Users' Guide for ColorForth 2.0a

Information for engineers using colorForth

Revision 1.00
02/08/08

1. Introduction

1.1 Welcome to Release 2.0a

Release 2.0a is the result of the efforts of the OKAD development team during September through December of 2006. The colorForth kernel was converted from Intel to FORTH assembler source, then considerably revised and reorganized so that a single source base could be used for both native x86 and 32-bit Windows platforms. If you have used the previous native version of colorForth version 1.0, you will find very little difference in the human interface but considerable change in the way you move data to and from the Windows version and the outside world (change for the better, of course!). The Windows version is designed specifically to be a proper superset of the Native version with no unnecessary surprises in human interface or operational methods, and in particular is designed to produce the same application results as does the Native version using the same design file.

The following two sections reflect this recommendation; first, discussing differences in the new Native system, and secondly discussing what a Native user will find different in Windows.

1.2 Native System 2.0a relative to 1.0

The first thing you will notice on booting 2.0a is that instead of reading 90 blocks into memory at boot time, the system reads 1440. Colorforth 1.0 read from between one to eighty cylinders from a floppy with **`nc`**, cylinders from a (native) floppy, set to 5 for loading 90 blocks to memory at boot.

The second thing you might notice is that you will not necessarily be booting from a floppy. In fact, any medium from which the BIOS on your machine can boot using the disk boot structure – floppies, hard disks, USB floppies things that look like hard disks such as USB hard disks and flash memories – can be used to boot the 1440 block floppy image without change, provided it is possible to transfer that image onto the medium. Optical disks (CD/DVD) are excluded from this class because the data structure used on them is radically different.

Unfortunately, while we are able to boot using the BIOS for device independence, we must still kill the BIOS after we have finished booting because letting it live interferes with using the PC freely as a 32 bit minicomputer. Consequently, we are still limited on writing to floppies and, at this time, to a specific set of USB flash devices on specific USB hardware using a specific System Management Mode BIOS.

Because we are able to boot now on a machine that does not have a floppy disk at all, and because we have seen DMA chips walk memory when a channel is enabled without a device attached to drive its request line, the code in the boot routine which enabled the floppy DMA has been factored out and as of 2.0a1 must be executed manually before attempting floppy access after each boot.

> **`homef`** is the word which does this. In 2.0a1 it must be executed before any floppy operation is performed. There should be no harm in executing it repeatedly.

The next big difference you should notice is that things are in different places in memory than they were before.

block works as it always has in colorForth – converting a block number to a word address in memory. Absolute block zero is still at absolute memory address zero. However, `block` internally adds the value of a variable called **offset** to incoming block numbers before multiplying them by 256. The value set in `offset` reflects the absolute memory address at which the disk image was read at boot. Various BIOSes have forced this address to be two binary million (200000 hex as a byte address; 80000 as a word address) so that the value in `offset` is hex 800, decimal 2048. This affects all uses of `block` including `load` and `edit` .

offset has been added to the dictionary so it is accessible from colorForth source.

The disk image was moved up that high because there was no room under the PC "hole" at 640k to 1 megabyte for 1440k of data. However, in moving the disk image, several unrelated things that were not actually part of the disk image have been left where they were, or moved elsewhere; so that there are no more buffers and such things in the range of address space occupied by the disk image. (Naturally, data such as the native kernel, the icon table, and ROM images, which *are* logically part of the disk image, have been left within it.)

As of 2.0a, the things which remain in low memory are the stacks (for keyboard, screen, and communication tasks), the cylinder buffer for floppy operations used mainly to avoid DMA operations across a 64k byte boundary in memory, and the "trash" buffer used by the editor for deleted words that might be re-inserted later.

The working Forth dictionary lies immediately after the disk image, at `offset` -relative block 1440. The dictionary is followed immediately by the initial position of **here** as of the start of loading block 18.

Other fixed allocations are made relative to very large block numbers and float above the point we call "block0" (200000 hex as a byte address in Native systems.)

In addition to offset and homef as described above, there are several other additions to the dicitionary:

abort This function existed in the 1.0 kernel but did not have a "head" and was accessed via kludging. The head now exists. Terminates execution and goes back to keyboard processing. Sets editor pointers if W addresses block 18 or higher. Empties return stack and resets a number of cells in the `spaces` table. Echoes a ?.

h (- a) This is the name of the variable whose current value `here` reports. Simply added the head to avoid kludging.

aper (- a) Again, simply adds a head to an already existing variable, the one which points to the frame buffer that is to be used for refreshing the display (frame, which is used by the drawing primitives, is not given a head at this time.)

tic (_ - a) is a brand new function intended for interpretive use only. Type `tic` and hit space. Then type another word and hit space. tic will search for this word and leave its address on the stack. Intended for use with garden variety, non macro red words to help interpret register values on hardware traps.

winver (- tf) is a new flag exported by both Native and Windows kernels; returns both a value and CPU flags reflecting that value. For Native systems it returns zero. For Windows systems it returns 1.

New coding practices are required. If you add something that depends on the native environment – such as priviliged code for I/O or CPU control register manipulation, or such as allocation of memory below block 0 or above 512 megs, you must do so in a manner that will not cause the windows version to fail. See the definition of env in block 18, and the environmental dependency blocks 34 and 36 for the two systems. At minimum make it possible to load both systems without failure; if the function won't work on both systems but has to be executed for some reason, put a no op in the native system.

This of course applies equally to things done in the Windows system that cannot be done natively – such as network access, playing of videos or music, taking of mouse or finger pointer input, speech recognition, et cetera.

1.3 Windows System 2.0a relative to Native System 2.0a

When you execute the program CF2.exe two windows will appear. The first, and the main window for the program at this time, is a console window in which status and diagnostic information are displayed. The second is the colorForth window.

1.3.1 Console Window

We leave a console window open so that there is a place in which diagnostic information and status feedback can be given outside the constraints of the colorForth human interface. On loading the program a typical display might look like this:

```
cmcf 2.0a
0012FF98
Opening OKAD source file OkadWork.cf 00000003 00168000
0012FF98
One time operations:
1000547B  10000000 0011075C 0053083F 00010029

00000034 00000038
10004C8E 0000003C
10004D8E  00000040 0091FF8C 00000CB8 000003FF

Win32 API Error: 00000578 1000625D 10003F22 10013400 10004C8E
```

The last line may or may not appear; it is due to a race condition between Windows and our event service loop. Some times our event handling is efficient enough that we get the signal to repaint the window and act on it before Windows has finished creating the window, resulting in this error message. It is not significant.

File writing operations each write a line documenting what they did and any abnormal status returned.

Hardware traps such as accessing unmapped memory, or divide by zero, display machine state and stack dumps. For example, writing 0 @ produces a trap since location zero is not accessible by the program:

```
Trap/Fault:
10006424  C0000005 00000000 00000000 1018C47B
          Windows code
10006434  00000002 00000000

1000643C  0001003F 00000000 00000000 00000000
1000644C  00000000 00000000 00000000 FFFF027F
1000645C  FFFF0000 FFFFFFFF 00000000 04910000
1000646C  00000000 FFFF0000 01B8CF60 0176CB94
1000647C  EE180000 07387C90 FFFF7C91 7C9106AB
1000648C  7C9106EB 0034D054 0000C000 00000000
1000649C  00000000 020A0014 0732CC1C 00057C91
100064AC  07780000 01B557A8 0176CBF4 EE18BED8
100064BC  07387C90 FFFF7C91 00000000 00000000
100064CC  0000003B 00000023 00000023 10003F22
                                     W (7/di)
100064DC  10015800 10004C8E 10004248 00000050
           S (si)   3 (bx)   2 (dx)   1 (cx)
100064EC  00000000 0012FFF0 1018C47B 0000001B
           0 (ax)   5 (bp)      IP   CS (seg)
100064FC  00010246 0012FF90 00000023
             flg    R (sp)

10015800  00000000 00000000 00000000 00000000
          Data stack top

0012FF90  100044AB 10006799 00000000 0012FFE0
          Return stack top
```

If you were not expecting the trap, or do not immediately see what mistake caused it, this information can be used to identify the source of the problem. The last two lines are the top four cells of the data and return stacks, respectively, top of stack on the left. The first cell of the dumps, C0000005, is a Windows code; the actual hardware trap number is unfortunately not returned to us. The machine registers are at the bottom of the main body of the dump. The annotations below the values above, in bold italics, are not of course part of the actual console display. Copy/paste are available in console window; thus these values can easily be transcribed or emailed to facilitate debugging of the offending code.

1.3.2 ColorForth Window

This window presents a human interface which is as faithful to that of the native system as we have been able to achieve. The window is fed by a 1024 by 768 Device Independent Bitmap (DIB). Actually, in v2.0 this DIB is sized somewhat taller than 768 rows because the editor will go ahead and display more rows of text than fit on the display, and will also display as many numbers are on the stack even if this number is huge. In both cases, the editor will write past the end of the DIB which causes one class of problems on Native systems and another (trap) on Windows. The extra vertical space in the v2.0 DIB is "headroom" to keep the editor from trapping until we fix these problems in a usable way. Even then it is possible to overrun the display and produce a trap, when for example listing a garbage block or coming back to the keyboard with a very large stack. Visual checking of whether this has happened can be done by dragging something across the bottom part of the colorForth window, or minimizing it, to force it to be refreshed.

The window is resizable, movable, and minimizable. If you have multiple screens you can move it to any screen you like. If you wish to run two instances of **cf2** at the same time you may do so; management of changes to output files is up to you. As of v2.0 we do not force the window to have constant aspect ratio on resize, partly because windows display devices are not guaranteed to have square pixels in the first place.

The keyboard is a faithful rendition of the Native keyboard with two major exceptions:

1. The native system has a certain number of characters of typeahead buffer kept inside of the keyboard processor chip. The keyboard processor is polled, not serviced by interrupt code, so when you hold down a key such as a cursor movement, you can have a fair amount of movement "queued" up in the chip when you lift your finger. The Windows version receives keyboard events as they occur and as a result there is exactly one character of type ahead buffer. When combined with the fast drawing speed of the Windows system, this leads to very satisfying results when navigating among blocks in the editor.
2. Because we are ignoring the extended keyboard flags, the Windows keyboard recognizes both left and right "alt" keys if the keyboard in use has them.

The hardware specific operations from the Native system (floppy and USB flash I/O) are ***not*** supported and ***must not*** be run. Violating this rule will produce trap dumps on the console window.

New words unique to the Windows system:

`utime ( - u)` returns the unix time in seconds since midnight UTC on Friday 1 Jan 1970.

`endram ( - a)` returns the address of the first byte past the end of memory allocated to the application.

`keych ( - a)` is a double cell variable. The first cell if nonzero is a keyboard character (Windows key code) waiting to be processed by the keyboard task. The second cell contains extended key flags which are not now relevant to us and probably never will be.

`retain` writes the 1440 block image from memory into a file called OkadWork.cf. This is the same file name that is read when starting the program. If you wish to save the original, do so at some time before retain - the file is not open while the program is running, so it can be renamed after starting the program. Note however that the file must exist by that name, even if zero length.

2. Program Installation

2.1 Native / Standalone Version

2.1.1 System Requirements

Floppy disk or other BIOS bootable medium that follows the boot conventions of floppies and hard disks. Floppy drive (connected to a true FDC, not a USB floppy) if you wish to write output – exception is some specific machines Chuck has which can write USB memory.

NVidia or ATI graphic chip which supports the VESA modes hex 4118 ("nVidia") or 4123 ("ati") to give 1024x768x32 bit frame buffer in the PCI resource which shows as prefetch enabled RAM in PCI configuration space.

2.2 Windows Version

You might wish to make a shortcut for **colorForth2.0a** (CF2.exe); there are icons in the release package to make shortcuts, and the windows they invoke, recognizable.

2.2.1 System Reqirements

Windows NT4, 2000, or XP Professional. The faster the better, the more RAM the better, any fast graphic chip, dual processors are not wasted. Preliminary results indicate Vista is also compatible.

3. ColorForth Human Interface

The colorForth human interface has not been changed intentionally as of v2.0. However a qwerty keyboard option has been added. When colorForth is configured for the qwerty keyboard, different keys must be used to exit text entry mode of the editor and to put colorForth into hexadecimal mode. These changes are covered below.

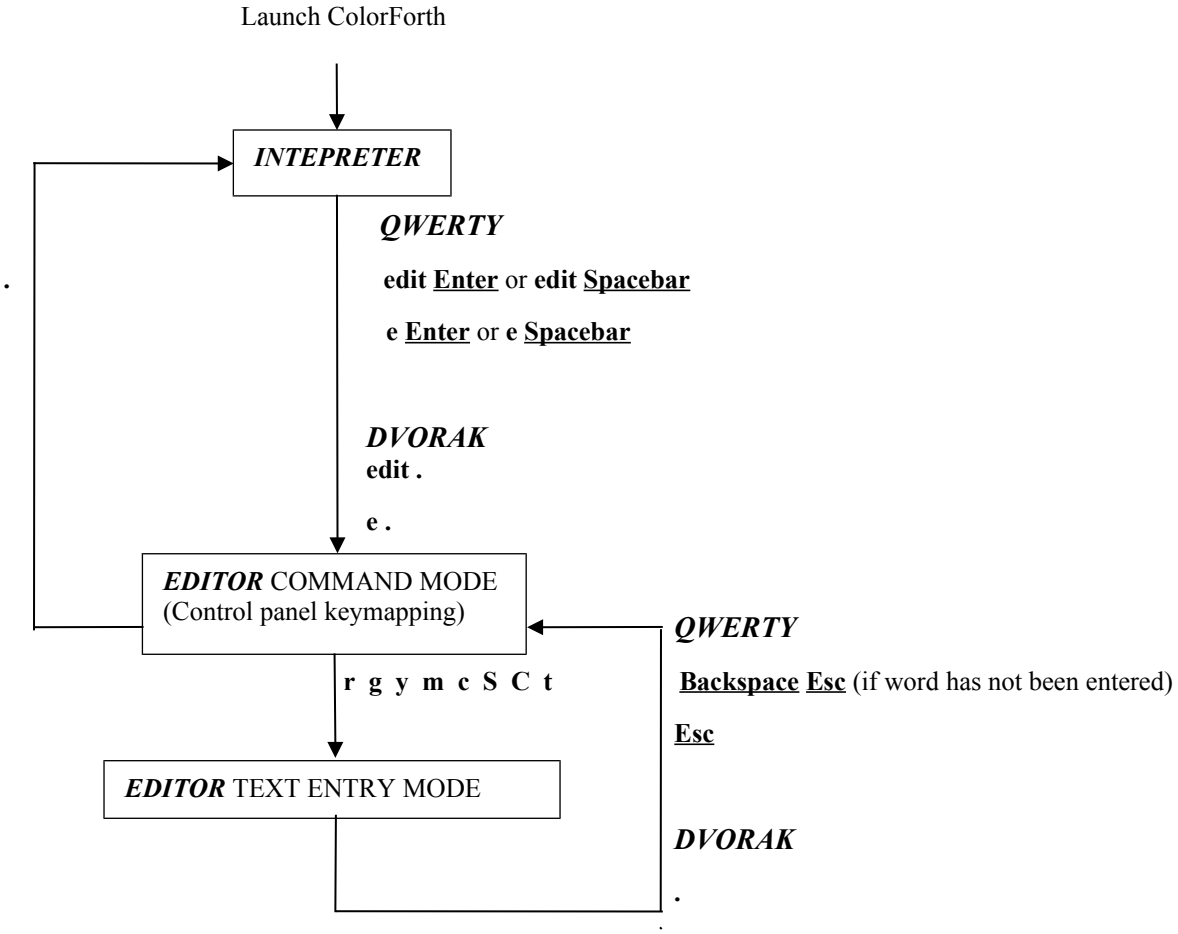
3.1 Keyboard Layout and Hinting

ColorForth is an integrated development environment consisting of an interpreter, compiler, editor, and dictionary. It is used to build all of the other tools that make applications. Keys are mapped for a specific tool being used. ColorForth can be configured to use a qwerty or Dvorak keyboard. This keyboard is used for the interpreter and text entry mode of the editor. A control panel is mapped for the command mode of the editor. The same control panel is used regardless of whether ColorForth is configured for qwerty of Dvorak keyboards. Colors are assigned to specific identifiers of the colorForth programming language. White is used for comments, red is used for names of words, and green is used for calls and macros.

ColorForth is launched in the interpretative mode with keys mapped to the qwerty keyboard. To switch to Dvorak key mappings comment out qwerty from block 18 and reboot (make it white). In this document, qwerty key mappings are in **Bold Underline** and the corresponding Dvorak key mappings as well as editor and interpreter commands are in **BOLD**. A state diagram showing how the interpreter and editor modes transition through different modes is shown in figure 1.

Figure 1: Editor and interpreter transitions

NOTE: Refer to figure 6. Editor controls for editor commands. These are not standard keyboard characters.



3.1.1 Qwerty Keyboard Mapping

As mentioned above, text entry mode of the editor and the interpretive mode behave the same. The lower right corner of the terminal is a hint area. The hint area indicates the type of text being entered and whether hex mode is on. Examples of how the hint area changes with different types of entries are shown below. In qwerty mode, the word "**qwer**" is always displayed in the hint area. The hint area will display the state of the keyboard (text or numeric) after the first character is typed. All of the state indicators and the text entry area are positioned relative to "**qwer**". Below is an example of the bottom of the screen showing the locations of the stack, current text being entered, and hinting area.

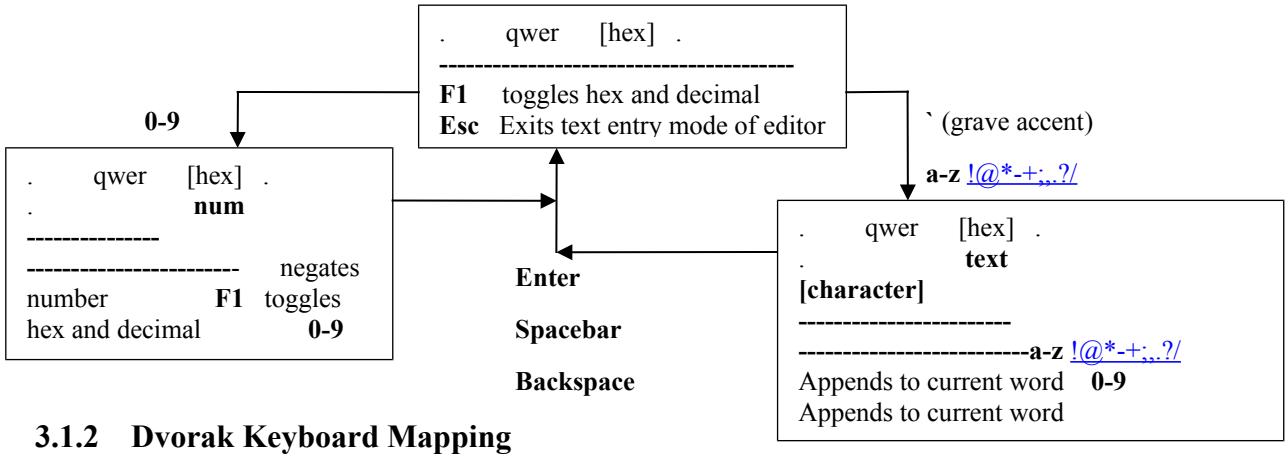
Bottom of screen

```
                                                        qwer  [hex]

                                                              [text/num]

stack                                           current text
```

While in text entry mode of the editor, the Esc key is disabled while a word is in the current text area. The following table lists the keys, corresponding actions, a sample key sequence, and changes in the hint area. Enter or Spacebar will place something on the stack or execute if in the interpreter. They will place the word after the cursor if in text entry mode of the editor.

Key	Action	Example Key Sequence	Hint Area
a-z !@*-+;,.?/	Enters text.	add	. qwer [hex] . **text add**
0-9	Enters a number. The number is written in the stack area; not in the current text area.	356	. qwer [hex] . **num**
-	Negates the current number. A digit must be entered before - or it will be entered as text.	5-	. qwer [hex] . **num .**
` (grave accent)	Forces text for numbers.	`2this	. qwer [hex] . **text 2this**
F1	Toggles between hex and decimal.	F1	. qwer [**hex**] . [text].
Backspace	Aborts current word.	Backspace	. qwer [hex] . **.**
Enter Spacebar	Enters a word on the stack in the interpreter. Places the current word after the cursor in a block in editor text entry mode. Clears current text after entry.	Enter	. qwer [hex] . **.**
Esc	Exits text entry mode of editor. NOTE: Current word must be entered or aborted or the key will have no effect.	Esc	. qwer [hex] . .

As can be seen above "**–** **"** can be entered as text or to negate a number depending on the state of the keyboard. These states are shown in the following state diagram. As mentioned above the first character entered determines the state of the keyboard and how the next keys entered will be interpreted.



3.1.2 Dvorak Keyboard Mapping

The Dvorak keyboard begins in the character mode whether in the interpreter or text entry mode of the editor. The hint area will show the key mapping for all keyboard states. The keyboard states are shown in figure 1. and mapping of the Dvorak character

keypad is shown in figures 2. – 5. . These mappings will change as each tool is changed. The last command executed appears to the left of these key mappings. The last row of characters on the Dvorak keyboard, control the keyboard mappings. These characters are the "**n**", "**space**", and "**alt**" of the qwerty keyboard. The state diagram below shows how the keyboards change with the corresponding control keys. All keyboards will return to the character key board after a word is either entered on the stack, aborted, or the "**a"** control key is pressed.

Figure 1: Keyboard states for Dvorak keyboard setting

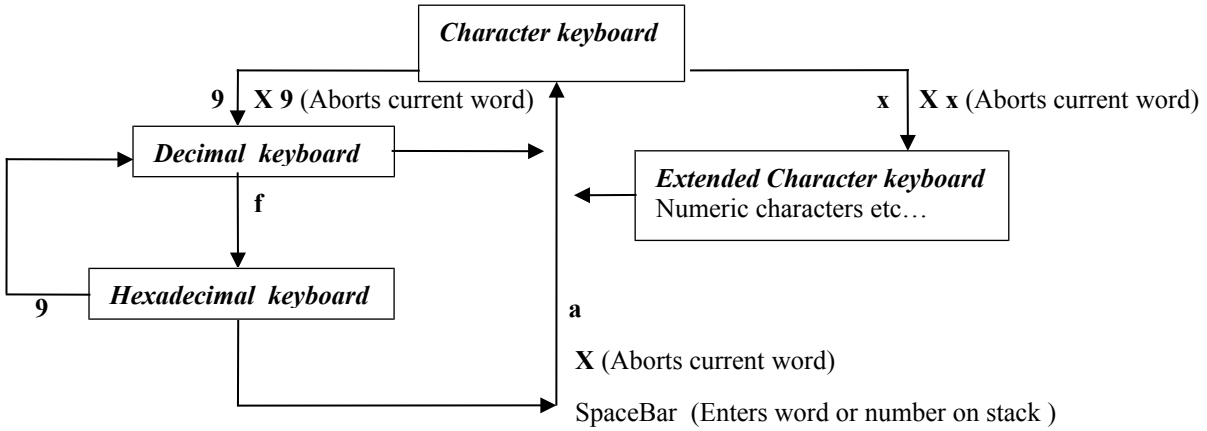


Figure 2: Dvorak character key mappings.

Q P	W Y	E F	R I			U G	I C	O R	P L
A A	S O	D E	F U			J H	K T	L N	; S
Z Q	X K	C X	V D	N		M B	. M	. W	/ V
				space 9		alt x			

With the exception of the "**X**" key, all of the Dvorak keys in the first three rows of the display will print the indicated characters. The "**9**" on the bottom row will switch to the numeric key mappings shown in figure 4 and figure 5. The "**x**" on the bottom row will switch to the extended keys shown in figure 6. This "**x**" should not be confused with the "**X**" mapped to "**n**" that is displayed after the first character has been typed. This "**X**" is used to delete the current characters. The "**.**" that is mapped to the "**sp**" after typing the first character places a number on the stack or executes a command and will be described in the next section. The Dvorak keyboard mappings that occur after typing the first character are shown in figure 4.

Figure 3: Dvorak character key mappings after typing the first character.

Q P	W Y	E F	R I					U G	I C	O R	P L
A A	S O	D E	F U					J H	K T	L N	; S
Z Q	X K	C X	V D	N X	M B	, M	. W	/ V
				space .		alt x			

The numeric keypad can be changed from decimal to hexadecimal. The "**alt**" key will change as the keypad changes from decimal to hexadecimal. An "**f**" is shown in the last row of the decimal keypad display to indicate a change to hexadecimal and a "**9**" is shown in the last row of the hexadecimal keypad display to indicate a change back to the decimal keypad. The "**a**" on the bottom row of the display will return to the character keypad described above. Typing "-" before a number will produce a negative number. As in the character key mappings after the first character is entered "**n**" will be mapped to "**X**" and "**sp**" will be mapped to "**.**".

Figure 4: Dvorak decimal key mappings.

Q	W	E	R					U 1	I 2	O 3	P
A	S	D	F					J 4	K 5	L 6	; :
Z	X	C	V	N -	M 7	, 8	. 9	/ /
				space a		alt f			

Figure 5: Dvorak hexadecimal key mappings.

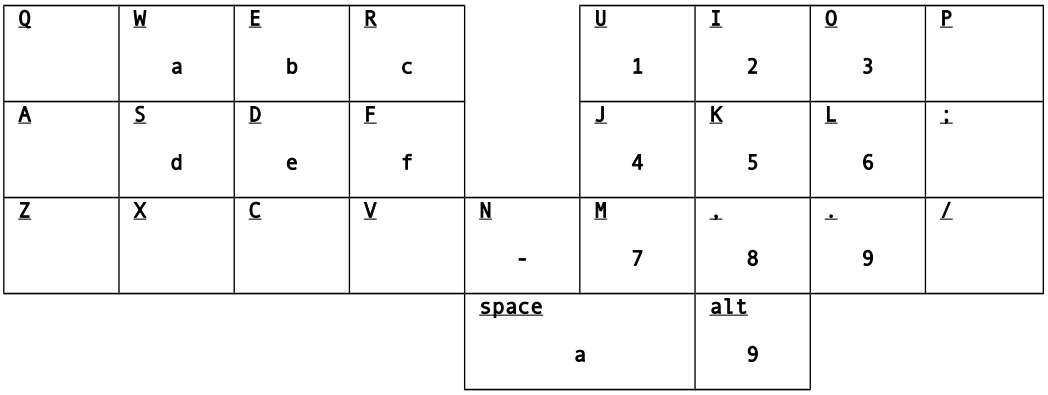
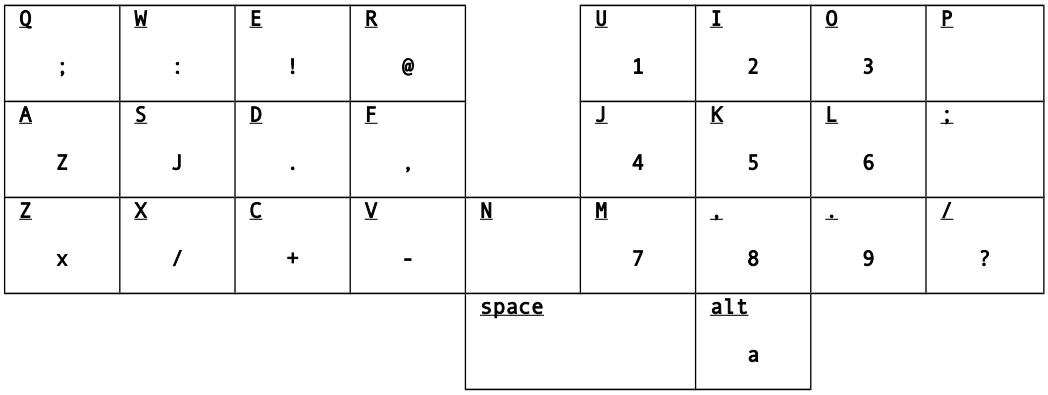


Figure 6: Dvorak extended character key mappings.



While using the extended character keypad, the "**a**" on the last row will return to the character keypad as it does from the numeric keypad. Again, as in the character and numeric key mappings after the first character is entered "**n**" will be mapped to "**X**" and "**sp**" will be mapped to "**.**".

3.2 Keyboard Semantics

3.2.1 Interpreter and editor text entry mode

All of the commands listed below in table 1 are executed from the interpreter. However some of them will enter the editor and begin to search memory blocks. Some of the keys used to execute a command or place a number on the stack are different for the qwerty and Dvorak keyboard mappings. These keys are shown in Table 2. Examples of key sequences used to enter commands are shown in table 3.

Table 1:Interpreter commands

COMMAND	ACTION
Bye	Closes colorForth
e or **edit**	Enters editor
Find	Waits for a second word to be typed then searches for the first occurrence of that word starting from block 18. **find** will enter the editor after execution if the word is found. If the word is not found it will stay in the interpreter. Note: Fall does not find literals.
literal (n)	Same as **find** but looks for `n` as a yellow or green number in either hex or decimal, matching the value and not the representation in either radix
Def	Same as **find** but looks for a (red word) / definition.
from (n)	Same as **find** but start at block n ie n from
C	Clears stack

Table 2: Text entry commands that differ depending on key mappings.

QWERTY COMMAND	DVORAK COMMAND	ACTION
Sp or **Enter**	.	Places number on stack or after cursor in block. Executes command in interpreter
Backspace	n	Deletes current word
F1	F (from numeric key mapping)	Changes to hex mode
- Note: - must be pressed after a digit is entered or text will be expected.	- (from numeric key mapping) Note: - must be pressed before number is entered or number will be deleted. Does not toggle as in qwerty mode	Negates number

Table 3: Qwerty and Dvorak Text Entry.

Entry	QWERTY KEYSTROKES	DVORAK KEYSTROKES
2 3 +	**2 Enter 3 Enter + Enter** NOTE: **Sp** can be used in place of **Enter**.	**9 2 . 9 3 . x + .**
2bee	**` 2 b e e Enter**	**x 2 a b e e .**
ffff (hexadecimal)	**F1 0 f f f f Enter**	**9 f f f f f .**
-5	**5-**	**-5**

3.2.1.1 Entering commands using the Dvorak keyboard

Using the Dvorak keyboard in interpretive mode, "**X**" will erase the current character or number that has been typed unless it is pressed before entering a number. In this case the number will be negated. However, if nothing has been typed and the number key mappings are not active this key does nothing. Also, after the first character is typed the key mapping changes as explained above. The space bar, "**sp**", will display a "**.** ". This key will place a number on the stack from a numeric keypad or execute a command from the character or extended keypad. If the executed command is not found, the command will be displayed in the lower right next to the display of the key mapping with a question mark prefixed. Table 1 and Table 2 above provide descriptions of these keys and examples of how they are used.

3.2.1.2 Entering commands using the qwerty keyboard

Using the qwerty keyboard, "**backspace**" erases the current word. When numbers are hit, "num" is displayed to the right of qwerty. "-" toggles between negative and positive and positive numbers. When letters or any valid colorForth characters are pressed (a-z !@*-+;,.?/ ') , "text" is displayed to the right of qwerty. "`" (grave accent) is used to start a word beginning with a digit. When grave accent is pressed the word "text" will be displayed to the right of qwerty just as it is when a letter or valid colorForth character is entered. F1 toggles between hexadecimal and decimal numbers. When in hexadecimal mode, "hex" is displayed to the right of qwerty. To enter a hexadecimal number beginning with a-f, a 0 must prefix the number or it will be recognized as text. Again, Table 1 and Table 2 above provide descriptions of these keys and examples of how they are used.

3.2.2 Editor

As stated above, "**e**" or "**edit**" will launch the editor from the interpreter. The editor commands are the same regardless of whether an application is configured for qwerty or Dvorak keyboards. The editor comes up in command mode and the keyboard will be mapped to the editor command key mapping. This mapping is shown in figure 7. The keypads described above are used to enter text into the editor. Table 4 describes the editor commands. "**.**" will exit the editor however the previous interpreter screen will not be redrawn. The last block will be displayed and variables will change if updated on the screen if changed by through the interpreter.

To add a definition to a block, red is selected from the editor control panel. After the name of the word is entered the editor will automatically change to green. Another color can be selected to modify the definition behavior. For example, magenta can be used to enter a variable or cyan can be selected to force a compile. Returning to the editor control panel is accomplished by pressing **Es**c if configured for qwerty and "." if configured for Dvorak.

Figure 7: Editor Controls

Q	W S	E C	R t		U y	I r	O g	P x
A	S d	D f	F j		J l	K u	L d	; r
Z	X	C	V	N x	M -	. m	. c	/ +
				space .		alt i		

Table 1 : Editor commands

KEY	ACTION
S (Shift Lock) comment	Enter comment (white) in capitols
C (Capitalize) comment	Enter comment (white) first letter capitalized
t (text) comment	Enter comment (white) all lower case
y (yellow) Interpreter	Executes when encountered. Gets executed during compilation if used inside a definition.
r (red) Word	Word definition. After the word is completed the color switches to green to enter instructions that the word defines.
g (green) Default compilation.	Call Forth words or execute macros.
x is really an *	Toggles between an even (source) block and odd (shadow) block. This is normally used to document the associated text.
c (change)	Changes color of word that follows cursor.
d (definition)	Finds the (red word) / definition of the (green word) / call that is following the cursor.
f (find) NOTE: To use this key a search must be invoked from the interpreter.	Finds the next word color combination occurrence of the search that was entered from the interpreter.
J	Toggles between current block and the last block. To write a block to the last block, enter <nn> edit from the interpretive mode. <nn> is the number of the block to be edited. The <nn> block will be the last block and will be toggled with the current block.
l (left)	Moves cursor left one word
u (up)	Moves cursor up. (left eight words)
d (down)	Moves cursor down. (right eight words)
r (right)	Moves cursor right one word.
k (copy)	Copies word to the left of the cursor and places on the stack.
- (down block)	Moves down one block of memory
m (magenta) variable	Variable name must fit in one cell. Four characters will always fit in one cell. The maximum length is seven characters but this depends on Shannon code.
c (cyan)	Always compile. Forces a call to a word.
+ (up block)	Moves up one block of memory.
x (cut)	Deletes word to left of cursor and places on the stack.
.	Returns to interpreter.
i (insert)	Pops words off of stack and places to at cursor.
Esc	Returns to the editor control panel.

3.2.3 Search utility

This utility, integrated with the editor, facilitates searching the source in several practical and useful ways. When in the interpreter, there are four words for starting a search, and one for continuing it. The words for starting the search are **`find def from`** and **`literal`** . When a search is started, we begin at block 18 (except in the case of **`from`**) and scan forward looking for the particular target. If it is not found, we remain silently in the interpreter. If it is found, we enter the editor with the cursor set at the target. Once a search has been started, it may be continued from the point of the last target found by using the "**`f`**" control panel key, if in the editor, or by typing **`f`** if in the interpreter. In either case a find will display a block, while a failure to find will remain silently in whatever environment (editor or interpreter) the continuation was requested.

The searches continue until the end of `nc` cylinders. For the interpretive `f` below, searches continue from the point of last find in that same search. The continuation point is the current cursor position when the editor's "`f`" key is used.

The searches automatically consider only blocks of the same sort (source or shadow) as the block in which the search was started or continued. Thus, all searches start with source only; if they are continued using the interpretive `f` or if the editor cursor is never changed before using the editor's "`f`" key, shadows will never be considered in that search. To search shadows, begin a search, ignore the first find (or no find), edit block 19, and then continue the search with the editor's "`f`" key.

`find` awaits a word from the keyboard and starts a new search for that word. It will find all instances of that word as a definition (red or magenta), reference (green, yellow or cyan), or any sort of comment.

`def` awaits a word from the keyboard and starts a new search for that word, as a definition (red or magenta) only.

`from (n)` awaits a word from the keyboard and starts a new search for that word, in any form as in `find` but starting in block `n` instead of block 18.

`literal (n)` starts a new search for the value `n` as a yellow or green number in either hex or decimal, matching the value and not the representation in either radix.

`f` continues the most recently started search immediately after the point at which the last target was found in that search. Due to the new dichotomy between source and shadows, this word is of limited usefulness.

When in the editor, two keys activate word search functions without leaving the editor:

"**`f`**" control panel key continues the most recently begun search, beginning after the current editor cursor position. If the current block is source, only source will be searched; if it is a shadow, only shadows.

"**`d`**" control panel key starts a new search based on the word immediately preceding the cursor.

- If the word is a definition (red or magenta) then the new search will be for references (green, yellow, or cyan) to that word.
- If the word is a reference or a comment, then the new search will be for definitions of that word.
- If the word is a literal number, then we do not search as such but simply edit the block by that number.

4. Source Code and Mass Storage Maintenance

colorForth mass storage is kept on suitable media including floppies and, for the Windows system, most raw and file system supporting media. Except as otherwise noted, mass storage is viewed and addressed as a contiguous sequence of 1024-byte (256-word) ***blocks*** such that these blocks are mapped onto a sequence of integers beginning with zero. Like conventional FORTH systems, the word **`block (n – a)`** translates the number of a mass storage block into a memory address at which its contents may be found. Unlike conventional systems, however, colorForth explicitly caches, under programmer control, an entire region of interest of mass storage, writing updated data out only under explicit command, rather than caching one or more blocks in memory and writing them out silently when the space is needed. Accordingly, the word `update` is not needed or found in colorForth. Instead, we have explicit functions for reading and writing mass storage.

For reading and writing purposes, mass storage is further organized into ***cylinders*** (18 consecutive blocks starting on a zero mod 18 boundary on the medium), which have a hardware meaning on 3.5" floppies ane which is retained for compatibility of practices on other media. The next larger unit is the ***disk image,*** which is set by the programmer by storing a suitable number of cylinders in the variable `nc` . The default value of `nc` is 80, indicating a full 1440 block disk image, one that fits on a 3.5" floppy.

From the perspective of colorForth, a full disk image represents a system and its application(s). From the perspective of colorForth2, a full disk image can represent a suite of applications such as the entire OKAD2 VLSI CAD system, with all tools, and one or more complete chip designs.

4.1 Reading and Writing

4.2 Disk Layout

In ColorForth version 2.0a, the disk layout is as follows:

```
The top-level toc for the source is:

  0- 12     Native kernel (included in Windows systems)
 12- 18     Icon table (used in both systems)
 18-144     colorForth code and tools (public domain)
144-1439    application
```

4.3 Reading and Writing

On activating ColorForth2, the disk image file named OkadWork.cf is read into memory for 1440 blocks, and the file is closed. From then on you are working with a RAM image of virtual disk. The actual file is only updated when you explicitly say to do so.

In the native system, this is done by typing **`floppy archive`** . On the Windows system it is done by typing **`retain`** .

Disk operations have been adjusted to give equivalent functionality on both native and Windows systems. The following *resident* operations now exist on both platforms:

`save` writes the active disk image (`nc` cylinders at block 0) to (native) floppy or (Windows) `OkadWork.cf` file. This includes boot, native kernel, icon table and source code.

`wrtboot` writes the boot and kernel image from block 0 to a native floppy, as an economical way to update the native kernel on floppy after executing `ati` or `nvidia` in the `floppy` utility. It does nothing on Windows systems.

`!back ( n)` writes `nc` cylinders to a (native) floppy or (Windows) `OkadBack.cf` file from RAM block `n` .

`@back ( n)` reads `nc` cylinders from a (native) floppy or (Windows) `OkadBack.cf` file into RAM at block `n` .

`@cyls ( a c n)` reads `n` cylinders to cell address `a` from cylinder `c` on backup. For native, the read is direct from floppy. For Windows, the whole `OkadBack.cf` file is read into RAM at block 3000 and then the cylinder(s) in question are copied from there.

`readme` reads the two cylinders at `cfuse` from the same cylinders on backup medium, using `@cyls` .

`tapeout` copies two cylinders of tapeout configuration from `cftape` to `cfuse` within the working RAM image.

4.4 Mass Storage Utilities

The principal function of these utilities is the composition, maintenance, and reconcialiation of colorForth and application source code.

4.4.1 The Editor

4.4.2 `floppy` Utility

4.4.3 `disking` Utility

This is a nonresident application which is loaded atop an empty dictionary by saying xxxx. Its purposes are to aid in the moving, copying, comparing and reconciling of source code.

The following are the most basic operations for manipulating groups of source blocks:

bloks (s d n) copies `n` blocks starting at block `s` into the RAM image of disk starting at block `d`. Must be spelled this way because `blocks` is indistinguishable from `block` in the colorForth dictionary. This will in general be used less often than the next word:

+blocks (s d n) copies `2*n` blocks from block `s` to block `d`. Think of it as copying `n` blocks of source along with their shadows.

obliterate (f l) clears, to all zero, RAM blocks starting at block `f` and ending at, but not including, block `l`.

The following words aid in reconciling source. They compare two areas of disk, identified by starting block numbers, in which corresponding blocks are to be compared and reconciled. The area with the lower block numbers is called the ***lesser*** area; whether it is the target of the reconciliation depends on how you employ the words. When looking at a block in either of these two areas, the ***other*** block is the corresponding block in the other area. Three internal variables are used to keep track of the areas in question, so that the functions do not depend on stack integrity.

matching (s d) sets up a reconciliation between areas of RAM disk starting at blocks `s` and `d`. Whichever of these areas starts at the smaller block number is the *lesser* area.

to (n) sets the upper limit of the *lesser* area to block `n`. Comparison will stop at, and not include, block `n`.

check loads the backup image appropriate for native or Windows systems into RAM at block 3000. This is equivalent to saying `3000 @back`.

Once the parameters of the comparison have been set up, reconciliation is an interactive process which involves use of the editor for examining individual differences and making source changes when one wishes, and leaving the editor to use the following words:

v used after having looked at a block with the editor to view the *other* block, leaving the editor set so that the "j" button on the editor control panel toggles between the two blocks.

g used after having looked at a block with the editor, and exiting the editor, to scan forward starting with the next block in the comparison, searching for the next block in which there is a difference and displaying the *lesser* with the editor set to toggle (using "`j`" control panel button) between it and the *other*, or stopping without doing anything if there are no more differences. It does not make any difference which of the areas was last viewed. *The comparison is done in such a way that blocks whose only differences are the values of magenta variables are skipped.*

give used after having looked at a block with the editor, and exiting the editor, to copy the block that was just displayed to the *other* block (think of giving it to the other block.) This means that the text displayed on the screen will be what is kept in both areas of the comparison.

take used after having looked at a block with the editor, and exiting the editor, to replace the contents of the block that was just displayed with the contents of the other block (think of taking it from the other block.) This means that the text displayed on the screen will be discarded and replaced.

A typical use of disking is to reconcile two complete disk images. To reconcile the active disk image (at block 0) with a backup image, load the desired backup image into RAM at block 3000 with `check` and then start the comparison with

all starts a comparison between the active image at block 0 and a backup image at block 3000, beginning the comparison at block 18 and stopping before the ROM image which begins at block 1433 for the AR24C18. Returns having done nothing if the source area matches perfectly (except for variable values,) otherwise stops on the first difference.

?kernel compares kernel binary leaving a list of differing lesser block numbers on the stack.

?ico compares icon image binary leaving a list of differing lesser block numbers on the stack.

?rom compares ROM binary leaving a list of differing lesser block numbers on the stack.

?bin compares all these binaries leaving a list of differing lesser block numbers on the stack.

5. Colorforth vocabulary

The following is a description of colorForth words. The stack effect for each word is placed to the right of the word name. The stack effect notation is as follows "(" begins the stack effect and ")" ends the stack effect. Variables to the left of the "-" are consumed from the stack and variables to the right of the "-" are placed on the stack by the word being described. If there is no "-" preceding the variables they are consumed by the word. If there is a "-" preceding the variables they are produced by the word. As with the variables, flags that are set by words precede "-" and flags that are used by the words are placed after the "-". Flags, however do not reside on the stack. Therefore they may not be in a deterministic state unless changed by the previous command. For this reason flags are represented in *italics*.

N	Number
A	Cell address
Ba	Byte address
M	Multiplier
Q	Quotient
D	Divisor
B	Block number
X	Non specified 1 cell of memory
R	Remainder
C	Cylinder or character
F	Flag
ZF	Zero flag
OF	Overflow flag
CF	Carry flag
SF	Sign flag

5.1 ColorForth

5.1.1 ! (a) Macro

Store word at address.

5.1.2 ! (a)

Store word at address.

5.1.3 !back (b)

Writes **nc** cylinders to a (native) floppy or (Windows) **OkadBack.cf** file from RAM block **b.**

5.1.4 * (n1 n2 – n3 *CF OF)* Macro

32 bit product.

5.1.5 * (n1 n2 – n3 *CF OF)*

32 bit product

5.1.6 */ (m n d – q *CF OF ZF SF*) Macro

64 bit product then quotient.

5.1.7 */ (m n d – q *CF OF ZF SF)*

64 bit product then quotient.

5.1.8 + (n1 n2 – n3 *CF OF ZF SF)* Macro

32 bit sum

5.1.9 + (n1 n2 – n3 *CF OF ZF SF)*

32 bit sum

5.1.10 +! (n a - *CF OF ZF SF*) Macro

Adds "n" to the addressed variable.

5.1.11 +at (n1 n2 - *CF OF ZF SF*) Kernel

Adds two sixteen bit numbers from the stack to the variable xy.

5.1.12 – (n1 – n2)

Ones complement. Not.

5.1.13 –if (*SF - SF*) Macro

Jumps if previous statement sets sign flag to 0. Sign flag is not reset or consumed.

5.1.14 –offset (n1 – n2 *CF OF ZF SF*)

Fetches offset and subtracts from n1.

5.1.15 , (n) Kernel

Lays down 4 bytes of the value given in the dictionary.

5.1.16 . (n) Kernel

Formats a signed number in decimal with leading 0's suppressed.

5.1.17 ; Macro

Exit from a definition. If immediately preceded by a call turns it into a direct jump. Otherwise compiles a RET.

5.1.18 / (n d – q *CF OF ZF SF*) Macro

Divide and do not return remainder.

5.1.19 / (n d – q *CF OF ZF SF*)

Divide and do not return remainder.

5.1.20 /mod (n d – r q *CF OF ZF SF*) Macro

Divide places the quotient at the top of the stack and remainder in the next entry.

5.1.21 ? (- *SF ZF*) Macro

Test bits and set flags but only for literals. Does a bitwise and without changing data.

5.1.22 ?dup (n1 – n1 | n1 n1) Macro

If h (see h) indicates only one byte exists at List, and if that byte is a drop, then h is decremented (List is not changed) and nothing is compiled. Otherwise we compile a dup.

5.1.23 ?lit (-*ZF(0)* | n *ZF(1)*) Kernel

Looks at preceding instructions to see if there is a literal to unravel. If one is found, it is returned with ZF flag set to 1. Otherwise returns zero flag set to 0 and no value. If preceded by DUP literal then kills both dup and the literal; if there is just a literal, it kills the literal and inserts a DROP.

NOTE: only macros.

5.1.24 @ (a) Macro

Fetch value at address.

5.1.25 @ (a)

Fetch value at address.

5.1.26 @back (b)

Reads `nc` cylinders from a (native) floppy or (Windows) `OkadBack.cf` file into RAM at block `b`

5.1.27 @cyls (a c n)

Same as reads. Reads `n` cylinders to cell address `a` from cylinder `c` on backup. For native, the read is direct from floppy. For Windows, the whole `OkadBack.cf` file is read into RAM at block 3000 and then the cylinder(s) in question are copied from there.

5.1.28 0 Macro

Identical to number 0. Uses XOR to produce value at top of stack.

5.1.29 1, (n) Kernel

Lays down 1 byte of the value given in the dictionary in little endian format.

5.1.30 1word (-n)

Returns first cell of word.

5.1.31 2* (n1 – n2 *SF ZF*) Macro

Shift left 1 bit. Multiply by two.

5.1.32 2, (n) Kernel

Lays down 2 bytes of the value given in the dictionary in little endian format.

5.1.33 2/ (n1 – n2 *SF ZF CF*) Macro

Shift right 1 bit. Divide by 2.

5.1.34 2/ (n1 – n2 *SF ZF CF*)

Shift right 1 bit. Divide by 2.

5.1.35 2emit (n) Kernel

Draws a 2x size character. Unlike emit there is no check for carriage return.

5.1.36 3, (n) Kernel

Lays down 3 byte of the value given in the dictionary in little endian format.

5.1.37 5*

Emit letters.

5.1.38 7pop Macro

Pop W (EDI).

5.1.39 7push Macro

Push W (EDI)

5.1.40 a Macro

Moves value at register 2 to register 0. EDX to EAX.

5.1.41 a! Macro

Moves value at register 0 to register 2. EAX to EDX

5.1.42 a,

Compile word address.

5.1.43 abs (n1 – n2)

Returns absolute value.

5.1.44 abort Kernel

Terminates execution and goes back to keyboard processing. Sets editor pointers if W addresses block 18 or higher. Empties return stack and resets a number of cells in the spaces list. Echoes a?.

5.1.45 accept Kernel

Sets the keyboard for beginning of an alpha string and begins accepting it. If first key is not a shift, "first" changes active shift table to the next one defined in memory; accepts a new Shannon coded string; does what is appropriate with it in aword ; and accepts the next string.

5.1.46 align Macro

Aligns next call to end on word boundry.

5.1.47 altfrm

Byte address of alternate frame.

5.1.48 and (n1 n2 – n3 *ZF SF*) Macro

Bitwise and. Zero flag and sign flag are set accordingly.

5.1.49 aper (-a) Kernel

Byte address of aperture used for screen refresh

5.1.50 at (n1 n2) Kernel

Moves two sixteen bit numbers from the stack to the xy location in memory. Sets cursor location to given pixel coordinates.

5.1.51 b! (n ba) Macro

Store n in literal byte address ba.

5.1.52 beep

Return;

5.1.53 black

Black foreground

5.1.54 blk Kernel

Variable used for current block number being edited.

5.1.55 blks (n1 – n2)

Multiplies number by 256. 256 is the number of words/block.

5.1.56 block (n – a)

Returns word address of a given block number.

5.1.57 blue

Blue. Foreground.

5.1.58 boot Kernel

Pulls the CPU reset line using the keyboard processor output port pin attached to same.

5.1.59 box (n1 n2) Kernel

Draws a solid box from upper left corner (cursor position xy) to lower right corner just inside absolute position limx limy limiting limx limy to screen size before start of arithmetic.

5.1.60 bswap (n1 – n2) Macro

Swap bytes.

5.1.61 buffer Kernel

Variable used as a four byte memory buffer.

5.1.62 bye Kernel

Terminates colorForth in an orderly fashion so that it may be called from and return to a BAT file.

5.1.63 c Kernel

Empties data stack for keyboard task ONLY - first cell to be stored into the stack goes at keyd.

5.1.64 cbg

Change to big green. Toggle decimal to hex.

5.1.65 cbw

Command block wrapper.

5.1.66 cby

Change to big yellow. Toggle decimal to hex.

5.1.67 cad (- a)

Adds 3 to blk (current block). Kernel hook to cursor address variable.

5.1.68 cf

Displays double size colorForth.

5.1.69 change

Change color of word behind cursor. See section 3 colorforth human interface.

5.1.70 clog Kernel

Closes the file.

5.1.71 color (n) Kernel

Sets foreground for the given pixel value.

5.1.72 comment

Find white word in OKAD. See section 3 colorforth human interface.

5.1.73 copy (n) Kernel

Copies the current block and shadow to block to the given block number, n, leaving the editor set to the destination block. It will not copy anything to a block number less than 12.

5.1.74 cpoint Kernel

Variable is a vector for post 18 LOAD cursor correction on abort.

5.1.75 curs Kernel

Cursor position (absolute cell address).

5.1.76 cr Kernel

Invokes a "carriage return" by moving xy down ih pixels and left to l.

5.1.77 csg

Change word behind cursor to small green.

5.1.78 csw (a n f)

Command status wrapper. Increment cbw tag.

5.1.79 csy

Change word behind cursor to small yellow.

5.1.80 ctg

Change word behind cursor to green.

5.1.81 ctt

Change word behind cursor to text (white).

5.1.82 cty

Change word behind cursor to yellow.

5.1.83 debug Kernel

Sets cursor position to …Then displays four numbers: Keyboard task saved return stack pointer, top value on return stack, value of scr, and the running task's data stack pointer.

5.1.84 def

Find definition. See section 3 Colorforth human interface.

5.1.85 device (n - a) Kernel

Device searches for a PCI device class whose number is aligned in the high octet of the argument, returning the base of its configuration registers. It searches downward starting at the literal address for literal number of steps, indexing device number field. If no match, returns the last device that was examined. This is why not finding the display often crashes the machine.

5.1.86 digit (n) Kernel

Displays a digit value.

5.1.87 drop (n) Kernel

(LODS instruction) Removes the top entry of the stack

5.1.88 dup (n1 – n1 n1) Kernel

Duplicates the number on the top of the stack.

5.1.89 dup (n1 – n1 n1)

Duplicates the number on the top of the stack

5.1.90 e (n) Kernel

Starts editor in block n. If the block number is not provided the editor starts in block 18.

5.1.91 edit (n) Kernel

Starts editor in block n. If the block number is not provided the editor starts in block 18.

5.1.92 ekt Kernel

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.

5.1.93 emit (n) Kernel

Emit draws the character code given at the position xy using 16x24 patterns in icons with fore pixel value, updating xy . It checks for carriage return before start.

5.1.94 empt Kernel

Restores the forget point in mk and sets class zero.

5.1.95 empty

Empties dictionary and displays logo.

5.1.96 end (a) Macro

Jump to begin.

5.1.97 erase (b n) Kernel

Stores zeroes into a string of blocks starting at b for n consecutive blocks. Shadows are included

5.1.98 f

Find next occurrence after a find is executed. See section 3 Colorforth human interface.

5.1.99 fall

Find all words regardless of color. See section 3 Colorforth human interface.

5.1.100 fifo (n) Kernel

Synonym for drop. Removes top stack entry.

5.1.101 fill (n1 a n2)

Writes n1 into a cell string address. Writes the third stack entry into the address that is in the second stack entry.

5.1.102 find

Finds a forth (green) word explained in cF user interface. Following short compiled word blocks 18 through number of cylinders searched for a 32 bit match that means 1st 4 bytes of name.

5.1.103 fk

Key in edit keyboard. Drops key and block number.

5.1.104 fkc

Find def of word to left of cursor.

5.1.105 format (a c – a c) Kernel

formatf is given cell address a of a format command under a cylinder number, with a cylinder's worth of format data in buffer . Sets DMA for write from the buffer, sets cylinder in command and ensures motor running, executes the format command, then resets DMA for a cylinder write.

This is a noop in Windows. The reason is explained in section TBD.

5.1.106 forth Kernel

Sets state of system so that subsequent definitions are regular forth definitions and can be used by the interpreter unlike macros.

Forth sets aDEFINE to forthd. The definition behavior is to add an entry to the forth dictionary as appropriate. The encoded name is taken from the cell preceding the cell address in W. The value for the definition is the contents of H. Last is left holding the cell address of the dictionary pointer table just filled. List is set to the address past the call to this function. lit is reset to adup. Finally if class is nonzero, we jump off to it rather than returning. On normal exit, 1 is bytadr of new name in dictionary, 2 is celladr of value field.

5.1.107 for Macro

Push count onto the return stack begin.

5.1.108 fov Kernel

fov is an abstract scale factor used in some applications.

5.1.109 freeze Kernel

freeze is a simple graphic process. Usage is: : defn ... freeze <thingstodo> ; The graphic task will endlessly do <thingstodo>. screen is the address of the thingstodo.

5.1.110 from (n)

From is the same as find but searches from a specific block number n. See section 3 colorforth human interface.

5.1.111 graphic Kernel

Graphic is a no-op that is called from show. It looks as though either it is never used or show would have to be patched.

5.1.112 green

Green foreground

5.1.113 h Kernel

Variable containing byte address of next available location in the dictionary. here returns this address.

5.1.114 h. (n) Kernel

Displays an 8 digit hex number.

5.1.115 h.n (n1 n2) Kernel

Displays the low order w hex digits of a number.

5.1.116 here (a) Kernel

Returns location counter H without destroying list.

5.1.117 hsvv Kernel

hsvv is stuffed from colorForth code with byte addresses and values by hardsim.

5.1.118 i Macro

Copy loop index to data stack

5.1.119 if (*ZF - ZF)* Macro

Jumps if previous statement is 0 or equal. If does not reset or consume the zero flag.

5.1.120 jump (n) Kernel

Jump implements a transfer vector. Usage: i jump zero one two three.... Indexes the ith call and makes a jump to the routine in question. Jump exits the calling definition as well.

5.1.121 keyboard Kernel

Displays keyboard mapping in lower right of display.

5.1.122 key?

Exits calling definition if key is struck.

5.1.123 last Kernel

? last 0 , (must follow H till patch in 150 fixed)

5.1.124 less (n1 n2 – n1 n2 *ZF*) Kernel

Execution time behavior for < .

5.1.125 line (n1 n2) Kernel

Line draws a line of fore pixels, n2 pixels long, starting at the cursor position xy adjusted to the left by 2 n1 pixels. Line advances the cursor downward one pixel afterward.

5.1.126 lit (n)

Finds compiled literal. See section 3 Colorforth human interface.

5.1.127 lm (n) Kernel

Lm sets the left margin.

5.1.128 load (b) Kernel

Sets the interpreter pointer to the start of the given block and begins interpreting it.

5.1.129 loads (b n)

Load block b and n successive blocks. NOTE: This will load successive executable blocks shadows are not counted (ie 2 96 loads will load blocks 96 and 98).

5.1.130 logo

Displays colorForth logo.

5.1.131 loop Macro

repeat

5.1.132 macro Kernel

Sets state of system so that subsequent definitions are regular macro definitions and are not executable from the interpreter as forth commands are.

Forth sets aDEFINE to macrod. The definition behavior is to add an entry to the macro dictionary as appropriate. The encoded name is taken from the cell preceding the celladdr in W. The value for the definition is the contents of H. last is left holding the cell address of the dictionary pointer table just filled. List is set to the address past the call to this function. lit is reset to adup. Finally if class is nonzero, we jump off to it rather than returning. On normal exit, 1 is bytadr of new name in dictionary, 2 is the cell address of value field.

5.1.133 mark Kernel

Saves a forget point in mk - #macros, #forths, and H.

5.1.134 max (n1 n2 – n)

Returns maximum value.

5.1.135 min (n1 n2 – n)

Returns minimum value

5.1.136 minute

Minutes past midnight.

5.1.137 mod (n d – r) Macro

Divide but only return remainder

5.1.138 move (a1 a2 n)

Copies contents of address a1, which is the third entry in the stack, into address a2, which is in the second entry in stack.

5.1.139 negate (n1 – n2)

Negates value 2's complement.

5.1.140 next (a – *SF OF ZF*) Macro

Decrement count. Jump if not zero to for. Pop return stack.

5.1.141 nc Kernel

Variable containing number of cylinders.

5.1.142 nip (n1 n2 n3 – n1 n3) Macro

Removes the second entry in the stack.

5.1.143 nop Macro

No operation

5.1.144 null

Return. Useful as null jump word.

5.1.145 octant(p q - p q n) Kernel

?

5.1.146 offset Kernel

Offset is a block number added to all regular uses of BLOCK.

5.1.147 ok

Start register display.

5.1.148 olog Kernel

Olog opens a file by the name OkadSim.log which will be created if not there, and which will be truncated if it exists. Due to new clib extra argument, if it is created it will probably be read only.

5.1.149 or (n1 n2 – n3 *OF CF ZF SF*) Macro

or is an exclusive or, **xor**. *OF* and *CF* are always set to 0. *ZF* and *SF* are set according to the result.

5.1.150 or! (n1 a - *OF CF ZF SF*) Macro

Bitwise or with location in memory. *OF* and *CF* are always set to 0. *ZF* and *SF* are set according to the result.

5.1.151 over (n1 n2 n3 – n1 n2 n3 n2) Macro

Copies 2nd entry in stack to top of stack.

5.1.152 p@ Macro

Read register.

5.1.153 pad Kernel

pad is Chuck's best redesign of the control pad definition mechanism as of the time he stopped maintaining the kernel.

5.1.154 pause Kernel

Gives up the machine. Saved state for current task is who called pause, data stack pointer on return stack, and R saved in the ROUND table.

5.1.155 pci (a – n) Kernel

Pci fetches a cell from PCI configuration space given address. Uses Config Mechanism 1 as defined by PCI Local Bus Spec 2.0

5.1.156 pop (-n) Macro

Pops return stack to top of data stack.

5.1.157 push (n) Macro

Pushes top of data stack on to top of return stack

5.1.158 rback (b – n) Kernel

Reads OkadBack.cf to RAM block b for n blocks.

5.1.159 reclr

Recolor cycles yellow green white

5.1.160 read (c a - `c `a) Kernel

Reads the given cylinder into the given CELL address, returning both incremented for the next cylinder.

5.1.161 reads (a c n)

Reads `n` cylinders to cell address `a` from cylinder `c` on backup. For native, the read is direct from floppy. For Windows, the whole `OkadBack.cf` file is read into RAM at block 3000 and then the cylinder(s) in question are copied from there.

5.1.162 red

Red foreground

5.1.163 rm (n) Kernel

Sets right margin.

5.1.164 ruu

Boot.

5.1.165 save

Write entire image. Save all changes made during current OKAD session. When OKAD is restarted the changes will be present.

save writes the active disk image (`nc` cylinders at block 0) to (native) floppy or (Windows) `OkadWork.cf` file. This includes boot, native kernel, icon table and source code.

5.1.166 screen

Produces a 1024x768 screen of current color.

5.1.167 serve Kernel

Serve is a simple receive process, endlessly performing PAUSE <thingstodo> where the latter is in receive.

5.1.168 sec

Seconds past midnight.

5.1.169 show Kernel

Show is a simple graphic process. Usage is: : defn ... show <thingstodo> ; The graphic task will endlessly do <thingstodo> switchm graphic is a fixed nop; switchm refreshes screen and PAUSEs.

5.1.170 silver

Silver foreground.

5.1.171 sp Kernel

sp is the vector table for token types while interpreting. It is initialized later; some cells stuffed as state changes during compile/interpret; and these are reset on abort.

5.1.172 space Kernel

Advances cursor, xy, by one character.

5.1.173 swap Macro

Exchanges top of stack with next entry.

5.1.174 stop Kernel

Stops the floppy motor and points trash floppy buff.

5.1.175 switch Kernel

switch in the native system copies RAM bitmap into the graphic frame buffer. For functional equivalence, in Windows we invalidate the entire current window and call for a complete update from our DIB.

5.1.176 t,

Compile word into workspace.

5.1.177 td (a n)

Transfer descriptor.

5.1.178 text Kernel

Makes word white

5.1.179 tic (–ba) Kernel

Tic used at keyboard or interpretively only; returns byte address of a Forth word. To use type tic and hit space. Tic will search for this word and leave it's address on the stack. See section 1.

5.1.180 time (n1) Macro

Current value of processor time stamp. Pentium cycle counter, calibrate to actual clock rate.

5.1.181 top Kernel

Sets xy and xycr to (lm, vs).

5.1.182 trash Kernel

??????? LABEL trash buffer 4* ,

5.1.183 tsim Kernel

tsim replaces type in the transistor simulation loop. Sets W and obtains contact voltages. On completion processes delta charge.

5.1.184 u+ (n3 n2 n1 – n4 n2) Macro

Adds top of stack to third entry. n4 = n3 + n1.

5.1.185 unpack (w - w' c) Kernel

unpack removes the leftmost / high order Shannon coded cF character from the packed cell given.

5.1.186 v+ (v1 v2 – v3)

Adds a 2 dimensional vector. v = (x,y).

5.1.187 vesa Kernel

Vesa is a mode number for INT 10 4F02. Bit 14 means use linear flat frame buffer. Mode number is VESA-defined (Ralf Brown):4118 is 1024x768x16m 4123 is not standard - 1600x1200x16m?? Note 411B is 1280x1024x16m Note that DAC may be limited to 6 bits after mode set, may need to do 4F08 afterward 4118 seems to be 24 bit not 32 bit 888`wadr (-a)` Address of word behind cursor.

5.1.188 warm Kernel

warm is in the dictionary; analogous to RELOAD.

5.1.189 wback (b n) Kernel

wback writes RAM block b for n blocks to start of OkadBack.cf

5.1.190 white

White foreground.

5.1.191 winver (- t | f) Kernel

winver returns 1 if windows version, 0 if native, with indicators set for if .

5.1.192 wlog (a n1) Kernel

wlog writes a string into the file given *byte* address and *byte* count.

5.1.193 word (*) Kernel

Accepts a cF word from keyboard and returns words cells of Shannon coded characters on the stack.

5.1.194 write (a c - a' c') Kernel

Writes the given cell address to the given cyl and incrs

5.1.195 writes

Address, cylinder, cylinder count

WARNING: Do not hit any keys while floppy is being written.

5.1.196 wrtboot

Writes boot and kernel. wrtboot writes the boot and kernel image from block 0 to a native floppy, as an economical way to update the nativekernel on floppy after executing `ati` or `nvidia` in the `floppy` utility. It does nothing on Windows systems.

5.1.197 xy Kernel

xy position vector for text display cursor, packed halfcell values y first x second so that fetched as a cell it looks like xy.

5.1.198 `yel`

Find yellow word. See section 3 colorforth human interface.