SCRATCH_
    ret

wFetch_:      ; ( a -- w ) \ w@
    xor _SCRATCH_, _SCRATCH_
    mov word _SCRATCH_x_, [ _TOS_ ] ;
    mov _TOS_, _SCRATCH_
    ret

fetch_:       ; ( a -- u ) \ @
    mov dword _TOS_, [ _TOS_ ] ;
    ret

two_fetch_:     ; ( a -- x1 x2 )
   sub esi, 4                      ; make room on stack
   mov _SCRATCH_, [ _TOS_ + 4 ]     ; read x1 from addr+4
   mov [ esi ], _SCRATCH_           ; write onto stack
   mov _TOS_, [ _TOS_ ]             ; read x2 from addr+0, replacing tos
   ret

cStore_:      ; ( c a -- )          \ c!
    mov _SCRATCH_, [ esi ]
    mov byte [ _TOS_ ], _SCRATCH_l_
    _DROP_
    _DROP_
    ret
```

```
wStore_:      ; ( w a -- )          \ w!
    mov _SCRATCH_, [ esi ]
    mov word [ _TOS_ ], _SCRATCH_x_
    _DROP_
    _DROP_
    ret

store_:       ; ( u a -- )          \ !
    mov _SCRATCH_, [ esi ]
    mov dword [ _TOS_ ], _SCRATCH_
    _DROP_
    _DROP_
    ret

two_store_:     ; ( x1 x2 a -- )    \ 2!
    mov _SCRATCH_, [ esi ]          ; x2 into scratch
    mov [ _TOS_ ], _SCRATCH_        ; write x2 to addr+0
    mov _SCRATCH_, [ esi + 4 ]      ; x1 into scratch
    mov [ _TOS_ + 4 ], _SCRATCH_    ; write x1 to addr+4
    _DROP_                          ; drop the stack
    _DROP_
    _DROP_
    ret

plus_store_:  ; ( n addr -- )       \ +!
    mov _SCRATCH_, [ esi ]          ; copy the value n into the scratch register
    add [ _TOS_ ], _SCRATCH_        ; add to value at addr
    _DROP_                          ; drop the stack
    _DROP_
    ret

; ****************************************************************************
; double number operators
; ****************************************************************************

d_negate_:  ; ( d1 -- d2 )
   not dword [ esi ]                ; invert d1-lo
   not _TOS_                        ; invert d1-hi
   add dword [ esi ], 1             ; make two's complement
   adc _TOS_, 0                     ; from invert + 1
   ret

d_plus_:   ; ( d1 d2 -- d3 )   add d2 to d1 to give d3
   mov _SCRATCH_, [ esi ]           ; get d2-lo
   add _SCRATCH_, [ esi + 8 ]       ; add d1-lo
   adc _TOS_, [ esi + 4 ]           ; add d1-hi and carry to d2-hi
   mov [ esi + 8 ], _SCRATCH_       ; write d3-low
   add esi, 8                       ; and clean up stack
   ret

d_minus_:   ; ( d1 d2 -- d3 )   subtract d2 from d1 to give d3
    call d_negate_
    call d_plus_
    ret

; ****************************************************************************
; stack operators
; ****************************************************************************

two_dup_:       ; ( a b -- a b a b )
;    sub esi, byte 0x08 ; lea esi, [ esi - 0x08 ]    ; pre-decrement the stack pointer, adding 2 cells
;    mov [ esi + 4 ], _TOS_   ; copy x2 to Third On Stack ( second on the real stack )
;    mov _SCRATCH_, [ esi + 8 ]    ; copy x1 to register ebx
;    mov [ esi ], _SCRATCH_        ; copy register ebx to Fourth On Stack
    _OVER_
    _OVER_
    ret

two_drop_:      ; ( a b -- )
```

```
    _DROP_
    _DROP_
    ret

two_swap_:      ; ( a b c d -- c d a b )
    mov _SCRATCH_, [ esi + 8 ]
    xchg _SCRATCH_, [ esi ]
    mov  [ esi + 8 ], _SCRATCH_
    xchg _TOS_, [ esi + 4 ]
    ret

two_over_:      ; ( a b c d -- a b c d a b )
    lea esi, [ esi - 8 ]
    mov [ esi + 4 ], _TOS_
    mov _SCRATCH_, [ esi + 0x10 ]
    mov [esi], _SCRATCH_
    mov _TOS_, [ esi + 0x0C ]
    ret

rot_:           ; ( a b c -- b c a)
    mov _SCRATCH_,[ esi + 4 ]
    mov ebp, [ esi ]
    mov [ esi + 4 ], ebp
    mov [ esi ],_TOS_
    mov _TOS_, _SCRATCH_
    ret

minus_rot_:     ; -rot ( a b c -- c b a)
    mov _SCRATCH_, [ esi + 4 ]
    mov ebp, [ esi ]
    mov [ esi + 4 ], _TOS_
    mov [esi], _SCRATCH_
    mov _TOS_, ebp
    ret

tuck_:          ; ( a b -- b a b )
    _SWAP_
    _OVER_
    ret

pick_:          ; ( ... n -- ... u ) where u is the n'th stack item
    mov _TOS_, [ esi + ( _TOS_ * 4 ) ]
    ret

%define CELL_WIDTH  0x04    ; this is a 32 bit wide system = 4 bytes

cell_:          ; ( -- c )
    _DUP_
    mov _TOS_, CELL_WIDTH
    ret

cell_minus_:    ; ( u -- u' )
    sub _TOS_, CELL_WIDTH
    ret

cell_plus_:    ; ( u -- u' )
    add _TOS_, CELL_WIDTH
    ret

cells_:         ; ( u -- u' )
    add _TOS_, _TOS_    ; this code must be changed if CELL_WIDTH is changed
    add _TOS_, _TOS_
    ret

; ****************************************************************************
; save and restore the Interrupt Descriptor Table and Interrupt Mask Registers
; ****************************************************************************

lidt_:  ; ( a -- )  \ set a into the Interrupt Descriptor Table (IDT) register
    cli
```

```
    push ebp
    mov ebp, ( PIC_BIOS_IDT_SETTINGS )   ; 6 bytes of RAM used to store the IDT info
    mov word [ ebp ], 0x03B7
    mov [ ebp + 2 ], _TOS_   ; save IDT base address from eax
    lidt [ ebp ]  ; db 0x0F, 0x01, 0x18
    _DROP_
    pop ebp
    ret

sidt_:  ; ( -- a )  \ return the address contained in the Interrupt Descriptor Table (IDT) register
    cli
    _DUP_
    push ebp
    mov ebp, ( IDT_AND_PIC_SETTINGS_PAD )   ; 6 bytes of RAM used to interface to the stack
    sidt [ ebp ]        ; write the 6-byte IDT to memory location pointed to by  ebp
    mov _TOS_, [ ebp + 2 ]   ; save IDT base address to eax
    pop ebp
    ret

save_BIOS_idt:  ; ( -- )  \ save the Interrupt Descriptor Table (IDT) register value
    cli
    push ebp
    mov ebp, ( PIC_BIOS_IDT_SETTINGS )   ; 6 bytes of RAM used to save the values in
    sidt [ ebp ]        ; write the 6-byte IDT to memory location pointed to by  ebp
    pop ebp
    ret

restore_BIOS_idt:  ; ( -- )  \ restore the saved IDT value into the Interrupt Descriptor Table (IDT)
register
    cli
    push ebp
    mov ebp, ( PIC_BIOS_IDT_SETTINGS )   ; 6 bytes of RAM used to restore from
    lidt [ ebp ]  ; db 0x0F, 0x01, 0x18
    pop ebp
    ret

save_BIOS_idt_and_pic: ; ( -- )   \ save the PIC1 and PIC2 IMR values into IDT_AND_PIC_SETTINGS at startup
    cli
    call save_BIOS_idt
    push ebp
    mov ebp, ( PIC_BIOS_IMR_SETTINGS )   ; 2 bytes of RAM used to save the IMR for PIC1 and PIC2
; PIC1
    in al, 0x21     ; read PIC1's IMR value
    mov [ ebp ], al
; PIC2
    inc ebp
    in al, 0xA1     ; read PIC 2's IMR value
    mov [ ebp ], al
    pop ebp
    ret

restore_BIOS_idt_and_pic: ; ( -- )   \ restore the saved BIOS PIC and IMR values into PIC1 and PIC2
    cli
    call restore_BIOS_idt
    push ebp
    mov ebp, ( PIC_BIOS_IMR_SETTINGS )   ; 2 bytes of RAM used to save the IMR for PIC1 and PIC2
; PIC1
    mov al, 0x11    ; init command
    out 0x20, al    ; init PIC1                 ( $11 $20 p! )
    mov al, 0x00    ; PIC1 Interrupt Vector table start address
    out 0x21, al    ; PIC1 now starts at 0x00   ( $00 $21 p! )
    mov al, 0x04    ; master mode command
    out 0x21, al    ; set PIC1 as master, sets up cascading of PIC1 and PIC2 ( $04 $21 p! )
    mov al, 0x01    ; 8086 command
    out 0x21, al    ; set 8086 mode         ( $01 $21 p! )
    mov al, [ ebp ] ; Interrupt Mask Register ( IMR )
    out 0x21, al    ; set PIC1's IMR, BIOS = 0xB8 ( $xx $21 p! )
; PIC2
    inc ebp
    mov al, 0x11    ; init command
```

```
    out 0xA0, al    ; init PIC2
    mov al, 0x08    ; PIC2 Interrupt Vector table start address
    out 0xA1, al    ; PIC2 now starts at 0x08   $08 $A1 p!
    mov al, 0x02    ; slave mode command
    out 0xA1, al    ; set PIC2 as slave     ( $02 $A1 p! )
    mov al, 0x01    ; 8086 command
    out 0xA1, al    ; set 8086 mode         ( $01 $A1 p! )
    mov al, [ ebp ] ; Interrupt Mask Register ( IMR )
    out 0xA1, al    ; set PIC2's IMR, BIOS = 0x8F ( $xx $A1 p! )
    pop ebp
    ret

restore_new_idt_and_pic: ; ( -- )   \ restore the new IDT and PIC IMR values
    cli
    push ebp
    mov ebp, ( PIC_NEW_IMR_SETTINGS )   ; 2 bytes of RAM used to save the IMR for PIC1 and PIC2
; PIC1
    mov al, 0x11    ; init command
    out 0x20, al    ; init PIC1                 ( $11 $20 p! )
    mov al, 0x20    ; PIC1 Interrupt Vector table start address
    out 0x21, al    ; PIC1 now starts at 0x20   ( $20 $21 p! )
    mov al, 0x04    ; master mode command
    out 0x21, al    ; set PIC1 as master, sets up cascading of PIC1 and PIC2 ( $04 $21 p! )
    mov al, 0x01    ; 8086 command
    out 0x21, al    ; set 8086 mode         ( $01 $21 p! )
    mov al, [ ebp ] ; Interrupt Mask Register ( IMR )
    out 0x21, al    ; set PIC1's IMR, BIOS = 0xB8 ( $xx $21 p! )
; PIC2
    inc ebp
    mov al, 0x11    ; init command
    out 0xA0, al    ; init PIC2
    mov al, 0x28    ; PIC2 Interrupt Vector table start address
    out 0xA1, al    ; PIC2 now starts at 0x28   $28 $A1 p!
    mov al, 0x02    ; slave mode command
    out 0xA1, al    ; set PIC2 as slave     ( $02 $A1 p! )
    mov al, 0x01    ; 8086 command
    out 0xA1, al    ; set 8086 mode         ( $01 $A1 p! )
    mov al, [ ebp ] ; Interrupt Mask Register ( IMR )
    out 0xA1, al    ; set PIC2's IMR, BIOS = 0x8F ( $xx $A1 p! )
    pop ebp
    ret

init_default_PIC_IMRs:  ; ( -- )
    pushf
    cli

    pusha
    mov esi, 0x0000                 ; source address = the BIOS interrupt vector table
    mov edi, INTERRUPT_VECTORS      ; destination address
    mov ecx, ( 1024 / 4 )           ; 1024 bytes in cells
    rep movsd                       ; does not change AX , it moves DS:SI to ES:DI and increments SI and
DI
    ; now copy Interrupts 0x00 to 0x0F up to 0x20 to 0x2F
    mov esi, 0x0000                 ; source address = the BIOS interrupt vector table
    mov edi, ( INTERRUPT_VECTORS + ( 0x20 * 4 ) )  ; destination address
    mov ecx, ( 0x10 )               ; 16 vectors in cells
    rep movsd                       ; does not change AX , it moves DS:SI to ES:DI and increments SI and
DI

    popa

    push ebp
    mov ebp, ( PIC_NEW_IMR_SETTINGS )   ; 2 bytes of RAM used to save the IMR for PIC1 and PIC2
    mov byte [ ebp ] , 0xFA   ; Interrupt Mask Register ( IMR ) saved value for PIC1
    inc ebp
    mov byte [ ebp ] , 0xFF   ; Interrupt Mask Register ( IMR ) saved value for PIC2
    pop ebp
    popf
    ret
```

```
set_PIC1_IMR:   ; ( c -- )   \ set the Interrupt Mask Register for PIC1 and copy to PIC_NEW_IMR_SETTINGS
    pushf
    cli
    push ebp
    mov ebp, ( PIC_NEW_IMR_SETTINGS )   ; 1 byte of RAM used to save the IMR for PIC1
    mov [ ebp ] , al    ; Interrupt Mask Register ( IMR )
    out 0x21, al        ; set PIC1's IMR ( $xx $21 p! )
    pop ebp
    popf
    _DROP_
    ret

set_PIC2_IMR:   ; ( c -- )   \ set the Interrupt Mask Register for PIC2 and copy to PIC_NEW_IMR_SETTINGS+1
    pushf
    cli
    push ebp
    mov ebp, ( PIC_NEW_IMR_SETTINGS + 1 )   ; 1 byte of RAM used to save the IMR for PIC1
    mov [ ebp ] , al    ; Interrupt Mask Register ( IMR )
    out 0xA1, al        ; set PIC2's IMR ( $xx $A1 p! )
    pop ebp
    popf
    _DROP_
    ret

; ****************************************************************************
; lp support for GRaphics demo
; ****************************************************************************

lp_:
;; test what lodsd actually does
;    push esi
;    mov esi, [ v_trash ]    ; setup EDI to point to the current trash buffer address
;    mov [ v_lcad ], esi
;    lodsd  ; loads a 32 bit dword from [ds:esi] into _TOS_, increments  esi  by 4 : true
;    mov [ v_pcad ], esi
;    pop esi
;    _DUP_
;    mov _TOS_, [ v_lcad ]
;    _DUP_
;    mov _TOS_, [ v_pcad ]
;    ret
    nop
    nop
    nop
    db 0x8B , 0xE8 ; mov ebp,eax
    lodsd  ; loads a 32 bit dword from [ds:esi] into _TOS_, increments  esi  by 4
    db 0x8B , 0xC8 ; mov ecx,eax
    lodsd  ; loads a 32 bit dword from [ds:esi] into _TOS_, increments  esi  by 4
    mov ebx,[edx+0x20]
    .back:
    mov [ebx],bp
    db 0x23 , 0xC0 ; and eax,eax   21C0                        and eax,eax
    js .forward
    add eax,[edx]
    add ebx,[edx+0x18]
    .forward:
    add eax,[edx+0x8]
    add ebx,[edx+0x10]
    loop .back
;    dd 0x8B909090 , 0xC88BADE8 , 0x205A8BAD , 0x232B8966
;    dd 0x030578C0 , 0x185A0302 , 0x03084203 , 0xECE2105A
    ret

; ****************************************************************************
; maths operators
; The ANSI/ISO Forth Standard (adopted in 1994) mandates the minimal set
; of arithmetic operators +  -  *  /  MOD  */  /MOD  */MOD and  M* .
; ****************************************************************************

two_slash_:                         ; ( n -- n' )   "2/" arithmetic divide by 2
```

```
    sar _TOS_, 0x01
    ret

u_two_slash_:                       ; ( u -- u' )   "u2/" unsigned divide by 2
    shr _TOS_, 0x01
    ret

rshift_:                            ; ( u c -- u' )   shift TOS right by c bits
    mov ecx, _TOS_
    _DROP_
    shr _TOS_, cl
    ret

lshift_:                            ; ( u c -- u' )   shift TOS left by c bits
    mov ecx, _TOS_
    _DROP_
    shl _TOS_, cl
    ret
%if 0
; untested!!!
mod_:                               ; ( n1 n2 -- n3 )
   mov _TOS_, [ esi ]               ; get dividend
   cdq                              ; sign extend dividend
   idiv _SCRATCH_                   ; do the divide
   add esi, 4                       ; clean up stack
   mov _TOS_, edx                   ; and return remainder in tos
   ret

%endif

; "idiv ecx" divides the signed double dividend EDX:EAX, by the divisor in ECX
; and stores the remainder in EDX and quotient in EAX
slash_mod_:                         ; /mod ( n1 n2 -- r q )
   mov ecx, _TOS_                   ; get n2 the divisor
   mov _TOS_, [ esi ]               ; get n1 the dividend
   cdq                              ; sign extend into edx
   idiv ecx                         ; do the divide
   mov [ esi ], edx                 ; and remainder to stack
   ret

; "imul ecx" multiplies ECX by EAX and stores the result in EDX:EAX
; ToDo: fix and test this properly...
star_slash_mod_:                    ; */mod ( n1 n2 n3 -- r q )
   push _TOS_
   mov ecx, [ esi ]                 ; get n2
   mov _TOS_, [ esi + 4 ]           ; get n1
   imul ecx                         ; n1*n2 => edx:eax
   add esi, 4                       ; clean up stack
   pop ecx
   idiv ecx                         ; n1*n2/n3
   mov [ esi ], edx                 ; remainder to stack
   ret

star_slash_:                        ; */ ( n1 n2 n3 -- n )
   push _TOS_
   mov ecx, [ esi ]                 ; get n2
   mov _TOS_, [ esi + 4 ]           ; get n1
   imul ecx                         ; n1*n2 => edx:eax
   add esi, 8                       ; clean up stack
   pop ecx
   idiv ecx                         ; n1*n2/n3
   ret

; U*/  is an unsigned  */  with the twist of rounding up
; It adds one less than the divisor ( u3 ) to the dividend before dividing
; ToDo: fix and test this properly...
u_star_slash_:                      ; U*/ ( u1 u2 u3 -- u )
   mov _SCRATCH_, _TOS_
   dec _SCRATCH_                    ; divisor
   mov ecx, [ esi ]                 ; get n2
```

```
   mov edx, [ esi + 4 ]            ; get n1
   mul ecx                         ; u1 * u2
   add _TOS_, _SCRATCH_              ; round up
   adc edx, 0                      ;
   inc _SCRATCH_                   ; restore the original u3 divisor
   div _SCRATCH_                   ; do the division
   add esi, 8                      ; clean up stack
   ret

cmove_:                            ; ( from to count -- )
    test _TOS_, _TOS_
    jz .forward
        mov _SCRATCH_, _TOS_
        mov edx, [ esi + 0 ]
        mov ecx, [ esi + 0x04 ]
        .back:
            mov byte al, [ ecx + 0 ]
            mov byte [ edx + 0 ], al
            inc ecx
            inc edx
            dec _SCRATCH_
        jnz .back
    .forward:
    mov _TOS_, [ esi + 0x08 ]
    add esi, 0x0C
    ret

two_star_:                         ; 2* ( u -- u' )    u' = 2 * u
    shl _TOS_, 1
    ret

two_star_star_:                    ; 2** ( c -- u )    u  = 2 ** c
    mov ecx, _TOS_
    mov _TOS_, 0x00000001
    shl _TOS_, cl
    ret

; ******************************************************************************
; Random and Pseudo Random Number Generators
; ******************************************************************************

GetCPUIDsupport:    ; ( -- )  equal flag is set if no CPUID support
    ; check to see if CPUID is supported
    pushfd              ; save EFLAGS
    pop eax             ; store EFLAGS in EAX
    mov ebx, eax        ; save in EBX for later testing
    xor eax, 00200000h  ; toggle bit 21
    push eax            ; push to stack
    popfd               ; save changed EAX to EFLAGS
    pushfd              ; push EFLAGS to TOS
    pop eax             ; store EFLAGS in EAX
    cmp eax, ebx        ; see if bit 21 has changed
    ret

GetRDRANDsupport:   ; zero flag is set if no support for RDRAND, the hardware Random Number generator
    mov _TOS_, 0x00000001     ; select the 'features' CPU information
    CPUID           ; get CPU information into eax, ebx, ecx and edx
    test eax, 0x40000000 ; Bit 30 of ECX returned by CPUID => RDRAND present if true
    ret

GetCPUID_:   ; ( -- u )
    _DUP_
    mov _TOS_, 0x00000001     ; select the 'features' CPU information
    CPUID           ; get CPU information into eax, ebx, ecx and edx
    ret

rdtsc_:  ; ( -- u )   \ return the current processor instruction counter
    _DUP_
    rdtsc ; db 0x0F, 0x31
    ret
```

```
randInit_:
    call rdtsc_
    push ebp
    mov ebp, v_random
    xor [ ebp ], _TOS_      ; vRandom ! , if the value was 0
    pop ebp
    _DROP_
    ret

%if 0
\ Marsaglia, "Xorshift RNGs".  http://www.jstatsoft.org/v08/i14/paper
: Random32 ( -- u )
    vRandom @
    dup 0= or
    dup 6 lshift xor
    dup 21 rshift xor
    dup 7 lshift xor
    dup vRandom ! ;
%endif

; \ Marsaglia, "Xorshift RNGs".  http://www.jstatsoft.org/v08/i14/paper
getRandMarsaglia: ; ( -- u )   \ load a 32 bit pseudo random number into TOS
    _DUP_
    push ebp
    mov ebp, v_random
    mov _TOS_, [ ebp ]      ; vRandom @
    test _TOS_, _TOS_
    jnz .forward            ; dup 0= or
        mov _TOS_, 0xFFFFFFFF
    .forward:

    mov _SCRATCH_, _TOS_    ; dup 6 lshift xor
    shl _SCRATCH_, 0x06
    xor _TOS_, _SCRATCH_

    mov _SCRATCH_, _TOS_    ; dup 21 rshift xor
    shr _SCRATCH_, 0x15
    xor _TOS_, _SCRATCH_

    mov _SCRATCH_, _TOS_    ; dup 7 lshift xor
    shl _SCRATCH_, 0x07
    xor _TOS_, _SCRATCH_

    mov [ ebp ], _TOS_      ; vRandom !

    pop ebp
    ret

rand_:  ; ( -- u )   \ load a 32 bit true random number into TOS
    _DUP_
    call GetCPUIDsupport
    je .NO_CPUID             ; if no change to bit 21, no CPUID
        ; CPUID is supported, so check if RDRAND is supported
        call GetRDRANDsupport
        jz .NO_CPUID         ; test for RDRAND support
            RDRAND _TOS_    ; supported, so call the instruction
            ret
    .NO_CPUID:
    _DROP_
    call getRandMarsaglia
    ret

randq_:     ; ( -- f )   \ returns true if the processor supports the RDRAND random number instruction
    _DUP_
    call GetCPUIDsupport
    jz .NO_CPUID             ; if no change, no CPUID
        ; CPUID is supported, so check if RDRAND is supported
        call GetRDRANDsupport
        jz .NO_CPUID         ; test for RDRAND support
```

```
            mov _TOS_, 0xFFFFFFFF
        ret
    .NO_CPUID:
    xor _TOS_, _TOS_
    ret

; *****************************************************************************
; CRC32 Cyclic Redundancy Checksum (32 bit)
; The International Standard 32-bit cyclical redundancy check defined by :
; [ITU-T-V42] International Telecommunications Union, "Error-correcting
; Procedures for DCEs Using Asynchronous-to-Synchronous Conversion",
; ITU-T Recommendation V.42, 1994, Rev. 1.
; and
; [ISO-3309]
; International Organization for Standardization,
; "Information Processing Systems--Data Communication High-Level Data Link
; Control Procedure--Frame Structure", IS 3309, October 1984, 3rd Edition.
; *****************************************************************************

crc32_table:
    dd 000000000h, 077073096h, 0EE0E612Ch, 0990951BAh, 0076DC419h, 0706AF48Fh, 0E963A535h, 09E6495A3h,
00EDB8832h, 079DCB8A4h
    dd 0E0D5E91Eh, 097D2D988h, 009B64C2Bh, 07EB17CBDh, 0E7B82D07h, 090BF1D91h, 01DB71064h, 06AB020F2h,
0F3B97148h, 084BE41DEh
    dd 01ADAD47Dh, 06DDDE4EBh, 0F4D4B551h, 083D385C7h, 0136C9856h, 0646BA8C0h, 0FD62F97Ah, 08A65C9ECh,
014015C4Fh, 063066CD9h
    dd 0FA0F3D63h, 08D080DF5h, 03B6E20C8h, 04C69105Eh, 0D56041E4h, 0A2677172h, 03C03E4D1h, 04B04D447h,
0D20D85FDh, 0A50AB56Bh
    dd 035B5A8FAh, 042B2986Ch, 0DBBBC9D6h, 0ACBCF940h, 032D86CE3h, 045DF5C75h, 0DCD60DCFh, 0ABD13D59h,
026D930ACh, 051DE003Ah
    dd 0C8D75180h, 0BFD06116h, 021B4F4B5h, 056B3C423h, 0CFBA9599h, 0B8BDA50Fh, 02802B89Eh, 05F058808h,
0C60CD9B2h, 0B10BE924h
    dd 02F6F7C87h, 058684C11h, 0C1611DABh, 0B6662D3Dh, 076DC4190h, 001DB7106h, 098D220BCh, 0EFD5102Ah,
071B18589h, 006B6B51Fh
    dd 09FBFE4A5h, 0E8B8D433h, 07807C9A2h, 00F00F934h, 09609A88Eh, 0E10E9818h, 07F6A0DBBh, 0086D3D2Dh,
091646C97h, 0E6635C01h
    dd 06B6B51F4h, 01C6C6162h, 0856530D8h, 0F262004Eh, 06C0695EDh, 01B01A57Bh, 08208F4C1h, 0F50FC457h,
065B0D9C6h, 012B7E950h
    dd 08BBEB8EAh, 0FCB9887Ch, 062DD1DDFh, 015DA2D49h, 08CD37CF3h, 0FBD44C65h, 04DB26158h, 03AB551CEh,
0A3BC0074h, 0D4BB30E2h
    dd 04ADFA541h, 03DD895D7h, 0A4D1C46Dh, 0D3D6F4FBh, 04369E96Ah, 0346ED9FCh, 0AD678846h, 0DA60B8D0h,
044042D73h, 033031DE5h
    dd 0AA0A4C5Fh, 0DD0D7CC9h, 05005713Ch, 0270241AAh, 0BE0B1010h, 0C90C2086h, 05768B525h, 0206F85B3h,
0B966D409h, 0CE61E49Fh
    dd 05EDEF90Eh, 029D9C998h, 0B0D09822h, 0C7D7A8B4h, 059B33D17h, 02EB40D81h, 0B7BD5C3Bh, 0C0BA6CADh,
0EDB88320h, 09ABFB3B6h
    dd 003B6E20Ch, 074B1D29Ah, 0EAD54739h, 09DD277AFh, 004DB2615h, 073DC1683h, 0E3630B12h, 094643B84h,
00D6D6A3Eh, 07A6A5AA8h
    dd 0E40ECF0Bh, 09309FF9Dh, 00A00AE27h, 07D079EB1h, 0F00F9344h, 08708A3D2h, 01E01F268h, 06906C2FEh,
0F762575Dh, 0806567CBh
    dd 0196C3671h, 06E6B06E7h, 0FED41B76h, 089D32BE0h, 010DA7A5Ah, 067DD4ACCh, 0F9B9DF6Fh, 08EBEEFF9h,
017B7BE43h, 060B08ED5h
    dd 0D6D6A3E8h, 0A1D1937Eh, 038D8C2C4h, 04FDFF252h, 0D1BB67F1h, 0A6BC5767h, 03FB506DDh, 048B2364Bh,
0D80D2BDAh, 0AF0A1B4Ch
    dd 036034AF6h, 041047A60h, 0DF60EFC3h, 0A867DF55h, 0316E8EEFh, 04669BE79h, 0CB61B38Ch, 0BC66831Ah,
0256FD2A0h, 05268E236h
    dd 0CC0C7795h, 0BB0B4703h, 0220216B9h, 05505262Fh, 0C5BA3BBEh, 0B2BD0B28h, 02BB45A92h, 05CB36A04h,
0C2D7FFA7h, 0B5D0CF31h
    dd 02CD99E8Bh, 05BDEAE1Dh, 09B64C2B0h, 0EC63F226h, 0756AA39Ch, 0026D930Ah, 09C0906A9h, 0EB0E363Fh,
072076785h, 005005713h
    dd 095BF4A82h, 0E2B87A14h, 07BB12BAEh, 00CB61B38h, 092D28E9Bh, 0E5D5BE0Dh, 07CDCEFB7h, 00BDBDF21h,
086D3D2D4h, 0F1D4E242h
    dd 068DDB3F8h, 01FDA836Eh, 081BE16CDh, 0F6B9265Bh, 06FB077E1h, 018B74777h, 088085AE6h, 0FF0F6A70h,
066063BCAh, 011010B5Ch
    dd 08F659EFFh, 0F862AE69h, 0616BFFD3h, 0166CCF45h, 0A00AE278h, 0D70DD2EEh, 04E048354h, 03903B3C2h,
0A7672661h, 0D06016F7h
    dd 04969474Dh, 03E6E77DBh, 0AED16A4Ah, 0D9D65ADCh, 040DF0B66h, 037D83BF0h, 0A9BCAE53h, 0DEBB9EC5h,
047B2CF7Fh, 030B5FFE9h
    dd 0BDBDF21Ch, 0CABAC28Ah, 053B39330h, 024B4A3A6h, 0BAD03605h, 0CDD70693h, 054DE5729h, 023D967BFh,
0B3667A2Eh, 0C4614AB8h
```

```
    dd 05D681B02h, 02A6F2B94h, 0B40BBE37h, 0C30C8EA1h, 05A05DF1Bh, 02D02EF8Dh

; CRC-32 with polynomial $04c11db7, as specified in IEEE 802.3 ( Ethernet )
 crc32_:    ; ( a n -- u )   \ CRC32 Cyclic Redundancy Checksum
    push    _SCRATCH_
    push    ecx
    push    edx

    mov ecx, _TOS_
    _DROP_
    mov _SCRATCH_, _TOS_
    ; address in ebx, count in ecx, result in eax

    xor edx, edx

    mov  _TOS_, 0xFFFFFFFF     ; initial CRC value

    test ecx, ecx
    jz .forward

        .back:
        mov dl, byte [_SCRATCH_]
        xor dl, al
        shr _TOS_, 8
        xor _TOS_, dword [ crc32_table + ( 4 * edx ) ]
        inc _SCRATCH_
        dec ecx
        jnz .back

        not _TOS_     ; invert the final CRC value
    .forward:

    pop edx
    pop ecx
    pop _SCRATCH_
    ret

; ****************************************************************************
; MD5
; ****************************************************************************

; From : https://github.com/rwfpl/rewolf-md5/blob/master/nasm/rewolf_md5.inc

;---------------------------------------------------------------------------
;|                      The MD5 Message-Digest Algorithm                   |
;---------------------------------------------------------------------------
;|   Description:                                                          |
;|   ============                                                          |
;|                                                                         |
;|   The MD5 algorithm is designed to be quite fast on 32-bit machines. In |
;|   addition,  the MD5 algorithm  does not require any large substitution |
;|   tables, the algorithm can be coded quite compactly.                   |
;|                                                                         |
;|   The MD5 algorithm is an extension of the MD4 message-digest algorithm |
;|   1,2]. MD5 is  slightly slower than MD4, but is more "conservative" in |
;|   design. MD5  was designed  because it  was felt  that MD4 was perhaps |
;|   being adopted  for use more  quickly than  justified by  the existing |
;|   critical  review, because MD4  was designed to be exceptionally fast, |
;|   it  is "at the edge"  in terms of  risking  successful  cryptanalytic |
;|   attack.  MD5 backs off  a bit, giving up a little in speed for a much |
;|   greater  likelihood  of  ultimate  security.  It  incorporates  some  |
;|   suggestions  made  by  various  reviewers,  and  contains  additional |
;|   optimizations. The MD5 algorithm is being placed in the public domain |
;|   for review and possible adoption as a standard.                       |
;|                                                                         |
;---------------------------------------------------------------------------
;|   Implementation based on rfc1321 (fully rewritten in asm, not ripped :))|
;---------------------------------------------------------------------------
;|   Usage:                                                                |
;|   ======                                                                |
```

```
;|                                                                          |
;|   Simply include this file to your project:                              |
;|   exp: include \..path..\rewolf_md5.inc                                  |
;|                                                                          |
;|   Target compiler...: NASM-YASM                                          |
;|   Calling convention:                                                    |
;|                                                                          |
;|       push    size of datablock                                          |
;|       push    datablock                                                  |
;|       push    destHash                                                   |
;|       call    _rwf_md5                                                   |
;|                                                                          |
;|   datablock -> (input)  -> buffer that contains data to hash             |
;|   destHash  -> (output) -> 16-bytes buffer for hashed data               |
;|                                                                          |
;|   Modified registers: none                                               |
;|   Stack is automatically cleared                                         |
;----------------------------------------------------------------------------
;|   Coder.: ReWolf^HTB                                                     |
;|   Date..: 17.XII.2004                                                    |
;|   E-mail: rewolf@poczta.onet.pl                                          |
;|   WWW...: http://www.rewolf.prv.pl                                       |
;----------------------------------------------------------------------------
;|   Adaptation for NASM/YASM:  Ange Albertini                              |
;----------------------------------------------------------------------------

S11 equ 7
S12 equ 12
S13 equ 17
S14 equ 22
S21 equ 5
S22 equ 9
S23 equ 14
S24 equ 20
S31 equ 4
S32 equ 11
S33 equ 16
S34 equ 23
S41 equ 6
S42 equ 10
S43 equ 15
S44 equ 21

%macro FF 7 ;a,b,c,d,k,s,i
    mov edi,%2
    mov ebp,%2
    and edi,%3
    not ebp
    and ebp,%4
    or  edi,ebp
    lea %1, [%1+edi+%7]
    add %1, dword [esi+%5*4]
    rol %1,%6
    add %1,%2
%endmacro

%macro GG 7
    mov edi,%4
    mov ebp,%4
    and edi,%2
    not ebp
    and ebp,%3
    or  edi,ebp
    lea %1, [%1+edi+%7]
    add %1, dword [esi+%5*4]
    rol %1,%6
    add %1,%2
%endmacro

%macro  HH  7
```

```
    mov ebp,%2
    xor ebp,%3
    xor ebp,%4
    lea %1, [%1+ebp+%7]
    add %1,dword [esi+%5*4]
    rol %1,%6
    add %1,%2
%endmacro

%macro  II  7
    mov ebp,%4
    not ebp
    or  ebp,%2
    xor ebp,%3
    lea %1, [%1+ebp+%7]
    add %1,dword [esi+%5*4]
    rol %1,%6
    add %1,%2
%endmacro

;|       push    size of datablock                                              |
;|       push    datablock                                                      |
;|       push    destHash
md5_:    ; ( a n -- md5_address )
    push _TOS_      ; [ a -- ]
    _DROP_
    push _TOS_  ; [ a n -- ]
    mov _SCRATCH_, MD5_OUTPUT_BUFFER
    push _SCRATCH_  ; [ a n md5_output -- ]
    call _rwf_md5
    mov _TOS_, MD5_OUTPUT_BUFFER
    ret

_rwf_md5:   ; ( a n outputPtr 0 -- )
    pushad
    mov esi,dword [esp+04h+8*4]
    mov dword [esi], 067452301h
    mov dword [esi+04h], 0efcdab89h
    mov dword [esi+08h], 098badcfeh
    mov dword [esi+0Ch], 010325476h
    mov eax,dword [esp+0Ch+8*4]
    push eax
    xor edx,edx
    mov ecx,64
    div ecx
    inc eax
    pop edx
    sub esp,64
    mov ebx,esp
    mov esi,dword [esp+08h+24*4]
    xchg    eax,edx
_n0:
    mov edi,ebx
    dec edx
    jne _n1
    test eax,eax
    js  _nD
    mov byte [ebx+eax],80h
    jmp _nC
_nD:
    xor eax,eax
    dec eax
_nC:
    mov ecx,64
    sub ecx,eax
    add edi,eax
    push eax
    xor eax,eax
    inc edi
    dec ecx
```

```
    rep stosb
    pop eax
    test eax,eax
    js  _nB
    cmp eax,56
    jnb _nE
_nB:
    push eax
    mov eax,dword [esp+0Ch+25*4]
    push edx
    xor edx,edx
    mov ecx,8
    mul ecx
    mov dword [ebx+56],eax
    mov dword [ebx+60],edx
    pop edx
    pop eax
    jmp _n1
_nE:
    inc edx
_n1:
    test eax,eax
    js  _nA
    cmp eax,64
    jnb _n2
    jmp _n10
_nA:
    xor eax,eax
_n10:
    mov ecx,eax
    jmp _n3
_n2:
    mov ecx,64
_n3:
    mov edi,ebx
    rep movsb
    push eax
    push edx
    push ebx
    push esi
    lea esi, [esp+10h]
    mov edi, dword [esp+4+28*4]
    push edi
    mov eax, dword [edi]
    mov ebx, dword [edi+04h]
    mov ecx, dword [edi+08h]
    mov edx, dword [edi+0Ch]

    FF  eax, ebx, ecx, edx, 0, S11, 0d76aa478h
    FF  edx, eax, ebx, ecx, 1, S12, 0e8c7b756h
    FF  ecx, edx, eax, ebx, 2, S13, 0242070dbh
    FF  ebx, ecx, edx, eax, 3, S14, 0c1bdceeeh
    FF  eax, ebx, ecx, edx, 4, S11, 0f57c0fafh
    FF  edx, eax, ebx, ecx, 5, S12, 04787c62ah
    FF  ecx, edx, eax, ebx, 6, S13, 0a8304613h
    FF  ebx, ecx, edx, eax, 7, S14, 0fd469501h
    FF  eax, ebx, ecx, edx, 8, S11, 0698098d8h
    FF  edx, eax, ebx, ecx, 9, S12, 08b44f7afh
    FF  ecx, edx, eax, ebx, 10, S13, 0ffff5bb1h
    FF  ebx, ecx, edx, eax, 11, S14, 0895cd7beh
    FF  eax, ebx, ecx, edx, 12, S11, 06b901122h
    FF  edx, eax, ebx, ecx, 13, S12, 0fd987193h
    FF  ecx, edx, eax, ebx, 14, S13, 0a679438eh
    FF  ebx, ecx, edx, eax, 15, S14, 049b40821h

    GG  eax, ebx, ecx, edx, 1, S21, 0f61e2562h
    GG  edx, eax, ebx, ecx, 6, S22, 0c040b340h
    GG  ecx, edx, eax, ebx,11, S23, 0265e5a51h
    GG  ebx, ecx, edx, eax, 0, S24, 0e9b6c7aah
    GG  eax, ebx, ecx, edx, 5, S21, 0d62f105dh
```

```
    GG  edx, eax, ebx, ecx,10, S22, 002441453h
    GG  ecx, edx, eax, ebx,15, S23, 0d8a1e681h
    GG  ebx, ecx, edx, eax, 4, S24, 0e7d3fbc8h
    GG  eax, ebx, ecx, edx, 9, S21, 021e1cde6h
    GG  edx, eax, ebx, ecx,14, S22, 0c33707d6h
    GG  ecx, edx, eax, ebx, 3, S23, 0f4d50d87h
    GG  ebx, ecx, edx, eax, 8, S24, 0455a14edh
    GG  eax, ebx, ecx, edx,13, S21, 0a9e3e905h
    GG  edx, eax, ebx, ecx, 2, S22, 0fcefa3f8h
    GG  ecx, edx, eax, ebx, 7, S23, 0676f02d9h
    GG  ebx, ecx, edx, eax,12, S24, 08d2a4c8ah

    HH  eax, ebx, ecx, edx, 5, S31, 0fffa3942h
    HH  edx, eax, ebx, ecx, 8, S32, 08771f681h
    HH  ecx, edx, eax, ebx,11, S33, 06d9d6122h
    HH  ebx, ecx, edx, eax,14, S34, 0fde5380ch
    HH  eax, ebx, ecx, edx, 1, S31, 0a4beea44h
    HH  edx, eax, ebx, ecx, 4, S32, 04bdecfa9h
    HH  ecx, edx, eax, ebx, 7, S33, 0f6bb4b60h
    HH  ebx, ecx, edx, eax,10, S34, 0bebfbc70h
    HH  eax, ebx, ecx, edx,13, S31, 0289b7ec6h
    HH  edx, eax, ebx, ecx, 0, S32, 0eaa127fah
    HH  ecx, edx, eax, ebx, 3, S33, 0d4ef3085h
    HH  ebx, ecx, edx, eax, 6, S34, 004881d05h
    HH  eax, ebx, ecx, edx, 9, S31, 0d9d4d039h
    HH  edx, eax, ebx, ecx,12, S32, 0e6db99e5h
    HH  ecx, edx, eax, ebx,15, S33, 01fa27cf8h
    HH  ebx, ecx, edx, eax, 2, S34, 0c4ac5665h

    II  eax, ebx, ecx, edx, 0, S41, 0f4292244h
    II  edx, eax, ebx, ecx, 7, S42, 0432aff97h
    II  ecx, edx, eax, ebx,14, S43, 0ab9423a7h
    II  ebx, ecx, edx, eax, 5, S44, 0fc93a039h
    II  eax, ebx, ecx, edx,12, S41, 0655b59c3h
    II  edx, eax, ebx, ecx, 3, S42, 08f0ccc92h
    II  ecx, edx, eax, ebx,10, S43, 0ffeff47dh
    II  ebx, ecx, edx, eax, 1, S44, 085845dd1h
    II  eax, ebx, ecx, edx, 8, S41, 06fa87e4fh
    II  edx, eax, ebx, ecx,15, S42, 0fe2ce6e0h
    II  ecx, edx, eax, ebx, 6, S43, 0a3014314h
    II  ebx, ecx, edx, eax,13, S44, 04e0811a1h
    II  eax, ebx, ecx, edx, 4, S41, 0f7537e82h
    II  edx, eax, ebx, ecx,11, S42, 0bd3af235h
    II  ecx, edx, eax, ebx, 2, S43, 02ad7d2bbh
    II  ebx, ecx, edx, eax, 9, S44, 0eb86d391h

    pop edi
    add dword [edi],eax
    add dword [edi+04h],ebx
    add dword [edi+08h],ecx
    add dword [edi+0Ch],edx
    pop esi
    pop ebx
    pop edx
    pop eax
    sub eax,64
    test edx,edx
    jne _n0
    add esp,64
    popad
    ret 0Ch

; ******************************************************************************
; ******************************************************************************

align 4, nop

tens:
    dd 10
    dd 100
```

```
    dd 1000
    dd 10000
    dd 100000
    dd 1000000
    dd 10000000
    dd 100000000
    dd 1000000000

x_numberDisplay:    ; either dotDecimal or dotHex , depending on the BASE to use to display numbers
    dd dotDecimal

v_help_counter:     ; cycles through the help screens used by "help" ( F1 key )
    dd 0

v_saved_v_blk:      ; the block number saved by "help"
    dd 0xFF

v_locatedBlock:     ; the block of the located word
    dd 510          ; demo only to see if the interface is OK - toggling the located word on "L" in the
editor keypad
v_locatedCurs:      ; the address of the located word within its block
    dd 10           ; demo only to see if the interface is OK - toggling the located word on "L" in the
editor keypad

v_blk:              ; the currently edited block
    dd START_BLOCK_NUMBER ; the default edited block
v_curs:             ; the offset in cells of the cursor within the currently edited block
    dd 0

v_otherBlock:       ; the previously edited block
    dd 510          ; the default other block is the highest non-shadow block saved by 'sa'
v_otherCursor:      ; the v_curs cursor position in the other block
    dd 0

; v_otherBlocks:      ; the previously edited help block array
;     dd START_BLOCK_NUMBER           ; the default edited block
;     dd START_BLOCK_NUMBER + 1       ; the default other block is the shadow of the default edited block
;     dd START_BLOCK_NUMBER + 2       ; the default other block is the shadow of the default edited block
;     dd START_BLOCK_NUMBER + 3       ; the default other block is the shadow of the default edited block

v_cursPtr:          ; variable to count the cursor offset from the start of the block
    dd 0

v_cursLine:         ; which line we want to display the cursor on
    dd 0

v_curs_number_down: ; to limit the steps down
    dd 0

v_display_token_number: ; how many tokens we have displayed in the editor screen
    dd 0

v_numberOfMagentas:
    dd 0

v_numberOfBigConstants:
    dd 0

v_numberOfRedAndMagentas:
    dd 0

v_numberOfTokens:   ; in the current block
    dd 0

v_cad:              ; the address of the cursor as an offset from the start of the currently displayed
block
    dd 0

v_pcad:             ; saved pointer to current cursor address (?)
    dd 0
```

```
v_lcad:             ; saved length of 32 bit cells to move (?)
    dd 0

v_trash:                            ; pointer to "trash" buffer, saves words deleted while editing
    dd TRASH_BUFFER

v_offset:
    dd ( RELOCATED >> ( 2 + 8 ) )

v_bitsPerPixel:
    dd 16   ; default, set using VESA info

v_iconw:
    dd 0 ; iconw

v_iconh:
    dd 0 ; iconh

v_keypadY_iconh:
    dd 0    ; keypadY * iconh

v_nine_iconw:
    dd 0

v_twentytwo_iconw:  ; width of 12 history characters, 1 space and 9 keypad characters
    dd 0            ; to calculate the start of the history display, subtracted from the right edge of the
screen

v_10000_iconw:
    dd 0    ; iconw*0x10000

x_qwerty:           ; selects non-QWERTY if set to 0, else jumps to the address
    dd 0xFFFFFFFF   ;

x_abort:
    dd abort_action

x_colourBlind:  ; ( state -- state )
    dd colourBlindAction

; byte variables
v_seeb:             ; if = 255, show blue words in editor
    db 0            ; 255 enable, 0 disable

v_colourBlindMode:  ; if = 255, select ANS style editor display
    db 0            ; 255 enable, 0 disable

v_not_cr:   ; true to disable the  cr  before a red word is displayed in the editor
    db 0

v_quitMode:         ; if non zero, the keypad is in Edit mode, else immediate (TIB) mode
    db 0            ; 255 enable, 0 disable

v_hintChar:         ; the character to display in the bottom right hand corner of the keyboard
    dd 0            ; as a hint to the colour being used

v_random:           ; the current Marsaglia Pseudo Random Number Generator state
    dd 0

v_show_ASCII:       ; if true show the ASCII keyboard entry field at the cursor
    db 0

; align 4, nop    ; so we can read these variables easily from Forth
; md5_output: times 4 dd 0    ; the MD5 hash output

align 4

currentKeypadIcons:
    dd ( alphaKeypad - 4 )
```

```
shiftAction:        ; the table of Forth words to execute for the current keypad
    dd alpha0

vars:       ; colorForth system variables start here
base:
    dd 10

setCurrentBase:                     ; set the base to either decimalor hexadecimal
    dd setBase_decimal

keypad_colour:
    dd colour_yellow   ; current key colour for displaying key presses

chars:
    dd 1

aword:
    dd ex1

anumber:
    dd nul

v_words:
    dd 1

v_qwerty_key:
    dd 0

v_digin:
    dd 0

lit:
    dd adup

v_washColour:
    dd colour_background

v_spare:        ; just to align v_pad to a 16 byte boundary
    dd 0
    dd 0
    dd 0

mark_MacroWordCount:
    dd MACRO_INITIAL_WORD_COUNT ; initial #macros
    ; number of Macro words, saved by  mark , empty restores to this value
mark_v_ForthWordCount:
    dd FORTH_INITIAL_WORD_COUNT ; initial #words
    ; number of Forth words, saved by  mark , empty restores to this value
mark_v_BlueWordCount:
    dd BLUE_INITIAL_WORD_COUNT ; initial #words
    ; number of Blue words, saved by  mark , empty restores to this value

mark_H:
    dd H0       ; top of dictionary pointer H , saved by  mark , empty restores to this value

v_H:
    dd H0       ; variable H , dictionary pointer HERE, where new definitions go

v_lblk:
    dd 0        ; the last block loaded by  ld

; this variable address is 16 below pad
v_x:            ; general purpose variable e.g. for  dump
    dd 0

v_y:            ; general purpose variable e.g. for  dump
    dd 0

v_z:            ; general purpose variable
```

```
    dd 0

v_lastToken:
    dd 0

v_font:         ; a pointer to the current font table
    dd ( font16x24 )           ; default font

v_onesec:       ; one second's worth of counter counts
    dd 0
    dd 0
v_khz:          ; the Processor clock, scaled down to kHz
    dd 0
v_mhz:          ; the Processor clock, scaled down to MHz
    dd 0

v_lastAddress:
    dd 0
v_lastAddress_copy:
    dd 0                    ; saves a copy of v_lastAddress

v_ForthWordCount:
 dd FORTH_INITIAL_WORD_COUNT ; initial #words   ; number of words in the Forth wordlist, empty resets this
value

v_MacroWordCount:
 dd MACRO_INITIAL_WORD_COUNT ; initial #macros  ; number of words in the Macro wordlist, empty resets this
value

v_BlueWordCount:
 dd BLUE_INITIAL_WORD_COUNT ; initial #macros  ; number of words in the Blue wordlist, empty resets this
value

v_test1:                            ; two 32 bit variables to save test values in. Use ' pad 8 - 2@ ' to
view the values
    dd 0
v_test2:
    dd 0
v_pad:                              ; the standard Forth PAD, 84 bytes long
    times 84 db 0x00

tokenActions:       ;
    dd qignore      ; 0  extension token
    dd execute_lit  ; 1
    dd num          ; 2
adefine: ; compile time action for each type of token, 3 to 12
    dd forthd       ; 3
    dd qcompile     ; 4
    dd cnum         ; 5
    dd cshort       ; 6
    dd compile      ; 7
    dd short_       ; 8
    dd nul          ; 9
    dd nul          ; A
    dd nul          ; B
    dd m_variable   ; C magenta variable
    dd nul          ; D
    dd nul          ; E
    dd nul          ; F

v_gr_xy:  ; variable that holds the XY position for drawing characters, ( 0, 0 ) is top left
v_gr_y:
    dw 0x0003
v_gr_x:
    dw 0x0003

v_leftMargin:
    dd 0x00000003  ; left margin

v_rightMargin:
```

```
    dd 0           ; right margin

; xycr:
    ; dd 0

v_fov:  ; abstract display scale
    dd 0    ; 10 * ( 2 * scrnh + scrnh / 2 )

vframe:  ; pointer to display frame buffer where we create our image, down from top of 32 Mbytes RAM (
0x2000000 )
    dd 0x2000000 - ( MAX_SCREEN_WIDTH * MAX_SCREEN_HEIGHT * BYTES_PER_PIXEL )

; v_frameBuffer:          ; framebuffer address
;    dd 0x00000000       ;

v_foregroundColour:
    dd 0x00000000       ; the display foreground colour, set by  set_color_

v_xc:
    dd 0x00000000       ;
v_yc:
    dd 0x00000000          ;

MacroNamesROM:
    dd 0xF0000000       ; semicolon ";"
    dd 0xC19B1000       ; dup
    dd 0xCF833620       ; qdup
;    dd 0xFF833620       ; ?dup
    dd 0xC0278800       ; drop
    dd 0x2C88C000       ; then
    dd 0xC6957600       ; begin_
; MacroNamesROM_end:

MacroJumpTableROM:         ; jump table for the macro wordlist
    dd semicolon        ; ;
    dd cdup             ; compile dup
    dd qdup             ; qdup
    dd cdrop            ; compile drop
    dd then             ;
    dd begin_           ;
MacroJumpTableROM_end:

ForthNamesROM:      ; displayed using cf2ansi
    dd 0xC6664000   ; boot
    dd 0xBA8C4000   ; warm
    dd 0xC4B9A080   ; pause
    dd 0x8AC84C00   ; macro     macro_
    dd 0xB1896400   ; forth     forth_
    dd 0x90000000   ; c
    dd 0x1A635000   ; rlba      Read_Sector_LBA
    dd 0xBD31A800   ; wlba      Write_Sector_LBA
    dd 0x145C1000   ; reads     ReadSectors
    dd 0xB8B92400   ; writes    WriteSectors
    dd 0x84200000   ; sss       SaveAll_
;    dd 0x2C800000   ; th        th_ ( thunk to BIOS Int 0x13 )
    dd 0x145C0000   ; read      bios_read
    dd 0xB8B92000   ; write     bios_write
;    dd 0x18248800   ; rsect
    dd 0xF9832800   ; @dx       fetchDX_
    dd 0xF5817100   ; !dap
    dd 0x59100000   ; act(tivate)
    dd 0x8643B800   ; show
    dd 0xA1AE0000   ; load
    dd 0x6A1AE000   ; nload
    dd 0xF7435C00   ; +load
    dd 0x2C839800   ; thru
    dd 0xF6590730   ; +thru
    dd 0x963A7400   ; cblk      return the block number currently being compiled, calculated from  edi
    dd 0x1C74E800   ; rblk      return the block number offset of the RELOCATED address
    dd 0x5C74E800   ; ablk      4 / cellAddressToBlock
```

```
    dd 0x41582000   ; erase
    dd 0xC8828000   ; here      here_
    dd 0xFF472000   ; ?lit
    dd 0xD7F80000   ; 3,
    dd 0xD5F80000   ; 2,
    dd 0xD3F80000   ; 1,
    dd 0x97E00000   ; c,
    dd 0xFC000000   ; ,
    dd 0xA2420000   ; less
    dd 0xE59A3880   ; jump
    dd 0xCF99C800   ; quit_      was accept = dd 0x59493110
    dd 0xC4B80000   ; pad_
    dd 0xC3019640   ; vsrch_
    dd 0x80CB2000   ; srch_
    dd 0xE893C580   ; keypd_
    dd 0xBBE24000   ; wipe
    dd 0xBBE24800   ; wipes     was erase
    dd 0x91E29800   ; copy
    dd 0x8A8F4000   ; mark
    dd 0x48E22980   ; empty
    dd 0x48B90000   ; emit
    dd 0x29E24000   ; type   type_
    dd 0xC0F57200   ; digit
    dd 0xD4917200   ; 2emit
    dd 0xEA000000   ; .
    dd 0xC9D75000   ; h.2    dotHex2_
    dd 0xC9D76000   ; h.4    dotHex4_
    dd 0xC9D40000   ; h.     dotHex8_
    dd 0xC9D58000   ; h.n
    dd 0x90800000   ; cr
    dd 0x86259200   ; space
    dd 0xC0776000   ; down
    dd 0x4C0E4000   ; edit
    dd 0x40000000   ; e
    dd 0xA4400000   ; lm
    dd 0x18800000   ; rm
    dd 0xA8AE2C80   ; graph
    dd 0x24CA4000   ; text
    dd 0xE893C4A0   ; keypa(d)  displayTheKeypad_  was 0xE893C660 keybo(ard)
    dd 0xC098F300   ; debu(g)
    dd 0x52000000   ; at
    dd 0xF6A40000   ; +at
    dd 0xCB300000   ; xy
    dd 0xC4B54000   ; page
    dd 0x84851180   ; screen
    dd 0xB1E10000   ; fov
;     dd 0xB3D8C000   ; fifo
    dd 0xC6794000   ; box
    dd 0xA3B20000   ; line
    dd 0x91D0C400   ; color     set_color_
    dd 0x3912B100   ; octant
    dd 0x86200000   ; sp
    dd 0xA2C08000   ; last
    dd 0xCCD89640   ; unpac(k)
    dd 0xC4B2E800   ; pack
    dd 0xC74E8000   ; blk
    dd 0x8485AE00   ; scrnw   screen width in pixels
    dd 0x8485B200   ; scrnh   screen height in pixels
    dd 0xC78B1000   ; bpp     bits per pixel
    dd 0xB1B10000   ; font    address of font pointer, containing by default font16x24
    dd 0x791B5C00   ; iconw   icon width in pixels
    dd 0x791B6400   ; iconh   icon height in pixels
    dd 0x91E66240   ; counte(r) counter
    dd 0x8C000000   ; ms      ms_
    dd 0x36482480   ; onesec  onesec_
    dd 0xE993B000   ; khz     khz_
    dd 0x1297C000   ; rtc@    rtc_fetch_
    dd 0x1297A000   ; rtc!    rtc_store_
    dd 0x92D25D00   ; calck   calck_
    dd 0xC2820000   ; ver
```

```
    dd 0x96618000   ; curs
    dd 0xC7439740   ; block_    block number to address
    dd 0x5D58F9D0   ; a2blk_    address to block number
    dd 0xC36158A0   ; vframe  video frame address, where we create the image to be displayed
    dd 0xC2A30000   ; vars
; new words
    dd 0x82263000   ; seeb     ( see blue words, toggle )
    dd 0x812CBA40   ; stacks_
    dd 0xC0650B00   ; dotsf     type a ShannonFano token
    dd 0xA22E1400   ; leave
;    dd 0x12312310   ; txtq
    dd 0x1AE30000   ; rgb
    dd 0xC7340000   ; bye
    dd 0xB98E0000   ; word
    dd 0x4E840000   ; ekt
    dd 0x5C662400   ; abort_
    dd 0x27974C80   ; tickh  HERE variable address
    dd 0xC79AD640   ; buffe(r)  buffer_
    dd 0x3B5A0840   ; offset
    dd 0x27900000   ; tic   tic_
    dd 0xC2905000   ; vesa
    dd 0xC2905880   ; vesam
    dd 0x21586400   ; trash trash_
;    dd 0xC90C3840   ; hsvv_
    dd 0xC3731C00   ; vword
;    dd 0xC2295800   ; vregs
    dd 0x7C292000   ; ivec
    dd 0x14863000   ; resb  restore_BIOS_idt_and_pic
    dd 0xC4F20000   ; pic
    dd 0xC0B88000   ; dap
    dd 0x82488000   ; sect
    dd 0xB98E0800   ; words
    dd 0xE8930000   ; key
    dd 0xCFD12600   ; qkey
    dd 0xC0F57600   ; digin
    dd 0xCF741200   ; qwert
    dd 0x1FE00000   ; r?
    dd 0x6CD40000   ; nul
    dd 0x92E00000   ; cad
    dd 0xC525C000   ; pcad
    dd 0xC0F0C540   ; displ(ay)
    dd 0x59148000   ; actc
    dd 0xF7478100   ; +list
    dd 0x72797400   ; itick
    dd 0xA3C00000   ; lisl
    dd 0xF6800000   ; +e
    dd 0x820E1D20   ; serv1     serv1_
    dd 0x780E1D20   ; isrv1_    initserv1_
    dd 0x820E1D40   ; serv2     serv2_
    dd 0x780E1D40   ; isrv2_    initserv2_
    dd 0x4C0E4A00   ; edita     editAddress_
    dd 0x963A3B60   ; cblind

    dd 0x97C00000   ; c@        cFetch_
    dd 0xBFC00000   ; w@        wFetch_
    dd 0xF8000000   ; @         fetch_          no longer replaced by optimising verson in block 68
    ; dd 0xB214B200   ; fetch     fetch_          was replaced by optimising verson in block 68

    dd 0xD5F00000   ; 2@        two_fetch_      no longer replaced by optimising verson in block 68

    dd 0x97A00000   ; c!        cStore_
    dd 0xBFA00000   ; w!        wStore_
    dd 0xF4000000   ; !         store_          no longer replaced by optimising verson in block 68
    dd 0xF7E80000   ; +!        plus_store_
    dd 0xD5E80000   ; 2!        two_store_      no longer replaced by optimising verson in block 68

    dd 0xC0C95000   ; dneg      d_negate_
    dd 0xC1EC0000   ; d+        d_plus_
    dd 0xC1CC0000   ; d-        d_minus_
    dd 0xD5833620   ; 2dup      two_dup_
```

```
    dd 0xD5804F10   ; 2drop     two_drop_        bug fix from Marco Nicola
    dd 0xD50BAE20   ; 2swap     two_swap_
    dd 0xD4785040   ; 2over     two_over_
    dd 0x13200000   ; rot       rot_
    dd 0xE6264000   ; -rot      minus_rot_
    dd 0x2CD2E800   ; tuck      tuck_
    dd 0xC4F2E800   ; pick      pick_

    dd 0x92528000   ; cell      cell_
    dd 0x92529CC0   ; cell-     cell_minus_
    dd 0x92529EC0   ; cell+     cell_plus_
    dd 0x92529000   ; cells     cells_

    dd 0xA6200000   ; lp        lp_

    dd 0xA3E02000   ; lidt      lidt_
    dd 0x83E02000   ; sidt      sidt_

    dd 0xD5DC0000   ; 2/        two_slash_
    dd 0xCDABB800   ; u2/       u_two_slash_
    dd 0x18647B10   ; rshift    rshift_
    dd 0xCDABB800   ; lshif     lshift_
    dd 0xFBDC0000   ; */        star_slash_
    dd 0xCDF7B800   ; u*/       u_star_slash_
    dd 0xeF13C000   ; /mod      slash_mod_
    dd 0xFBDE2780   ; */mod     star_slash_mod_
    dd 0x944F0A00   ; cmove_    cmove
    dd 0xD5F40000   ; 2*        two_star_
    dd 0xD5F7E800   ; 2**       two_star_star_
;    dd 0xD5DC0000  ; u/        u/_
    dd 0x962CCF80   ; cpuid     GetCPUID_
    dd 0x1C050900   ; rdtsc
    dd 0x156C0000   ; rand      rand_
    dd 0x156C1DC0   ; rand/     randInit_
    dd 0x156C19C0   ; randq     randq_
    dd 0x90CB5EA0   ; crc32     crc32_
    dd 0x8E0DA000   ; md5       md5_
    dd 0x8E0DB8C0   ; md5b      md5buf_

;    dd 0xB18C5480   ; format
;    dd 0xC5270000   ; pci
;    dd 0x68248000   ; nsec       was devic(e)
    dd 0x85DCA590   ; switch    switch
    dd 0xB0A27640   ; freeze    freeze
    dd 0x23C40000   ; top       top_

    dd 0x8AC94000   ; maca      maca_
    dd 0x8AC98000   ; macn      macn_
    dd 0x8ACA8000   ; macl      macl_
    dd 0x8AC88000   ; mact      mact_

    dd 0xB1645000   ; ftha      ftha_
    dd 0xB1646000   ; fthn      fthn_
    dd 0xB164A000   ; fthl      fthl_
    dd 0xB1642000   ; ftht      ftht_

    dd 0xC74CCA00   ; blua      blua_
    dd 0xC74CCC00   ; blun      blun_
    dd 0xB63A6640   ; fblue     find_Blue_word_
    dd 0xA1C80000   ; loc       loc_
    dd 0x82200000   ; see       see_
    dd 0xB3600000   ; fnd       fnd_
    dd 0xC11C4000   ; dmp       dmp_
    dd 0x4CA92000   ; exec      execute_

    dd 0xCA000000   ; x         x_
    dd 0x98000000   ; y         y_
    dd 0xEC000000   ; z         z_
    dd 0xA63A7400   ; lblk      lblk_
```

```
    dd 0            ; terminating null at the end of the list

ForthJumpTableROM:  ; jumptable:
    dd boot         ;
    dd warm         ;
    dd pause_       ; pause
    dd macro_       ; macro     select the macto wordlist
    dd forth_       ; forth     select the forth wordlist
    dd c_           ; c
    dd Read_Sector_LBA   - $$ + BOOTOFFSET  ; jmp Read_Sector_LBA
    dd Write_Sector_LBA  - $$ + BOOTOFFSET  ; jmp Write_Sector_LBA
    dd ReadSectors  - $$ + BOOTOFFSET  ; jmp ReadSectors    reads
    dd WriteSectors - $$ + BOOTOFFSET  ; jmp WriteSectors   writes
    dd SaveAll_     - $$ + BOOTOFFSET  ; jmp SaveAll_
;    dd th_        - $$ + BOOTOFFSET ; jmp th_ ( thunk to BIOS Int 0x13 )
    dd bios_read  - $$ + BOOTOFFSET ; jmp bios_read   'read'
    dd bios_write - $$ + BOOTOFFSET ; jmp bios_write  'write'
;    dd XXXrsect_  - $$ + BOOTOFFSET ; jmp rsect_  'rsect'
    dd fetchDX_     ; @dx
    dd setupDAP_    ; !dap
    dd activate     ; act
    dd show         ;
    dd _load_       ;
    dd nload        ; nload
    dd plusLoad     ; +load
    dd thru_        ; thru
    dd plusThru_    ; +thru
    dd cblk_        ;           return the block number currently being compiled, calculated from  edi
    dd rblk_        ;           return the block number offset of the RELOCATED address
    dd ablk_        ;           convert byte address to block number
    dd erase_       ;
    dd here_        ; here      returns the current dictionary pointer
    dd qlit         ; ?lit
    dd comma3_      ; 3,
    dd comma2_      ; 2,
    dd comma1_      ; 1,
    dd comma1_      ; c,
    dd comma_       ; ,
    dd less         ; less
    dd jump         ; jump
    dd quit_        ; quit
    dd pad_         ; pad
    dd vsrch_       ; vsrch
    dd srch_        ; srch
    dd keypd_       ; keypd  ( alias of pad )
    dd wipe         ;
    dd wipes        ;
    dd copy_        ; copy
    dd mark         ;
    dd empty_       ; empty
    dd emit_        ; emit
    dd type_        ; type
    dd digit        ;
    dd two_emit     ; 2emit
    dd dotDecimal   ; .
    dd dotHex2_     ; h.2
    dd dotHex4_     ; h.4
    dd dotHex8_     ; h.
    dd h_dot_n      ; h.n
    dd cr_          ; cr
    dd space_       ; space
    dd down         ;
    dd edit_        ;
    dd e_           ; e
    dd lm           ;
    dd rm           ;
    dd graphAction  ; graph
    dd setupText_   ; text
    dd displayTheKeypad_ ; keypa (d)
```

```
    dd debug        ;
    dd _at          ; at
    dd plus_at      ; +at
    dd xy_          ;
    dd page_        ; page
    dd screen_      ; screen
    dd fov_         ;
;    dd fifo         ;
    dd box_         ; box
    dd line_        ; line
    dd set_color_    ; color
    dd octant       ;
    dd tokenActions_ ; tokenActions table
    dd last         ;
    dd unpack       ;
    dd pack_        ;
    dd blk_         ;
    dd scrnw_       ; scrnw  screen width in pixels
    dd scrnh_       ; scrnh  screen height in pixels
    dd bpp_         ; bpp    bits per pixel
    dd font_        ; font   address of font pointer, containing by default font16x24
    dd iconw_       ; iconw  icon width in pixels
    dd iconh_       ; iconh  icon height in pixels
    dd counter_     ; counter
    dd ms_          ; ms
    dd onesec_      ; onesec
    dd khz_         ; khz
    dd rtc_fetch_   ; rtc@
    dd rtc_store_   ; rtc!
    dd calck_       ; calclk  calibrate the clock for ms
    dd version_     ; ver
    dd curs         ; curs
    dd block_       ; block
    dd a2blk_       ; a2blk
    dd vframe_      ; vframe
    dd vars_        ; vars
; new words
    dd seeb         ; seeb
    dd stacks_      ;
    dd dotsf_       ; dotsf
    dd leave_       ; leave
;    dd txtq_        ;
    dd rgb          ; rgb
    dd bye_         ; bye
    dd _word        ;
    dd ekt          ;
    dd abort_       ;
    dd tickh        ;
    dd buffer_      ; buffe(r)
    dd offset_      ;
    dd tic_         ; tic
    dd vesa         ;
    dd vesamode_    ;
    dd trash_       ; trash
;    dd hsvv_        ; hsvv
    dd vword_       ; ('%s')", DB_NAME,
;    dd vregs_       ; vregs
    dd ivec_         ; ivec
    dd restore_BIOS_idt_and_pic ; resb
    dd pic_         ; pic  Programmable Interrupt Controller settings, as set by the BIOS
    dd dap_         ; dap
    dd sect_        ; sect
    dd words_       ; words
    dd get_key_     ; key
    dd get_qwerty_key_ ; qkey
    dd digin        ;
    dd qwert        ;
    dd rquery       ; r?
    dd nul          ;
    dd cad          ;
```

```
    dd pcad         ;
    dd displ        ;
    dd actc         ;
    dd plusList     ; +list
    dd itick        ;
    dd refresh      ; lis
    dd plus_e       ; +e
    dd serv1_       ; serv1
    dd initserv1_   ; isrv1_
    dd serv2_       ; serv2
    dd initserv2_   ; isrv1_
    dd editAddress_ ; edita
    dd cBlindAddr_  ; cblind

    dd cFetch_      ; c@
    dd wFetch_      ; w@
    dd fetch_       ; @     was replaced by optimising verson in block 68
    dd two_fetch_   ; 2@    was replaced by optimising verson in block 68
    dd cStore_      ; c!
    dd wStore_      ; w!
    dd store_       ; !     was replaced by optimising verson in block 68
    dd plus_store_  ; +!
    dd two_store_   ; 2!    was replaced by optimising verson in block 68

    dd d_negate_    ; dneg
    dd d_plus_      ; d+
    dd d_minus_     ; d-
    dd two_dup_     ; 2dup
    dd two_drop_    ; 2drop
    dd two_swap_    ; 2swap
    dd two_over_    ; 2over
    dd rot_         ; rot
    dd minus_rot_   ; -rot
    dd tuck_        ; tuck
    dd pick_        ; pick

    dd cell_        ; cell
    dd cell_minus_  ; cell-
    dd cell_plus_   ; cell+
    dd cells_       ; cells
    dd lp_          ; lp
    dd lidt_        ; lidt
    dd sidt_        ; sidt
    dd two_slash_   ; 2/
    dd u_two_slash_ ; u2/
    dd rshift_      ; rshift
    dd lshift_      ; lshif     lshift
    dd star_slash_  ; */
    dd u_star_slash_; u*/
    dd slash_mod_   ; /mod
    dd star_slash_mod_ ; */mod
    dd cmove_       ; cmove
    dd two_star_    ; 2*
    dd two_star_star_ ; 2**   _
;    dd u/_          ; u/

    dd GetCPUID_    ; cpuid
    dd rdtsc_       ; rdtsc
    dd rand_        ; rand
    dd randInit_    ; rand/
    dd randq_       ; randq
    dd crc32_       ; crc32
    dd md5_         ; md5
    dd md5buf_      ; md5b      md5buf_

;    dd format       ;
;    dd pci          ;
;    dd device       ;
    dd switch       ;
    dd freeze       ;
```

```
    dd top_         ;

    dd maca_        ; maca
    dd macn_        ; macn
    dd macl_        ; macl
    dd mact_        ; mact

    dd ftha_        ; ftha
    dd fthn_        ; fthn
    dd fthl_        ; fthl
    dd ftht_        ; ftht

    dd blua_        ; blua
    dd blun_        ; blun
    dd find_Blue_word_ ; fblue
    dd loc_         ; loc
    dd see_         ; see
    dd fnd_         ; fnd
    dd dmp_         ; dmp
    dd execute_     ; exec

    dd x_           ; x
    dd y_           ; y
    dd z_           ; z
    dd lblk_        ; lblk

ForthJumpTableROM_end:

; times 200 NOP   ;  enable this line to see how much space is left. If NASM reports :
; "cf2022.nasm:6282: error: TIMES value -28 is negative" with "times 200" you have (200 - 28) bytes left

; fill with no-ops to 55AA at end of boot sector, less $40 for the info string
times ( ( START_BLOCK_NUMBER - SIZE_OF_FONT_IN_BLOCKS ) * 0x400 ) - ($ - $$)  NOP

; the above produces a 26K boot image, we then add the 6K font and colorForth source blocks:
font16x24:
; incbin "cf2023_font.img"
; incbin "cf2023Ref.img",( OFFSET_OF_FONT +(SIZE_OF_FONT_IN_BLOCKS *1024) ), ( 512 * 1024 ) ; append the
font and colorForth source blocks from the reference image, skip the kernel code
; colorForth:                         ; the colorForth source blocks
incbin "cf2023Ref.img", OFFSET_OF_FONT, ( ( 512 - START_BLOCK_NUMBER + SIZE_OF_FONT_IN_BLOCKS ) * 1024 );
append the font and colorForth source blocks from the reference image, skip the kernel code
times 32768 db 'U'

; end of file
```

Appendix C colorForth Source Code

```
\ C:\Users\User\Dropbox\colorForth\cf2023\cf2023Ref.img converted by colorForthScan V1.0 2023 Jan 29
\ File MD5 = 649C8F306CE629E514E5A0812BF1C912  roomed-zebra

\ MagentaV  is the colorForth Magenta Variable
: MagentaV ( initial -- )   create , ; \ Runtime: ( -- a )

\ Block 64
( colorforth cf2023 2023 Apr 04 )

( processor clock ) #0 MagentaV mhz

( dump ) #5952 MagentaV x #0 MagentaV y ( ld ) #64 MagentaV lblk

#2 #12 +thru
: dump #78 load ;
: words #112 ld ;
: icons #80 ld ;
: serve #506 ld ;
: north #92 ld ;
: rtc #96 ld ;
: lan #98 ld ;
: colors #102 ld ;
: wood #106 ld ;
: mand #108 ld ;
: sound #114 ld ;
: gr #118 ld ;
: eth #176 ld ;
: life #272 ld ;
: ed #252 ld ;
: slime #246 ld ;
: int #288 ld ;
: xx #278 load ;
: info ver dump ;
: staks #504 ld ;

( hardware ) #0 MagentaV rng
: chm ( -- ) #0 mhz ! md5b x ! #0 y ! #64 lblk ! $00010000
: ch ( n-- ) #64 block swap md5 dump ;
: hlp randq rng ! logo pause calkhz
   onesec @ #1000 / mhz ! e ;

mark empty hlp

( Press the * key to see the comment block )

( Press F1 )

\ Block 65
\ ( Based on colorforth 2001 Jul 31 by Chuck Moore )

\ ( released into the Public Domain. )

\ ( This block is loaded at power up. Press F1 for help )
\ : dump ( instant compile version of DUMP )
\ : icons ( edit the character font icons )
\ : north ( North Bridge PCI chip display )
\ : rtc ( Real Time Clock display )
\ : colors ( 3-axis rgb colour display )
\ : wood ( imitation pine blockboard )
\ : mand ( display the Mandeldrot set )
\ : sound ( control the PC speaker )
\ : gr ( graphics - type ok to run the demo )
\ : life ( Conways game of life )
\ : ed ( the editor partly converted to colorforth )
\ : slime ( watch out for the slugs! )
\ : int ( 1000 Hz timer interrupt )
\ : xx ( colorforth explorer )
\ : ch ( show MD5 of n bytes starting at block 64 )
\ : chm ( show MD5 of system blocks )
```

```
\ : help ( press the space bar to leave the editor, then type the keys indicated in the keypad in the bottom right of the
\ screen, then the space bar to execute the word. Type ) e ( or ) #64 edit ( to run the editor. )

\ info ( to view the boot system version )

\ seeb ( to toggle display of blue words )

\ hlp ( shows help and clock speed )

\ Block 66
macro
: ?f $C021 2, ;
: 0if $75 2, here ;
: +if $78 2, here ;
: 1+ ( n-n ) $40 1, ;
: 1- ( n-n ) $48 1, ;
: 2/ ( n-n ) $F8D1 2, ;
: time ( -u ) qdup $310F 2, ;
: shl ( uc-u ) ?lit $E0C1 2, 1, ;
: shr ( uc-u ) ?lit $E8C1 2, 1, ;
: r@ qdup $8B 1, $C7 1, ;
: sti $FB 1, ; ( enable interrupts )
: cli $FA 1, ; ( disable interrupts )

forth
: cli cli ;
: sti sti ;
: nul ;
: time time ;

\ Block 67
\ ( Pentium macros: )
\ : ?f ( set flags to reflect tos )
\ : 0if ( if zero ... then jnz aids in clarity )
\ : +if ( js, this complements the set )
\ : 1- ( subtract 1 )
\ : 2/ ( divide by 2 )
\ : qdup ( is the new name for ?dup )
\ : time ( return Pentium instruction counter )
\ : lshift ( shift u left c places )
\ : rshift ( shift u right c places )
\ : r@ ( copies the top of the return stack to TOS )
\ : sti ( enable device interrupts )
\ : cli ( disable them )
\ : a,

\ Block 68
( more macros ) macro
: swap $168B 2, $C28B0689 , ;
: 0 qdup $C031 2, ;
: if $74 2, here ;
: -if $79 2, here ;
: a qdup $C28B 2, ;
: a! ?lit if $BA 1, , ; then $D08B 2, drop ;
: 1@ $8A 2, ;
: 1! a! $0288 2, drop ;
: p@ ( a-n ) qdup a! $EC 1, ;
: p! ( na- ) a! $EE 1, drop ;
: 2* $E0D1 2, ;
: a, , ;
: @@ ?lit if qdup $058B 2, , ; then $8B 2, 0 , ;
: !! ?lit if ?lit if $05C7 2, swap , , ; then $0589 2, , drop ; then a! $0289 2, 0 , drop ;
: nip $0004768D 3, ;
: + ?lit if $05 1, , ; then $0603 2, nip ;
: xor $0633
: binary ?lit if swap #2 + 1, , ; then 2, nip ;
: and $0623 binary ;
: or $060B binary ;
: u+ ?lit if $0681 2, , ; then $00044601 3, drop ;
: ? ?lit $A9 1, , ;
```

```
\ Block 69
\ ( Pentium macros: 1, 2, 3, , compile 1-4 bytes )
\ : drop ( lodsd, flags unchanged, why sp is in ESI )
\ : over ( sp 4 + @ )
\ : swap ( sp xchg )
\ : 0 ( 0 0 xor, macro 0 identical to number 0 )
\ : a ( 2 0 mov, never used? )
\ : a! ( 0 2 mov, unoptimized )
\ : 1@ ( fetch byte from byte address )
\ : 1! ( store byte to byte address )
\ : p@ p-n ( fetch byte from port )
\ : p! np ( store byte to port )
\ : @ ( EAX 4 *, unoptimized )
\ : ! ( EDX 4 * )
\ : nop ( used to thwart look-back optimization )
\ : - ( ones-complement )
\ : 2*
\ : 2/
\ : if ( jz, flags set, max 127 bytes, leave address )
\ : -if ( jns, same )
\ : then ( fix address - in kernel )
\ : push ( EAX push )
\ : pop ( EAX pop )
\ : u+ ( add to 2nd number, literal or value )
\ : ? ( test bits, set flags, literal only! )

\ Block 70
( even more macros )
: over qdup $0004468B 3, ;
: push $50 1, drop ;
: pop qdup $58 1, ;
: invert ( n-n ) $D0F7 2, ;
: for push begin ;
: *next swap
: next $75240CFF
: 0next , here invert + 1, $0004C483 3, ;
: -next $79240CFF 0next ;
: i qdup $0024048B 3, ;
: *end swap
: end $EB 1, here invert + 1, ;
: +! ?lit if ?lit if $0581 2, swap , , ; then $0501 2, , drop ; then a! $0201 2, drop ;
: nop $90 1, ;
: align here invert #3 and drop if nop align ; then ;
: or! a! $00950409 3, 0 , drop ;
: * $0006AF0F 3, nip ;
: */ $C88B 2, drop $F9F72EF7 , nip ;
: /mod swap $99 1, $16893EF7 , ;
: / /mod nip ;
: mod /mod drop ;

\ Block 71
\
\ : - n-n ( ones complement negate , xor )
\ : for n ( push count onto return stack, falls into ) begin
\ : begin -a ( current code address - byte )
\ : *next aa-aa ( swap ) for ( and ) if ( addresses )
\ : next a ( decrement count, jnz to ) for, ( pop return stack when done )
\ : -next a ( same, jns - loop includes 0 )
\ : i -n ( copy loop index to data stack )
\ : end a ( jmp to ) begin
\ : +! na ( add to memory, 2 literals optimized )
\ : align ( next call to end on word boundary )
\ : or! na ( inclusive-or to memory, unoptimized )
\ : * mm-p ( 32-bit product )
\ : */ mnd-q ( 64-bit product, then quotient )
\ : /mod nd-rq ( remainder and quotient )
\ : / nd-q ( quotient )
\ : mod nd-r ( remainder )
\ : time -n ( Pentium cycle counter, calibrate to get actual clock rate )

\ Block 72
```

```
( Compiled macros ) forth
: r@ ( -n ) r@ ;
: @@ ( a-n ) @ ;
: !! ( an- ) ! ;

( Arithmetic )
: + ( nn-n ) + ;
: 1+ ( u--u ) 1+ ;
: 1- ( u--u ) 1- ;
: invert ( n-n ) invert ;
: */ ( nnn-n ) */ ;
: * ( nn-n ) * ;
: / ( nn-n ) / ;
: 2* ( n-n ) 2* ;
: 2/ ( n-n ) 2/ ;
: drop ( n- ) drop ;
: dup ( n-nn ) dup ;
: swap ( nn-nn ) swap ;
: over over ;
: negate ( n-n ) invert #1 + ;
: - ( nn-n ) negate + ;
: min ( nn-n ) less if drop ; then swap drop ;
: abs ( n-u ) dup negate
: max ( nn-n ) less if swap then drop ;
: v+ ( vv-v ) push u+ pop + ;
: save sss ;
: sa sss e ;

\ Block 73
\ ( These macros may be ) yellow, ( others may not )
\ : block n-a ( block number to word address )
\ : r@ ( copies the top of the return stack to stack )
\ : @ etc ( Arithmetic )
\ : negate n-n ( when you just cant use ) -
\ : min nn-n ( minimum )
\ : abs n-u ( absolute value )
\ : max nn-n ( maximum )
\ : v+ vv-v ( add 2-vectors )
\ : save ( write colorforth to a bootable USB drive )
\ : sa ( save, then show edit screen )

\ Block 74
( Relative load blocks )
: ll ( -- ) blk @ load ;
: sect ( --asn ) blk @ block blk @ 2* #2 ;
: ss ( -- ) sect writes drop drop ;
: uu ( -- ) sect reads drop drop ;


: ld ( n- ) dup lblk ! load ;
: vv ( -- ) lblk @ edit ;
: help ( -- ) lblk @ #1 + edit ;

( Real Time Clock )
: rtc@ ( t-c ) $70 p! $71 p@ ;
: rtc! ( ct- ) $70 p! $71 p! ;
: hi ( -- ) #10 rtc@ $80 and drop 0if hi ; then ;
: lo ( -- ) #10 rtc@ $80 and drop if lo ; then ;


: calkhz ( -- ) hi lo counter hi lo counter swap -
   dup onesec ! #1 rshift #250 + #500 / dup khz ! ;
: ms ( n- ) khz @ * counter + begin pause dup counter
   invert + drop -if drop ; then end drop ;
: secs ( n- ) for pause lo hi next ; macro
: swapb ( w-w ) $E086 2, ; forth
: split ( w--cc ) dup swapb $FF and swap $FF and ;

\ Block 75
\
\ : nload ( loads the next source block : b+2 )
```

```
\ : +load ( loads the source block : b+n )
\ : blk ( where the current blk happens to be kept )
\ : ll ( load the current edit blk )
\ : ss ( save the sector containing the current edit block to the floppy disc )
\ : lblk ( holds the last block loaded by )
\ : ld
\ : vv ( edits the last block loaded by ld )
\ : rtc@ reg-n ( fetch reg from rtc )
\ : rtc! n reg- ( store in rtc register )
\ : hi ( wait till Update In Progress bit is high )
\ : lo ( wait till UIP bit is low )
\ : calkhz ( calibrate the processor clock using the RTC )
\ : ms ( wait for n milliseconds )
\ : secs ( wait for n seconds )
\ : swapb ( swap the two low bytes )
\ : split ( split the low two bytes )
\ : vframe ( byte address of the video frame buffer )

\ Block 76
( Colors etc )
: white $00FFFFFF rgb color ; : red $00FF0000 rgb color ;
: green $FF00 rgb color ; : blue $FF rgb color ;
: silver $00BFBFBF rgb color ; : yellow $FFE0 color ;
: orange $00E04000 rgb color ; : black $00 rgb color ;
: 5* #5 for 2emit next ;
: cf #25 dup at red $72 $6F $6C $6F $63 5* green $68 $74 $72 $6F $46 5* ;
: logo show black screen #800 #710 blue box #600 #50 at #1024 #620 red box #200 #100 at #700 #500 green box text cf keypa
d ;
: noshow show keypad ;
: lshift ( uc-u ) $1F and ?f 0if drop ; then for #1 shl next ;
: rshift ( uc-u ) $1F and ?f 0if drop ; then for #1 shr next ;
: rand32 ( -n ) time dup #16 lshift xor ;
: string pop ;
: 1@ ( a-c ) 1@ $0F and ;
: 1! ( ac- ) 1! ;

\ Block 77
\
\ : colors ( specified as rgb: 888 )
\ : screen ( fills screen with current color )
\ : at xy ( set current screen position )
\ : box xy ( lower-right of colored rectangle )
\ : 5* ( displays 5 large characters )
\ : cf ( displays ) colorforth
\ : logo ( displays colorforth logo )
\ : empty ( also displays the logo )
\ : lshift ( shift u left c places )
\ : rshift ( shift u right c places )
\ : show ( background task executes following code repeatedly )
\ : keypad ( displays keypad and stack )
\ : string ( returns the address of the string following )
\ : rand32 ( returns a 3 bit random number )

\ Block 78
( Dump names )
: .cell ( a-a ) orange dup @ #4 for dup $FF and emit $0100 / next drop white ;
: one dup dup @ dup push h. space dup h. pop space swap .cell drop space space space space dup dotsf drop white cr ;
: lines for one #4 + next drop ;
: dump ( a- ) $0FFFFFFC and x !
: r show black screen x @ #16 text lines cr x @ #16 for .cell #4 + next drop keypad ;
: it @ + @ dup h. space ;
: lines for white i x it i y it xor drop if red then i . cr -next ;
: cmp show blue screen text #19 lines red x @ h. space y @ h. keypad ;
: u $40
: +xy dup x +! y +! ;
: d $FFFFFFC0 +xy ;
: ati $F4100000 ( ff7fc000 ) xor
: byte #4 / dump ;
: fix for #0 over ! #1 + next ; dump

\ Block 79
```

```
\ ( Does not say empty, compiles on top of application )
\ : x -a ( current address )
\ : one a-a ( line of display )
\ : lines an
\ : dump a ( background task continually displays memory : decodes the value as a name and ASCII )
\ : u ( increment address )
\ : d ( decrement )
\ : ati ( address of AGP graphic registers )
\ : byte a ( byte address dump )
\ : fix an-a ( test word )
\ : ver ( show the kernel version information )
\ : cmp ( shows data at both ) x ( and ) y ( addresses )

\ Block 80
( App: Icons font editor ) empty

( icon number ) #28 MagentaV ic ( cursor ) #351 MagentaV cu

macro : @w $8B66 3, ; : !w a! $00028966 3, drop ;
: *byte $C486 2, ; forth


: sq xy @ $00010000 /mod #16 + swap #16 + box #17 #0 +at ;
: loc ic @ $FF and
: tofont ( n--a ) #16 #24 #8 */ * font @ + ;
: 0/1 $8000 ? if green sq ; then blue sq ;
: row dup @w *byte #16 for 0/1 2* next drop #-17 #16 * #17 +at ;
: cpl #32 ;
: showall ( -- ) #2 lm iconw cpl * rm ic @ cpl /mod iconh * #448 #2 - + swap iconw * swap over over at red #16 #4 + u+ #24
+ #4 + box white
   #0 #2 #448 at #256 for dup emit #1 + next drop ;
: ikon loc #24 for row #2 + next drop ;
: adj #17 * swap ;
: cursor cu @ #16 /mod adj adj over over at red #52 u+ #52 + box ;
: ok show page cursor #18 dup at ikon text blue #400 #400 at    ef #416 #424 box #400 #400 at white    ef ic @ dup emit space
dup green . $30 emit $78 emit #2 h.n showall keypad ;


: fcopy tofont swap tofont swap #16 #24 #8 */ cmove ;

nload ok h

\ Block 81
\ ( Draw big-bits icon )
\ : @w a-n ( fetch 16-bit word from byte address )
\ : !w na ( store same )
\ : *byte n-n ( swap bytes )
\ : ic -a ( current icon )
\ : cu -a ( cursor )
\ : sq ( draw small square )
\ : xy -a ( current screen position, set by ) at
\ : loc -a ( location of current icons bit-map )
\ : 0/1 n-n ( color square depending on bit 15 )
\ : row a-a ( draw row of icon )
\ : +at nn ( relative change to screen position )
\ : ikon ( draw big-bits icon )
\ : adj nn-nn ( magnify cursor position )
\ : cursor ( draw red box for cursor )
\ : ok ( background task to continually draw icon, icon number at bottom )

\ Block 82
( Edit icon )


: icmv ( n-- ) ic @ + $FF and ic ! ;
: +ic #1 icmv ;
: -ic #-1 icmv ;
: ++ic cpl icmv ;
: --ic cpl negate icmv ;
: bit cu @ 2/ 2/ 2/ 2/ 2* loc + $00010000 cu @ $0F and #1 + for 2/ next *byte ;
: toggle bit over @w xor swap !w ;
```

```
: td toggle
: d #16
: wrap cu @ + #16 #24 * dup u+ /mod drop cu ! ;
: tu toggle
: u #-16 wrap ;
: tr toggle
: r #1 wrap ;
: tl toggle
: l #-1 wrap ;
: nul ;
: h keypd

nul nul quit nul        tl tu td tr

l u d r             -ic --ic ++ic +ic

nul nul nul nul         nul nul nul toggle
   nul nul nul nul

$2500 , $13121110 dup , , $2B16152D , #0 , $80000000 , #0 ,

\ Block 83
\ ( Edit icon )
\ : t ( toggles the current pixel )
\ : ludr ( left up down right )
\ : . ( top row toggles and moves )
\ : -+ ( select icon to edit )

\ Block 84
( Print PNG to disk ) #1024 MagentaV w #768 MagentaV h #1 MagentaV d

#6 +load #4 +load #2 +load
: -crc ( a ) here over negate + crc . ;
: crc -crc ;
: wd ( -a ) here #3 and drop if #0 1, wd ; then here #2 2/s ;
: bys ( n-a ) . here swap , ;
: plte $45544C50 #48 bys $00 3, $00FF0000 3, $FF00 3, $00FFFF00 3, $FF 3, $00FF00FF 3, $FFFF 3, $00FFFFFF 3, $00 3,
$00C00000
3, $C000 3, $00C0C000 3, $C0 3, $00C000C0 3, $C0C0 3, $00C0C0C0 3, crc ;
: png ( awh ) d @ / h ! d @ / w ! wd swap $474E5089 , $0A1A0A0D , ( ihdr ) $52444849 #13 bys w @ . h @ . $0304 , $00 1, crc
plte ( idat ) $54414449 #0 bys swap deflate crc ( iend ) $444E4549 #0 bys crc wd over negate + ;
: at #1024 * + 2* vframe + ;
: full #4 d ! #0 dup at #1024 #768 png ;
: pad #1 d ! #46 #-9 + #22 * nop #25 #-4 + #30 * at #9 #22 * nop #4 #30 * png ;
: go #1 d ! #1024 w ! #768 h ! #0 #0 at #1024 #768 png raw ; go e

\ Block 85
\ ( Print PNG to disk )
\ : frame ( the video frame buffer )
\ : -crc ( a )
\ : crc
\ : wd ( -a )
\ : bys ( n-a )
\ : plte
\ : png ( awh )
\ : at
\ : full
\ : pad
\ : go ( copy the screen image as a PNG file to the floppy disk block 270 and up. )

\ Block 86
( lz77 ) macro
: @w $8B66 3, ;
: *byte $C486 2, ;
: !b a! $0289 2, drop ; forth
: *bys dup #16 2/s *byte swap $FFFF and *byte $00010000 * + ;
: . *bys , ;
: +or over invert and or ;
: 0/1 $10 ? if $1E and $1E or drop if #7 ; then $0F ; then #0 and ;
```

```
: 4b dup 0/1 #9 and over #6 2/s 0/1 $0A and +or swap #11 2/s 0/1 $0C and +or $08 or ;
: pix dup @w d @ 2* u+ 4b ;
: row 1, dup w @ 2/ dup #1 + dup 2, invert 2, #0 dup 1, +adl for pix #16 * push pix pop or dup 1, +adl next drop +mod d @
#1024 #2 * * + ;
: deflate $0178 2, #1 #0 adl! h @ #-1 + for #0 row next #1 row drop ad2 @ *byte 2, ad1 @ *byte 2, here over #4 + negate +
*bys over #-4 + !b ;

\ Block 88
( Crc ) macro
: 2/s ?lit $E8C1 2, 1, ;
: 1@ $8A 2, ; forth #36054 MagentaV ad1 #54347 MagentaV ad2
: array ( -a ) pop #2 2/s ;
: bit ( n-n ) #1 ? if #1 2/s $EDB88320 or ; then #1 2/s ;
: fill ( nn ) for dup #8 for bit next , #1 + next drop ;
: table ( -a ) align array #0 #256 fill
: crc ( an-n ) #-1 swap for over 1@ over or $FF and table + @ swap #8 2/s or #1 u+ next invert nip ;
: +adl ( n ) $FF and ad1 @ + dup ad2 @ +
: adl! ad2 ! ad1 ! ;
: +mod ad1 @ #65521 mod ad2 @ #65521 mod adl! ;

\ Block 90
( DOS file )
: blks #256 * ;
: w/c #18 blks ;
: buffer block ;
: size ( -a ) buffer #0 #1 reads buffer $098F + ;
: set ( n ) ! buffer     s #1 writes ;
: cyls ( n-nn ) #1 swap w/c #-1 + + w/c / ;
: put ( an ) dup 2* 2* size set cyls writes /flop ;
: raw ( an- ) #15 swap 2* 2* w/c #-1 + + w/c / writes /flop ;
: get ( a ) size @ #3 + 2/ 2/ cyls reads /flop ;
: .com #0 #63 blocks put ;

\ Block 91
\
\ : blks n-n ( size in blocks to words )
\ : w/c -n ( words per cylinder )
\ : buffer -a ( 1 cylinder required for floppy dma )
\ : size -a ( locate size of 2nd file. Floppy has first FILLER then FILE allocated. FILLER is 2048 bytes, to fill out cylind
\er 0. Names at most 8 letters, all caps. Directory starts at ) buffer $0980 +
\ : set n ( size. FILE must be larger than your file. )
\ : cyls n-nn ( starting cylinder 1 and number of cylinders )
\ : raw an ( write raw data to cyl 15 , block 270 )
\ : put an ( write file from address )
\ : get a ( read file to address )

\ Block 92
( App: North Bridge ) empty macro
: 4@ dup $ED 1, ;
: 4! $EF 1, drop ; forth #2048 MagentaV dev
: nb $00 dev ! ;
: sb $3800 dev ! ;
: agp $0800 dev ! ;
: ess $6800 dev ! ;
: ric $7800 dev ! ;
: win $8000 dev ! ;
: ati $00010000 dev ! ;
: add $0CF8 a! 4! $0CFC a! ;
: q $80000000 + add 4@ ;
: en $8004 q #-4 and xor 4! ;
: dv dup $0800 * q swap #1 + ;
: regs dev @ #19 #4 * + #20 for dup q h. space dup h. cr #-4 + next drop ;
: devs #0 #33 for dup q dup #1 + drop if dup h. space drop dup #8 + q dup h. space over h. cr then drop $0800 + next drop
;
: ok show black screen text regs keypad ;
: ko show black screen text devs keypad ;
: u $40 dev +! ;
: d #-64 dev +! ;
: test $FF00 + a! 4@ ; ok

\ Block 93
```

\ (Display the PCI interface chip registers)

\ Block 94
(ASCII)
: cf-ii string (0*00) $6F747200 , $696E6165 , $79636D73 , $7766676C , (0*10) $62707664 , $71757868 , $33323130 , $37363534 , (0*20) $2D6A3938 , $2F7A2E6B , $2B213A3B , $3F2C2A40 , (0*30) $4F545200 ,
: ch $FFFFFFF0 and unpack cf-ii + 1@ $FF and ;
: ii-cf string (0x20) $64632A00 , $7271706F , $2B2D6E6D , $2725232E , (0x30 3210) $1B1A1918 , (7654) $1F1E1D1C , (..98) $28292120 , $2F6C6B6A , (0x40 CBA@) $3A43352C , (GFED) $3D3E3440 , (KJIH) $54523744 , (ONML) $3336393C , (0x50
SRQP) $38314742 , (WVUT) $3F414632 , (.ZYX) $58563B45 , $75745973 , (0x60 cba.) $0A130576 , (gfed) $0D0E0410 , (kjih
) $24220714 , (onml) $0306090C , (0x70 srqp) $08011712 , (wvut) $0F111602 , (.zyx) $77260B15 , $62617879 ,
: chc $FFFFFFE0 + ii-cf + 1@ $FF and ;
: tst #2000 block dup #4 * #-1 + $60 for $01 + $80 i negate + over 1! next drop dump ; #51 MagentaV qch
: rr (c-c) qch ! $20 $60 for $01 + dup chc qch @ negate + drop 0if pop drop ; then next $7F and ;

\ Block 95
\ (Convert colorforth chars to and from ASCII)
\ : cf-ii (conversion table)
\ : ch (convert colorforth character to ASCII)
\ : ii-cf (conversion table)
\ : chc (convert ASCII to colorforth)
\ : tst (create a table of ASCII characters)
\ : r (scan the ii-cf table to perform cf-ii . Used to cross-reference the two tables)
\ : info (display the ASCII version information in the last 64 bytes of block 11 . Type u to see more .)
\   (dump takes a byte address)

\ Block 96
(App: RTC Real Time Clock) empty

: bcd (-c) rtc@ #16 /mod #10 * + ;
: hms (-n) lo #4 bcd #100 * #2 bcd + #100 * #0 bcd + ;   s
: ymd (-n) lo #9 bcd #2000 + #100 * #8 bcd + #100 * #7 bcd + ;
: day (-c) lo #6 bcd ;
: crlf (Port Dump)
: one (n-n) space yellow dup rtc@ h.2 blue space dup . cr ;
: lines (sn-) for one #-1 + next drop ;
: ok show page text cr #15 #16 lines white cr ymd .
   hms . day . keypad ;
: h

keypd nul nul quit   nul nul nul nul

nul nul nul nul   nul nul nul nul

nul nul nul nul   nul nul nul nul
   nul nul nul nul nul

$00250000 , #0 , #0 , #0 , #0 , #0 , #0 ,

ok

\ Block 97
\ (RTC Real Time Clock)
\ : . (displays the PC clock registers)
\ : bcd bcd-n (bcd to binary)
\ : hms -n (hours+mins+secs)
\ : ymd -n (year+month+day)
\ : day -n (day of the week)
\ : rtc (display the Real Time Clock registers)
\ : one (display one line)
\ : lines (display n lines starting at s)
\ : ok (display task)

\ Block 98
(LAN) empty $03F8 nload init
: no block #4 * #1024 ;
: send no for dup 1@ xmit #1 + next drop ;
: receive no for rcv over 1! #1 + next drop ;

```
: no #18 #7 #18 * ;
: backup no for dup send #1 + next drop ;
: accept no for dup receive #1 + next drop ;

\ Block 99
\

\ Block 100
( Serial 3f8 2e8 1050 ) macro
: 1@ $8A 2, ;
: 1! a! $0288 2, drop ; forth
: r #0 + + ;
: 9600 #12 ;
: 38400 #3 ;
: 115200 #1 ;
: b/s $83 #3 r p! 38400 #0 r p! #0 #1 r p! #3 #3 r p! ;
: init b/s ( 16550 ) #1 #2 r p! #0 #4 r p! ;
: xmit ( n ) #5 r p@ $20 and drop if #0 r p! ; then pause xmit ;
: cts #6 r p@ $30 and $30 xor drop if cts ; then xmit ;
: st #6 r p@
: xbits $30 and $10 / dup #1 and 2* 2* + 2/ ;
: st! #4 r p! ;
: ?rcv #5 r p@ #1 and drop if #0 r p@ then ;
: rcv ?rcv if ; then pause rcv ; lblk @ edit

\ Block 101
\
\ : 1@ a-n ( fetch byte from byte address )
\ : 1! na ( store byte to byte address )
\ : r n-p ( convert relative to absolute port address. Base port on stack at compile time. Compiled as literal at yellow
\-green transition )
\ : 9600
\ : 115200 ( baud-rate divisors. These are names, not numbers )
\ : b/s ( set baud rate. Edit to change )
\ : init ( initialize uart )
\ : xmit n ( wait for ready and transmit byte )
\ : cts n ( wait for clear-to-send then xmit )
\ : st -n ( fetch status byte )
\ : xbits n-n ( exchange status bits )
\ : st! n ( store control byte )
\ : ?rcv ( fetch byte if ready. Set flag to be tested by ) if
\ : rcv -n ( wait for ready and fetch byte )

\ Block 102
( App: Colors ) empty

#4210752 MagentaV col #4210752 MagentaV del
: lin dup 2/ 2/ dup 2* line ;
: hex xy @ #7 and over 2/ for lin #7 + next over for lin next swap 2/ for #-7 + lin next drop ;
: +del del @ nop
: petal and col @ + $00F8F8F8 and rgb color #100 hex ;
: -del del @ $00F8F8F8 xor $00080808 + ;
: rose #0 +del #-176 #-200 +at $00F80000 -del petal #352 #-200 +at $00F80000 +del #-264 #-349 +at $F800 -del petal #176 #-200
+at $F8 +del #-176 #98 +at $F8 -del petal #176 #-200 +at $F800 +del ;
: ok show page #512 #282 at rose text col @ h. space del @ $FF and h. keypad ; nload ok h e

\ Block 103
\ ( Draws 7 hexagons. Colors differ along red, green and blue axes. )
\ : col ( color of center hexagon )
\ : del ( color difference )
\ : lin n ( draws 1 horizontal line of a hexagon )
\ : hex n ( draws top, center and bottom. Slope 7 x to 4 y is 1.750 compared to 1.732 )
\ : +del n ( increment color )
\ : -del n
\ : petal n ( draw colored hexagon )
\ : rose ( draw 7 hexagons )
\ : ok ( describe screen. Center color at top )

\ Block 104
( Colors keypad )
: in del @ 2* $00404040 min del ! ;
```

: out del @ 2/ $00080808 max del ! ;
: r $00F80000
: +del del @
: +col and col @ + $00F8F8F8 and col ! ;
: g $F800 +del ;
: b $F8 +del ;
: -r $00F80000 -del +col ;
: -g $F800 -del +col ;
: -b $F8 -del +col ;
: nul ;
: h keypd nul nul quit nul -r -g -b nul r g b nul out nul nul in nul nul nul nul nul nul nul nul nul nul nul nul $00250000
, $00626772 dup , , $2B00002D , #0 , #0 , #0 ,

\ Block 105
\
\ : in (increment color difference)
\ : out (decrement it)
\ : r
\ : g
\ : b (increment center color)
\ : -r
\ : -g
\ : -b (decrement it)
\ : +del (redefine with ;)
\ : +col (change center color)
\ : nul (ignore)
\ : h (describe keypad)

\ Block 106
(App: Wood) empty #125810090 MagentaV x #-1123891786 MagentaV y

#8286477 MagentaV inc #33554432 MagentaV frame #39 MagentaV dep #65056 MagentaV hole
: h0 #400000 inc ! #15 dep !
: home inc @ scrnw #2 / * negate x s ! inc @ scrnh #2 / * y ! ; macro
: f* $2EF7 2, #26 shr $E2C1 2, #6 1, $C20B 2, nip ;
: w! a! $00028966 3, drop ; forth
: wf+ frame @ w! #2 frame +! ;
: om negate $FF + ;
: o5 om $03 shr $07E0 xor ;
: o4 $FC and #3 shl $1F xor ;
: o3 om $F8 and #8 shl $1F xor ;
: o2 #3 shr $F800 xor ;
: o1 om $FC and #3 shl $F800 xor ;
: o0 $F8 and #8 shl $07E0 xor ;
: order jump o0 o1 o2 o3 o4 o5 o0
: hue #8 shl #26 / dup $FF and swap #8 shr order ;
: vlen dup f* swap dup f* + ;
: vdup over over ;
: vndp push push vdup pop pop ;
: itr over dup f* over dup f* negate + push f* 2* pop swap v+ over 2* + 2/ vndp + + ;
: data ; #4 +load ok draw h

\ Block 107
\ (Display an imitation pine blockboard screen)

\

\ (This is based on a skewed Mandelbrot set with)

\ (modified colors)

\ Block 108
(App: Mandelbrot Set) empty

#-204800000 MagentaV x #153600000 MagentaV y #400000 MagentaV inc

#34 MagentaV dep #33554432 MagentaV frame #0 MagentaV hole
: h0 #400000 inc ! #34 dep !
: home inc @ scrnw #2 / * negate x s ! inc @ scrnh #2 / * y ! ; macro
: f* $2EF7 2, #26 shr $E2C1 2, #6 1, $C20B 2, nip ;
: w! a! $00028966 3, drop ; forth

```
: wf+ frame @ w! #2 frame +! ;
: hue ( n-n ) #8191 * ; dup dup + dup dup + + + dup dup + dup dup    ef + + ; #3142 * ; @ ;    ef
: vlen dup f* swap dup f* + ;
: vdup over over ;
: vndp push push vdup pop pop ;
: itr over dup f* over dup f* negate + push f* 2* pop swap v+ ;
: x: ( c- ) emit $3D emit ;
: data text #0 #0 at $78 x: x @ . $79 x: y @ . $69 x: inc @ . $64 x: dep @ . ; nload ok draw h

\ Block 109
\

\ Block 110
( Mandelbrot Set )
: o 0 0 dep @ #1 max for vndp itr vdup vlen $F0000000 + drop -if *next drop drop hole @ ; then drop drop pop hue ;
: mh x @ swap scrnw for o wf+ inc @ u+ next nip ;
: mv y @ scrnh for mh inc @ negate + next drop ;
: +d #2 dep +! : -d #-1 dep +! dep @ #1 max dep !
: draw vframe frame ! mv data ;
: ok c show keypad ;
: l inc @ scrnw #1 - #8 */ negate x +! draw ;
: u inc @ scrnh #1 - #8 */ y +! draw ;
: d inc @ scrnh #1 - #8 */ negate y +! draw ;
: r inc @ scrnw #1 - #8 */ x +! draw ;
: +z inc @ #3 max dup scrnw #1 - #8 */ x +! dup scrnh #1 - #8 */ negate y +! #3 #4 */ #3 max inc ! draw ;
: -z inc @ #10000000 min dup scrnw #1 - #8 */ negate x +! dup scrnh #1 - #8 */ y +! #4 #3 */ inc ! draw ;
: hh home draw ; : hh2 h0 draw ;
: h keypd nul nul quit nul -d nul nul +d l u d r -z hh hh2 +z nul nul nul nul nul nul nul nul nul nul $2500 , $2B00002D
, $13121110 , $2B30482D , #0 , #0 , #0 ,

\ Block 111
\ ( More Mandelbrot )

\ ( ludr move the cursor left right up down )

\ ( - + top row change depth detail )

\ ( - + bottom row change zoom )

\ ( h centres the image to the home location )

\ ( 0 resets depth and zoom )

\ Block 112
( App: words )

#74878 MagentaV from

#10315920 MagentaV tobl

#16 MagentaV num

#10010 MagentaV blkn
: tost ( u-- ) tobl @ ! #4 tobl +! ;
: tb $25C7C00E tost ; ( blue tab token )
: docr $9080000E tost ; ( blue cr token )
: setbl ( u-- ) blkn ! blkn @ block dup #1024 erase tobl ! #0 num ! ;
: newblk blkn @ #2 + setbl ;
: one ( -- ) from @ @ $FFFFFFF0 and $09 or tost tb #4 from +! #1 num +! ;
: chk1 num @ #6 mod #0 + 0if docr then drop ;
: chk2 num @ #96 mod #0 + 0if newblk then drop ;
: ww ( -- ) #10000 setbl
  ( FORTH ) $B189640B tost docr
  ftha from ! fthn @ for one chk1 chk2 next
  ( MACRO ) newblk $8AC84C0B tost docr
  maca from ! macn @ for one chk1 chk2 next
  ( BLUE ) newblk $C74CC80B tost docr
  blua from ! blun @ for one chk1 chk2 next
  #10000 edit ; ww

\ Block 113
```

```
\

\ Block 114
( App: Sounds make a noise ) empty

#25 MagentaV tempo #0 MagentaV mute #2259 MagentaV period
: tn ( ft- ) tempo @ * swap #660 #50 */
: hz ( tf- ) push #1000 #1193 pop */
: osc ( tp- ) dup period ! split $42 p! $42 p!
: tone ( t- ) mute @ #0 + drop if drop ; then $4F $61 p! ms $4D $61 p! #20 ms ;
: click #1 #90 osc ;
: t #3 tn ;
: q #8 tn ;
: c #16 tn ;
: 2tone #75 q #50 q ;
: h1 #50 c #54 q #50 q #45 c #60 c ;
: h2 #40 c #45 q #50 q #50 c #45 c ;
: h3 #54 c #60 q #54 q #50 c #45 q #40 q #50 t #45 t #50 t #45 t #45 #12 tn #40 q #40 #32 tn ;
: hh
: handel h1 h2 h3 ;
: piano #55 #7 for dup q #3 #2 */ next drop ;
: cetk #6 c #10 c #8 c #4 c #6 #32 tn ;
: bomb mute @ #0 + drop if ; then $4F $61 p! #500 for #1000 i invert + split $42 p! $42 p! #1 ms next $4D $61 p! #1 #32 tn
; handel

\ Block 115
\ ( Sounds : using the PC internal speaker )
\ : tempo ( in ms per 1/8 quaver )
\ : mute ( equals -1 to disable sound )
\ : period ( test only - value sent to hardware )
\ : tn ( ft- play f Hz for t * 11 ms )
\ : hz ( tf- play t ms at f Hz )
\ : osc ( tp- play t ms of period p )
\ : tone ( t- play the current tone for t ms )
\ : click ( makes a click )
\ : t ( triplet )
\ : q ( quaver )
\ : c ( crotchet )
\ : 2tone ( 2 tones )
\ : h1
\ : h2
\ : h3
\ : hh
\ : handel ( part of Handels Gavotte )
\ : piano
\ : cetk ( Close Encounters of the Third Kind )
\ : bomb ( - well sort of .... )

\ Block 116
( Colourblind Editor Display )

#1 MagentaV state $01 MagentaV state*
: +txt white $6D emit space ;
: -txt white $6E emit space ;
: +imm yellow $58 emit space ;
: -imm yellow $59 emit space ;
: +mvar yellow $09 emit $11 emit $05 emit $01 emit space ;
: txts string $03010100 , $07060504 , $09090901 , $0F0E0D0C , ( ; )
: tx ( c-c ) $0F and txts + 1@ $0F and ;
: .new state @ $0F and jump nul +imm nul nul nul nul nul nul nul +txt nul nul +mvar nul nul nul ;
: .old state* @ $0F and jump nul -imm nul nul nul nul nul nul nul -txt nul nul nul nul nul nul ;

here
: cb ( n-n ) #0 + 0if ; then tx
  state @ swap dup state ! - drop if .old .new
  state @ #0 + if dup state* ! then then ;
: cbs ( -- here ) #0 + $00 + cblind ! ;

\ Block 117
\
\ : state
```

```
\ : cb ( acts on a change of token type. It ignores extension tokens )

\ Block 118
( Graphics demo Todo: fix this! ) empty

#2 #22 +thru


: htm #116 load ( html ) ;

ok

\ Block 119
\ ( A graphics extension package )
\ : . ( Type ) ok ( after loading this block )

\ Block 120
( added macros ) forth
: mfill #24 for cr space #5 for rand32 h. space next next ;
: matrix show black screen green mfill keypad ;

\ Block 121
\ ( added macros )
\ : 1+ ( increment tos )
\ : 1- ( decrement tos )
\ : @b ( fetch byte from absolute addr. )
\ : @w ( fetch word from absolute addr. )
\ : @l ( fetch long from absolute addr. )
\ : !b ( store byte in absolute addr. )
\ : !w ( store word in absolute addr. )
\ : !l ( store long in absolute addr. )
\ : matrix ( What is the Matrix? )
\ : ver ( returns the address of the CFDOS version - use as ) ver dump

\ Block 122
( Stack juggling + misc. )
: v- ( v-v ) push invert 1+ u+ pop invert 1+ + ;
: vn push rot less if rot pop -rot ; then -rot pop ; #2222 MagentaV pen #236986408 MagentaV bs
: vloc ( xy-a ) scrnw 2* * over + + vframe + ;

macro
: @w $8B66 3, ;
: !w a! $00028966 3, drop ;

forth
: point ( xy- ) pen @ swap w! ;
: at? ( -xy ) xy @ $00010000 /mod swap ;
: @r ( a-a ) 1+ dup #4 u+ @ + ;
: !r ( aa- ) 1+ dup push negate #-4 + + pop ! ;
: select ( an- ) #5 * over + @r swap @r !r ;

\ Block 123
\ ( Stack juggling words. small and fast. )
\ : addr -a ( absolute address )
\ : rot abc-bca ( stack pictures are best .. )
\ : -rot abc-cab ( ..described with letters, in )
\ : tuck ab-bab ( ..this case. )
\ : 2swap abxy-xyab
\ : 2over abxy-abxyab
\ : 2dup ab-abab
\ : v- v1v2 - v1-v2 ( vector subtract. )
\ : vn vv-vv ( sort vectors so x1 is less x2 )
\ : vframe -addr ( address of screen. )
\ : pen -addr ( current color. )
\ : bs -addr ( base for elements )
\ : vloc xy-a ( convert xy into addr. )
\ : point xy- ( set point at xy to current pen. )
\ : at? -xy ( return current screen location. )
\ : @r a-a ( get absolute addr from jump/call )
\ : !r aa- ( set jump/call to absolute addr. )
\ : select an- ( select call n from table a. Store it in table call 0 )
```

```
\ Block 124
( new logo )
: .co $72 $6F $6C $6F $63 5* ;
: .fo $68 $74 $72 $6F $46 5* ;
: cf #27 dup at silver .co .fo #25 dup at red .co green .fo ;
: log1 show black screen text cf keypad ;
: ckb black #0 #740 at #1023 #767 box #800 #650 at #1023 #740 box ;
: grads #0 #128 for i 2* 1- rgb color dup #10 at #5 + dup #120 box next
  iconw #21 * - #128 for #257 i 2* negate + dup #256 * + rgb color dup #10 at #5 + dup #100 box next drop ;

\ Block 125
\ ( New logo )
\ : log1 ( a simple text demo )
\ : ok ( the graphics demo )

\ Block 126
( Circles ) #-16977 MagentaV c-cd #0 MagentaV c-ff
: point4 #4096 * swap #4 * 2dup + 2/ negate bs @ + pen @ over w! over push + pen @ over w! + pen @ over w! pop negate + pen
@ swap w! ;
: opnts 2dup point4 2dup swap point4 ;
: d? c-cd @ ?f drop -if ; then dup invert c-cd +! 1- #1 c-ff ! ;
: cfl 1+ 1+ push pen @ swap pop 2/ for over over w! 1+ 1+ next drop drop ;
: cfl4 #4096 * swap #4 * 2dup + 2/ negate bs @ + swap 2dup cfl push + pop cfl ;
: fvrt ?f drop if cfl4 #0 c-ff ! ; then point4 ;
: fpnts 2dup c-ff @ fvrt 2dup swap cfl4 ;
: points opnts ;
: addr pop ;
: pntst addr points opnts fpnts ;
: framed pntst #1 select ;
: filled pntst #2 select ;
: circle ( rxyc- ) #0 c-ff ! pen ! #1024 * + 2* vframe + bs ! #0 swap dup negate c-cd !
: crcl less if points #1 u+ over c-cd +! d? crcl ; then points drop drop ;

\ Block 127
\ ( Circles )
\ : point4 ( .. all other words are internal. )
\ : points ( acts like a deferred word. )
\ : pntst ( table of calls to different point routines. Select alters ) points
\ : framed ( set ) circle ( to draw outlined circles. )
\ : filled ( set ) circle ( to draw filled circles. )
\ : circle rxyc- ( draw circle with radius ) r ( center ) xy ( in color ) c

\ Block 128
( lines )

#-1456 MagentaV ax #0 MagentaV ay #2048 MagentaV sx #2 MagentaV sy #31987278 MagentaV lbase

macro
: lp $8B909090 , $C88BADE8 , $205A8BAD , $232B8966 , $030578C0 , $185A0302 , $03084203 , $ECE2105A , ;

forth
: !base ( xy- ) #2048 * over + + vframe + lbase ! ;
: bline ( xy- ) abs 2* dup ay ! over 2* negate ax ! over
  negate + swap 1+ pen @ ax a! lp drop ;
: ?xd ( vv-vv ) 2over 2over v- abs swap abs swap less
  drop drop #-1 if 1+ then ?f drop ;
: !sy ( yn-y ) push ?f pop -if negate then sy ! bline ;
: xdom ( xyxy- ) 2swap !base #2 sx ! #2048 !sy ;
: ydom ( xyxy- ) swap 2swap swap !base swap #2048 sx !
  #2 !sy ;
: aline ( vv- ) ?xd if vn 2over v- xdom ; then push push
  swap pop pop swap vn 2over v- ydom ;
: line ( xy- ) at? 2over aline at ;
: frame ( xy- ) at? 2over drop over line 2over line 2swap
  push drop over pop line line ;

\ Block 129
\ ( line drawing Do Not Mess With Variables. They are indexed by lp. )
\ : lp ( macro inner loop for speed. Draws point and moves location. )
\ : !base x y -- ( set base address )
```

```
\ : bline dx dy -- ( draw a line using bresenham x dominant )
\ : ?xd v1 v2 -- v1 v2 ( set flag if line is x-dominant )
\ : !sy dy n -- dy ( store n in sy set sign to match sign of dy )
\ : xdom x y dx dy ( draw an x-dominant line )
\ : ydom x y dx dy ( draw a y-dominant line )
\ : aline v1 v2 ( draw any straight line )
\ : line x y ( draw line from current at to xy. Moves at to given xy. )
\ : frame xy- ( trace outline of rectangle with corners at and xy. Pen position is not altered. )

\ Block 130
( Utils )
: xxcoy ( sf st ) $E7C1F88B , $368B560A , $B90AE6C1 , $0100 , $AD5EA5F3 , $C3AD 2,
: xrcopy ( sf sl st ) push dup push swap negate + pop swap pop over + swap for over over copy push 1- pop 1- -next drop drop
;

\ Block 131
\ ( Utils )
\ : copy from to- ( copy from to block numbers. Unlike orig copy; no change to blk )
\ : rcopy first last to- ( multiple block copy routine )

\ Block 132
( fillstack ) #1114112 MagentaV fstak $00 MagentaV fstakn
: fstini ( - ) $0400 block fstak ! 0 fstakn ! ;
: fpop ( -uuu ) fstak @ #3 for dup @ swap cell- next
  fstak ! #-3 cells fstakn +! ;
: fpsh ( uuu- ) #3 for cell fstak +! fstak @ ! next
  #3 cells fstakn +! ;
: fst? ( - ) fstakn @ ?f drop ; fstini

macro
: 2- 1- 1- ;
: 2+ 1+ 1+ ;

forth
: 5drop ( uuuuu- ) drop drop drop drop drop ;
: rtre ( a-n ) #2048 #1 - and negate #2048 + ;
: enstak ( dlrlr-dlrlr ) 2- #4 pick dup #3 pick + over
  #3 pick + fpsh over #4 pick negate + 2+ drop
  -if #4 pick negate dup #3 pick +

\ Block 133
\ ( fillstack: stack of spans to fill. )
\ : fstini ( initialize )
\ : fpop ( pop the next element from the stack )
\ : fpsh ( push element on the stack )
\ : fst? ( set 0 flag if empty )
\ : 2- ( screen pixels are 2 bytes. )
\ : 2+
\ : 5drop ( unload forth stack. )
\ : rtre a-n ( return remaining to right screen edge. )
\ : enstak dlrlr-dlrlr ( push a span or element onto the stack. Also push a left hand direction reversal and a right hand
\ reversal if needed. )

\ Block 134
( area filling ) #25702 MagentaV tfc #14660 MagentaV fc
: pset ( a-f ) dup dup w@ $FFFF and tfc @ negate + drop
  if drop 0 ; then fc @ swap w! 0 1+ ;
: bcup ( a-a ) dup #2048 #1 - and 2- begin -if drop ; then
  push 2- pset drop pop if 2- *end then drop 2+ ;
: ispan pset if ; then push enstak pop ;
: xgr dup negate #3 pick + drop ;
: nispan ( dlrlx- ) xgr -if 5drop pop pop pop drop drop
  drop ; then pset if push nip dup pop then ;
: dosp ( dlrlx-dlrlxi ) jump nispan ispan ;
: sha2 over rtre begin ( dlrlxic ) -if drop ; then push
  dosp #2 u+ pop 2- end
: sha1 ( dlr- ) over pset over ( dlrxil ) if bcup ( dlrxil ) then
  swap push swap 2+ pop ( dlrlxi ) sha2 ?f drop
  if enstak then 5drop ;
: sha begin fst? if fpop sha1 *end then ;
: fsln ( a-lr ) dup bcup swap dup rtre
```

```
   begin -if drop ; then push pset drop if
   2+ pop 2- *end then pop drop 2- ;
: afill ( xyc- ) fstini fc ! vloc dup w@ $FFFF and tfc !
   fsln over over #-2048 u+ #-2048 + #-2048 -rot fpsh
   #2048 u+ #2048 + #2048 -rot fpsh sha ;
: afill drop drop drop ;

\ Block 135
\ ( area filling )
\ : pset a-0/1 ( set pixel at a, if pixel equals tfc. Return 0 if not, 1 if pixel was set. )
\ : bcup a-a ( adjust a until left edge is found. Limited to screen edge. )
\ : ispan ( stack if the right edge is found. )
\ : xgr ( Set neg flag if x is greater then parent-r )
\ : nispan ( exit if beyond right edge of span, else start a new span. )
\ : dosp dlrlx - dlrlxi ( jump table. )
\ : sha2 ( let x go over each pixel and set it or start/end new spans. )
\ : sha1 ( starting at left edge, find the new left edge and init x to next pixel. stack if run into right screen edge while
\ in span. )
\ : sha ( pop the next span and color it. )
\ : fsln a-lr ( Starting at screen address a, find the left edge and right edge of the seed line. Color it in the proces
\s. )
\ : afill xyc ( starting with screen location xy, and color c, fill the color found there with c until the color found change
\s. )

\ Block 136
( random ) #-1896373196 MagentaV rsav #-526774649 MagentaV rseed
: rand ( -- ) time rsav ! $E09A0E87 rseed ! ;
: ror ( u-u ) $D3ADC88B , $C3C8 2,
: random ( w-w ) push rseed @ #0 #32 for 2* swap 2* swap -if rsav @ xor then next nip #15 ror dup rsav ! abs pop mod abs
; rand
: tt $0100 random ;

\ Block 137
\ ( random )
\ : rand - ( set random variables )
\ : ror nm-n ( rotate n m times right )
\ : random n-0..n-1 ( return a random number range 0..n-1 limited to a 16 bit number. )

\ Block 138
( demos )
: xlate #384 + #512 u+ ;
: xat xlate at ;
: xline xlate line ;
: 4lines over #0 xat #0 over xline over - #0 xline negate #0 swap xline #0 xline ;
: art #70 for #71 i - #5 * i #5 * 4lines next ;
: radius #8 ;
: lrc push dup dup + negate pop + random + ;
: shade 2over #2 + 2over drop #3 + #0 circle circle ;
: dotty filled #100 for radius random dup #397 lrc #621 + over #176 lrc #121 + $FFFF random shade next ;
: blbx black #6 #121 at #404 #299 box ; #-17 MagentaV xyzz
: fillit #-1 xyzz +! xyzz @ #200 + drop -if blbx 0 xyzz ! then framed #3 for #8 random #2 + dup #398 lrc #6 + over #178 lrc
#121 + $FFFF circle next

; #6 #210 $FFF0 random afill ;

\ Block 140
( new logo 2 )
: lnes framed #20 for i 2* #40 + #250 #584 $FF07 circle next filled #30 #250 #584 $F800 circle framed $FFFF pen ! #620 #120
at #1020 #300 frame #5 #120 at #405 #300 frame ;
: ok show black screen grads lnes text cf dotty fillit ckb keypad ; ( ok )

\ Block 141
\ ( New logo )
\ : log1 ( a simple text demo )
\ : ok ( the graphics demo )

\ Block 142
( html0 ) #80 load #2222119 MagentaV h-dd #0 MagentaV ppt macro
: 2/s ?lit $E8C1 2, 1, ; forth
: temit h-dd @ !b #1 h-dd +! ;
: tspc $20 temit ;
```

```
: .dc ?f #1 -if - then swap abs
: dcl #10 /mod swap $30 + push ?f 0if drop ?f drop -if $2D temit then pop temit ; then dcl pop temit nop ;
: .hx $39 over #15 and $30 + less nip if $27 + then push #4 2/s 0if drop pop temit ; then .hx pop temit nop ;
: strt dup @b $FF and if temit 1+ strt ; then drop drop ;
: str: pop strt ;
: header str: $6D74683C , $3C0A3E6C , $6B6E696C , $6C657220 , $7974733D , $6873656C , $20746565 , $65707974 ,
$6574223D ,
$632F7478 , $20227373 , $66657268 , $3D 1, $6C6F6322 , $6F66726F , $2E687472 , $22737363 , $703C0A3E , $0A3E 3,
: trailer str: $74682F3C , $0A3E6C6D , $00 1,

\ Block 143
\ ( html0. Block 80 has ascii conversion tables. )
\ : h-dd ( data destination. ) ppt ( pre- parsed type. )
\ : 2/s ( macro, right shift by n. )
\ : temit c- ( emit char to target. )
\ : tspc ( emit space )
\ : .dc n- ( signed decimal print. Recursive! )
\ : dcl ( dec print loop. )
\ : .hx n- ( unsigned hex print. Also recursive. Both routines have no leading zeroes. )
\ : strt a- ( Print bytes from address until first null byte. )
\ : str: ( Output what follows up to null byte. )
\ : header ( Lay down html header to display blocks. The header is very minimal. It expects colorforth.css in the same direct
\ory. )
\ : trailer ( Closing html stuff. )

\ Block 144
( html1 )
: .code 1- drop -if ; then str: $6F632F3C , $003E6564 ,
: .all str: $646F633C , $6C632065 , $3D737361 , $00 1,
: same? ppt @ over ppt ! swap over - 1+ + drop ;
: comn same? 0if drop tspc pop drop ; then .code .all ;
: .def str: $3E666564 , $20 2,
: .com #2 comn str: $3E6D6F63 , $20 2,
: .chx #3 comn str: $3E786863 , $20 2,
: .exe #4 comn str: $3E657865 , $20 2,
: .xhx #5 comn str: $3E786878 , $20 2,
: .cpm #6 comn str: $3E6D7063 , $20 2,
: .var #7 comn str: $3E726176 , $20 2,
: .txt #8 comn str: $3E747874 , $20 2,
: .txc #9 comn str: $3E637874 , $20 2,
: .tac #10 comn str: $3E636174 , $20 2,

\ Block 145
\ ( html1 )
\ : .code n- ( output /code in brackets if n is larger then 0. )
\ : .all ( common part to start a new code tag. )
\ : same? n-o ( set ppt to the new type. Return the old type with flags set from comparison. )
\ : comn n- ( if this is a new tag, close prev tag and print common part. If not: print space AND EXIT CALLER )
\ : .def ( Each of these words correspond to a )
\ : .com ( .. code tag as defined in colorforth.css )
\ : .chx ( .. The numbers are positional, and bare )
\ : .exe ( .. no correspondence to the pre parsed )
\ : .xhx ( .. types. They will output if a change )
\ : .cpm ( .. in tag is required. Comn will exit )
\ : .var ( .. by doing a pop-drop if the tag is the )
\ : .txt ( .. same. )
\ : .txc
\ : .tac

\ Block 146
( html2 )
: .str ch if temit .str ; then drop drop ;
: bs1 #0 ppt ! str: $3E72683C , $6C627B0A , $206B636F , $00 1,
: bs2 str: $643C0A7D , $63207669 , $7373616C , $786F623D , $0A3E 3,
: bend ppt @ .code str: $69642F3C , $000A3E76 ,
: .br 1- drop -if ; then str: $3E72623C , $0A 2,
: pp0 .str ;
: pp1 .exe .str ;
: pp3 ppt @ dup .code .br #1 ppt ! .all .def .str ;
: pp4 .com .str ;
: pp7 .cpm .str ;
```

```
: pp9 .txt .str ;
: ppa .txc .str ;
: ppb .tac .str ;
: ppc .var .str 1+ dup @ .com .dc ;

\ Block 147
\ ( html2 )
\ : .str n- ( Unpack n and print as ascii. )
\ : bs1 ( clear the type and print html stuff for the start of a block. )
\ : bs2 ( second half of block header. )
\ : bend ( Block end html stuff. )
\ : .br n- ( Html line break, if n larger then 0 )
\ : pp0 ( The preparsed words in a block are )
\ : pp1 ( .. printed by the ppn words. Eg pp0 is )
\ : pp3 ( .. word continuation pp1 is for executed )
\ : pp4 ( .. words, etc. They unpack and print. )
\ : pp7 ( .. They also print html tags. )
\ : pp9
\ : ppa
\ : ppb
\ : ppc

\ Block 148
( html3 )
: dbn push 1+ dup @ pop ?f drop ;
: sln dup 2/ 2/ 2/ 2/ 2/ swap #16 and drop ;
: xnb if .xhx .hx ; then .exe .dc ;
: cnb if .chx .hx ; then .com .dc ;
: pp2 dbn xnb ;
: pp5 dbn cnb ;
: pp6 sln cnb ;
: pp8 sln xnb ;
: ppdo jump pp0 pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9 ppa ppb ppc ;
: index dup #15 and dup push or pop ;
: dblk dup bs1 .dc bs2 block begin dup @ ?f 0if drop drop bend ; then index ppdo 1+ end
: hbuf #2000 block ;
: html hbuf #4 * h-dd ! header swap over for over i - 1+ + over + dblk next drop drop trailer hbuf h-dd @ #3 + #4 / over
- 1+ + #3 for tspc next ;

\ Block 149
\ ( html3 )
\ : dbn an-an ( Fetch next word. Set hex flag. )
\ : sln n-n ( Make full word and set hex flag. )
\ : xnb n- ( print n as hex/dec executed number. )
\ : cnb n- ( print n as hex/dec compiled number. )
\ : pp2 an-a ( A double executed number. )
\ : pp5 an-a ( A double compiled number. )
\ : pp6 n- ( A single compiled number. )
\ : pp8 n- ( A single executed number. )
\ : ppdo ( Table of words. The index is the pre- parsed type type. )
\ : index n-ni ( extract index from n. )
\ : dblk b- ( print block b in html. )
\ : hbuf -a ( start of buffer. )
\ : html bn-al ( Output n blocks starting with block b in html. Leaves addr and length on the stack, so it can be saved using
\ ) file put ( on a floppy. )

\ Block 150
( html3 )
: dbn push 1+ dup @ pop ?f drop ;
: sln dup 2/ 2/ 2/ 2/ 2/ swap #16 and drop ;
: xnb if .xhx .hx ; then .exe .dc ;
: cnb if .chx .hx ; then .com .dc ;
: pp2 dbn xnb ;
: pp5 dbn cnb ;
: pp6 sln cnb ;
: pp8 sln xnb ;
: ppdo jump pp0 pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9 ppa ppb ppc ;
: index dup #15 and dup push or pop ;
: dblk dup bs1 .dc bs2 block begin dup @ ?f 0if drop drop bend ; then index ppdo 1+ end
: hbuf #2000 block ;
: html hbuf #4 * h-dd ! header swap over for over i - 1+ + over + dblk next drop drop trailer hbuf h-dd @ #3 + #4 / over
```

```
- 1+ + #3 for tspc next ;

\ Block 151
\ ( html3 )
\ : dbn an-an ( Fetch next word. Set hex flag. )
\ : sln n-n ( Make full word and set hex flag. )
\ : xnb n- ( print n as hex/dec executed number. )
\ : cnb n- ( print n as hex/dec compiled number. )
\ : pp2 an-a ( A double executed number. )
\ : pp5 an-a ( A double compiled number. )
\ : pp6 n- ( A single compiled number. )
\ : pp8 n- ( A single executed number. )
\ : ppdo ( Table of words. The index is the pre- parsed type type. )
\ : index n-ni ( extract index from n. )
\ : dblk b- ( print block b in html. )
\ : hbuf -a ( start of buffer. )
\ : html bn-al ( Output n blocks starting with block b in html. Leaves addr and length on the stack, so it can be saved using
\ ) file put ( on a floppy. )

\ Block 152
( simpler and slower bresenham line drawing. For reference. ) #-360 MagentaV ax #0 MagentaV ay #2 MagentaV sy #0 MagentaV
sw

: bpoint push 2dup sw @ ?f drop if swap then point pop ;
: bline abs 2* dup ay ! over 2* negate ax ! over negate + swap 1+ for bpoint ?f +if sy @ u+ ax @ + then ay @ + push #1 u+
pop next drop drop drop ;
: ?xd 2over 2over v- abs swap abs swap less drop drop #-1 if 1+ then ?f drop ;
: !sy push ?f pop -if negate then sy ! bline ;
: xdom #0 sw ! #1 !sy ;
: ydom #1 sw ! #1 !sy ;
: aline ?xd if vn 2over v- xdom ; then push push swap pop pop swap vn 2over v- ydom ;

\ Block 154


\ Block 155
\ ( fillstack: stack of spans to fill. )
\ : fstini ( initialize )
\ : fpop ( pop the next element from the stack )
\ : fpsh ( push element on the stack )
\ : fst? ( set 0 flag if emtpy. )
\ : pick ( copy n from the stack. )
\ : 2- ( screen pixels are 2 bytes. )
\ : 2+
\ : 5drop ( unload forth stack. )
\ : rtre a-n ( return remaining to right screen edge. )
\ : enstak dlrlr-dlrlr ( push a span or element onto the stack. Also push a left hand direction reversal and a right hand
\ reversal if needed. )

\ Block 160
( Timing ) empty macro
: out $E1E6 2, ; forth
: tare time invert #1000 for next time + ;
: tare+ time invert push #1000 for dup next c pop time + ;
: test +     s #1000 for out next time + ; ( next 3 loop 5.7 /next 2 /swap 25 swap 7.2 ) macro
: c! $C88B 2, drop here ;
: loop $49 1, $75 1, ( e2 ) here invert + 1, ; forth
: try time invert #1000 c! loop time + ;

\ Block 162
( Spy ) empt $03F8 #54 load init
: ry #5 r p@ ; nload init
: buffer #2000 block ; #2000 #1 wipes #0 MagentaV buf #0 buf !
: b! swap $FF and + buf @ buffer + ! #1 buf +! ;
: dev r2 if dup xmit $0100 b! dev ; then ;
: pc ?rcv if dup x2 0 b! pc ; then ;
: relay s2 st s2! st! dev pc ;
: .1 $0F and digit ;
: .byte dup $10 / .1 .1 ;
: traffic text buffer buf @ #1 max #400 min for dup @ green $0100 ? if red then .byte #1 + next drop ;
: ok show black screen relay traffic keypad ;
```

```
: k show black screen relay keypad ;
: q #6000 for relay next ;
: test st! st ; #84 load

\ Block 164
( Serial 2 )
: r $02F8 + ;
: b/s $83 #3 r p! 9600 #262 #0 r p! #0 #1 r p! #3 #3 r p! ;
: init b/s ( 16550 ) #1 #2 r p! #0 #4 r p! ;
: x2 #5 r p@ $20 and drop if #0 r p! ; then x2 ;
: c2 #6 r p@ $30 and $30 or drop if c2 ; then x2 ;
: s2 #6 r p@ xbits ;
: s2! #4 r p! ;
: r2 #5 r p@ #1 and drop if #0 r p@ ; then ;

\ Block 166
( Dynapulse 200m )
: send pop swap for dup 1@ x2 #1 + next drop ;
: reset #2 send $2323 ,
: 1st #12 send $37269A12 , $39027AFD , $23C75680 ,

\ Block 168
( Test sidt and lidt )

#7168 MagentaV vidt sidt vidt !
: resi cli vidt @ lidt ;

\ Block 169
\ ( This block is used by the next block as the interrupt vector table. )

\ Block 170
( Interrupts ) macro
: 1ld ( n ) ?lit $B9 1, , ;
: p! ( na ) a! $EE 1, drop ;
: 2push $5250 2, ;
: 2pop $585A 2, ;
: forth: 2push $00BE5651 3, ivec $0100 + a, ;
: ;forth $595E 2, 2pop ;
: clear $20E620B0 , ;
: 8clear $A0E620B0 , $20E6 2, ;
: i; $CF 1, ; forth
: interrupt ( n ) 2* 2* 2* ivec + here $FFFF and $00080000 + over ! here $FFFF0000 and $8E00 + swap #4 + ! ;
: ifill ( an ) for dup interrupt #1 + next drop ; $00 $70 ifill
: ignore i; $20 $08 ifill
: ignore 2push clear 2pop i; $28 $08 ifill
: ignore 2push 8clear 2pop i; $00 interrupt
: 0div $7FFFFFFF 1ld i;

\ Block 171
\
\ : idt -a ( table of 2-word interrupts. Edit convenient block number )
\ : 1ld n ( load register 1 with literal )
\ : lidt ( load interrupt descriptor table from byte address on stack )
\ : 2push ( save registers 0 and 2 )
\ : 2pop ( restore 2 and 0 )
\ : forth: ( save registers used by Forth )
\ : ;forth ( restore registers used by Forth )
\ : clear ( store 20 to port 20 to clear irq 0-7 )
\ : 8clear ( also 20 to port a0 to clear irq 8-f )
\ : i; ( return from interrupt - restore flags )
\ : !idt b ( execute lidt )
\ : interrupt n ( construct interrupt to ) here. ( Avoid yellow-green literal with red comment )
\ : ifill an ( n entries in default interrupt table )
\ : ignore ( clear ) ( grey = $01644001  ) ( interrupt. Doesnt clear the device )
\ : 0div ( make divisor +infinity, quotient 0 )

\ Block 172
( Admtek Comet An983b ) macro
: align here #7 and #3 xor drop if nop align ; then ; forth
: array pop 2/ 2/ ;
: us ( n ) khz @ #1000 #3 * / * for next ;
```

```
: r ( n-a ) $DB000000 + 2/ 2/ ;
: rom ( a-n ) $A4 + r @ ;
: 3rom ( nnn ) #4 rom #0 rom dup #16 for 2/ next swap ;
: reset #1 $00 r ! #1000 us ;
: frag #0 , $02000000 , $00 , here #4 + , ;
: tx align array frag frag frag frag frag frag
: n tx #1 + ;
: a tx #2 + ; #16 MagentaV f
: fr! f @ + ! ;
: first ( an ) #0 f ! $20000000 or
: send ( an ) $01000000 or n fr! a fr! $80000000 tx fr! #4 f +! ;
: last ( an ) $42000000 or send #1 us
: poll #-1 $08 r ! ;

\ Block 173
\
\ : array -a ( returns word-aligned address in dictionary )
\ : us n ( delay n microseconds. Edit cpu clock rate )
\ : r n-a ( word address of register. Edit base address from ) north ( PCI device configuration )
\ : rom a-n ( fetch 2 bytes of ethernet id )
\ : 3rom -nnn ( 3 byte-pairs of id. )
\ : reset ( controller )
\ : tx -a ( transmit descriptor ring )
\ : n -a ( fragment length/control )
\ : a -a ( fragment address )
\ : send an ( fragment into descriptor queue )
\ : first an ( fragment. )
\ : last an ( fragment. Start transmission )

\ Block 174
( Receive ) #281880 MagentaV rxp
: rx align array $80000000 , $01000600 , $2000 block #4 * dup , here #4 + , $80000000 , $01000600 , $0600 + , rx #4 * ,
: init reset rx #2 * 2* $18 r ( receive ) ! #1 us tx #2 * 2* $20 r ( transmit ) ! #1 us $00202002 ( start ) $30 r ! #1 us
$00010040 $38 r ! sti #-1 $28 r ! ;
: link #3 + @ 2/ 2/ ;
: own? @ #0 or drop ;
: /int rxp @ $80000000 over ! link own? -if #-1 $28 r ! then ;
: rcvd rx nop
: wait dup own? -if link wait ; then dup rxp ! #2 + @ ;
: reg dup r @ h. space #2 h.n cr ;
: regs $B8 reg $A0 reg $98 reg $94 reg $78 reg $60 reg $48 #10 for dup reg #-8 + next drop ;
: ok show $00400000 rgb color screen text regs keypad ;
: rx1 $2000 block dump ;
: rx2 $2000 block $0180 + dump ; ok

\ Block 175
\
\ : rx -b ( receive descriptor ring )
\ : init ( ialize controller. Set tx/rx address/on and perfect match )
\ : link a-b ( next link in descriptor ring )
\ : own? a ( is this descriptor owned? )
\ : /int ( give up ownership of received packet , clear interrupt if no packet remains )
\ : rcvd -a ( return address of recieved packet )
\ : wait -b ( till packet received )
\ : reg a ( display register and address )
\ : regs ( display interesting registers )
\ : ok ( diagnostic display )

\ Block 176
( App: Ethernet ) empty

( interrupts ) #170 load

( hardware interface ) #172 load #174 load macro
: w $66 1, ;
: w@ $8B 2, ;
: w! $0289 2, drop ;
: *byte $C486 2, ; forth
: n@ w w@ $FFFF and *byte ;
: 2! a! w w! ;
: n! a! *byte w w! ;
```

```
: n, *byte 2, ;
: string pop ;
: packet string #-1 dup , 2, 3rom 2, 2, 2, #0 n,
: length ( n ) packet #12 + n! ;
: broadcast #-1 dup dup packet nop
: 3! swap over 2! #2 + swap over 2! #2 + 2! ;
: ethernet ( n ) length packet #14 first ;
: +ethernet ( -a ) rcvd #14 + ; fixthis

#2 #16 +thru breakhere ( todo fix this )

$2A interrupt
: serve forth: receive /int 8clear ;forth i; init ok discover

\ Block 177
\
\ : empty ( redefined to disable interrupts )
\ : w ( 16-bit prefix )
\ : w@ b-n ( fetch 16-bits from byte address )
\ : w! nb ( store 16-bits )
\ : *byte n-n ( swap bytes 0 and 1 )
\ : n@ b-n ( fetch 16-bit network-ordered number )
\ : 2! nb ( store 16-bit number )
\ : n! nb ( store 16-bit number in network order )
\ : n, n ( compile 16-bit number in network order )
\ : string -b ( returns byte address )
\ : packet -b ( ethernet packet header )
\ : dest -b ( destination field in packet )
\ : src -b ( source field )
\ : length n ( store length into packet )
\ : 3! nnnb ( store 3-word MAC )
\ : ethernet n ( send header with type/length )
\ : @ethernet -b ( return payload address of received packet )

\ Block 178
( ARP for a single correspondent ) macro
: move ( sdn ) $C189 2, drop $00C78957 3, drop $00C68956 3, $A4F3 2, $5F5E 2, drop ; forth
: . ( n ) 1, ;
: message string $01 n, $0800 n, $06 . $04 . $01 n,
: me 3rom 2, 2, 2, ( IP ) #0 . #0 . #0 . #0 .
: to #0 #0 #0 2, 2, 2, ( IP ) #0 . #0 . #0 . #0 .
: sender #8 + ;
: target #18 + ;
: dir #6 + ;
: ip #6 + w@ ;
: ar ( n ) message dir n! $0806 ( ARP ) ethernet message #28 last ;
: arp cli broadcast #1 ar sti ;
: -arp ( b-b ) dup #-2 + n@ $0806 or drop if ; then pop drop
: me? dup target ip message sender ip or drop if ; then dup sender packet #6 move
: query? dup dir n@ #1 or drop if ; then sender message target #10 move #2 ar ;

\ Block 179
\ ( Set ip addresses with Edit. Normal order, net bytes first )
\ : move sdn ( move n bytes from source to destination. Register 1 is used, 6 and 7 are saved )
\ : . n ( compile byte. Resembles URL punctuation )
\ : message -b ( 28-byte string )
\ : me ( comment marking my mac/ip address )
\ : to ( comment marking correspondent )
\ : sender
\ : target
\ : dir -b ( fields in either ) message ( or received message )
\ : ip b-n ( fetch ip address )
\ : ar n ( send query 1, or reply 2 )
\ : arp ( broadcast query )
\ : -arp b-b ( return if not ARP. Otherwise process and skip out. )
\ : me? b ( return if broadcast not for me. Save sender only in packet )
\ : query? b ( if a request, reply )

\ Block 180
( ipv4 )
: header align string $4500 n, #0 n, #1 n, #0 n, $FF11 n, #0 n, #0 , #0 ,
```

```
: length ( n ) header #2 + n! ;
: +id header #4 + dup n@ #1 + swap n! ;
: -sum for dup n@ u+ #2 + next drop dup $00010000 / + invert ;
: sum header #10 + n! ;
: checksum 0 sum #0 header #10 -sum sum ;
: source header #12 + ;
: destination header #16 + ;
: ip ( n-n ) dup #20 + $0800 ethernet length +id checksum header #20 send ;
: +ip dup #-2 + n@ $0800 or drop if pop ; then #20 + ;

\ Block 181
\ ( Set ip addresses with Edit. Normal order, net bytes first )
\ : header -a ( 40-byte ipv6 header )
\ : length n ( store 2-byte length in header )
\ : dest -a ( 4-byte destination ip address )
\ : src -a ( source ip )
\ : ip n ( send ip header embedded in ethernet packet )
\ : +ip b-b ( skip out if not IP. Otherwise return payload address )

\ Block 182
( UDP )
: xid 3rom + + ;
: b@ ( b-n ) w@ $FF and ;
: header string xid n, #0 n, #8 n, #0 n, #0 n,
: length ( n ) #8 + header #4 + n! ;
: port header #2 + n! ;
: from? over #-8 + n@ or drop ;
: udp ( n ) dup #8 + ip length ;
: +udp ( b-b ) dup #-11 + b@ #17 or drop if pop ; then #8 + ;

\ Block 183
\
\ : b@ b-n ( fetch byte )
\ : header -a ( 8-byte udp header )
\ : length n ( store length in header )
\ : port p ( set destination port )
\ : from? ap ( udp packet from port ) p ( ? )
\ : udp n ( send ip header for n-byte packet )
\ : +udp b-b ( skip out if not UDP. Otherwise return payload address )

\ Block 184
( DNS resolver ) $0CF42A44 MagentaV server #1671948608 MagentaV host
: msg string #0 , #1 n, #0 2, #0 , #1 n, #1 n,
: ptr? dup n@ $C000 and $C000 or drop ;
: skip ptr? if dup b@ if + #1 + skip ; then drop #1 + ; then #2 + ;
: length dup negate swap skip + ;
: 4! a! w! ;
: query server @ destination 4! #53 port dup length dup #16 + udp drop header #8 send msg #12 send send msg #12 + #4 last
;
: answer dup #12 + skip #4 + swap #6 + n@ ;
: resolve ( a-h ) #0 host ! query
: wait host @ #0 or if ; then drop wait ;
: rr+ #8 + dup n@ + #2 + ;
: -dns #53 from? if ; then pop drop answer
: rr #-1 + -if #-1 host ! ; then swap skip dup n@ #1 or drop if rr+ swap rr ; then
: address #10 + dup w@ host ! ;

\ Block 185
\ ( Assumtions )
\ : 1 ( a response contains one entry in the question section )
\ : 2 ( the first address in the answer section, if any, sufficiently resolves the query )
\ : server ( name server )
\ : host ( the resolved IP address )
\ : skip a-b ( skip past a domain field )
\ : length a-n ( length of a domain in bytes )
\ : query a- ( send DNS query to the DNS server )
\ : answer a-bn ( give the answer section and the number of resource records )
\ : resolve a-h ( resolve domain name to host address )
\ : wait -h ( wait for a response from the server )
\ : rr+ a-b ( skip a resource record )
\ : -dns ( dns packet recieved , search for address )
```

\ : rr a-b (process resource record)
\ : address a-b (set the host address)

\ Block 186
(Domain names) #62 load macro
: 1! a! $0288 2, drop ;
: interp qdup $F889 2, ; forth
: word ch if 1, #1 u+ word ; then drop drop ;
: . here #0 1, interp #0 over @ #-16 and word #1 u+
: words over @ $0F ? if drop nip swap 1! ; then word #1 u+ words ;
: end #0 1, ;
: cf string . (www) . (colorforth) . (com) end
: google string . (www) . (google) . (com) end
: none string . (none) end

\ Block 187
\
\ : 1! xa- (write byte at byte address)
\ : interp -a (word address of next word to be interpreted)
\ : word w- (compile packed word as ASCII characters)
\ : . (compile counted ASCII string)
\ : words an- (compile extentions words as ASCII)
\ : end (of domain)
\ : none (test of a non-existant domain)

\ Block 188
(DHCP client)
: fill for #0 , next ;
: msg align string $00060101 , xid , #5 fill 3rom 2, 2, 2, #0 2, #50 fill $6382 n, $5363 n, $00010135 3, $06030237 , #12 1, . (colorforth) $FF 2, #0 , $3204 n, #0 , $FF 1,
: eq over over or drop ;
: skip over #1 + b@ #2 + u+ ;
: find over b@ if eq if $FF or if drop skip find ; then then drop drop #2 + ; then drop #1 u+ find ;
: your #16 + w@ ;
: ack dup #6 find w@ server ! #3 find w@ message target #6 + 4! your dup source 4! message sender #6 + 4! #1 ar ;
: -dhcp #67 from? if ; then dup #4 + w@ xid or drop if ; then dup #240 + dup #53 find w@
: type #2 or if #7 or drop if ack then drop ; then drop
: offer #54 find w@ msg #261 + 4! your msg #267 + 4!
: request #272 $3604 $0103 msg #241 + n!
: bootp msg #259 + n! broadcast #-1 destination 4! #67 port udp header #8 send msg swap last ;
: discover #260 $FF00 bootp ;

\ Block 189
\
\ : xid -v (a unique identifier used in all DHCP correspondence with this client)
\ : fill n (fill) n (words)
\ : msg (the DHCP message , both discover and request are contained , discover is ends at) $FF 2,
\ : eq xy-xy (test equality)
\ : skip at-bt (skip DHCP option)
\ : find at-b (find option of type) t (in option list)
\ : your a-h (IP address)
\ : ack ao (server acknowledge , assign your IP , router IP , and DNS server IP)
\ : -dhcp a (receive DHCP packet with) xid
\ : type aot (recieve offer) 2 (or ack) 5
\ : offer ao (recieved an offer , send a request)
\ : request (request the offered parameters)
\ : bootp nt (send a discover or request message)
\ : discover (broadcast a discover message)

\ Block 190
(ICMP)
: header string $0800 n, $00 n, $00 ,
: icmp dup #-34 + b@ #1 or drop if ; then ;
: ping #8 ip header #8 last ;

\ Block 191
\ (Client can get or put blocks to server)
\ : payload n-bn (2 bytes were appended to UDP header for block number)
\ : +put nn (send block number. Append block as last fragment. Packet length distinguishes two messages)
\ : it b (move 1024 bytes from packet to offset block)
\ : -got b-b (if a 2-byte message, return. Otherwise move block to archive - 300+ - and skip out)

```
\ : receive ( check and decode received packet. ) +test ( returns if true, ) -test ( returns if false. Otherwise they ) pop
\ ( - skip-out - return from ) receive. ( Resulting stack need not be empty, since ) /forth ( will restore pre-interrupt
\ stack. ) pop ( must be in a word called by ) receive, ( it cant be nested )
\ : +get b ( send requested block from archive )
\ : get n ( send block number to request. Interrupt disabled lest reply interfer )
\ : put n ( send block )
\ : archive ( send blocks 0-161 - 9 cylinders ) icmp dhcp

\ Block 192
( Blocks to/from server )
: payload ( n-bn ) header #8 + n! header #10 ;
: +put ( nn ) #1026 udp over payload send + block 2* 2* #1024 last ;
: it ( b ) dup #2 + swap n@ #300 + block 2* 2* #1024 move ;
: -got ( b-b ) dup #-4 + n@ #2 #8 + or drop if it pop ; then ;
: receive +ethernet -arp +ip +udp -dns -dhcp -got
: +get ( b ) n@ #300 +put ;
: ... ( interrupt-protect words that transmit )
: get ( n ) cli #2 udp payload last sti ;
: put ( n ) cli #0 +put sti ;
: archive #161 for i put #1000 us -next ;

\ Block 193
\ ( Client can get or put blocks to server )
\ : payload n-bn ( 2 bytes were appended to UDP header for block number )
\ : +put nn ( send block number. Append block as last fragment. Packet length distinguishes two messages )
\ : it b ( move 1024 bytes from packet to offset block )
\ : -got b-b ( if a 2-byte message, return. Otherwise move block to archive - 300+ - and skip out )
\ : receive ( check and decode received packet. ) +test ( returns if true, ) -test ( returns if false. Otherwise they ) pop
\ ( - skip-out - return from ) receive. ( Resulting stack need not be empty, since ) /forth ( will restore pre-interrupt
\ stack. ) pop ( must be in a word called by ) receive, ( it cant be nested )
\ : +get b ( send requested block from archive )
\ : get n ( send block number to request. Interrupt disabled lest reply interfer )
\ : put n ( send block )
\ : archive ( send blocks 0-161 - 9 cylinders ) icmp dhcp

\ Block 194
( Format floppy ) empt forth #1 MagentaV hd
: array pop 2/ 2/ ;
: com align array $1202004D , $6C 2,
: done $03F4 a! p@ $D0 or drop if done ; then ;
: byte ( n ) ready p! ;
: sectors ( nn-n ) #18 for over byte hd @ byte dup #18 mod #1 + byte #2 byte #1 + next drop ;
: head ( nn-n ) dup hd ! $0400 * $1202004D + com ! seek com #6 command dup 2* - #1801 + sectors done ;
: cylinders ( n ) #0 swap for #0 head #1 head #1 + next stop drop ;
: format #12 cylinders ;

\ Block 195
\ ( Increase speed from 2 cylinders/s to 3 )
\ : array -a ( return next word address )
\ : com -a ( address of command string )
\ : done ( wait till last sector formatted. Till ready to read )
\ : byte n ( send byte to fdc when ready )
\ : sectors nn-n ( send 4 format bytes to each of 18 sectors. Sector number from 1 to 18 )
\ : head nn-n ( set head number. Issue seek and format commands. Starting sector number depends on cylinder, allowing 2 sector
\ times to step heads. Cylinder 1: 17 18 1 2 ... 16. 1801 + adjusts for 1s complement and for unsigned mod )
\ : cylinders n ( format both heads of each cylinder, starting at 0 )
\ : format ( standard number of cylinders. Smaller is faster )

\ Block 196
( Hard disk ) empt macro ( use this at your own ) risk
: 2/s ?lit $F8C1 2, 1, ;
: p!+ $42EE 2, ;
: 1! $91 1, drop ;
: insw 1! $97 1, $006DF266 3, $97 1, ;
: outsw 1! $96 1, $006FF266 3, $96 1, ; forth
: 2dup over over ;
: bsy $01F7 p@ $80 and drop if bsy ; then ;
: rdy ( -n ) $01F7 p@ #8 and drop if $01F0 a! #256 ; then rdy ;
: sector $01F3 a! swap p!+ #8 2/s p!+ #8 2/s p!+ #8 2/s $E0 or p!+ drop p!+ drop 2* 2* ;
: read ( an ) $20 sector #256 for rdy insw next drop ;
: write ( an ) bsy $30 sector #256 for rdy outsw next drop ; nload
```

```
\ Block 198
( boot: 3f fat0: 5f fat1: 25a5 dir: 2 cl forth: 8e6d cl )
: reg dup p@ $FF and #2 h.n space #3 h.n cr ;
: regs #7 for i $01F0 + reg -next ;
: ok show blue screen text regs keypad ;
: cl $20 * $4AAB + ;
: buffer $2000 block ;
: ?fort dup @ $54524F46 or drop ;
: cl0 dup #5 + @ $00010000 * swap #6 + @ #16 2/s $FFFF and or ;
: find ( -n ) buffer dup #2 cl read #256 for ?fort if #8 + *next drop ; then cl0 pop drop ;
: fort $8E6D cl ;
: +2 $8000 u+ $0100 + ;
: reads for 2dup read +2 next drop drop ;
: writes for 2dup write +2 next drop drop ;
: get buffer fort #9 reads ;
: cf! #0 fort #2 writes ;

\ Block 200
( Deskjet ) empty #2 +load
: nb #768 #3 * ; #4 +load
: pixels for pix next drop drop ;
: drow string $33622A1B , $622A1B4D , $5730 2,
: rpt drow #10 type drop ;
: columns for $0264 #2 wipes dup buffer #8 * #768 pixels line rpt rpt #2 + next drop ;
: res #300 2, #300 2, #2 2, ;
: esci string $306C261B , $6F2A1B4C , $1B4D312E , $3033742A , $2A1B5230 , $55342D72 , ( 32672a1b 4025736 res res res res
) $32722A1B , $53343033 , $30722A1B , $722A1B41 , $000C4362 3,
: print esci #37 type $F0000000 #767 #1024 * #2 * + #1024 columns #6 type drop ;
: tx string $3F and if $3F or if ; then $C0 or ; then ;
: text tx map ! print ;
: it table map ! print ;

\ Block 202
( Printer ) macro
: p@ $EC 1, ;
: p! $EE 1, ;
: @w $8B66 3, ;
: @b $8A 2, ;
: +a $C2FF 2, ;
: bts $0010AB0F 3, drop ;
: 2/s ?lit $F8C1 2, 1, ; forth
: ready p@ $80 and if ; then ready ;
: delay for next ;
: emit $0378 a! p! +a ready +a $8D or p! #30 delay #1 or p! drop ;
: type for dup @b emit #1 + next ;
: buffer $0264 block #4 * ;
: string pop ;
: !b dup - #7 and a! dup #3 2/s bts #1 + ;
: three !b
: two !b
: one !b
: nul drop ;
: white $FFFF and dup $FFFF or drop if - then ;

\ Block 204
( Deskjet )
: -nb nb negate u+ ;
: bcmy string $10243800 , $3033 , $00200022 , $10000011 , $C00F , $4003 , $00 , $00 , $0008000A , $00 , $00800002 , $00 ,
$04000005 , $00 , $00 , $C0000001 ,
: ye nb #3 * u+
: all over over #3 and jump nul one two three
: ma -nb #2 2/s all ;
: cy -nb #2 2/s all ;
: bl -nb #2 2/s all ; #1050918 MagentaV map
: 6b $C618 and #3 2/s dup #3 2/s or $03C3 and dup #4 2/s or $3F and ;
: table string bcmy + @b ;
: ex map @ push ;
: pix over @w 6b ex $FF and if ye ma cy bl then drop #3 + #1024 #-2 * u+ ;
: arow string $30622A1B , $4D 1,
: trbp string $32622A1B , $00563838 3,
```

```
: trbr string $32622A1B , $00573838 3,
: color #7 type drop nb #8 / type ;
: line arow #5 type drop buffer #3 for trbp color next trbr color drop ;

\ Block 206
( x18 simulator ) empty macro
: 2/s ?lit $F8C1 2, 1, ; forth
: state $1FFF block ; nload
: reset r #26 for $00100000 over ! #1 + next drop $0180 mem @ ir ! $0181 pc ! $00 slot ! ;
: un. #5 for #37 emit next ;
: undef $00100000 ? if drop un. ; then #5 h.n ;
: r. ( a-a ) dup @ undef cr #1 + ;
: stack sp @ $08 for dup ss r. drop #-1 + next drop ;
: return rp @ #8 for #1 + dup rs r. drop next drop ;
: ok show black screen text green return r r. blue r. r. white r. r. green r. r. drop stack keypad ;

reset ok

\ Block 207
\
\ : 2/s n ( shift right n bits )
\ : state -a ( address of state vector for current computer )
\ : reset ( set registers undefined, execute from ROM )
\ : un. ( display undefined register )
\ : h.n nn ( display n hex digits of number )
\ : undef n ( bit 20 set means undefined )
\ : r. ( display register )
\ : stack ( display stack, top at top )
\ : return ( display return stack, top at bottom )
\ : ok ( display registers, b a blue, pc ir white )

\ Block 208
( Registers )
: r state ;
: b state #1 + ;
: ar state #2 + ;
: pc state #3 + ;
: ir state #4 + ;
: t state #5 + ;
: s state #6 + ;
: slot state #7 + ;
: ss #7 and #8 + state + ;
: rs #7 and #16 + state + ;
: rp state #24 + ;
: sp state #25 + ;
: mem $2000 block + ; #4 +load #2 +load
: s1 ir @ #8 2/s inst ;
: s2 ir @ #3 2/s inst ;
: s3 #0 slot ! ir @ #4 and drop if ret then pc @ mem @ ir ! #1 pc +!
: s0 ir @ #13 2/s inst ;
: step slot @ jump s0 s1 s2 s3
: steps for step next ;

\ Block 209
\ ( Name 26 registers in state vector )
\ : ar -a ( A register. Cannot be named a because Pentium macro takes precedence )
\ : s0-s3 ( execute instruction from slot 0-3 )
\ : step ( execute next instruction )
\ : steps n ( execute n instructions )

\ Block 210
( Instructions )
: nul ;
: call pc @ +r
: jmp ir @ $01FF and pc ! ;
: jz t @ dup or
: jc drop if #3 slot ! ; then jmp ;
: jns t @ $00020000 and jc ;
: ret -r pc ! ;
: @b b @
: @x mem @ +t ;
```

```
: @+ ar @ #1 ar +! @x ;
: n pc @ #1 pc +! @x ;
: @a ar @ @x ;
: !b b @ #1 b +!
: !x -t swap mem ! ;
: !+ ar @ #1 ar +! !x ;
: !a ar @ !x ;
: inst ( n ) #1 slot +! $1F and jump jmp jmp call call jz jz jns jns @b @+ n @a !b !+ nul !a -x 2*x 2/x +* orx andx nul +x
r@ a@ t@ s@ r! a!x nul t!

\ Block 211
\ ( Define action of each instruction )
\ : inst n ( jump vector for 32 instruction codes )

\ Block 212
( Instructions )
: +r ( n ) r @ rp @ #1 + dup rp ! rs ! r ! ;
: -r ( -n ) r @ rp @ dup rs @ r ! #-1 + rp ! ;
: +t ( n ) t @ s @ sp @ #1 + dup sp ! ss ! s ! t ! ;
: -t ( -n ) t @ s @ t ! sp @ dup ss @ s ! #-1 + sp ! ;
: -x t @ $0003FFFF or t ! ;
: 2*x t @ 2* $0003FFFF and t ! ;
: 2/x t @ dup $00020000 and 2* or 2/ t ! ;
: +* t @ #1 ? if s @ + then 2/ t ! ;
: orx -t t @ or t ! ;
: andx -t t @ and t ! ;
: +x -t t @ + $0003FFFF and t ! ;
: r@ -r +t ;
: a@ ar @ +t ;
: t@ t @ +t ;
: s@ s @ +t ;
: r! -t +r ;
: a!x -t ar ! ;
: t! -t drop ;

\ Block 213
\
\ : +r n ( push onto return stack )
\ : -r -n ( pop from return stack )
\ : +t n ( push onto data stack )
\ : -t -n ( pop from data stack )
\ : -x ( some instructions named with terminal x to avoid Pentium conflict )

\ Block 214
( x18 target compiler ) empt #2097556 MagentaV h #2097555 MagentaV ip #2 MagentaV slot macro
: 2*s ?lit $E0C1 2, 1, ; forth
: memory $2000 block ;
: org ( n ) memory + dup h ! ip ! #0 slot ! ;
: , ( n ) h @ ! #1 h +! ;
: s3
: s0 h @ ip ! #13 2*s , #1 slot ! ;
: s1 #8 2*s
: sn ip @ +! #1 slot +! ;
: s2 #3 2*s sn ;
: i, slot @ jump s0 s1 s2 s3
: 25x #174 load ; #8 +load #2 +load #4 +load n x18 call class 25x

\ Block 215
\ ( Prototype for target compilers )
\ : h ( address of next available word in target memory )
\ : ip ( address of current instruction word )
\ : slot ( next available instruction slot )
\ : 2*s n ( shift left n bits )
\ : memory -a ( host address for target memory )
\ : org n ( set current target memory location )
\ : , n ( compile word into target memory )
\ : s0-s3 ( assemble instruction into slot 0-3 )
\ : i, ( assemble instruction into next slot )
\ : 25x ( compile code for multicomputer )

\ Block 216
```

```
( Instructions )
: nop $1E i, ;
: adr ( n-a ) slot @ #2 or drop if nop then i, ip @ ;
: call defer ( a ) #2 adr +! ;
: if ( -a ) #4 adr ;
: -if ( -a ) #6 adr ;
: then ( a ) h @ $01FF and swap +! ;
: @+ $08 i, ;
: @b $09 i, ;
: n defer #8 f@ execute $0A i, , ;
: @ $0B i, ;
: !+ $0C i, ;
: !b $0D i, ;
: ! $0F i, ;
: - $10 i, ;
: 2* $11 i, ;
: 2/ $12 i, ;
: +* $13 i, ;
: or $14 i, ;
: and $15 i, ;
: + $17 i, ;

\ Block 217
\ ( Words being redefined for the target computer. These Pentium words can no longer be executed. Although Pentium macros
\ still take precedence during compilation, they will no longer be used. )
\ : adr n-a ( assembles instruction, but not in slot 2, where address goes. Instruction address left on stack )
\ : call ( deferred to class. Executed for target defined words )
\ : then a ( puts address in low 9 bits of previous instruction word )
\ : n ( executed for green short-numbers. All 18-bit target numbers are short. Executes white short-number to put interp
\reted number on stack. Then assembles literal instruction with number in next location )

\ Block 218
( Instructions )
: pop $18 i, ;
: a $19 i, ;
: dup $1A i, ;
: over $1B i, ;
: push $1C i, ;
: a! $1D i, ;
: drop $1F i, ;
: ; #4 ip +! ;

\ Block 219
\ ( More target instructions )
\ : ; ( since it will be executed, it does not conflict with the Pentium macro )

\ Block 220
( 25x ROM ) $0180 org $00 dup - dup - dup - dup - dup - dup - dup - dup - dup push push push push push push push push push
a! a nop

\ Block 222
( Target )
: defer ( -a ) pop ;
: execute ( a ) push ;
: class ( a ) last #1 + ! ;
: f! ( an ) sp + ! ;
: f@ ( n-a ) sp + @ ; #1445 MagentaV ?com #1369 MagentaV csho
: empty empt #0 class csho @ ?com @
: functions ( aa ) #4 f! #6 f! ;
: x18 ( a ) #4 f@ ?com ! #6 f@ csho ! #1 f@ functions ;

\ Block 224


\ Block 225
\

\ Block 226
( Realtek rtl8139b ) macro
: move ( sdn ) $C189 2, drop $00C78957 3, drop $00C68956 3, $A4F3 2, $5F5E 2, drop ; forth
: 1us #1
```

```
: us ( n ) #550 #3 / * for next ;
: r ( n-a ) $02000000 device $14 + pci + 2/ 2/ ;
: rom ( a-n ) r @ ;
: 3rom ( nnn ) #4 rom #0 rom dup #16 for 2/ next swap ;
: tx ( -b ) $2000 block #4 * ;
: rx ( -b ) tx #1536 + ; #1 MagentaV ds #42 MagentaV fr
: n ( -a ) ds @ $10 r + ;
: send ( an ) fr @ tx + swap dup fr +! move ;
: first ( an ) n @ $2000 and drop if ds dup @ #1 + #3 and swap ! #0 fr ! send ; then first ;
: last ( an ) send tx ds @ $20 r + ! fr @ #60 max n ! ;
: reset $10000000 $34 r ! #100 us ;
: init rx $30 r ! 1us reset $0C000000 $34 r ! 1us $8A $44 r ! #3 ds ! $FB dup $21 p! $A1 p! sti
: /int $FFFF0001 $3C r ! ;
: rcvd ( -b ) $38 r @ dup $00010000 / $1FFF and $FFFFFFF0 + $38 r ! $10 + $1FFF and rx #4 + + ;

\ Block 227
\
\ : move sdn ( move n bytes from source to destination. Register 1 is used, 6 and 7 are saved )
\ : us n ( delay n microseconds. Edit cpu clock rate )
\ : r n-a ( word address of register )
\ : rom a-n ( fetch 2 bytes of mac )
\ : 3rom nnn ( 3 byte-pairs of mac )
\ : tx -a ( transmit buffer. 1536 bytes. Fragments must be assembled for transmission )
\ : rx -b ( receive buffer. 8k + 1532 byte overrun )
\ : ds -a ( must cycle thru 4 tx descriptors )
\ : fr -a ( must accumulate fragments in tx buffer )
\ : n -a ( tx status/length. Writing starts transmission )
\ : send an ( fragment into transmit buffer )
\ : first an ( fragment. Wait till buffer empty )
\ : last an ( fragment. Start transmission )
\ : reset ( controller )
\ : init ( ialize controller. Set tx/rx address/on and mac/broadcast. Enable irq10 )
\ : rcvd -b ( received packet. Register 38 is 10 bytes before start of next packet. Register 3a is end of current packet
\ )

\ Block 228
( Display registers )
: reg ( a ) dup r @ h. space #2 h.n cr ;
: regs $48 #19 for dup reg #-4 + next drop ;
: ok show red screen text regs keypad ;

\ Block 229
\
\ : reg a ( display register and address )
\ : regs ( display interesting registers )
\ : ok ( diagnostic display )
\ : 48 ( counter. Neat! )
\ : 44 ( rx configuration )
\ : 40 ( tx configuration )
\ : 3c ( interrupt )
\ : 38 ( rx count/address )
\ : 34 ( command )
\ : 30 ( rx 8k ring buffer )
\ : 2c-20 ( tx address )
\ : 1c-10 ( tx status )
\ : c-8 ( multicast id, unused )
\ : 4 ( mac 54 )
\ : 0 ( mac 3210 )

\ Block 230
( Ethernet ) empty #124 load
: empty empt logo cli ; macro
: w $66 1, ;
: w@ $8B 2, ;
: w! w $0289 2, drop ;
: *byte $C486 2, ; forth #126 load #128 load
: n@ w w@ $FFFF and *byte ;
: 2! a! w! ;
: n! a! *byte w! ;
: n, *byte 2, ;
: string pop ;
```

```
: packet string #-1 dup dup 2, 2, 2, 3rom 2, 2, 2, #0 n,
: length ( n ) packet #12 + n! ;
: 3! swap over 2! #2 + swap over 2! #2 + 2! ;
: ethernet ( n ) length packet #14 first ;
: +ethernet ( -a ) rcvd #14 + ; #132 load #134 load #136 load #138 load $72 interrupt
: serve forth receive /int 8clear /forth i; init ok

\ Block 231
\
\ : empty ( redefined to disable interrupts )
\ : w ( 16-bit prefix )
\ : w@ b-n ( fetch 16-bits from byte address )
\ : w! nb ( store 16-bits )
\ : *byte n-n ( swap bytes 0 and 1 )
\ : n@ b-n ( fetch 16-bit network-ordered number )
\ : 2! nb ( store 16-bit number )
\ : n! nb ( store 16-bit number in network order )
\ : n, n ( compile 16-bit number in network order )
\ : string -b ( returns byte address )
\ : packet -b ( ethernet packet header )
\ : dest -b ( destination field in packet )
\ : src -b ( source field )
\ : length n ( store length into packet )
\ : 3! nnnb ( store 3-word MAC )
\ : ethernet n ( send header with type/length )
\ : @ethernet -b ( return payload address of received packet )

\ Block 232
( ARP for a single correspondent )
: . ( n ) 1, ;
: message string $01 n, $0800 n, $06 . $04 . $01 n,
: me 3rom 2, 2, 2, ( IP ) #0 . #0 . #0 . #2 .
: to #0 #0 #0 2, 2, 2, ( IP ) #0 . #0 . #0 . #1 .
: sender #8 + ;
: target #18 + ;
: dir #6 + ;
: ip #6 + w@ ;
: ar ( n ) message dir n! $0806 ethernet message #28 last ;
: arp cli #-1 dup dup packet 3! #1 ar sti ;
: -arp ( b-b ) dup #-2 + n@ $0806 or drop if ; then pop drop
: me? dup target ip message sender ip or drop if ; then dup sender packet #6 move
: query? dup dir n@ #1 or drop if ; then sender message target #10 move #4 ar ;

\ Block 233
\ ( Set ip addresses with Edit. Normal order, net bytes first )
\ : . n ( compile byte. Resembles URL punctuation )
\ : message -b ( 28-byte string )
\ : me ( comment marking my mac/ip address )
\ : to ( comment marking correspondent )
\ : sender
\ : target
\ : dir -b ( fields in either ) message ( or received message )
\ : ip b-n ( fetch ip address )
\ : ar n ( send query 1, or reply 4 )
\ : arp ( broadcast query )
\ : -arp b-b ( return if not ARP. Otherwise process and skip out. )
\ : me? b ( return if broadcast not for me. Save sender only in packet )
\ : query? b ( if a request, reply )

\ Block 234
( ipv6 )
: header string $01000060 , $00 n, $17 . #64 .
: to $00 , $00 , $00 , ( IP ) #0 . #0 . #0 . #2 .
: me $00 , $00 , $00 , ( IP ) #0 . #0 . #0 . #1 .
: length ( n ) header #4 + n! ;
: dest header #20 + ;
: src header #36 + ;
: ip ( n ) $86DD ethernet length header #40 send ;
: +ip ( b-b ) dup #-2 + n@ $86DD or drop if pop ; then #40 + ;

\ Block 235
```

```
\ ( Set ip addresses with Edit. Normal order, net bytes first )
\ : header -a ( 40-byte ipv6 header )
\ : length n ( store 2-byte length in header )
\ : dest -a ( 4-byte destination ip address )
\ : src -a ( source ip )
\ : ip n ( send ip header embedded in ethernet packet )
\ : +ip b-b ( skip out if not IP. Otherwise return payload address )

\ Block 236
( UDP )
: b@ ( b-n ) w@ $FF and ;
: header string #0 n, #0 n, #8 n, #0 n, #0 n,
: length ( n ) #8 + header #4 + n! ;
: udp ( n ) dup #8 + ip length ;
: +udp ( b-b ) dup #-34 + b@ $17 or drop if pop ; then #8 + ;

\ Block 237
\
\ : b@ b-n ( fetch byte )
\ : header -a ( 8-byte udp header )
\ : length n ( store length in header )
\ : udp n ( send ip header for n-byte packet )
\ : +udp b-b ( skip out if not UDP. Otherwise return payload address )

\ Block 238
( Blocks to/from server )
: payload ( n-bn ) header #8 + n! header #10 ;
: +put ( nn ) #1026 udp over payload send + block 2* 2* #1024 last ;
: it ( b ) dup #2 + swap n@ #300 + block 2* 2* #1024 move ;
: -got ( b-b ) dup #-4 + n@ #2 #8 + or drop if it pop ; then ;
: receive +ethernet -arp +ip +udp -got
: +get ( b ) n@ #300 +put ;
: ... ( interrupt-protect words that transmit )
: get ( n ) cli #2 udp payload last sti ;
: put ( n ) cli #0 +put sti ;
: archive #161 for i put #1000 us -next ; lblk @ edit

\ Block 239
\ ( Client can get or put blocks to server )
\ : payload n-bn ( 2 bytes were appended to UDP header for block number )
\ : +put nn ( send block number. Append block as last fragment. Packet length distinguishes two messages )
\ : it b ( move 1024 bytes from packet to offset block )
\ : -got b-b ( if a 2-byte message, return. Otherwise move block to archive - 300+ - and skip out )
\ : receive ( check and decode received packet. ) +test ( returns if true, ) -test ( returns if false. Otherwise they ) pop
\ ( - skip-out - return from ) receive. ( Resulting stack need not be empty, since ) /forth ( will restore pre-interrupt
\ stack. ) pop ( must be in a word called by ) receive, ( it cant be nested )
\ : +get b ( send requested block from archive )
\ : get n ( send block number to request. Interrupt disabled lest reply interfer )
\ : put n ( send block )
\ : archive ( send blocks 0-161 - 9 cylinders )

\ Block 240
( ipv4 )
: header align string $4500 n, #0 n, #1 n, #0 n, $FF00 n, #0 n, #0 , #0 ,
: length ( n ) #20 + header #2 + n! ;
: +id header #4 + dup n@ #1 + swap n! ;
: checksum ;
: source header #12 + ;
: destination header #16 + ;
: ip ( n-n ) dup #20 + $0800 ethernet length +id checksum header #20 send ;

\ Block 242
( Howerds test block ) empty macro
: gtend $7E 1, here invert + 1, ;
: init $B803F0BA , $EEEE0055 , ; forth
: h $01E5 ; ( h last class macros forths )
: allot ( n- ) h +! ;
: mk2 here $10 + ; $40 allot
: mk $01E2 ;
: class $01E9 ;
: macros $01EA ;
```

```
: forths $01EB ;
: mk macros @ mk2 ! forths @ mk2 #1 + ! h @ mk2 #2 + ! ;
: mt mk2 @ macros ! mk2 #1 + @ forths ! mk2 #2 + @ h ! ;
: reload #0 push ;
: qkey #3 for i next ; #57 MagentaV ky
: key pause $64 p@ #1 and drop if $60 p@ dup $3A - drop -if ky ! ; then drop then key ;
: kk key ky @ #57 - drop if kk then ;
: pt $03F0 ; here $04 / $12345678 , ,
: conf cli init $00 pt p! pt $01 + p@ $01 pt p! pt $01 + p@ ;

\ Block 243
\
\ : kk ( shows key values . press esc to exit )

\ Block 244
( IR remote ) empty macro
: 2/s ?lit $F8C1 2, 1, ;
: p@ $EC 1, ;
: p! $EE 1, drop ;
: 1@ $8A 2, ;
: 1! a! $0288 2, drop ; forth
: ba #10 /mod $011F a! p! $0118 + a! ;
: b@ ba #0 p@ ;
: b! ba p! ;
: us #748 * time + -
: till dup time + drop -if till ; then drop ;
: ms #1000 * us ;
: array pop #2 2/s ;
: nul ; #3 MagentaV onf #145 load #146 load #50 load #147 load #148 load #149 load #150 load #151 load #152 load #153 load
#155 load #154 load
: h keypd nul nul quit bye +db -db mute nul +xx -ch jp vcr tv0 dvd cd fm nul nul nul nul nul nul nul nul nul nul nul
$00152500 , $00091016 , $11001016 , $0E0A1002 , #0 , #0 , #0 ,

\ Block 245
\ ( smsc ircc2.0 IR Consumer mode ) $32 #10 b! #0 #12 b! #0 #20 b!
\ : buffer #200 block #4 * ;
\ : reset $10 #7 b! $80 #4 b! ;
\ : on $40 #5 b! ;
\ : off #2 #4 b! #200 ms ;
\ : emit #6 b@ $40 and drop if emit ; then #0 b! ;
\ : rdy #6 b@ $80 and drop ;
\ : get #0 b@ over 1! #1 + ;
\ : bytes for
\ : byte rdy if get dup buffer #4096 + or drop if byte ; then drop pop drop ; then next drop ;
\ : r #200 #1 wipes $80 dup #4 b! #5 b! buffer #1000000 bytes #0 #5 b! ;
\ : word - #4 for dup emit #8 2/s next drop ;
\ : cmd for word next #1
\ : sp for #0 word next ;
\ : rate #22 b! #21 b! ;
\ : sync $80 #20 b! ;

\ Block 246
( App: Slime : simple game ) empty ( sounds ) #4 +load

macro
: @w $8B66 3, ; forth #2 MagentaV speed #13631840 MagentaV alice #29360784 MagentaV bob #0 MagentaV once #-1048576
MagentaV da
#-16 MagentaV db #17 MagentaV delay #25 MagentaV /del #-1 MagentaV off #0 MagentaV done
: mova da @ alice +! ;
: movb db @ bob +! ;
: qpel ( a- ) @ $00010000 /mod at vframe xy @ $00010000 /mod swap $0400 * + $02 * + @w $FFFF and #0 + if #1 done ! #1 off
! white bomb then ;
: clr #13 #65536 * #16 * #320 + alice ! #28 #65536 * #16 * #688 + bob ! #16 da ! #-16 db ! #0 delay ! #1 off ! #0 done !
#1 #1000 tn
: bgnd silver screen #16 #16 at black #1008 #672 box
: draw $FFFF color alice mova qpel #132 emit red bob movb qpel #133 emit ;
: tick off @ #0 + drop if ; then delay @ #-1 + delay ! -if /del @ delay ! draw click then ;
: b. ( c- ) $30 + 2emit ;
: ok show silver once @ #0 + drop if clr #0 once ! then silver #0 #708 at #600 #768 box #48 #708 at $00FFFF00 color #135
mute @ #0 + drop if #1 + then 2emit #0 emit speed @ #1 + b. tick keypad ; nload x ok h
```

```
\ Block 247
\ ( slime ) empt macro
\ : @w ( 16bit fetch )
\ : speed ( selected speed )
\ : alice ( 16:16 bit xy coordinate of left slug )
\ : bob ( 16:16 bit xy coordinate of right slug )
\ : once ( is set to initialise the game )
\ : mova ( move alice by the value in da )
\ : movb ( move bob by the value in db )
\ : delay ( counts the ticks for each move )
\ : /del ( the reset value for delay )
\ : qpel ( check for slime coloured pixel )
\ : clr ( set alice and bob to start positions )
\ : bgnd ( draw the background )
\ : draw ( the slugs )
\ : tick ( do this every screen update )
\ : ok ( the screen display )

\ Block 248
( Slime keypad )
: +speed #1
: +/-s speed @ + #0 max #9 min speed ! #10 speed @

invert + dup * #7 + #2 / #2 invert + /del ! ;
: -speed #-1 +/-s ;
: down #16 #65536 * da ! ; : up #-16 #65536 * da ! ;
: r #16 da ! ; : l #-16 da ! ;
: d2 #16 #65536 * db ! ; : u2 #-16 #65536 * db ! ;
: r2 #16 db ! ; : l2 #-16 db ! ;
: nul ;
: go #0 off ! ; : stop #-1 off ! ;
: x #1 once ! ;
: t off @ #0 + drop if #0 off ! ; then #-1 off ! ;
: help #249 edit ;
: mutet mute a @ invert mute ! ;
: h keypd nul quit t nul nul nul nul nul l2 u2 d2 r2 x nul stop go nul nul nul nul l up down r -speed help mutet +speed $742E
, #0 , $13121110 , $31302078 , #0 , $13121110 , $2B6E682D ,

\ Block 249
\ ( Slime keypad )
\ : ludr ( move Alice and Bob left up down up )
\ : x ( reset the game )
\ : 0 ( stop the game )
\ : 1 ( start the game )
\ : - ( decrease the speed )
\ : h ( to see this help screen )
\ : m ( mute the sound - on/off )
\ : + ( increase the speed )
\ : . ( quit )
\ : t ( toggle on/off )
\ : slime: ( two players control Alice and Bob. The first to hit any slime or the edges loses. )
\ : credits: ( Coded by Howerd Oakford from an idea by Alan Crawley and Paul Chapman )
\ : tested: ( by Hannah Oakford )
\ : type slime ( to play again )

\ Block 250
( Sounds ) #20 MagentaV tempo #-1 MagentaV mute #90 MagentaV period
: tn ( ft- ) tempo @ * swap #660 #50 */
: hz ( tf- ) push #1000 #1193 pop */
: osc ( tp- ) dup period ! split $42 p! $42 p!
: tone ( t- ) mute @ #0 + drop if drop ; then $4F $61 p! ms $4D $61 p! #20 ms ;
: click #1 #90 osc ;
: t #3 tn ;
: q #8 tn ;
: c #16 tn ;
: 2tone #75 q #50 q ;
: h1 #50 c #54 q #50 q #45 c #60 c ;
: h2 #40 c #45 q #50 q #50 c #45 c ;
: h3 #54 c #60 q #54 q #50 c #45 q #40 q #50 t #45 t #50 t #45 t #45 #12 tn #40 q #40 #32 tn ;
: hh
: handel h1 h2 h3 ;
```

```
: piano #55 #7 for dup q #3 #2 */ next drop ;
: cetk #6 c #10 c #8 c #4 c #6 #32 tn ;
: bomb mute @ #0 + drop if ; then $4F $61 p! #500 for #1000 i invert + split $42 p! $42 p! #1 ms next $4D $61 p! #1 #32 tn
;

\ Block 251
\ ( Sounds )
\ : tempo ( in ms per 1/8 quaver )
\ : mute ( equals -1 to disable sound )
\ : period ( test only - value sent to hardware )
\ : tn ( ft- play f Hz for t * 11 ms )
\ : hz ( tf- play t ms at f Hz )
\ : osc ( tp- play t ms of period p )
\ : tone ( t- play the current tone for t ms )
\ : click ( makes a click )
\ : t ( triplet )
\ : q ( quaver )
\ : c ( crotchet )
\ : 2tone ( 2 tones )
\ : h1
\ : h2
\ : h3
\ : hh
\ : handel ( part of Handels Gavotte )
\ : piano
\ : cetk ( Close Encounters of the Third Kind )
\ : bomb ( - well sort of .... )

\ Block 252
( App: colorforth editor ) empty nload qinit
: eddd jblk @ ok h ( drop ) ;
: edd ( b- ) jblk @ jlast ! jblk ! eddd ; blk @ jblk ! #206 jlast ! eddd

\ Block 253
\ ( The colorforth editor in colorforth )

\ Block 254
( Editor circular buffers ) #0 MagentaV cbn #0 MagentaV ends
: data ( - ) cbn @ $01 invert and cbn ! ;
: ptrs ( - ) cbn @ $01 or cbn ! ;
: heads ( - ) cbn @ $02 invert and cbn ! ;
: tails ( - ) cbn @ $02 or cbn ! ;
: cb@ ( -c ) ends @ cbn @ #8 * rshift $FF and ;
: cb! ( c- ) $FF and cbn @ #8 * lshift ends @ $FF cbn @ #8 * lshift invert and or ends ! ;
: cbnum ( -n ) cbn @ heads cb@ tails cb@ - $FF and swap cbn ! ;
: cbuf ( -a ) r@ $0100 / #2 + cbn @ $01 and + block ;
: tl- ( -n ) cbnum ?f drop 0if $00 ; then tails cb@ cbuf + @ cb@ #1 + cb! ;
: tl+ ( n- ) tails cbnum $FF - drop 0if tl- drop then cb@ $01 - cb! cb@ cbuf + ! ;
: hd@ ( -n ) heads cb@ cbuf + @ ;
: hd- ( -n ) cbnum $00 - drop 0if $00 ; then hd@ cb@ #1 - cb! ;
: hd! ( n- ) heads cb@ cbuf + ! ;
: hd+ ( n- ) cbnum $FF - drop 0if tl- drop then heads cb@ $01 + cb! hd! ; #4 +load

\ Block 255
\
\ : cbn ( bit 0 selects one of two circular buffers. Bit 1 selects head or tail value )
\ : cb@
\ : cb! ( read/write a byte to one of the 4 in ends selected by cbn )
\ : ptrs ( selects the pointer buffer )
\ : data ( selects the data buffer )
\ : heads ( selects the head value )
\ : tails ( selects the tail value )
\ : cbnum ( gives the number of items in the currently selected buffer )
\ : cbuf ( returns the address of the start of both buffers - the next 2 blocks )
\ : tl+
\ : tl-
\ : hd+
\ : hd- ( add or subtract from the head or tail of the currently selected buffer )
\ : ... ( note the tl- in hd+ . if the buffer is full we remove the oldest from the tail )

\ Block 256
```

```
( r App:ay buffer string Undo Display r   i r   s r   t r   l r   f r   d r   0 r   o r   ; r  r r  rr r  rt
r   e r  re r  ra r  rn r  ri r   a r  rc r  rl r  rf r  rd r   n r  r8 r  r; r  t r  tr r   i r  to r  te
r  ta r  tn r   s r  ts r  tc r  tl r  tf r   c r  t0 r  t8 r  t; r  o r   l r  ot r  oo r  oe r  oa r   f
r  oi r  os r  oc r  ol r   d r  od r  o0 r  o8 r  o; r   0 r  er r  et r  eo r  ee r   8 r  en r  ei r  es
r  ec r   ; r  ef r  ed r  e0 r  e8 r  r r  a r  ar r  at r  ao r  rr r  aa r  an r  ai r  as r  rt r  al
r  af r  ad r  a0 r  ro r  a; r  n r  nr r  nt r  re r  ne r  na r  nn r  ni r  ra r  nc r  nl r  nf r  nd
r  rn r  n8 r  n; r  i r  ir r  ri r  io r  ie r  ia r  in r  rs r  is r  ic r  il r  if r  rc r  i0 r  i8
r  i; r  s r  rl r  se r  sn r  ss r  sl r  rf r  s8 r  m r  mt r  me r  rd r  ms r  ml r  md r  m8 r  r0
r  ct r  ce r  cn r  cs r  r8 r  cd r  c8 r  y r  yt r  r; r  yn r  ys r  yl r  yd r  t r  l r  lt r  le
r  ln r  tr r  ll r  ld r  l8 r  g r  tt r  ge r  gn r  gs r  gl r  to r  g8 r  f r  ft r  fe r  te r  fs
r  fl r  fd r  f8 r  ta r  wt r  we r  wn r  ws r  tn r  wd r  w8 r  d r  ds r  ti r  vs r  p r  ps r  b
r  ts r  h r  hs r  x r  xs r  tc r  us r  q r  qs r  0 r  tl r  1 r  1s r  2 r  2s r  tf r  3s r  4 r  4s
r  5 r  td r  6 r  6s r  7 r  7s r  t0 r  8s r  9 r  9s r  j r  t8 r  - r  -s r  k r  ks r  t; r  .s r  z
r  zs r  ; r  : r  !s cccc cccc r  !s r  + r  +s r  @ bbbb r  @ r  ot bbbb r  *s )

\ Block 257
\

\ Block 258
( Display Undo string buffer ) #0 MagentaV jcur #64 MagentaV jblk
: sze ( -n ) $E0 ;
: qinit #0 ends ! $00 ptrs hd! $10000009 data hd! ;
: qnew ( - ) #0 ptrs hd+ ;
: qnum ( -c ) ptrs hd@ ;
: qpop ( -n ) data hd- ptrs hd@ #1 - if hd! ; then hd- drop drop ;
: qpush ( n- ) data hd+ ptrs hd@ $FF - drop 0if drop then ptrs hd@ #1 + hd! #0 #0 MagentaV pos #0 MagentaV lpos
: 2toc ( n-a ) jblk @ block pos @ + + ;
: xtoc? ( -n ) #1 2toc @ $0F and ;
: rtocs ( - ) jcur @ pos !
: ntocs ( -n ) #0 2toc @ $0F and #12 - ?f drop 0if #2 ; then #1 xtoc? ?f drop 0if #1 + then $FF and ;
: ltocs ( -n ) #0 pos !
: ltcs pos @ jcur @ - drop -if pos @ lpos ! ntocs pos +! ltcs drop then jcur @ lpos @ - ;
: mx ( n- ) jcur @ + #0 max #255 min jcur ! ;
: ml ltocs negate mx ;
: mu #8 for ml next ;
: mr rtocs mx ;
: md #8 for mr next ; nload

\ Block 259
\
\ : qinit ( initialises the queue pointers )
\ : qnew ( starts a new string entry )
\ : qnum ( -c number of cells in the top string )
\ : qpop ( -n returns the top cell of the top string )
\ : qpush ( n- stores n in the top string )
\ : ntocs ( number of tokens in the top string )
\ : qq ( n- ) qnew for qnum @ $0100 * $10000009 + qpush next cbnum drop ;
\ : qqq qinit #50 for #5 qq next #3 qq ;
\ : vvv ptrs cbnum data cbnum ;
\ : kk c vvv qpop ptrs hd@ ;
\ : gg cbuf dump ;

\ Block 260
( Editor Display ) #0 MagentaV cblind
: cb cblind @ #0 + drop ; #16 MagentaV state $10 MagentaV state*
: yellow $00FFFF00 color ;
: +txt white $6D emit space ;
: -txt white $6E emit space ;
: +imm yellow $58 emit space ;
: -imm yellow $59 emit space ;
: +mvar yellow $09 emit $11 emit $05 emit $01 emit space ;
: txts string $03010100 , $07060504 , $09090901 , $0F0E0D0C , ( ; )
: tx ( c-c ) $0F and txts + 1@ $0F and ;
: .new state @ $0F and jump nul +imm nul nul nul nul nul nul nul +txt nul nul +mvar nul nul nul ;
: .old state* @ $0F and jump nul -imm nul nul nul nul nul nul nul -txt nul nul nul nul nul nul ;
: state! ( n-* ) dup #0 + drop 0if drop ; then tx cb 0if drop ; then state @ swap dup state ! - drop if .old .new state @
#0 + if dup state* ! then drop then ; nload

\ Block 261
\
\ : state
```

```
\ : state! ( acts on a change of token type. It ignores extension tokens )

\ Block 262
( Editor Display ) macro
: @b $8A 2, ; forth #160 MagentaV jcnt #206 MagentaV jlast #2 MagentaV jcol
: bksp xy @ #22 $00010000 * negate + xy ! ;
: ?.cur jcnt @ #1 + #255 min jcnt ! jcur @ jcnt @ negate + #1 + drop 0if $00FF4040 color bksp $30 emit white then ;
: x xy @ $00010000 / ;
: ?cr x #1000 negate + drop -if ; then
: ncr xy @ #30 + $FFFF and $00030000 xor xy ! ;
: emt ?cr emit ;
: emit emt ;
: emitw unpack if emit emitw ; then space drop drop ;
: emitcs unpack if #48 + emit emitcs ; then space drop drop ;
: dig pop + @b $FF and emit ;
: edig dig $1B1A1918 , $1F1E1D1C , $13052120 , $0E04100A ,
: odig dup $0F and swap 2/ 2/ 2/ 2/ $0FFFFFFF and ; nload

\ Block 263
\
\ : ncr ( new cr -does not get confused with original )

\ Block 264
( CAPITALS HPO 2004 Editor Display )
: .hex odig if .hex edig ; then drop edig ;
: .dec #-1 ? -if negate #35 emit then
: n #10 /mod #-1 ? if .dec edig ; then drop edig ;
: num if $00C0C000 and color cb if #24 emit #21 emit then .hex space ; then color .dec space ;
: txt $00FFFFFF color emitw ;
: blu $FF color emitw ;
: cap $00FFFFFF color unpack #48 + emit emitw ; $00 MagentaV caps?
: caps $00FFFFFF color emitcs #-1 caps? ! ;
: ex bksp caps? @ ?f drop if caps ; then emitw ;
: gw $FF00 color emitw ;
: cw $FFFF color emitw ;
: yw $00FFFF00 color emitw ;
: coly #2 jcol ! ;
: colr #4 jcol ! ;
: colg #5 jcol ! ;
: colm #13 jcol ! ;
: colc #8 jcol ! ;
: colb #14 jcol ! ;
: rot $8B045E8B , $046E892E , $C38B0689 , $C3 1, #1220107268 MagentaV last nload

\ Block 265
\
\ : caps
\ : caps? ( is true if the extension token is CAPITALS )
\ : txt? ( returns true if the last token was text )
\ : .hex
\ : .dec

\ Block 266
( Editor display )
: short push dup 2/ 2/ 2/ 2/ 2/ swap $10 and drop pop num ;
: ys $00FFFF00 short ;
: long push #1 u+ $10 and drop dup @ pop num ;
: yn $00FFFF00 long ;
: gs $FF00 short ;
: gn $FF00 long ;
: var $00FF00FF color emitw #0 gn ;
: x xy @ $00010000 / ;
: rcr x #0 xor drop if cr then ;
: rw xy @ $FFFCFFFD + drop if rcr then $00FF0000 color cb if #41 emit space then emitw ;
: nuld drop ;
: .word ( w- ) dup #-16 and swap $0F and if $00 caps? ! then dup state! jump ex yw yn rw gw gn gs cw ys txt cap caps var
blu nuld nuld ( ; )
: t #0 jcnt ! jblk @ block text #3 lm #1024 rm #3 #3 at $10 state ! $10 state* !
: n dup @ #-1 ? if ?.cur .word #1 + n ; then drop drop $0F state! ; white #103 emit ;
: ok show $00200040 color screen t keypad ; nload
```

```
\ Block 267
\ ( CAPITALSALLTHEWAY! )

\ Block 268
( Editor aaaa bbbb cccc dddd keypad insertion )
: ripple ( a- ) dup dup @ over #1 + @ rot ! swap #1 + ! ;
: toc ( -a ) jblk @ block jcur @ + ;
: toend ( -n ) sze jcur @ - #0 max sze min ;
: del toc @ qpush toc toend for dup ripple #1 + next #0 swap ! drop ;
: dels jcur @ ?f drop 0if ; then ml qnew rtocs for del next ;
: ins ( n- ) sze jcur @ - ?f drop -if ; then jblk @ block sze + toend for #1 - dup ripple next ! ;
: undo qpop ins ;
: undos qnum ?f 0if drop ; then for undo next mr ; #25 MagentaV ky
: key pause $64 p@ #1 and drop if $60 p@ dup $3A - drop -if ky ! ; then drop then key ;
: lst ( n- ) jblk ! ok key drop ; nload

\ Block 269
\ ( Editor main keypad )
\ : ripple ( a- swaps the values at a and a+1 )
\ : bpush
\ : bpop ( push and pop the edit stack TBD )
\ : del ( removes the cell at the current cursor )
\ : dels ( removes the extension cells and one non extension coll before the cursor )
\ : undo ( puts back one cell )
\ : undos ( puts back one word which may have extension cells )

\ Block 270
( Editor keypad cursor )
: btog ( n-n ) dup #1 and drop if #1 invert and dup jblk ! ; then #1 xor dup jblk ! ;
: cbtog cblind @ invert cblind ! ;
: lastb ( n-n ) jlast @ dup jblk ! swap jlast ! ;
: blkld jblk @ $FFFFFFFE and #-32 + drop -if ; then jblk @ load ;
: -blk ( n-n ) #-2 + #18 max dup jblk ! ;
: +blk ( n-n ) #2 + #252 min dup jblk ! ;
: accep drop xx ;
: h keypd nul dels accep undos coly colr colg btog ml mu md mr -blk colm colc +blk colb nul nul nul cbtog nul nul lastb blkld
nul nul nul $00072515 , $2D0D010B , $0110160C , $2B0A0923 , $023A3800 , $03000029 , $3C ,

\ Block 272
( App: Conways Game of Life ) empty nload
: 1cell ( n-- ) #32 /mod adj adj over over at #16 u+
   #16 + box ;
: nocell ( n-- ) drop ;
: draw ( n-- ) dup old @ #1 and jump nocell 1cell
: allcls ( -- ) #1023 for i draw -next ;
: gen ( -- ) #1023 for i tick swap new ! -next #1023 for i new @ i old ! -next ;
: locn ( --n ) row @ #32 * col @ + ;
: cur ( -- ) locn dup old @ $FF * $00FF0000 + color 1cell ;
: back ( -- ) black screen $00303010 color #40 #40 at
   #583 dup box ;
: g ( -- ) show back green allcls gen keypad ;
: s ( -- ) gen show back blue allcls cur keypad ;
: clear ( -- ) #1500 #8 wipes #16 row ! #16 col ! s ;
: t ( -- ) locn old dup @ #1 xor swap ! ;
: col! ( n-- ) col +! col @ #31 and col ! ;
: l1 ( -- ) #-1 col! ;
: r1 ( -- ) #1 col! ;
: row! ( n-- ) row +! row @ #31 and row ! ;
: up1 ( -- ) #-1 row! ;
: dn1 ( -- ) #1 row! ;
: h keypd nul nul quit nul nul nul nul nul l1 up1 dn1 r1 nul nul nul nul glide glid2 glid3 glid4 clear s g t nul nul nul
rando $2E00 , #0 , $13121110 , #0 , $1C1B1A19 , $74677378 , $52000000 , clear glide g h

\ Block 273
\
\ : s ( stop )
\ : g ( go )
\ : t ( toggle the square )
\ : ludr ( left up down right )
\   ( press s to stop then draw a shape using ludr and t to toggle )
\   ( then press g to go or s to single step )
```

```
\ : 1234 ( create gliders which move to the four corners counting clockwise from the top left )

\ ( R loads random numbers )

\ Block 274
( Conways Game of Life ) #16 MagentaV row #16 MagentaV col
: old ( n-a ) cells #1500 block + ;
: new ( n-a ) cells #1504 block + ;
: rando ( -- ) #0 old $03FF for rand over ! cell+ next drop ;
: pos swap #32 /mod swap ;
: val #32 * + swap over old @ #1 and + ;
: up pos swap #31 + #31 and val ;
: dn pos swap #1 + #31 and val ;
: lt pos #31 + #31 and swap val ;
: rt pos #1 + #31 and swap val ;
: nul ;
: n2 #0 ;
: s2 dup old @ #1 and ;
: y2 #1 ;
: tick dup #0 up lt dn dn rt rt up up nip jump n2 n2 s2 y2 n2 n2 n2 n2 n2
: adj ( nn--nn ) swap #17 * #40 + ;
: st ( rc- ) col @ + swap row @ + #32 * + old #1 swap ! ;
: glide ( -- ) #0 $02 st #0 #1 st #0 #0 st #1 #0 st #2 #1 st ;
: glid2 #0 #0 st #0 #1 st #0 #2 st #1 #2 st #2 #1 st ;
: glid3 ( -- ) #0 #2 st #1 #2 st #2 #2 st #2 #1 st #1 #0 st ;
: glid4 ( -- ) #0 #0 st #1 #0 st #2 #0 st #2 #1 st #1 #2 st ;

\ Block 276
( Wave audio SB, 8 bit, mono, no DMA ) empt macro
: pb@ 0 $EC 1, ;
: pb! $EE 1, drop ;
: /8 $0008F8C1 3, ; forth
: +base $0220 + ; ( * )
: ?rd $0E +base a!
: *?rd pb@ $80 ? drop if ; then *?rd ;
: ?wr $0C +base a!
: *?wr pb@ $80 ? drop if *?wr then ;
: dsp@ ?rd $0A +base a! pb@ ;
: dsp! ?wr pb! ;
: ?init dsp@ $AA or drop if ?init ; then ;
: 0dsp #6 +base a! #1 pb! #30 for pb@ drop next #0 pb! ?init $D1 dsp! ; 0dsp
: *dac! $10 dsp! dup dsp! /8 ;
: dac! *dac! *dac! *dac! *dac! drop ;
: length #2 + dup #-1 + @ 2/ 2/ ;
: ?data dup @ $61746164 or drop if length + ?data ; then length ;
: sound #100 block #3 + ?data ; ( * )
: play for dup @ dac! #1 + next drop ;

\ Block 277
\
\ : pb@ ( -n get byte from port )
\ : pb! ( n- put byte to port )
\ : /8 ( n-n shift 8 bit right )
\ : +base ( n-n add base adress )
\ : ?rd ( wait for DSP read ready )
\ : ?wr ( wait for DSP write ready )
\ : dsp@ ( -n read DSP )
\ : dsp! ( n- write DSP )
\ : ?init ( wait until initialized )
\ : 0dsp ( reset 3 us DSP, turn on speaker )
\ : dac! ( n- write 4 byte to DAC )
\ : length ( a-an return length of record )
\ : ?data ( a-an search data record )
\ : sound ( -an return address and length of sound data )
\ : play ( an- play sound )

\ Block 278
( App: colorforth Explorer ) empty #9 MagentaV strt
: ?sze ( a- ) dup #510 block - drop ;
: crs ( n- ) ?f if for cr next #0 then drop ;
: docrs cr strt @ negate #0 max crs ;
```

```
: up1 ( a-a ) ?sze +if ; then $0400 + dup @ $FFFFFFF0 and $5C58BC80 - drop 0if ; then up1 ;
: upn ( n-a ) #0 max #64 block up1 swap ?f if for up1 next ; then drop ;
: ln ( a- ) #4 for cell+ dup @ dup $0F and $01 - ?f drop
   0if drop leave then if dotsf then next drop ;
: .line ( a- ) ?sze +if drop ; then cr dup ablk . ln ;
: lines ( -- ) strt @ #0 max upn #20 strt @ negate #0 max - ?f if for blue dup .line up1 next then drop ;


: marker iconh #11 * #4 - ;
: qok show $4228 color screen #240 #0 at cblk block ln $00 color #0 marker at #1023 marker #30 + box #0 #0 at

docrs lines keypad ;

nload

\ Block 279
\ ( Scans the first cell of each block for App: )

\ ( and displays the first 4 words after App: )
\ : +
\ : - ( step through the applications )
\ : ? ( displays the applications first shadow block )
\ : o ( loads the application )
\ : . ( requires ) .word ( from the editor )
\ : up1 ( takes the address of the start block and steps through even blocks until it finds a token App: )

\ Block 280
( explorer )
: go strt @ #9 + upn ablk noshow ld ;
: md strt @ #1 - #-9 max strt ! ;
: mu strt @ #1 + #512 #4 / min strt ! ;
: qed strt @ #9 + upn ablk dup blk ! edit ;
: qh keypd nul quit go nul qed nul nul qed nul nul nul nul md nul nul mu nul nul nul nul nul nul nul nul nul nul nul nul
$002D2500 , $3F202065 , $00 , $2B20202D , $00 , $00 , $00 , qok qh

\ Block 281
\ ( Scans the first cell of each block for App: )

\ ( and displays the first 4 words after App: )
\ : +
\ : - ( step through the applications )
\ : ? ( displays the applications first shadow block )
\ : o ( loads the application )
\ : . ( requires ) .word ( from the editor )
\ : up1 ( takes the address of the start block and steps through even blocks until it finds a token App: )

\ Block 282
( fix the font )
: glyph ( c--a ) #48 * font @ + ;
: fix ( from to -- ) swap glyph swap glyph $30 cmove ;
: fixa $27 $3A fix $5B $01 fix $5C $02 fix $5D $03 fix

$20 $04 fix ;
: fixb $28 $5B fix $29 $5D fix ;

\ Block 283
\

\ Block 284
( grey = #-29727166  )oken ( test ) ( grey = #19138560  ) ( all tokens )
: tok ( t-n ) $11110000 + ;
: loc #2000 block ;
: set $10 for i #1 - tok loc i + ! next #9 tok loc ! #0 loc #17 + ! loc dump ; set

\ Block 288
( App: Timer Interrupt ) empty

( Interrupts ) #170 load

#0 MagentaV ticks
```

```
: !pit $34 $43 p! ( lo ) $A9 $40 p! ( hi ) $04 $40 p! ; !pit
: pic1! $21 p! ; : pic2! $A1 p! ;
: p@ p@ ; : p! p! ;
: ttb $20 p@ $21 p@ $A0 p@ $A1 p@ ;
: bpic cli ( init ) $11 dup $20 p! $A0 p!

( irq ) $00 pic1! $08 pic2! ( master ) #4 pic1! ( slave ) #2 pic2! ( 8086 mode ) #1 dup pic1! pic2! ( mask irqs ) $8F pic2
! $B8 pic1! ;
: npic cli ( init ) $11 dup $20 p! $A0 p!

( irq ) $20 pic1! $28 pic2! ( master ) #4 pic1! ( slave ) #2 pic2! ( 8086 mode ) #1 dup pic1! pic2! ( mask irqs ) $8F pic2
! $B8 pic1! ;

npic ( Note: npic will break bochs )

$20 interrupt
: timer0 forth: #1 ticks +! clear ;forth i;

( cli to disable interrupts , sti to enable )

sti
: test cli #0 ticks ! #1 secs sti #100 secs cli ;
: tm cli #0 ticks ! ;

lblk @ edit

( Type bye after loading in bochs !!!! )

\ Block 289
\ ( Timer Interrupt )
\ : !pit ( sets up the programable interval timer to )
\    ( 1 khz for a 1 ms tick )
\    ( for a clock of 14.31818 / 12 or 1.19318167 Mhz )
\    ( +/- 400 Hz this is actually 0.99985 +/- 0.0004 )
\    ( ms or about 0.015 percent fast. )
\ : pic1! ( write an octet to interrupt controller 1 )
\ : pic2! ( write an octet to interrupt controller 2 )
\ : !pic ( sets up the PIC chips )

\ $20 interrupt ( is the timer interrupt )
\ : timer0 ( the Forth code to run every timer tick )
\    ( use ) sti ( to enable interrupts, ) cli ( to disable )
\ : test ( run a 100 second test to time the timer )
\    ( interrupt with respect to the Real Time Clock. )
\ : tm ( measure cpu ms in timer ticks )

\ Block 290
( App: Sounds ) jmk #20 MagentaV tempo #0 MagentaV mute #1807 MagentaV period
: tn ( ft- ) tempo @ * swap #660 #50 */
: hz ( tf- ) push #1000 #1193 pop */
: osc ( tp- ) dup period ! split $42 p! $42 p!
: tone ( t- ) mute @ #0 + drop if drop ; then $4F $61 p! ms $4D $61 p! #20 ms ;
: click #1 #90 osc ;
: t #3 tn ;
: q #8 tn ;
: c #16 tn ;
: 2tone #75 q #50 q ;
: h1 #50 c #54 q #50 q #45 c #60 c ;
: h2 #40 c #45 q #50 q #50 c #45 c ;
: h3 #54 c #60 q #54 q #50 c #45 q #40 q #50 t #45 t #50 t #45 t #45 #12 tn #40 q #40 #32 tn ;
: hh
: handel h1 h2 h3 ;
: piano #55 #7 for dup q #3 #2 */ next drop ;
: cetk #6 c #10 c #8 c #4 c #6 #32 tn ;
: bomb mute @ #0 + drop if ; then $4F $61 p! #500 for #1000 i - + split $42 p! $42 p! #1 ms next $4D $61 p! #1 #32 tn ; 2tone
jmt

\ Block 292
( App: Test block : ) empty #0 MagentaV strt #62976 MagentaV lstup
: sze #256 block ;
```

: crs (n-) ?f if for cr next ; then drop ;
: up1 (a-a) dup sze - drop +if ; then $0100 + dup @ $FFFFFFF0 and $5C58BC80 - drop 0if dup lstup ! ; then up1 then ;
: .line (a-) dup sze - drop +if drop ; then cr dup $0100 / . #4 for #1 + dup @ dotsf next drop ;
: upn (n-a) ?f if #0 swap for up1 next ; then #0 ;
: lines strt @ negate #0 max crs strt @ #0 max upn #16 for up1 blue dup .line next drop drop ;
: ok show $00444444 color screen #240 #0 at r@ $0100 / block #4 for #1 + dup @ dotsf next drop $00 color #0 #266 at #1023 #296 box #0 #0 at lines keypad ;
: go strt @ #9 + upn $0100 / ld xx ;
: md strt @ #1 - #-8 max strt ! ;
: mu strt @ #1 + #256 min strt ! ;
: ?? strt @ #9 + upn $0100 / #1 + lst xx ;
: h keypd nul nul accept nul go nul nul ?? nul nul nul nul md nul nul mu nul nul nul nul nul nul nul nul nul nul nul nul
$00 , $2F000003 ,

\ Block 293
\ (compilation)
\ : . (the code for) 2tone (only exists for as long as it is needed)

\ Block 294
(App: Serial terminal) empty #52 load #48 load

#65 MagentaV char #0 MagentaV qchar #0 MagentaV pos
: - (nn-n) negate + ;
: 0eq (n-) ?f if #0 #0 + drop ; then #1 #0 + drop ;
: 0neq #0 + drop ;
: eq (nn-) - 0eq ;
: crr pos @ $1E + $FFFF and pos ! ;
: cls black screen #0 pos ! ;
: act qchar @ 0eq if ; then pos @ $00010000 /mod swap at blue char @ chc emit xy @ pos ! char @ #13 eq if crr then char @ #12 eq if cls then #0 qchar ! ;
: wait pause qchar @ 0neq if wait then ;
: ch (c-) rkey? if rkey $FF and char ! #-1 qchar ! then ;
: ok c cls act #0 pos ! show ch act $00 #650 at $00202020 color #1024 #768 box keypad ;

\ Block 295
\ (The next two blocks are a 256 character 8*8 pixel font)
\ : . (display characters statically on the screen)
\ : . (type) ok (then) #65 ch #13 ch #66 ch

\ Block 296
(App: Mouse test) empty vars dump mark
: kk vars dump hex $04 for ekey ;is ekey ekey ekey ekey ekey c next ;
: tt
: ps2 $D4 $60 pc! ;
: mm kstat ;
: ;iso

\ Block 297
\

\ Block 298

\ Block 299
\

\ Block 301
\ (Help screen)

\ (F1) show this help screen or the start shadow

\ (F2) toggle number base between decimal and hex

\ (F3) toggle seeb display of blue words (-)

\ (F4) editor, toggle colorforth / colorblind mode

\ Block 302
(App: Floppy disk driver) macro
: - $35 1, $FFFFFFFF , ;
: delay $E1E6 2, ;

```
: p@ a! dup $EC 1, delay ;
: p! a! $EE 1, delay drop ;
: 1@ $8A 2, ;
: 1! a! $0288 2, drop ; forth
: on $1C $03F2 p! ;
: off $00 $03F2 p! ;
: err -if off warm ; then drop ;
: msr $03F4 p@ $C0 and ;
: out $00100000 for msr $80 or drop if *next $00 - ; then $03F5 p! pop drop 0 ;
: in $00100000 for msr $C0 or drop if *next $01 - ; then $03F5 p@ pop drop 0 ;
: cmd for out err next ;
: conf $00 $70 $00 $13 $04 cmd ; $03 $A2 $03 $03 cmd ;
: sense $08 $01 cmd ; nload off

\ Block 303
\
\ : - ( ones complement, sets flags )
\ : delay ( dummy write, some hardware seems to need this )
\ : on - ( activate floppy )
\ : off - ( turn motor off, reset FDC )
\ : err n - ( warm start if SF set )
\ : msr - n ( get main status register )
\ : out n - ? ( write a byte to the FIFO, return error on timeout )
\ : in - n ? ( read a byte from the FIFO, return error on timeout )
\ : cmd x n - ( send n bytes to the FIFO )
\ : conf - ( some FDC commands, )
\ : spec -
\ : sense - ( see documentation for details )

\ Block 304
( Floppy disk driver )
: clrfifo in -if drop ; then drop drop clrfifo ;
: clrintr sense in err $80 and drop if clrfifo ; then clrfifo clrintr ;
: wait sense in err $80 and drop if clrfifo wait ; then clrfifo ;
: cal $00 $07 $02 cmd wait ;
: reset /flop $03 $A2 $03 $03 cmd $00 $70 $00 $13 $04 cmd ;
: init on pause spec conf clrintr cal ;
: xfer for in err over 1! $01 + next drop ;
: rd init push $FF $1B $12 $02 $01 $00 pop $00 $E6 $09 cmd block $04 * $0400 $12 * xfer off ;
: readid $00 $4A $02 cmd $07 for in err next clrintr ;
: version $10 $01 cmd in err $90 or drop if $02 - ; then 0 ;

\ Block 305
\
\ : clrfifo - ( discard all remaining input from the FIFO )
\ : clrintr - ( clear all pending interrupts )
\ : wait - ( wait for interrupt )
\ : cal - ( calibrate: move head to track 0 )
\ : reset - ( put FDC back to original state )
\ : init - ( initialize controller )
\ : xfer a n - ( reads n bytes from the FIFO to byte address a )
\ : rd b c - ( reads cylinder c to block b )
\ : readid ( for debugging )
\ : version - ? ( tests if your FDC supports enhanced commands )

\ Block 314
( EEEEEEE qkqq )

\ Block 332


\ Block 333
\

\ Block 337
\

\ Block 344


\ Block 400
```

```
( App: words )

#4621480 MagentaV from

#10308116 MagentaV tobl

#298 MagentaV num

#10000 MagentaV blkn
: tost ( u-- ) tobl @ ! #4 tobl +! ;
: ww ( -- ) #0 num ! #10000 blkn ! blkn @ block tobl !
   ftha from !
   fthn @ for from @ @ #9 or tost $25C7C00E tost
   #4 from +! #1 num +!
   num @ $06 mod #0 + 0if $9080000E tost then drop
   num @ $06 #20 * mod #0 + mod 0if blkn @ #1 + blkn ! blkn @ block tobl ! #0 num ! then drop
   next
   #10000 edit

;

\ Block 401
\ ( Help screen )

\ ( F1 ) show this help screen or the start shadow

\ ( F2 ) toggle number base between decimal and hex

\ ( F3 ) toggle seeb display of blue words ( - )

\ ( F4 ) editor, toggle colorforth / colorblind mode

\ ( F5 ) rsn...

\ ( F6 ) shows the last block edited

\ Block 416


\ Block 428


\ Block 432


\ Block 433
\

\ Block 434


\ Block 459
\

\ Block 491
\

\ Block 500
( App: Icons font editor ) empty

#0 MagentaV ic



: showall ( -- ) #0 #0 #448 at #256 for dup emit #1 + next
   drop ic @ #42 /mod #24 * #448 + swap #16 *
   2dup at #16 #24 v+ red box ;

\ Block 501
\ ( Draw big-bits icon )
```

```
\ : @w a-n ( fetch 16-bit word from byte address )
\ : !w na ( store same )
\ : *byte n-n ( swap bytes )
\ : ic -a ( current icon )
\ : cu -a ( cursor )
\ : sq ( draw small square )
\ : xy -a ( current screen position, set by ) at
\ : loc -a ( location of current icons bit-map )
\ : 0/1 n-n ( color square depending on bit 15 )
\ : row a-a ( draw row of icon )
\ : +at nn ( relative change to screen position )
\ : ikon ( draw big-bits icon )
\ : adj nn-nn ( magnify cursor position )
\ : cursor ( draw red box for cursor )
\ : ok ( background task to continually draw icon, icon number at bottom )

\ Block 503
\

\ Block 504
( App: Stack usage analyser )


: oneline ( a-- ) cr dup $6600 - $08 * $6800 + blue h.4 space yellow $20 type ;
: ok show blue page text #8 lm #1024 rm
   stacks $10 for dup oneline $20 + next drop
   keypad ;

ok

\ Block 506
( App: Server tasks )

#0 MagentaV var1
: ttsv1 serv1 #1000 ms #1 var1 +! ;


: ksv1 isrv1 ;

ttsv1



#0 MagentaV var2
: ttsv2 serv2 #2000 ms #1 var2 +! ;
: ksv2 isrv2 ;

ttsv2




: ttclr #0 var1 ! #0 var2 ! ;


: ttstop isrv1 isrv2 ;



lblk @ edit

\ Block 507
\ ( Server tasks )

\ ( demonstrates how to run colorforth words in one of the two background tasks )

\
\ : ttsrv1 ( counts up var1 )
\ : ttsrv2 ( counts up var2 )

\
```

```
\ : ksrv1 ( kills the serv1 task by giving it a null )
\    ( action )
\ : ksrv2 ( the same for serv2 )
\ : ttclr ( clears both counter variables )
\ : ttstop ( stops both tasks from running )

\ Block 508
                    ( Arithm )

\ Block 509
\

\ Block 510
( Sandbox  o98 any old ASCII  ovk@ )



: rrrr push ;
: tttt pop ;



: test ( your code here ) ;

\ Block 511
\ ( Help screen )

\ ( F1 ) show this help screen or the start shadow

\ ( F2 ) toggle number base between decimal and hex

\ ( F3 ) toggle seeb display of blue words ( - )

\ ( F4 ) editor, toggle colorforth / colorblind mode

\ ( F5 ) rsn...

\ ( F6 ) shows the last block edited

\ Done.
```

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     008A07CF E8 8B 12 B6 FF        call     here (00401a5f)
2/       008A07D4 D1 F8                 sar      eax,1
2/       008A07D6 D1 F8                 sar      eax,1
push     008A07D8 8D 76 FC              lea      esi,[esi-4]
         008A07DB 89 06                 mov      dword ptr [esi],eax
         008A07DD 50                    push     eax
drop     008A07DE AD                    lods     dword ptr [esi]
a!       008A07DF 8B D0                 mov      edx,eax
0        008A07E1 B8 00 00 00 00        mov      eax,0
dup      008A07E6 8D 76 FC              lea      esi,[esi-4]
         008A07E9 89 06                 mov      dword ptr [esi],eax
a        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
pop      [edx*4],eax
@        008A07F7 58                    pop      eax
;
```

[redacted]

```
008A07F8 8B 04 85 00 00 00 00 mov          eax,dword ptr
[eax*4]
008A07FF C3                   ret
```

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.

.