valFORTH T.M.

VALPAR
INTERNATIONAL

valFORTH T.M

VALPAR
INTERNATIONAL

VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141

valFORTH T.M.
SOFTWARE SYSTEM
for ATARI*

GENERAL UTILITIES AND VIDEO EDITOR

*Atari is a trademark of Atari, Inc., a division of Warner Communications

valFORTH T.M.

Screen Oriented Video Editor

Version 1.1
March 1982

The FORTH language is a very powerful addition to the Atari home computer. Programs which are impossible to write in BASIC (usually because of limitations in speed and flexibility) can almost always be written in FORTH. Even when one has mastered the BASIC language, making corrections or additions to programs can be tedious. The video editor described here removes this problem from the FORTH environment. Similar to the MEMO PAD function in the Atari operating system, this editor makes it possible to insert and delete entire lines of code, insert and delete single characters, toggle between insert and replace modes, move entire blocks of text, and much more.

valFORTH T.M.
SOFTWARE SYSTEM

GENERAL UTILITIES AND VIDEO EDITOR

Stephen Maguire
Evan Rosen

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

GENERAL UTILITIES and VIDEO EDITOR

valFORTH UTILITIES/EOITOR USER'S MANUAL

Table of Contents

Page

XI. valFORTH 1.1 OISPLAY-ORIENTED VIOEO EOITOR
A user's manual for the valFORTH video editor

a) OVERVIEW 1
b) ENTERING THE EDIT MOOE 2
c) CURSOR MOVEMENT 5
d) EOITING COMMANDS 6
e) STORAGE BUFFER MANAGEMENT 8
f) CHANGING SCREENS; SAVING; ABORTING 13
g) SPECIAL COMMANDS 14
h) SCREEN MANAGEMENT 16
i) EDITOR COMMAND SUMMARY 18

XII. STRINGS, ARRAYS, CASE STATEMENTS, OOUBLE NUMBER EXTENSIONS

a) STRING PACKAGE 1
b) ARRAYS, TABLES, VECTORS 5
c) CASE:, CASE, SEL, CONO 8
d) DOUBLE NUMBER EXTENSIONS 14

XIII. HI-RES TEXT, MISC. UTILITIES, TRANSIENTS

a) HI-RESOLUTION (8 GR.) TEXT OUTPUT 1
b) MISCELLANEOUS UTILITIES 5
c) TRANSIENTS (DISPOSABLE ASSEMBLERS, ETC.) 9

XIV. UTILITIES/EOITOR SUPPLIED SOURCE LISTING

Overview

This editor is a powerful extension to the valFORTH system designed specifically for the Atari 400/800 series of microcomputers. The main purpose for this editor is to give the FORTH programmer an easy method of text entry to screens for subsequent compilation. The editor has four basic modes of operation:

1) It allows entering of new text to a FORTH screen as though typing on a regular typewriter.

2) It allows quick, painless modification of any text with a powerful set of single stroke editing commands.

3) It pinpoints exactly where a compilation error has occurred and sets up the editor for immediate correction and recompilation.

4) Given the name of a precompiled word, it locates where the original text definition of the word is on disk, if the "LOCATOR" option had been selected when the word was compiled.

The set of single stroke editing commands is a superset of the functions found in the MEMO PAD function of the standard Atari operating system. In addition to cursor movement, single character insertion/deletion, and line insertion/deletion, the editor supports a clear-to-end-of-line function, a split command which separates a single line into two lines, and a useful insert submode usually found only in higher quality word processors.

In addition, there are provisions for scrolling both forwards and backwards through screens, and to save or "forget" any changes made. This is useful at times when text is mistakenly modified.

Also provided is a visible edit storage buffer which allows the user to move, replace, and insert up to 320 lines of text at a time. This feature alone allows the FORTH programmer to easily reorganize source code with the added benefit of knowing that re-typing mistakes are avoided. Usage has shown that once edit-buffer management is learned, significant typing and programming time can be saved.

For those times when not programming, the editor can double as a simple word processor for writing letters and filling other documentation needs. The best method for learning how to use this powerful editor is to enter the edit mode and try each of the following commands as they are encountered in the reading.

As stated above, there are four ways in which to enter the video editor. The following four commands explain each of the possibilities. Note that the symbol "<ret>" indicates that the "RETURN" key is to be typed.

V   view screen   (scr# ---)

To edit a screen for the first time, the "View" command is to be used. The video display will enter a 32 character wide mode and will be broken into three distinct sections. For example,

50 V <ret>

should give something like the display shown in fig. 1.

```
 Screen # 50    U    #Bufs:  5
```

```
■ ( Example screen )  ( line 0 )

  : TEST1             ( line 2 )
    10 0
    DO
      I CR .
    LOOP ;

  : OCTAL                ( --- )
    8 BASE ! ;

  : +C!
    DUP C@ ROT +
    SWAP C! ;

                   ( bottom line )
```

```



```

Fig. 1

The top window, composed of a single line, indicates in decimal which screen is currently being edited. One should always make a practice of checking this screen number to insure that editing will be done on the intended screen. Often times, when working with other number bases, the wrong screen is called up accidentally and catching this mistake early can save time. Also shown is the size of the edit buffer (described later). In this example, the buffer is five lines in length. This window is known as the heading window.

FORTH screens typically are 1K (1024 characters) long. Since it is impossible to see an entire screen simultaneously, this editor reveals only half a screen at a time. There is an "upper" half and a "lower" half. In the center of the heading window, either a "U" or an "L" is displayed indicating which half of the current screen is being viewed. If the valFORTH system is in the half-K screen mode, neither "U" nor "L" is displayed since an entire half-K screen can be viewed at one time. In figure 1, the upper half of a full-K screen is being viewed.

The second window (the text window) contains the text found on the specified screen. This window is 32 characters wide and 16 lines high. The white cursor (indicated by the symbol "■") will be in the upper-lefthand corner of the screen awaiting editing commands.

The final five-line window found at the bottom of the screen is known as the buffer window. This is used for advanced editing and is described in greater detail in the section entitled "Buffer Management."

L re-edit last screen (---)

This command is used to re-edit the "Last" screen edited. It functions identically to the "V" command described above, except no screen number is specified.

Example: L <ret> (re-edit screen 50)

WHERE find location of error (---)

If, when compiling code, a compilation error occurs, the WHERE command will enter the edit mode and position the cursor over the last letter of the offending word. The word can then be fixed and the screen can be re-compiled. Bear in mind that using the WHERE command prior to any occurrence of an error could give strange results.

LOCATE locate definition cccc (---)

Once source text has been compiled into the dictionary, it loses easy readability to all but experts of the FORTH language. Often times, though, it is helpful to see what the original source code was. The DECOMP command found in the debugger helps tremendously in this regard, however, some structures such as IF and DO are still difficult to follow. For this reason, the LOCATE command is included with the editor.

This command accepts a word name, and if at all possible it will actually direct the editor to load in the screen where that word was defined. This is very helpful at times when one cannot remember where the original text was. If the screen shown in figure 1 were loaded and the command

LOCATE +C! <ret>

were given, the editor would call up screen 50 and position the cursor over the word ":" which is the beginning of the definition for "+C!". Typically, the LOCATE command will point to ":" , "CODE" , "CONSTANT" , and other defining words.

There is a drawback to this feature, however. In order to call up any word, the LOCATE command must know where the word actually is. Normally, when a word is compiled, there is no way of knowing where it was loaded from. Thus for the LOCATE command to work, each time a word is entered into the dictionary, three extra bytes of memory must be used to store this lookup information. For an application with many words, these extra bytes per word add up quickly, and this is not always desirable. For this reason, the LOCATOR command (described below) allows the user to enable or disable the storage of this lookup information. Only words that were compiled with the LOCATOR option selected can be located. If a word cannot be located, the user is warned, or if the DEBUGGER is loaded, the word is DECOMPed giving pseudo original code.

LOCATOR enable/disable location (ON/OFF ---)

In order for a word to be locatable using LOCATE, the LOCATOR option must have been selected prior to compiling the word. The LOCATOR option is selected by executing "ON LOCATOR" and deselected by executing "OFF LOCATOR". For example:

```
ON LOCATOR
: PLUS   ." = " + . ;              (partial view of a screen)
: STAR   42 EMIT ;
OFF LOCATOR
: NEGATE  MINUS ;
```

Only the words PLUS and STAR can be located. NEGATE cannot be located since the LOCATOR option was disabled. If the DEBUGGER were loaded, NEGATE would be decompiled (see the debugger), otherwise, the user would be given a warning. The default value for LOCATOR is OFF.

#BUFS set buffer length (#lines ---)

The #BUFS command allows the user to specify the length (in terms of number of lines) of the special edit storage buffer. The power of the edit buffer lies in the number of lines that can be stored in it. Although the default value is five, practice shows that at least 16 lines should be set aside for this buffer. The maximum number of lines allowable is 320 which is enough to hold 20 full screens simultaneously.

The following sections give a detailed description of all commands which the video editor recognizes. A quick reference command list can be found following these descriptions.

Cursor Movement

When the edit mode is first entered via the "V" command, a cursor is placed in the upper lefthand corner of the screen. It should appear as a white block and may enclose a black letter. Whenever any key is typed and it is not recognized as an editor command, it is placed in the text window where the cursor appears. Likewise, any line functions (such as delete line) work on the line where the cursor is found.

ctrl ∧ , ctrl ∨ , ctrl < , ctrl >   move-cursor commands

To change the current edit line or character, one of four commands may be given. These are known as cursor commands. They are the four keys with arrows on them. These keys move the cursor in the direction specified by the arrow on the particular key pressed. There are times, however, when this is not the case.

If the current cursor line is the topmost line of the text window, and the "cursor-up" command is issued (by simultaneously typing "ctrl" and "up-arrow"), the cursor will move to the bottom line of the text window. Likewise, a subsequent "cursor-down" command would return the cursor to the topmost line of the window. Similarly, if the cursor is positioned on the leftmost edge and the "cursor-left" command is given, the cursor will "wrap" to the rightmost character ON THE SAME LINE. Issuing "cursor-right" will wrap back to the first character on that line.

RETURN   next-line command

Normally, the RETURN key positions the cursor on the first character of the next line. If RETURN is pressed when the cursor is on the last line of the text window (i.e., when the last text line of the screen is current), the cursor is positioned in the upper lefthand corner of the screen.

TAB   tabulate command

The TAB key is used to tabulate to the next fixed four column tabular stop to the right of the current cursor character. TABbing off the end of the current line simply places the cursor at the beginning of that same line.

NOTE:

Many commands in the editor will "mark" a current FORTH screen as updated so that any changes made can be preserved on disk. As simple cursor movement does not change the text window in any way, these commands never mark the current FORTH screen. See the section on screen management for more information.

Editing Commands

Editing commands are those commands which modify the text in some predefined manner and mark the current FORTH screen as updated for later saving.

ctrl INS — character insert command

When the "insert-character" command is given, a blank character is inserted at the current cursor location. The current character and all characters to the right are pushed to the right by one character position. The last character of the line "falls off" the end and is lost. The inserted blank then becomes the current cursor character. This is the logical complement to the "delete-character" command described below.

ctrl DEL — delete character command

When the "delete-character" command is issued, the current cursor character is removed, and all characters to the right of the current cursor character are moved left one position, thus giving a "squeeze" effect. This is normally called "closing" a line. The rightmost character on the line (which was vacated) is replaced with a blank. This serves as the logical complement to the "insert-command" described above.

shift INS — line insert command

The "line-insert" command inserts a blank line between the current cursor line and the line immediately above it. The current line and all lines below it are moved down one line to make room for the new line. The last line on the screen falls off the bottom and is lost. If this command is accidentally typed, the "oops" command (ctrl-O) described later can be used to recover from the mistake. Also see the "from buffer" command described in the section on buffer management for a similar command. This command serves as the logical complement to the "line-delete" command described below.

shift DEL — line delete command

The "line-delete" command deletes the current cursor line. All lines below the current line are brought up one line and a blank line fills the vacated bottom line of the text window. The deleted line is lost. If this command is accidentally issued, recovery can be made by issuing the "oops" command (ctrl-O) described later. Also see the "to-buffer" command described in the section on buffer management for a similar command. The "delete-line" command serves as the logical complement to the "line-insert" command.

ctrl H erase to end of line

The "Hack" command performs a clear-to-end-of-line function. The current cursor character and all characters to the right of it on the current line are blank filled. All characters blanked are lost. The "oops" command described later can be used to recover from an accidentally hacked line.

ctrl I insert/replace toggle

In normal operation, any key typed which is not recognized by the editor as a control command will replace the current cursor character with itself. This is the standard replace mode. Normally, if one wanted to insert a character at the current cursor location, the insert character command would have to be issued before any text could be entered. If inserting many characters, this is cumbersome.

When active, the insert submode automatically makes room for any new characters or words and frees the user from having to worry about this. When the editor is called up via the "V" command, the insert mode is deactivated. Issuing the insert toggle command will activate it and the cursor will blink, indicating that the insert mode is on. Issuing the command a second time will deactivate the insert mode and restore the editor to the replace mode. Note that while in the insert mode, all edit commands (except BACKS, below) function as before.

BACKS delete previous character

The BACKS key behaves in two different ways, depending upon whether the editor is in the insert mode or in the replace mode. When issued while in the replace mode, the cursor is backed up one position and the new current character is replaced with a blank. If the cursor is at the beginning of the line, the cursor does not move, but the cursor character is still replaced with a blank.

If the editor is in the insert mode, the cursor backs up one position, then deletes the new current cursor character and then closes the line. If the cursor is at the beginning of the line, the cursor remains in the same position, the cursor character is deleted and the line closed.

NOTE:

As all of the above commands modify the text window in some manner, the screen is marked as having been changed. This is to be sure that all changes made are eventually saved on disk. The "quit" command described in the section on changing screens allows one to unmark a screen so that major mistakes need not be saved.

Buffer Management

Much of the utility of the valFORTH editor lies in its ability to temporarily save text in a visible buffer. To aid the user, it is possible to temporarily send text to the buffer and to later retrieve it. This storage buffer can hold as many as 320 lines of text simultaneously. This buffer is viewed through a 5 line "peephole" visible as the last window on the screen. Using this buffer, it is possible to duplicate, move, and easily reorganize text, in addition to temporarily saving a line that is about to be edited so that the original form can be viewed or restored if necessary. The following section will explain exactly how to accomplish each of these actions.

ctrl T — to buffer command

The "to-buffer" command deletes the current cursor line, but unlike the "delete-line" command where the line is lost, this command moves the "peephole" down and copies the line to the bottom line of the visible buffer window. This line is the current buffer line. The buffer is rolled upon each occurrence of this command so that it may be used repeatedly without the loss of stored text.

For example, if the cursor is positioned on line eight of the display shown in figure 1 and the "to-buffer" command is issued twice, the final result will be as shown in figure 2.

ctrl F — from buffer command

The "from-buffer" command does exactly the opposite of the "to-buffer" command described above. It takes the current buffer line and inserts it between the current cursor line and the line above it. The cursor line and all lines below it are moved down one line with the last line of the text window being lost. If the cursor were placed on line 14 of the above screen display and the "from-buffer" command were issued once, the display in figure 3 would result.

```
Screen # 50     U     #Bufs: 5
```

```
           ( Example screen )  ( line 0 )

           : TEST1             ( line 2 )
             10 0
             DO
               I CR .
             LOOP ;

Current:   ■
           : +C!
             DUP C@ ROT +
             SWAP C! ;

                          ( bottom line )
```

```
Current:   : OCTAL                ( --- )
             8 BASE ! ;
```

fig. 2

```
                    Screen # 50    U    #Bufs: 5
```

```
                    ( Example screen )  ( line 0 )

                    : TEST1             ( line 2 )
                      10 0
                      DO
                        I CR .
                      LOOP ;


                    : +C!
                      DUP C@ ROT +
                      SWAP C! ;
Current:          ■   8 BASE ! ;
                              ( bottom line )
```

```
line was              8 BASE ! ;
rolled to
the top

Current:            : OCTAL               ( --- )
```

fig. 3

If the "from-buffer" command is issued again, then lines 13 through 15 of the text window would look like:

```
Current:            : OCTAL               ( --- )
                      8 BASE ! ;
                              ( bottom line )
```

fig. 4

Note that a block of text has been moved on the screen. Larger blocks of text can be moved in the same manner.

ctrl K copy to buffer command

The "copy-to-buffer" command takes the current cursor line and duplicates it, sending the copy to the buffer. This commands functions identically to the "to-buffer" command described above, except that the current cursor line is NOT deleted from the text window.

ctrl U copy from buffer

The "copy-from-buffer" command replaces the current cursor line with the current buffer line. This command functions identically to the "from-buffer" command described above, except that the buffer line is not inserted into the text window, it merely replaces the current cursor line. The "oops" command described below can be used to recover from accidental usage of this command.

ctrl R roll buffer

The "roll-buffer" command moves the buffer "peephole" down one line and redisplays the visible window. If the buffer were the minimum five lines in length, the bottom four lines in the window would move up a line and the top line would "wrap" to the bottom and become the current buffer line. If there were more than five buffer lines, the bottom four lines would move up a line, the topmost line would be pushed up behind the peephole, and a new buffer line coming up from below the peephole would be displayed and made current. For example, if the buffer were five lines long and contained:

```
          ( Who?    )
          ( What?   )
          ( When?   )
          ( Where?  )
Current:  ( Why?    )
```

Fig. 5

the "roll-buffer" command gives:

```
          ( What?   )
          ( When?   )
          ( Where?  )
          ( Why?    )
Current:  ( Who?    )
```

Fig. 6

ctrl B back-roll-buffer command

The "back-roll-buffer" does exactly the opposite of the "roll-buffer" command described above. For example, if given the buffer in figure 6 above, the "back-roll" command would give the buffer shown in figure 5.

ctrl C clear buffer line command

The "clear-buffer-line" command clears the current buffer line and then "back-rolls" the buffer so that successive clears can be used to erase the entire buffer.

NOTE:

Any of the above commands which change the text window will mark the current screen as updated. Those commands which alter only the buffer window (such as the "roll" command) will not change the status of the current screen.

Changing Screens

There are four ways in which to leave a FORTH screen. These four methods are: moving to a previous screen, moving to a following screen, saving the current screen and exiting, or simply aborting the edit session. The four commands allowing this are now described:

ctrl P previous screen command

The "previous-screen" command has two basic functions. If the lower part of the current screen is being viewed in the text window, this command simply displays the upper portion of the screen. If the upper portion is already being viewed, then the "previous-screen" command saves any changes made to the current screen and then loads in the screen immediately before the current screen. The lower part of the screen will then be displayed. If in the half-K screen mode, however, this command simply changes screens.

ctrl N next screen command

Like the "previous-screen" command described above, the "next-screen" command also has two basic functions. If the upper part of a screen is being viewed, this command simply displays the lower portion. If, on the other hand, the lower part of the screen is being edited, any changes made to the current screen are saved and the next screen is loaded.

ctrl S save command

The "save" command saves any changes made to the current screen and exits the edit mode. The video screen is cleared, and the number of the screen just being edited is displayed for reference. Note that it is usually a good idea to immediately FLUSH (described in the section on screen management below) any unsaved screens.

ctrl Q quit command

The "quit" command aborts the edit session "forgetting" any changes made to the text visible in the text window. Changes made on previously edited screens will NOT be forgotten. The "quit" command is usually used when either the wrong screen has been called up, or if it becomes desirable to start over and re-edit the screen again.

Special Commands

There are four special commands in this editor which allow greater flexibility in programming on the valFORTH system:

ESCAPE — special key command

The "special-key" command instructs the video editor to ignore any command function of the key typed next and force a character to the screen. For example, normally when "ctrl >" is typed, the cursor is moved right. By typing "ESCAPE ctrl >" the cursor is not moved -- rather, the right-arrow is displayed.

ctrl A — arrow command

When dealing with FORTH screens, it is often necessary to put the FORTH word "-->" (pronounced "next screen") or the ValFORTH word "==>" (pronounced "next screen") or the ValFORTH word "==>" (pronounced "next half-K screen") at the end of a screen for chaining a long set of words together. This command automatically places, or erases, an arrow in the lower right hand corner of the text window. If "-->" is already there, it is replaced with "==>". If "==>" is found, it is erased. (This command marks the screen as updated.)

ctrl J — split line command

Often times, for formatting reasons, it is necessary to "split" a line into two lines. The split line command takes all characters to the left of the cursor and creates the first line, and with the remaining characters of the original line, a second line is created. Graphically, this looks like:

```
before:   | The quick■brown fox jumped.       |


after:    | The quick■                          |
          |            brown fox jumped.       |
```

Since a line is inserted, the bottom line of the text window is lost. Using the "oops" command below, however, this can be recovered.

ctrl O oops command

Occasionally, a line is inserted or deleted accidentally, half a line cleared by mistake, or some other major editing blunder is made. As the name implies, the "oops" command corrects most of these major editing errors. The "oops" command can be used to recover from the following commands:

1)	insert line command	(shift INS)
2)	delete line command	(shift DEL)
3)	hack command	(ctrl H)
4)	to buffer command	(ctrl T)
5)	from buffer command	(ctrl F)
6)	copy from buffer command	(ctrl U)
7)	split line command	(ctrl J)

Screen Management

In addition to the commands available while in the edit mode, there are several other commands which are for use outside of the edit mode. Typically, these commands deal with entire screens at a time.

FLUSH (---)

When any changes are made to the current text window, the current screen is marked as having been changed. When leaving the edit mode using the "save" command, the current screen is sent to a set of internal FORTH buffers. These buffers are not written to disk until needed for other data. Thus, if no other screen is ever accessed, the buffers will never be saved to disk. The FLUSH command forces these buffers to be saved if they have been marked as being modified.

Example: FLUSH <ret>

EMPTY-BUFFERS (---)

Occasionally, screens are modified temporarily or by accident, and get marked as being modified. The EMPTY-BUFFERS command unmarks the internal FORTH buffers and fills them with zeroes so that "bad" data are not saved to disk. Zero filling the buffers ensures that the next access to any of the screens that were in the buffers will load the unadulterated copy from disk. The abbreviation MTB is included in the valFORTH system to make the use of this command easier.

Examples: EMPTY-BUFFERS <ret>
MTB <ret>

COPY (from to ---)

To duplicate a screen, the COPY command is used. The screen "from" is copied to the screen "to" but not flushed.

Example: 51 60 COPY <ret>

(Copies screen 51 to screen 60.)

CLEAR (scr# ---)

The CLEAR command fills the specified screen with blanks so that a clean edit can be started. The screen is then made current so that the L command can be used to enter the edit mode.

Example: 50 CLEAR <ret>

(Clears screen 50 and makes it current.)

CLEARS (scr# #screens ---)

The CLEARS command is used to clear blocks of screens at a time. After user verification, it starts with the specified screen and clears the specified number of consecutive screens. The first screen cleared is made current so that the L command can be used to enter the edit mode.

Example:

```
25 3 CLEARS   <ret>
Clear from SCR 25
        to SCR 27  <Y/N> Y
```

(Screens 25-27 are cleared. Screen 25 is made current.)

SMOVE (from to #screens ---)

The SMOVE command is a multiple screen copy command used for copying large numbers of consecutive screens at a time. User verification is required by this command to avoid disastrous loss of data. All screens to be copied are read into available memory and the user is prompted to initiate the copy. This allows the swapping of disks between moves to make disk transfers possible. The number of screens the SMOVE command can copy at a time is limited only by available memory.

Example:

```
50 60 5 SMOVE   <ret>
SMOVE from 50 thru 54
        to 60 thru 64  <Y/N> Y
Insert source <RETURN>  <ret>
Insert dest. <RETURN>  <ret>
```

(Transfers the specified screens.)

Editor Command Summary

Below is a quick reference list of all the commands which the video editor recognizes.

Entering the Edit Mode: (executed outside of the edit mode)

V	Enter the edit mode and view the specified screen.	(scr# ---)
L	Re-view the current screen.	(---)
WHERE	Enter the edit mode and position the cursor over the word that caused a compilation error.	(---)
LOCATE cccc	Enter the edit mode and position the cursor over the word defining "cccc".	(---)
LOCATOR	When ON, allows all words compiled until the next OFF to be locatable using the LOCATE command above.	(ON/OFF ---)
#BUFS	Sets the length (in lines) of the storage buffer. The default is five.	(#lines ---)

Cursor Movement: (issued within the edit mode)

ctrl ∧	Move cursor up one line, wrapping to the bottom line if moved off the top.
ctrl ∨	Move cursor down one line, wrapping to the top line if moved off the bottom.
ctrl <	Move cursor left one character, wrapping to the right edge if moved off the left.
ctrl >	Move cursor right one character, wrapping to the left edge if moved off the right.
RETURN	Position the cursor at the beginning of the next line.
TAB	Advance to next tabular column.

Editing Commands: (issued within the edit mode)

ctrl INS	Insert one blank at cursor location, losing the last character on the line.
ctrl DEL	Delete character under cursor, closing the line.
shift INS	Insert blank line above current line, losing the last line on the screen.
shift DEL	Delete current cursor line, closing the screen.
ctrl I	Toggle insert-mode/replace-mode. (see full description of ctrl-I).
BACKS	Delete last character typed, if on the same line as the cursor.
ctrl H	Erase to end of line (Hack).

Buffer Management: (issued within the edit mode)

ctrl T	Delete current cursor line sending it to the edit buffer for later use.
ctrl F	Take the current buffer line and insert it above the current cursor line.
ctrl K	Copy current cursor line sending it to the edit buffer for later use.
ctrl U	Take the current buffer line and copy it to the current cursor line.
ctrl R	Roll the buffer making the next buffer line current.
ctrl B	Roll the buffer backwards making the previous buffer line on the screen current.
ctrl C	Clear the current buffer line and perform a ctrl-B.

Note: The current buffer line is last line visible on the video display.

Changing Screens: (issued within the edit mode)

ctrl P	Display the previous screen saving all changes made to the current text window.
ctrl N	Display the next screen saving all changes made to the current text window.
ctrl S	Save the changes made to the current text window and end the edit session.
ctrl Q	Quit the edit session forgetting all changes made to current text window.

Special Keys: (issued within the edit mode)

ESC	Do not interpret the next key typed as any of the commands above. Send it directly to the screen instead.
ctrl A	Put "-->", "==>", or erase the lower right-hand corner of the text window.
ctrl J	Split the current line into two lines at the point where the cursor is.
ctrl O	Corrects any major editing blunders.

Screen Management: (executed outside of the edit mode)

FLUSH (---)
Save any updated FORTH screens to disk.

EMPTY-BUFFERS (---)
Forget any changes made to any screens not yet FLUSHed to disk. Used in "losing" major editing mistakes. The abbreviation MTB is more commonly used.

COPY (from to ---)
Copies screen #from to screen #to.

CLEAR (scr# ---)
Blank fills specified screen. This performs the same functions as "WIPE" in Leo Brodie's book.

CLEARS (scr# #screens ---)
Blank fills the specified number of screens starting with screen scr#.

SMOVE (from to #screens ---)
Duplicate the specified number of screens Starting with screen number "from". Allows swapping of disks before saving screens to screen number "to".

STRING UTILITIES

The following collection of words describes the string utilities of the valFORTH Utilities Package. Strings have been implemented in the FORTH language in many different ways. Most implementations set aside space for a third stack -- a string stack. As strings are entered, they are moved (using CMOVE) to this stack. When strings are manipulated on this stack, many long memory moves are usually required. This method is typically much slower than the method implemented in valFORTH.

Rather than waste memory space with a third stack, valFORTH uses the already existing parameter stack. Unlike the implementation described above, valFORTH does not store strings on the stack. Rather, it stores the addresses of where the strings can be found.* Using this method, words such as SWAP , DUP , PICK , and ROLL can be used to manipulate strings. Routines such as string sorts which work on many strings at a time are typically much faster since addresses are manipulated rather than long strings. In practice, we have found few if any problems using this method of string representation.

String Glossary

For the purposes of this section, a string is defined to be a sequence of up to 255 characters preceded by a byte indicating its length. The first character of the string is referenced as character one. If the length of the string is zero, it has no characters and is called the "null" string. In stack notation, strings are represented by the symbol $ and the address of the string is stored on the stack rather than the string itself*.

-TEXT addr1 n addr2 -- flag

The word -TEXT compares n characters at address1 with n characters at address2. Returns a false flag if the sequences match, true if they don't. Flag is positive if the character sequence at address1 is alphabetically greater than the one at address2. Flag is zero if the character sequences match, and is negative if the character sequence at address1 is alphabetically less than the one at address2.

-NUMBER addr -- d

-NUMBER functions identically to the standard FORTH word NUMBER with the only difference being that -NUMBER does not abort program execution upon an illegal conversion. -NUMBER takes the character string at addr and attempts to convert it to a double number. On successful conversion, the value d is returned with the status variable NFLG set to one. On unsuccessful conversion, a double number zero is returned with the variable NFLG set to zero. -NUMBER is pronounced "not number".

*Representing strings on the stack by their addresses is a very useful concept borrowed from MMS Forth (TRS-80), authored by Tom Dowling, and available from Miller Microcomputer Services, 617-653-6136.

NFLG -- addr

A variable used by -NUMBER that indicates whether the last conversion attempted was successful. NFLG is true if the conversion was successful; otherwise, it is false.

UMOVE addr1 addr2 n --

UMOVE is a "universal" memory move. It takes the block of memory n bytes long at addr1 and copies it to memory location addr2. UMOVE correctly uses either CMOVE or <CMOVE so that when a block of memory is moved onto part of itself, no data are destroyed.

" cccc" -- (at compile time)
cccc: -- addr (at run time)

If compiling, the sequence cccc (delimited by the trailing ") is compiled into the dictionary as a string:

```
| len | c | c | c |...| c |
```

All valFORTH strings are represented in this fashion. Since a single byte is used to store the length, a maximum string length of 255 is allowed. A string with 0 length is called a "null" string. At execution time, " puts the address in memory where the string is located onto the stack.

Note that " is IMMEDIATE. When executed outside of a colon definition, the string is not compiled into the dictionary, but is stored at PAD instead.

Example: " This is a string"

$CONSTANT cccc $ -- (at compile time)
cccc: -- $ (at execution time)

Takes the string on top of the stack and compiles it into the dictionary with the name cccc. When cccc is later executed, the address of the string is pushed onto the stack.

Example: " Ready? <Y/N> " $CONSTANT VERIFY

$VARIABLE cccc n --
cccc: -- $

Reserves space for a string of length n. When cccc is later executed, the address of the string is pushed onto the stack.

Example: 80 $VARIABLE TEXTLINE

$. $ --

Takes the string on top of the stack and sends it to the current output device.

Example: " Hi there" $. <ret> Hi there

$! $ addr --

Takes the string at second on stack and stores it at the address on top of stack.

Example: " Store me!" TEXTLINE $!

$+ $1 $2 -- $3

Takes $2 and concatenates it with $1, leaving $3 at PAD.

Example: " Santa " $CONSTANT 1ST
" Claus" $CONSTANT LAST
1ST LAST $+
$. <ret> Santa Claus

LEFT$ $1 n -- $2

Returns the leftmost "n" characters of $1 as $2. $2 is stored at PAD.

Example: " They" 3 LEFT$ $. <ret> The

RIGHT$ $1 n -- $2

Returns the rightmost "n" characters of $1 as $2. $2 is stored at PAD.

Example: " mother" 5 RIGHT$ $. <ret> other

MID$ $1 n u -- $2

Returns $2 of length u starting with the nth character of $1. Recall that the first character of a string is numbered as one.

Example: " Timeout" 3 2 MID$ $. <ret> me

LEN $ -- len

Returns the length of the specified string.

ASC $ -- c

Returns the ASCII value of the first character of the specified string.

$COMPARE $1 $2 -- flag

Compares $1 with $2 and returns a status flag. The flag is a) positive if $1 is greater than $2 or is equal to $2, but longer, b) zero if the strings match and are the same length, and c) negative if $1 less than $2 or if they are equal and $1 is shorter than $2.

$= $1 $2 -- flag

Compares two strings on top of the stack and returns a status flag. The flag is true if the strings match and are equal in length, otherwise it is false.

$< $1 $2 -- flag

Compares two strings on top of the stack and returns a status flag. The flag is true if $1 is less than $2 or if $1 matches $2 but is shorter in length.

$> $1 $2 -- flag

Compares two strings on top of the stack and returns a status flag. The flag is true if $1 is greater than $2 or if $1 matches $2 but is longer in length.

SAVE$ $1 -- $2

As most string operations leave resultant strings at PAD, the word SAVE$ is used to temporarily move strings to PAD+512 so that they can be manipulated without being altered in the process.

Example: " Wash" SAVE$ " ington" $+

INSTR $1 $2 -- n

Searches $1 for first occurrence of $2. Returns the character position in $1 if a match is found; otherwise, zero is returned.
Example: " FDCBA" $CONSTANT GRADES
GRADES " A" INSTR 1- . <ret> 4

CHR$ c -- $

Takes the character "c" and makes it into a string of length one and stores it at PAD.

DVAL $ -- d

Takes numerical string $ and converts it to a double length number. The variable NFLG is true if the conversion is successful, otherwise it is false. See -NUMBER above.
Example: " 123" DVAL D. <ret> 123

VAL $ -- n

Takes the numerical string $ and converts it to a single length number. The variable NFLG is true if the conversion is successful, otherwise it is false. See -NUMBER above.

DSTR$ d -- $

Takes the double number d and converts it to its ASCII representation as $ at PAD.
Example: 123 DSTR$ $. <ret> 123

STR$ n -- $

Takes the single length number n and converts it to its ASCII representation as $ at PAD.

STRING$ n $1 -- $2

Creates $2 as n copies of the first character of $1.

#IN$ n -- $

#IN$ has three similar but different functions. If n is positive, it accepts a string of n or fewer characters from the terminal. If n is zero, it accepts up to 255 characters from the terminal. If n is negative, it returns only after accepting -n characters from the terminal. The resultant string is stored at PAD.

IN$ -- $

Accepts a string of up to 255 characters from the terminal.

$-TB $1 -- $2

Removes trailing blanks from $1 leaving new $2.

$XCHG $1 -- $2

Exchanges the contents of $1 with $2.

ARRAYS and their COUSINS

All of the words described below create structures that are accessed in the same way, i.e., by putting the index or indices on the stack and then typing the structure's name. The differences are in the ways the structures are created.

The concept of the array should be known from BASIC. While in fig-FORTH there is no standard way to implement arrays and similar structures, there does exist a general consensus about how this should be done.

The point on which there is the most divergence of opinion is whether the first element in an array should be referred to by the index 0 or 1. We select 0 for the first index since this gives much cleaner code and makes more sense than 1 after you get used to it. (We've worked with it both ways.)

ARRAY and CARRAY, and 2ARRAY and 2CARRAY

The size of an array, specified when it is defined, is the number of elements in the array. In other words, an array defined by

```
8  ARRAY BINGO
```

will have 8 elements numbered 0 - 7.

To access an element of an array, do

```
n  array-name
```

to get the address of the nth element on the stack. (You will not be told if the number n is not a legitimate index number for the array.) For example,

```
5 BINGO
```

will leave the address of element number 5 in BINGO on the stack. You can store to or fetch from this address as you require.

The word CARRAY defines a byte or character array. A c-array works the same as an array, except that you must use C@ and C! to manipulate single elements, rather than @ and !.

The words 2ARRAY and 2CARRAY each take two numbers during definition of a 2ARRAY or 2CARRAY, and 2ARRAYS and 2CARRAYS take two numbers to access an element. Note that when using a 2CARRAY named, say, CHESSBOARD, and a constant named ROOK, the two phrases

```
ROOK   4  6  CHESSBOARD C!
          and
ROOK   6  4  CHESSBOARD C!
```

don't do the same thing. Also note that the phrase

```
8  8  2CARRAY CHESSBOARD
```

defines a 2CARRAY of 8 x 8 = 64 elements, with both indices running from 0 to 7.

When an ARRAY or a CARRAY is defined, the initial values of the elements are undefined.

TABLE AND CTABLE

A cousin of ARRAY is TABLE. Example: The phrase

```
TABLE  THISLIST  14 , 18 , -34 , 16 ,
```

defines a table THISLIST of 4 elements. (The commas above are part of the code and must be included.) The number of elements does not have to be specified. The elements in THISLIST are accessed using the indices 0-3, the same as if it had been defined as an array. The word CTABLE works similarly, though using C, instead of , to compile in the numbers. Note that negatives won't be compiled in by a C, since in two's complement representation negative numbers always occupy the maximum number of bytes.

VECTOR and CVECTOR

The last array-type words in this package are CVECTOR and VECTOR. Vector is just another name for a list. These words are used when the elements of the array you want to create are on the stack, with the last element on top of the stack. You just put the number of elements on the stack and the VECTOR or CVECTOR, and the name you want to use. Example:

```
-3  8 127 899 -43     5  VECTOR    POSITIONS
```

creates an array named POSITIONS with 5 elements 0-4 with -3 in element 0 and -43 in element 4. CVECTOR works in a similar way.

EXAMPLES:

```
2   3   BINGO   !
```

Stores the value 2 into element 3 of array BINGO.

```
2   THISLIST   @
```

Will leave the value in element 2 of table THISLIST. According to the definition of THISLIST above, this value will be -34.

```
3  POSITION  @  .  <cr>  899
```

ARRAY WORD GLOSSARY

ARRAY cccc, n -- (compiling)
cccc: m -- addr (executing)
When compiling, creates an array named cccc with n 16-bit elements numbered 0 thru n-1. Initial values are undefined. When executing, takes an argument, m, off the stack and leaves the address of element m of the array.

CARRAY cccc, n -- (compiling)
cccc: m -- addr (executing)
When compiling, creates a c-array named cccc with n 8-bit elements numbered 0 thru n-1. Initial values are undefined. When executing, takes an argument, m, off the stack and leaves the address of element m of the c-array.

TABLE cccc, -- (compiling)
cccc: m -- addr (executing)
When compiling, creates a table named cccc but does not allot space. Elements are compiled in directly with , (comma). When executing, takes one argument, m off the stack and, assuming 16-bit elements, leaves the address of element m of the table.

CTABLE cccc, -- (compiling)
cccc: m -- addr (executing)
When compiling, creates a c-table named cccc but does not allot space. Elements are compiled in directly with C, (c-comma). When executing, takes one argument, m off the stack and, assuming 8-bit elements, leaves the address of element m of the c-table.

X! n0 ... nN count addr --
Stores count 16-bit words, n0 thru nN into memory starting at addr, with n0 going into addr. Pronounced "extended store."

XC! b0 ... bN count addr --
Stores count 8-bit words, b0 thru bN into memory starting at addr, with b0 going into addr. Pronounced "extended c-store."

VECTOR cccc, n0 ... nN count -- (compiling)
cccc: m -- addr (executing)
When compiling, creates a vector named cccc with count 16-bit elements numbered 0-N. n0 is the initial value of element 0, nN is the initial value of element N, and so on. When executing, takes one argument, m, off the stack and leaves the address of element m on the stack.

CVECTOR cccc, b0 ... bN count -- (compiling)
cccc: m -- addr (executing)
When compiling, creates a c-vector named cccc with count 8-bit elements numbered 0-N. b0 is the initial value of element 0, bN is the initial value of element N, and so on. When executing, takes an argument, m, off the stack and leaves the address of element m on the stack.

CASE STRUCTURES

It often becomes necessary to make many tests upon a single number. Typically, this is accomplished by using a series of nested "DUP test IF" statements followed by a series of ENDIFs to terminate the IFs. This is arduous and very wasteful of memory. valFORTH contains four very powerful Pascal-type CASE statements which ease programming and conserve memory.

The CASE: structure

Format:

```
CASE:  wordname
       word0
       word1
       ...
       wordN  ;
```

The word CASE: creates words that expect a number from 0 to N on the stack. If the number is zero, word0 is executed; if the number is one, the word1 is executed; and so on. No error checks are made to ensure that the case number is a legal value.

Example:

```
:  ZERO   ." Zero" ;
:  ONE    ." One" ;
:  TWO    ." Two" ;

CASE:  NUM
       ZERO
       ONE
       TWO  ;

0  NUM  <ret>  Zero
1  NUM  <ret>  One
2  NUM  <ret>  Two
```

Note that any other number (e.g. 3 NUM) will crash the system.

The CASE Structure

Format:

```
: wordname
  ...
  CASE
     word0
     word1
     ...
     wordN
( NOCASE  wordnone )          (optional)
 CASEND
 ... ;
```

The CASE...CASEND structure is always used within a colon definition. Like CASE: above, it requires a number from 0 and N. However, unlike CASE: above, boundary checks are made so that an illegal case will do nothing. If the optional NOCASE clause is included then wordnone is executed if an "out of bounds" number is used.

Examples:

I)

```
: ZERO     ." Zero"  ;
: ONE      ." One"   ;
: TWO      ." Two"   ;

: CHECKNUM              ( n -- )
  CASE
     ZERO
     ONE
     TWO
  CASEND  ;

  0 CHECKNUM  <ret>  Zero
  1 CHECKNUM  <ret>  One
999 CHECKNUM  <ret>  (nothing happens)
  2 CHECKNUM  <ret>  Two
```

```
II)     : GRADEA     ." A" ;
        : GRADEB     ." B" ;
        : GRADEC     ." C" ;
        : GRADED     ." D" ;
        : OTHER      ." Failed" ;

        DECIMAL
        :  GETGRADE             ( -- )
           KEY 65 -             (Convert A to 0, B to 1, etc)
           CASE
              GRADEA
              GRADEB
              GRADEC
              GRADED
           NOCASE  OTHER
           CASEND  ;


        GETGRADE <return and press A> A
        GETGRADE <return and press B> B
        GETGRADE <return and press F> Failed
        GETGRADE <return and press D> D
```

The SEL Structure

Format:

```
:  wordname
   ...
   SEL                          (Select)
      n1  ->  word0
      n2  ->  word1
      ...
      nN   >  wordN
(  NOSEL  wordnone  )           (optional)
   SELEND
   ... ;
```

The SEL...SELEND structure is used when the "selection" numbers (n1 etc.) are not sequential. This structure is somewhat slower than either CASE or CASE: , but is much more general. SEL is typically used in operations such as table driver menus where single keystroke commands are used. The valFORTH video editor uses the SEL structure to implement the many editing keystroke commands.

Example:

```
I)    : NICKEL   ." nickel."   ;
      : DIME     ." dime."   ;
      : QUARTER  ." quarter."   ;
      : 4BITS    ." fifty cent piece."   ;
      : SUSANB   ." dollar"  ;
      : BAD$$$   ." wooden nickel."   ;

      : MONEY-NAME                 ( n -- )
        ." That is called a "
        SEL
           5  ->  NICKEL
          10  ->  DIME
          25  ->  QUARTER
          50  ->  4BITS
         100  ->  SUSANB
       NOSEL  BAD$$$          ( this line is optional )
       SELEND  ;

     5 MONEY-NAME <ret> That is called a nickel.
    33 MONEY-NAME <ret> That is called a wooden nickel.
    25 MONEY-NAME <ret> That is called a quarter.
```

The COND Structure

Format:

```
    :  wordname
       ...
       COND
          condition0  <<  words0  >>
          condition1  <<  words1  >>
          ...
          conditionN  <<  wordsn  >>
    (  NOCOND  wordsnone  )               (optional)
       CONDEND
       ... ;
```

Unlike the three previous CASE structures which test for equality, the COND structure bases its selection upon any true conditional test (e.g. if n > 0 then...) COND can also be used for range cases. The NOCOND clause is optional and is only executed if no other condition passes. Only the code of the first condition that passes will be executed.

Example:

```
: EXAM                       ( score -- grade )
  COND
     90  >=   <<  ." Grade of A"  4  >>
     80  >=   <<  ." Grade of B"  3  >>
     70  >=   <<  ." Grade of C"  2  >>
     60  >=   <<  ." Grade of D"  1  >>
  NOCOND  ." Not too good"  0
  CONDEND  ;
```

Note that neither << nor >> are needed (nor allowed) around the "NOCOND" case. Also note that more than one word can be executed between the << and >> .

(intentionally left blank)

DOUBLE NUMBER EXTENSIONS

The following words extend the set of double number words to be as nearly identical as possible to the set in the book Starting FORTH. The exceptions are DVARIABLE and DCONSTANT which conform to the FIG standard by expecting initial values on the stack.

All of the single number operations comparable to the double number operations below were machine coded; all of the words below (with the exception of DVARIABLE) have high-level run time code and so are considerably slower than their single number counterparts.

DOUBLE NUMBER EXTENSION GLOSSARY

DVARIABLE cccc d --
cccc: -- addr
At compile time, creates a double number variable cccc with the initial value d. At run time, cccc leaves the address of its value on the stack.

DCONSTANT cccc d --
cccc: -- d
At compile time, creates a double number constant cccc with the initial value d. At run time, cccc leaves the value d on the stack.

0. -- 0.
A double number constant equal to double number zero.

1. -- 1.
A double number constant equal to double number one.

D- d1 d2 -- d3
Leaves d1-d2=d3.

D0= d -- flag
If d is equal to 0. leaves true flag; otherwise, leaves false flag.

D= d1 d2 -- flag
If d1 equals d2, leaves true flag; otherwise, leaves false flag.

D0< d -- flag
If d is negative, leaves true flag; otherwise, leaves false flag.

D< d1 d2 -- flag
If d1 is less than d2, leaves true flag; otherwise, leaves false flag.

D> d1 d2 -- flag
If d1 is greater than d2, leaves true flag; otherwise, leaves false flag.

DMIN d1 d2 -- d3
Leaves the minimum of d1 and d2.

DMAX d1 d2 -- d3
Leaves the maximum of d1 and d2.

D>R   d --
Sends the double number at top of stack to the return stack.

DR>   -- d
Pulls the double number at top of the return stack to the stack.

D,   d --
Compiles the double number at top of stack into the dictionary.

DU<   ud1 ud2 -- flag
If the unsigned double number ud1 is less than the unsigned double number ud2, leaves a true flag; otherwise, leaves a false flag.

M+   d1 n -- d2
Converts n to a double number and then sums with d1.

HIGH RESOLUTION TEXT OUTPUT

Occasionally, the need arises to print text in high resolution graphic displays (8 GR.). The following set of words explains how Graphic Characters can be used in valFORTH programs. The Graphic-Character output routines are designed to function identically to the standard FORTH output operations. There is an invisible cursor on the high resolution page which always points to where the next graphic-character will be printed. As with normal text output, this cursor can be repositioned at any time and in various ways. Because of the nature of hi-res printing, this cursor can also be moved vertically by partial characters. This allows for super/subscripting, over-striking, and underlining. Multiple character fonts on the same line are also possible.

GCINIT --

Initializes the graphic character output routines. This must be executed prior to using any other hi-res output words.

GC. n --

Displays the single length number n at the current hi-res cursor location.

GC.R n1 n2 --

Displays the single length number n1 right-justified in a field n2 graphic characters wide. See .R .

GCD.R d n --

Displays the double length number d right-justified in a field n graphic characters wide. See D.R .

GCEMIT c --

Displays the text character c at the current hi-res cursor location. Three special characters are interpreted by GCEMIT . The up arrow (↑) forces text output into the superscript mode; the down arrow (↓) forces the text into the subscript mode; and the left arrow (←) performs a GCBKS command (described below). See OSTRIKE below; also see EMIT.

GCLEN addr n -- len

Scans the first n characters at addr and returns the number of characters that will actually be displayed on screen. This is typically used to find the true length of a string that contains any of the non-printing special characters described in GCEMIT above. Used principally to aid in centering text, etc.

GCR --

Repositions the hi-res cursor to the beginning of the next hi-res text line. See CR .

GCLS --

Clears the hi-res display and repositions the cursor in the upper lefthand corner.

GCSPACE --

Sends a space to the graphic character output routine. See SPACE .

GCSPACES n --

Sends n spaces to the graphic character output routine. See SPACES .

GCTYPE addr n --

Sends the first n characters at addr to the graphic character output routine. See TYPE .

GC" cccc" --

Sends the character string cccc (delimited by ") to the graphic character output routine. If in the execution mode, this action is taken immediately. If in the compile mode, the character string is compiled into the dictionary and printed out only when executed in the word that uses it. See ." .

GCBKS --

Moves the hi-res cursor back one character position for overstriking or underlining.

GCPOS horz vert --

Positions the hi-res cursor to the coordinates specified. Note that the upper lefthand corner is 0,0.

GC$. addr --

Sends the string found at addr and preceded by a count byte to the graphic character output routine. See $. .

SUPER --

Forces the graphic character output routine into the superscript mode (or out of the subscript mode). See VMI below. May be performed within a string by the ↑ character.

SUB --

Forces the graphic character output routine into the subscript mode (or out of the superscript mode). See VMI below. May be performed within a string by the ↓ character.

VMI n --

Each character is eight bytes tall. The VMI command sets the number of eighths of characters to scroll up or down when either a SUPER or SUB command is issued. Normally, 4 VMI is used to scroll 4/8 or half a character in either direction.

VMI# -- addr

A variable set by VMI.

OSTRIKE ON or OFF --

The GCEMIT command has two separate functions. If OSTRIKE (overstrike) option is OFF, the character output will replace the character at the current cursor position. This is the normal method of output. If the OSTRIKE option is ON, the new character is printed over top of the previous character giving the impression of an overstrike. This allows the user to underline text and create new characters: Example: To do underline, a value of, say, 2 should be used with VMI, and then the ↓ character added in the string before the underline character.

GCBAS -- addr

A variable which contains the address of the character set displayed by GCEMIT. To change character sets, simply store the address of your new character set into this variable.

GCLFT -- addr

A variable which holds the column position of the left margin. Normally two, this can be changed to obtain a different display window.

GCRGT -- addr

A variable which holds the column position of the right margin. Normally 39, this can be changed to obtain a different display window.

(intentionally left blank)

MISCELLANEOUS UTILITIES

This is a grab-bag of useful words. Here they are...

XR/W   #secs addr blk flag --

"Extended read-write." The same as R/W except that XR/W accepts a sector count for multiple sector reads and writes. Starting at address addr and block blk, read (flag true) or write (flag false) #secs sectors from or to disk.

SMOVE   org des count --

Move count screens from screen # org to screen # dest.

The primary disk rearranging word, also used for moving sequences of screens between disks. This is a smart routine that uses all memory available below the current GR.-generated display list, with prompts for verification and disk swap if desired. See valFORTH Editor 1.1 documentation for further details.

LOADS   start count --

Loads count screens starting from screen # start. This word is used if you want to use words that are not chained together by --> 's. It will stop loading if a CONSOLE button is held down when the routine finishes loading its present screen.

THRU   start finish -- start count

Converts two range numbers to a start-count format. Example:

```
120 130 THRU PLISTS
```

will print screens 120 thru 130.

SEC   n --
Provides an n second delay. Uses a tuned do-loop.

MSEC   n --
Provides an n millisecond delay. (approx)
Uses a tuned do-loop.

H->L   n -- b
Moves the high byte of n to the low byte and zero's the high byte, creating b. Machine code.

L->H   n1 -- n2
Moves the low byte of n1 to the high byte and zero's the low byte, creating n2. Machine code.

H/L   n1 -- n1(hi) n1(lo)
Split top of stack into two stack items: New top of stack is low byte of old top of stack. New second on stack is old top of stack with low byte zeroed.
Example: HEX 1234 H/L .S <cr> 1200 0034

BIT   b -- n
Creates a number n that has only its bth bit set. The bits are numbered 0-15, with zero the least significant. Machine code.

?BIT   n b -- f
Leaves a true flag if the bth bit of n is set. Otherwise leaves a false flag.

TBIT   n1 b -- n2
Toggles the bth bit of n1, making n2.

SBIT   n1 b -- n2
Sets the bth bit of n1, making n2.

RBIT n1 b -- n2
Resets the bth bit of n1, making n2.

STICK   n -- horiz vert
Reads the nth stick (0-3) and resolves the setting into horizontal and vertical parts, with values from -1 to +1. -1 -1 means up and to the left.

PADDLE n1 -- n2
Reads the n1th paddle (0-7) and returns its value n2. Machine code.

ATTRACT f --
If the flag is true, the attract mode is initiated. If the flag is false, the attract mode is terminated.

NXTATR --
If the system is in the attract mode, this command cycles to the next color setup in the attract sequence. Disturbs the timer looked at by 16TIME.

HLDATR   --
If the system is in attract mode, zero's fast byte of the system timer so that attract won't cycle to next color setup for at least four seconds or until system timer is changed, say by NXTATR. Disturbs the timer looked at by 16TIME.

16TIME -- n
Returns a 16 bit timer reading from the system clock at locations 19 and 20, decimal. This clock is updated 60 times per second, with the fast byte in 20. Machine code, not fooled by carry.

8RND -- b
Leaves one random byte from the internal hardware. Machine code.

16RND -- n
Leaves one random word from the internal hardware. Machine code with 20 cycle extra delay for rerandomization.

CHOOSE u1 -- u2
Randomly choose an unsigned number u2 which is less than u1.

CSHUFL addr n --
Randomly rearrange n bytes in memory, starting at address addr.
Pronounced "c-shuffle."

SHUFL addr n --
Randomly rearrange n words in memory, starting at address addr. Pronounced "shuffle." SHUFL may also be used to shuffle items directly on the stack by doing SP@ n SHUFL.

H. n --
See DEBUG Glossary.

A. addr --
Print the ASCII character at addr, or if not printable, print a period. (Used by DUMP).

DUMP addr n --
Starting at addr, dump at least n bytes (even multiple of 8) as ASCII and hex. May be exited early by pressing a CONSOLE button.

BLKOP system use only

BXOR addr count b --
Starting at address addr, for count bytes, perform bit-wise exclusive or with byte b at each address. Useful for toggling an area of display memory to inverse video or a different color, and for other purposes. For instance, in 0 GR., do

```
DCX 88 @ 280 128 BXOR
```

Then do Shift-Clear to clear the screen. Pronounced "block ex or."

BAND addr count b --
Starting at address addr, for count bytes, perform bit-wise AND with byte b at each address. Applications similar to BXOR.
Pronounced "block and."

BOR addr count b -
Starting at address addr, for count bytes, perform bit-wise or with byte b at each address. Applications similar to BXOR.
Pronounced "block or."

STRIG n -- flag
Reads the button of joystick n (0-3). Leaves a true flag if the button is pressed, a false flag if it isn't.

PTRIG n -- flag
Reads the button of paddle n (0-7). Leaves a true flag if the button is pressed, a false flag if it isn't.

(intentionally left blank)

TRANSIENTS

One of the more annoying parts about common releases of FORTH concerns the FORTH machine code assemblers. On the positive side, FORTH-based assemblers can be extraordinarily smart and easy to use interactively, and can compile on the fly as you type, rather than in multiple-pass fashion. (The 6502 assembler provided with valFORTH is a good example of a smart, structured, FORTH-based assembler.) On the other hand, since the assembler loads into the dictonary one usually sacrifices between 3 and 4K of memory on a utility that is only a compilation aid, and is not used during execution. With the utility described below, however, you can use the assembler and then remove it from the dictionary when you're finished with it.

In the directory of the Utilities/Editor disk (screen 170) you will find a heading of Transients. Loading this screen brings in three words: TRANSIENT, PERMANENT, and DISPOSE, and a few variables. It also defines a new area of memory called the Transient area. This area is used to load utilities like the assembler, certain parts of case statements, and similar constructs, that have one characteristic in common: They have compile-time behavior only, and are not used at run-time. An example will help make clear the sequence of operations. You may recall that on the valFORTH disk, in order to load floating point words you needed the assembler. Let's make a disk that has floating point but no assembler:

* Boot your valFORTH disk. It can be the bare system, or your normal programing disk if it doesn't have the assembler already in it.

* Insert your Utilities/Editor disk, find the Transient section in the directory, and load it.

* Do MTB (EMPTY-BUFFERS) and swap in your valFORTH disk. (It is a VERY good idea to get into the habit of doing MTB before swapping disks.) Find the assembler in the directory, but before you load it, do TRANSIENT to cause it to be loaded into the transient dictionary area, in high memory. Now go ahead and load the assembler. When it is loaded, do PERMANENT so that the next entries will go into the permanent dictionary area, which is back where you started.

* Now find and load the floating point words.

* Finally, do DISPOSE to pinch off the links that tie the transient area (with the assembler in it) to the permanent dictionary, with the floating point words in it. Do a VLIST or two to prove it to yourself. (Note that there are about a half-dozen words in the assembler vocabulary in the kernel. These were in the dictionary on boot up and are not affected by DISPOSE.)

You can derive great benefit from the simple recipe above, and if you study the Transient code a bit, you may learn even more. We offer several comments:

* In the case of the above recipe, you didn't actually have to do PERMANENT and TRANSIENT because the assembler source code checks at the front to see if TRANSIENT exists, and does it if so. At the end if checks to see if PERMANENT exists, and does it if so. This conditional execution is accomplished with the valFORTH construct

```
'(    )(    )
```

which is described in valFORTH documentation. Take a look at the assembler source code to see how this is done.

* If you want to do assembly on more than one section of code, you needn't DISPOSE until you really finished with the assembler; or, if you have DISPOSED of the assembler, you can bring it back in later without harm, by the same method. You can also code high-level definitions, and then more assembly code, and so on, and only do DISPOSE when you were finished. Be sure to do DISPOSE before SAVE or AUTO, however, because either your system will crash or your SAVE'd or AUTO'd program won't work.

The situation is slightly different with "case" words, since if you bring them in more than once you'll get duplicate names on the run-time words like (SEL), (CASE) and CASE:, which uses extra space and defeats the purpose of Transients.

* If you use the Transient structures for otherpurposes, remember only to send code that is not used at run-time to the transient area. As an example of this distinction, look at the code for the "case" words on the valFORTH disk. Note that the '(   )(   ) construct is again used, but that some of the parts of the case constructs, for instance (SEL), stay in the permanent dictionary. That is because (SEL) actually ends up in the compiled code, while SEL does not.

* Look at the beginning of the code for the Transient structures, and notice that the Transient area has been set up 4000 bytes below the display list. (The byte just below the display list in normal modes is pointed to by memory location 741 decimal, courtesy of the Atari OS.) This is usually a good place if only the 0 Graphics mode is used. (8 GR., for example, will over-write this area,crashing the system.) After DISPOSE is executed, this area is freed for other purposes. If you want to use a different area for Transients, just substitute your address into the source code on the appropriate screen. Remember that you must leave enough room for whatever will go into the Transient dictionary, and that NOTHING else must write to the area until you have cleared it out with DISPOSE. (This includes SMOVE, DISKCOPY1, DISKCOPY2, etc.)

****** NOTE ***** NOTE ***** NOTE ***** NOTE ***** NOTE ******

In the above example, 4000 bytes have been set aside for the Transient area just below the 0 GR. display list. This amount of memory will generally hold the assembler and some case statement compiling words. REMEMBER that if you have relocated the buffers (see the section on Relocating Buffers) to this area as well, you will have a collision, and a crashed system in short order.

To cure this, simply locate the Transient area 2113 bytes lower in memory so that there will be no overlap.

****** NOTE ***** NOTE ***** NOTE ***** NOTE ***** NOTE ******

ACKNOWLEDGEMENT

Various implementations of the Transient concept have appeared. valFORTH adopts the names TRANSIENT, PERMANENT, and DISPOSE from a public domain article by Phillip Wasson which appeared in FORTH DIMENSIONS volume III no. 6. The Transient structure implemented in the article has been altered somewhat in the valFORTH implementation to allow DISPOSE to dispose of the entire Transient structure, including DISPOSE itself, thus rendering the final product perfectly clean.

FORTH DIMENSIONS is a publication available through FIG (address listed elsewhere) and can be a valuable source of information and ideas to the advanced FORTH programmer.

Screen: 36

```
( Transients:  setup              )
BASE @ DCX
HERE
741 @ 4000 - DP !
( SUGGESTED PLACEMENT OF TAREA )
HERE CONSTANT TAREA
    0 VARIABLE TP
    1 VARIABLE TPFLAG
      VARIABLE OLDDP
                                ==>
```

Screen: 37

```
( Xsients: TRANSIENT PERMANENT )
: TRANSIENT                 ( -- )
  TPFLAG @ NOT
  IF HERE OLDDP ! TP @ DP !
     1 TPFLAG !
  ENDIF ;
: PERMANENT                 ( -- )
  TPFLAG @
  IF HERE TP ! OLDDP @ DP !
     0 TPFLAG !
  ENDIF ;
                                -->
```

Screen: 38

```
( Transients:  DISPOSE           )
: DISPOSE   PERMANENT
  CR ." Disposing..." VOC-LINK
  BEGIN DUP    0 53279 C!
   BEGIN @ DUP TAREA U<
   UNTIL DUP ROT ! DUP 0=
  UNTIL DROP VOC-LINK @
  BEGIN DUP 4 -
   BEGIN DUP    0 53279 C!
    BEGIN PFA LFA @ DUP TAREA U<
    UNTIL
     DUP ROT PFA LFA ! DUP 0=
   UNTIL DROP @ DUP 0=
  UNTIL DROP [COMPILE] FORTH
  DEFINITIONS ." Done" CR ;
  PERMANENT             BASE !
```

Screen: 39

Screen: 40

Screen: 41

Screen: 42

```
( Utils:  CARRAY  ARRAY          )
BASE @ HEX
: CARRAY          ( cccc, n -- )
  CREATE SMUDGE ( cccc: n -- a )
    ALLOT
  ;CODE CA C, CA C, 18 C,
  A5 C, W C,  69 C, 02 C, 95 C,
  00 C, 98 C, 65 C, W 1+ C,
  95 C, 01 C, 4C C,
  ' + ( CFA @ ) , C;

: ARRAY           ( cccc, n -- )
  CREATE SMUDGE ( cccc: n -- a )
    2* ALLOT
  ;CODE 16 C, 00 C, 36 C, 01 C,
   4C C, ' CARRAY 08 + , C;  ==)
```

Screen: 43

```
( Utils:  CTABLE  TABLE          )

: CTABLE          ( cccc, -- )
  CREATE SMUDGE ( cccc: n -- a )
  ;CODE
  4C C, ' CARRAY 08 + , C;

: TABLE           ( cccc, -- )
  CREATE SMUDGE ( cccc: n -- a )
  ;CODE
  4C C, ' ARRAY 0A + , C;




                                 --)
```

Screen: 44

```
( Utils:  2CARRAY  2ARRAY        )

: 2CARRAY      ( cccc, n n -- )
  <BUILDS      ( cccc: n n -- a )
   SWAP DUP , * ALLOT
  DOES>
    DUP >R @ * + R> + 2+ ;

: 2ARRAY       ( cccc, n n -- )
  <BUILDS      ( cccc: n n -- a )
   SWAP DUP , * 2* ALLOT
  DOES>
    DUP >R @ * + 2* R> + 2+ ;



                                 ==)
```

Screen: 45

```
( Utils:  XC!  X!

: XC!    ( n0...nm cnt addr -- )
  OVER 1- + >R 0
  DO J I - C!
  LOOP R> DROP ;

: X!     ( n0...nm cnt addr -- )
  OVER 1- 2* + >R 0
  DO J I 2* - !
  LOOP R> DROP ;

( Caution: Remember limitation
( on stack size of 30 values
( because of OS conflict. )
                                 --)
```

Screen: 46

```
( Utils:  CVECTOR  VECTOR        )

: CVECTOR        ( cccc, cnt -- )
  CREATE SMUDGE ( cccc: n -- a )
  HERE OVER ALLOT XC!
  ;CODE
  4C C, ' CARRAY 08 + , C;

: VECTOR         ( cccc, cnt -- )
  CREATE SMUDGE ( cccc: n -- a )
  HERE OVER 2* ALLOT X!
  ;CODE
  4C C, ' ARRAY  0A + , C;


                      BASE !
```

Screen: 47

```
Screen:  48
( Utils:  HIDCHR  NOKEY  CURSOR)
BASE @ DCX
'( CASE )( 28 KLOAD )
: HIDCHR                        ( -- )
  65535 94 ! ;
: NOKEY                         ( -- )
  255 764 C! ;  )
: CURSOR                      ( f -- )
  0= 752 C!
  28 EMIT 29 EMIT ;
                                   ==>
```

```
Screen:  49
( Utils:  INKEY$                     )
DCX
: (INKEY$)                    ( c -- )
  702 C! NOKEY ;
: INKEY$                      ( -- c )
  764 C@
  COND
    252 = (( 128 (INKEY$) 0 ))
     191 > (( 0 ))
     188 = (( 0 ))
     124 = ((  64 (INKEY$) 0 ))
      60 = ((   0 (INKEY$) 0 ))
      39 = (( 0 ))
  NOCOND KEY
  CONDEND ;                        -->
```

```
Screen:  50
( Utils:  -Y/N                       )
: -Y/N                        ( -- f )
  BEGIN KEY
    COND
      89 = (( 1 1 ))
      78 = (( 0 1 ))
    NOCOND
      0
    CONDEND
  UNTIL ;
                                   ==>
```

```
Screen:  51
( Utils:  Y/N  -RETURN  RETURN )
: Y/N                         ( -- f )
  ."  (Y/N) " -Y/N DUP
  IF 89 ELSE 78 ENDIF
  EMIT SPACE ;
: -RETURN                       ( -- )
  BEGIN KEY 155 = UNTIL ;
: RETURN                        ( -- )
  ."  (RETURN) " -RETURN ;
                          BASE !
```

```
Screen:  52
( Screen code conversion words )
BASE @ HEX
CODE >BSCD              ( a a n -- )
  A9 C, 03 C, 20 C, SETUP ,
  HERE  C4 C, C2 C, D0 C, 07 C,
  C6 C, C3 C, 10 C, 03 C, 4C C,
  NEXT ,       B1 C, C6 C, 48 C,
  29 C, 7F C, C9 C, 60 C, B0 C,
  0D C, C9 C, 20 C, B0 C, 06 C,
  18 C, 69 C, 40 C, 4C C, HERE
2 ALLOT 38 C, E9 C, 20 C, HERE
SWAP !  91 C, C4 C, 68 C, 29 C,
                                 ==>
```

```
Screen:  53
( Screen code conversion words )
  80 C, 11 C, C4 C, 91 C, C4 C,
  C8 C, D0 C, D3 C, E6 C, C7 C,
  E6 C, C5 C, 4C C, ,        C;
CODE BSCD>              ( a a n -- )
  A9 C, 03 C, 20 C, SETUP ,
  HERE  C4 C, C2 C, D0 C, 07 C,
  C6 C, C3 C, 10 C, 03 C, 4C C,
  NEXT ,       B1 C, C6 C, 48 C,
  29 C, 7F C, C9 C, 60 C, B0 C,
  0D C, C9 C, 40 C, B0 C, 06 C,
  18 C, 69 C, 20 C, 4C C, HERE
2 ALLOT 38 C, E9 C, 40 C, HERE
                                 -->
```

Screen: 54

```
( Screen code conversion words )

SWAP !  91 C, C4 C, 68 C, 29 C,
  80 C, 11 C, C4 C, 91 C, C4 C,
  C8 C, D0 C, D3 C, E6 C, C7 C,
  E6 C, C5 C, 4C C, ,


: >SCD  SP@ DUP 1 >BSCD ;
: SCD>  SP@ DUP 1 BSCD> ;






                          BASE !
```

Screen: 55

Screen: 56

```
( Case Statements:  CASE        )
BASE @ DCX
'( PERMANENT PERMANENT )(  )
: (CASE)
  R C@ MIN -1 MAX 2*
  R 3 + + @EX
  R C@ 2* 5 + R> + >R ;
'( TRANSIENT TRANSIENT )(  )
: CASE
  ?COMP COMPILE (CASE)
  HERE 0 C,
  COMPILE NOOP 6 ;     IMMEDIATE

: NOCASE
  6 ?PAIRS 7 ;         IMMEDIATE
                              ==>
```

Screen: 57

```
( Case statements:  CASE
 
: CASEND
  DUP 6 =
  IF
    DROP COMPILE NOOP
  ELSE
    7 ?PAIRS
  ENDIF
  HERE 2- @ OVER 1+ !
  HERE OVER -
  5 - 2/ SWAP C! ;      IMMEDIATE

'( PERMANENT PERMANENT )(  )

                              -->
```

Screen: 58

```
( Case statements:  SEL         )

'( PERMANENT PERMANENT )(  )
: (SEL)
  R 1+ DUP 2+ DUP R C@
  2* 2* + R> DROP DUP >R SWAP
  DO I @ 3 PICK =
     IF I 2+ SWAP DROP LEAVE
     ENDIF
  4 /LOOP SWAP DROP @EX ;

'( TRANSIENT TRANSIENT )(  )
: SEL    ?COMP
  ?LOADING COMPILE (SEL) HERE
  0 C, COMPILE NOOP [COMPILE] [
  8 ; IMMEDIATE               ==>
```

Screen: 59

```
( Case statements:  SEL         )

: NOSEL
  8 ?PAIRS [COMPILE] ' CFA
  OVER 1+ ! 8 ;        IMMEDIATE

: ->
  SWAP 8 ?PAIRS , DUP C@ 1+
  OVER C! [COMPILE] '
  CFA , 8 ;            IMMEDIATE

: SELEND
  8 ?PAIRS
  DROP [COMPILE] ] ;   IMMEDIATE
'( PERMANENT PERMANENT )(  )
                              -->
```

Screen: 60

```
( Case statements:  COND        )
'( TRANSIENT TRANSIENT )(  )
: COND
  0 COMPILE DUP ;      IMMEDIATE

: ((
  1+ [COMPILE] IF
  COMPILE DROP ;       IMMEDIATE

: ))
  [COMPILE] ELSE COMPILE
  DUP ROT ;            IMMEDIATE

: NOCOND
  COMPILE 2DROP ;      IMMEDIATE
'( PERMANENT PERMANENT )(  ) ==>
```

Screen: 61

```
( Case statements:  COND        )

'( TRANSIENT TRANSIENT )(  )

: CONDEND
  0 DO
    [COMPILE] ENDIF
  LOOP ;               IMMEDIATE

'( PERMANENT PERMANENT )(  )






                               -->
```

Screen: 62

```
( Case statements:  CASE:       )

: CASE:
  <BUILDS
    SMUDGE !CSP
    [COMPILE] ]
  DOES>
    SWAP 2* + @EX ;








                        BASE !
```

Screen: 63

```
```

Screen: 64

```
( ValFORTH Video editor    V1.1 )

BASE @ DCX

'( XC! )( 21 KLOAD )
'( HIDCHR )( 24 KLOAD )
'( )BSCD )( 26 KLOAD )








                                ==>
```

Screen: 65

```
( ValFORTH Video editor    V1.1 )














                                -->
```

Screen: 66

```
( ValFORTH Video editor    V1.1 )
VOCABULARY EDITOR IMMEDIATE
EDITOR DEFINITIONS
0 VARIABLE XLOC    ( X coord.    )
0 VARIABLE YLOC    ( Y coord.    )
0 VARIABLE INSRT   ( insert on?  )
0 VARIABLE LSTCHR  ( last key    )
0 VARIABLE ?BUFSM  ( buf same?   )
0 VARIABLE ?PADSM  ( PAD same?   )
0 VARIABLE ?ESC    ( coded char?)
0 VARIABLE TBLK    ( top block   )
                              ==)
```

Screen: 67

```
( ValFORTH Video editor    V1.1 )
0 VARIABLE LNFLG  ( oops flag  )
4    ARRAY UPSTAT ( update map )
    15 CONSTANT 15
    32 CONSTANT 32
   128 CONSTANT 128
5 32 * CONSTANT BLEN
: LMOVE 32 CMOVE ;
: BOL 88 @ YLOC @ 1+ 32 * + ;
: SBL  88 @ 544 + ;
: PBL  PAD 544 +  ;
: PBLL PBL BLEN + 32 - ;
: !SCR 88 @ 32 + PAD 512 BSCD) ;
                              --)
```

Screen: 68

```
( ValFORTH Video editor    V1.1 )
: CURLOC                  ( -- )
  BOL XLOC @ + ;      ( SCR ADDR )
: CSHOW                   ( -- )
  CURLOC DUP     ( GET SCR ADDR )
  C@ 128 OR      ( INVERSE CHAR )
  SWAP C! ;      ( STORE ON SCR )
: CBLANK                  ( -- )
  CURLOC DUP    ( GET SCR ADDR )
  C@ 127 AND    ( STRIP MSB )
  SWAP C! ;     ( STORE IT )
                              ==)
```

Screen: 69

```
( ValFORTH Video editor    V1.1 )
: UPCUR                   ( -- )
  CBLANK YLOC @
  1 - DUP 0<
  IF DROP 15 ENDIF
  YLOC ! CSHOW ;
: DNCUR                   ( -- )
  CBLANK YLOC @
  1 + DUP 15 >
  IF DROP 0 ENDIF
  YLOC ! CSHOW ;
                              --)
```

Screen: 70

```
( ValFORTH Video editor    V1.1 )
: LFCUR                   ( -- )
  CBLANK XLOC @
  1 - DUP 0<         ( AT L-SIDE?)
  IF DROP 31 ENDIF ( FIX IF SO )
  XLOC ! CSHOW ;
: RTCUR                   ( -- )
  CBLANK XLOC @
  1+ DUP 31 >        ( AT R-SIDE?)
  IF DROP 0 ENDIF    ( FIX IF SO )
  XLOC ! CSHOW ;
: EDMRK
  1 YLOC @ 4 / UPSTAT ! ;     ==)
```

Screen: 71

```
( ValFORTH Video editor    V1.1 )
: INTGL                   ( -- )
  INSRT @ 0=         ( TOGGLE THE )
  INSRT ! ;          ( INSRT FLAG )
: NXTLN                   ( -- )
  CBLANK 0 XLOC !
  CSHOW DNCUR ;
: CLREOL                  ( -- )
  CBLANK !SCR
  1 LNFLG ! CURLOC       ( CLEAR )
  32 XLOC @ -           ( TO END )
  ERASE CSHOW           ( OF LINE)
  EDMRK ;                     --)
```

```
Screen:  72
( ValFORTH Video editor    V1.1 )
: HMCUR                       ( -- )
  CBLANK 0 XLOC !
  0 YLOC ! CSHOW ;
: BYTINS CBLANK               ( -- )
  XLOC @ 31 <      ( SPREAD LN )
  IF
    CURLOC DUP 1+  ( FROM, TO )
    31 XLOC @ -    ( # CHARS )
    <CMOVE         ( MOVE IT )
  ENDIF
  0 CURLOC C!      ( CLEAR OLD )
  CSHOW EDMRK ;    ( CHARACTER )
                              ==>
```

```
Screen:  73
( ValFORTH Video editor    V1.1 )
: BYTDEL                      ( -- )
  CBLANK           ( CLOSE LINE)
  XLOC @ 31 <
  IF
    CURLOC DUP     ( FROM ADDR )
    1+ SWAP        ( TO ADDR )
    31 XLOC @ -    ( # CHARS )
    CMOVE          ( MOVE IT )
  ENDIF
  0 CURLOC         ( BLANK OUT )
  31 XLOC @ - + C! ( CHAR AT )
  CSHOW EDMRK ;    ( END OF LN )
                              -->
```

```
Screen:  74
( ValFORTH Video editor    V1.1 )
: LNINS                       ( -- )
  CBLANK 2 LNFLG ! !SCR
  4 YLOC @ 4 /
  DO 1 I UPSTAT ! LOOP
  YLOC @ 15 <
  IF
    BOL DUP 32 +
    15 YLOC @ - 32 *
    <CMOVE
  ENDIF
  BOL 32 ERASE
  CSHOW EDMRK ;
                              ==>
```

```
Screen:  75
( ValFORTH Video editor    V1.1 )
: LNDEL                       ( -- )
  CBLANK 3 LNFLG ! !SCR
  4 YLOC @ 4 /
  DO 1 I UPSTAT ! LOOP
  YLOC @ 15 <
  IF BOL                ( FROM )
    DUP 32 + SWAP       (  TO  )
    15 YLOC @ - 32 *    ( # CH )
    CMOVE
  ENDIF
  BOL 15 YLOC @ -
  32 * + 32 ERASE
  CSHOW EDMRK ;
                              -->
```

```
Screen:  76
( ValFORTH Video editor    V1.1 )
: BFSHW                       ( -- )
  PBLL 128 -          ( F , T  )
  SBL 160 CMOVE ;     ( # MOVE )
: BFROT                       ( -- )
  PBL DUP
  BLEN + LMOVE
  PBL DUP 32 +
  SWAP BLEN 32 -
  CMOVE PBLL 32 +
  PBLL LMOVE
  BFSHW ;
                              ==>
```

```
Screen:  77
( ValFORTH Video editor    V1.1 )
: <BFROT                      ( -- )
  PBLL DUP
  32 + LMOVE
  PBL DUP 32 +
  BLEN 32 - <CMOVE
  PBL DUP BLEN +
  SWAP LMOVE
  BFSHW ;
: BFCLR                       ( -- )
  PBLL 32 ERASE
  <BFROT ;
                              -->
```

Screen: 78

```
( ValFORTH Video editor    V1.1 )
: BFCPY                        ( -- )
  CBLANK BFROT       ( BRING LN )
  BOL PBLL           ( DOWN TO )
  LMOVE BFSHW        ( BUFFER & )
  CSHOW ;            ( ROTATE )
: >BFNXT BFCPY NXTLN ;       ( -- )
: >BFLN BFCPY LNDEL ;        ( -- )
: BFLN>                      ( -- )
  LNINS PBLL       ( TAKE LINE)
  BOL LMOVE        ( UP FROM )
  CSHOW <BFROT ; ( BUFFER   ) ==>
```

Screen: 79

```
( ValFORTH Video editor    V1.1 )
: BFRPL                        ( -- )
  CBLANK
  !SCR 4 LNFLG !    ( TAKE LINE )
  PBLL BOL LMOVE    ( UP TO SCR )
  <BFROT CSHOW      ( & ROTATE  )
  EDMRK ;
: TAB                          ( -- )
  CBLANK XLOC @ DUP
  31 = IF DROP -1 ENDIF
  4 + 4 / 4 * DUP 30 >
  IF DROP 31 ENDIF
  XLOC ! CSHOW ;
                                 -->
```

Screen: 80

```
( ValFORTH Video editor    V1.1 )
: RUB                          ( -- )
  XLOC @ 0= NOT   ( ON L-EDGE? )
  IF
    LFCUR           ( RUB IF NOT )
    0 CURLOC C!
    CSHOW EDMRK
  ENDIF
  INSRT @
  IF
    BYTDEL
  ENDIF ;
                                 ==>
```

Screen: 81

```
( ValFORTH Video editor    V1.1 )
: ARROW                        ( -- )
  CBLANK
  88 @ 541 + DUP @
  COND
    3341 = (( 30 7453 ))
    7453 = (( 00 0000 ))
  NOCOND
    30 3341
  CONDEND
  3 PICK !
  SWAP 2+ C!
  1 3 UPSTAT !
  CSHOW ;
                                 -->
```

Screen: 82

```
( ValFORTH Video editor    V1.1 )
: OOPS                         ( -- )
  LNFLG @
  IF
    CBLANK
    PAD 88 @ 32 + 512 >BSCD
    CSHOW
    0 LNFLG !
  ENDIF ;
                                 ==>
```

Screen: 83

```
( ValFORTH Video editor    V1.1 )
: SPLIT                        ( -- )
  YLOC @ 15 <>
  IF
    CBLANK
    LNINS
    BOL DUP 32 + SWAP
    XLOC @ CMOVE
    BOL 32 +
    XLOC @ ERASE
    CSHOW
  ENDIF ;
                                 -->
```

Screen: 84

```
( ValFORTH Video editor   V1.1 )
: SCRSV                          ( -- )
  88 @ 32 + PAD 512 BSCD>
  4 0
  DO
    I UPSTAT @
    0 I UPSTAT !
    IF
      PAD 128 I * +
      TBLK @ I + BLOCK
      128 CMOVE UPDATE
    ENDIF
  LOOP
  0 INSRT !
  0 XLOC ! 0 YLOC ! ;             ==>
```

Screen: 85

```
( ValFORTH Video editor   V1.1 )
: SCRGT                          ( -- )
  4 0
  DO
    TBLK @
    I + BLOCK
    PAD 128 I * +
    128 CMOVE
  LOOP
  PAD 88 @ 32 +
  512 >BSCD ;
                                  -->
```

Screen: 86

```
( ValFORTH Video editor   V1.1 )
: NWSCR                  ( -1/0/1 -- )
  CBLANK DUP
  IF SCRSV ENDIF 2* 2*
  TBLK @ + 0 MAX TBLK ! SCRGT
  TBLK @ 8 /MOD
  DUP <ROT SCR !
  IF 44 ELSE 53 ENDIF
  ?1K NOT
  IF
    44 = SWAP 2* + DUP SCR ! 0
  ENDIF
  88 @ 17 + C!
  0 84 C! 11 85 ! 1 752 C!
  . 2 SPACES CSHOW ;              ==>
```

Screen: 87

```
( ValFORTH Video editor   V1.1 )
: PRVSCR -1 NWSCR ;              ( -- )
: NXTSCR  1 NWSCR ;              ( -- )
: SPLCHR  1 ?ESC ! ;             ( -- )
: EXIT                           ( -- )
  HMCUR 19 LSTCHR ! ;
: EDTABT                         ( -- )
  0 UPSTAT 8 ERASE
  EXIT ;
                                  -->
```

Screen: 88

```
( ValFORTH Video editor   V1.1 )
: PTCHR                          ( -- )
  INSRT @ EDMRK
  IF BYTINS ENDIF
  LSTCHR @ 127 AND
  DUP LSTCHR !
  >SCD CURLOC C!
  RTCUR XLOC @ 0=
  IF DNCUR ENDIF
  0 ?ESC ! CSHOW ;
: CONTROL                       ( n -- )
  SEL 19 -> EXIT     17 -> EDTABT
      28 -> UPCUR    29 -> DNCUR
                                  ==>
```

Screen: 89

```
( ValFORTH Video editor   V1.1 )
      30 -> LFCUR    31 -> RTCUR
     126 -> RUB     127 -> TAB
       9 -> INTGL   155 -> NXTLN
     255 -> BYTINS  254 -> BYTDEL
     157 -> LNINS   156 -> LNDEL
      18 -> BFROT     2 -> <BFROT
       3 -> BFCLR    11 -> >BFNXT
      20 -> >BFLN     6 -> BFLN>
      16 -> PRVSCR   14 -> NXTSCR
      27 -> SPLCHR    8 -> CLREOL
       1 -> ARROW    21 -> BFRPL
      15 -> OOPS     10 -> SPLIT
  NOSEL PTCHR
  SELEND ;                        -->
```

Screen: 90

```
( ValFORTH Video editor   V1.1 )
: (V)                          ( TBLK -- )
  DECIMAL
  DUP BLOCK DROP TBLK !
  1 PFLAG ! 0 GR. 1 752 C! CLS
  1 559 C@ 252 AND OR 559 C!
  112 560 @ 6 + C!
  112 560 @ 23 + C!
  ." Screen #" 11 SPACES
  ." #Bufs: " BLEN 32 / . HIDCHR
  0 UPSTAT 8 ERASE 0 NWSCR
  PAD ?PADSM @ OVER ?PADSM ! =
  PBL @ ?BUFSM @ = AND NOT
  IF PBL BLEN ERASE ENDIF
                                    ==>
```

Screen: 91

```
( ValFORTH Video editor   V1.1 )
  BFSHW
  BEGIN
    INKEY$ DUP LSTCHR ! -DUP
    IF
      ?ESC @
      IF DROP PTCHR 0 LSTCHR !
      ELSE CONTROL ENDIF
    ELSE
      INSRT @
      IF
        CBLANK CSHOW
      ENDIF
    ENDIF
    LSTCHR @ 19 =
  UNTIL                             -->
```

Screen: 92

```
( ValFORTH Video editor   V1.1 )
  CBLANK SCRSV 0 767 C!
  2 560 @ 6 + C!
  2 560 @ 23 + C!
  PBL @ ?BUFSM !
  2 559 C@ 252 AND OR 559 C!
  0 LNFLG ! 0 752 C! CLS CR
  ." Last edit on screen # "
  SCR @ . CR CR 0 INSRT ! ;
FORTH DEFINITIONS
: V                            ( s -- )
  1 MAX B/SCR *
  EDITOR (V) ;                      ==>
```

Screen: 93

```
( ValFORTH Video editor   V1.1 )
: L                            ( -- )
  SCR @ DUP 1+
  B/SCR * SWAP B/SCR *
  EDITOR TBLK @ DUP <ROT
  <= <ROT > AND
  IF
    EDITOR TBLK @
  ELSE
    SCR @ B/SCR *
  ENDIF
  EDITOR (V) ;
                                    -->
```

Screen: 94

```
( ValFORTH Video editor   V1.1 )
: CLEAR                        ( s -- )
  B/SCR * B/SCR O+S
  DO
   FORTH I BLOCK
   B/BUF BLANKS UPDATE
  LOOP ;
: COPY                         ( s1 s2 -- )
  B/SCR * OFFSET @ +
  SWAP B/SCR * B/SCR O+S
  DO DUP FORTH I
    BLOCK 2- !
    1+ UPDATE
  LOOP DROP ( FLUSH ) ;             ==>
```

Screen: 95

```
( ValFORTH Video editor   V1.1 )
: CLEARS                       ( s # -- )
  OVER >R O+S
  2DUP CR
  ." Clear from SCR " . CR
  ."         thru SCR " 1 - . Y/N
  IF
    DO
      FORTH I CLEAR
    LOOP
  ELSE
    2DROP
  ENDIF
  R> SCR ! FLUSH ;
                                    -->
```

Screen: 96

```
( ValFORTH Video editor   V1.1 )

: WHERE EDITOR          ( n n --- )
  OVER OVER
  DUP 65532 AND
  SWAP OVER - 128 *
  ROT + 32 /MOD
  YLOC C!
  2- 0 MAX XLOC C!
  1 INSRT !
  EDITOR (V) ;

: #BUFS                         ( # -- )
  5 MAX 320 MIN 32 * EDITOR
  ' BLEN ! 0 ?PADSM ! ;
                                    ==)
```

Screen: 97

```
( ValFORTH Video editor   V1.1 )

: (LOC)                         ( sys )
  BLK @ , IN @ C, ;

: LOCATOR                       ( f -- )
  IF
    [ ' (LOC) CFA ] LITERAL
  ELSE
    [ ' NOOP CFA ] LITERAL
  ENDIF
  ' CREATE ! ;



                                    --)
```

Screen: 98

```
( ValFORTH Video editor   V1.1 )

: LOCATE
  [COMPILE] ' DUP NFA 1- DUP
  2- @ DUP 1439 U( SWAP 0# AND
  IF
    SWAP DROP DUP C@
    SWAP 2- @ WHERE 2DROP
  ELSE
    CR ." Cannot locate"
    '( DCMPR DROP DCMPR
    )( 2DROP CR )
  ENDIF ;


                            BASE !
```

Screen: 99

Screen: 100

Screen: 101

Screen: 102

Screen: 103

Screen: 104

Screen: 105

Screen: 106

```
( Hi-resolution text printing  )

BASE @ DCX

'( )SCD )( 26 KLOAD )
'( COND )( 28 KLOAD )

57344 VARIABLE GCBAS
    0 VARIABLE GCPTR
    2 VARIABLE GCLFT
   39 VARIABLE GCRGT
    0 VARIABLE GMOD
    0 VARIABLE GCCOL
    0 VARIABLE GCROW
  120 VARIABLE VMI#
                                 ==>
```

Screen: 107

```
( Hi-res:  GCR                  )

: GCR                       ( -- )
  1 GCROW @ + DUP 20
  703 C@ MAX <
  IF GCROW !
  ELSE
    DROP 88 @ 320 O+S
    703 C@ 4 =
    IF 6400 ELSE 7680 ENDIF 2DUP
    + 320 - >R CMOVE
    R> 320 ERASE
  ENDIF
  GCROW @ 320 *
  GCLFT @ DUP GCCOL !
  + GCPTR ! ;                   -->
```

Screen: 108

```
( Hi-res:  [GCEMIT]                 )

: (GCEMIT)                   ( c -- )
  >SCD 8 * GCBAS @ +
  GCPTR @ 88 @ + 320 O+S
  DO
    DUP C@ GMOD C@
    IF I C@ OR ENDIF
    I C! 1+
  40 /LOOP
  DROP 1 GCPTR +!
  1 GCCOL @ + DUP GCRGT @ >
  IF DROP GCR
  ELSE GCCOL !
  ENDIF ;
                                   ==>
```

Screen: 109

```
( Hi-res:  GCBKS OSTRIKE GCINIT)

: GCBKS                       ( -- )
  GCCOL @ GCLFT @ >
  IF
    -1 GCCOL +!      ( backspace )
    -1 GCPTR +!
  ENDIF ;

: OSTRIKE                   ( f -- )
  GMOD ! ;           ( overstrike)

: GCINIT                      ( -- )
  0 GCROW ! GCLFT @ DUP
  GCCOL ! GCPTR ! ;
                                 -->
```

Screen: 110

```
( Hi-res:  GCPOS SUPER SUB      )

: GCPOS              ( col row -- )
  2DUP 320 * + GCPTR !
  GCROW ! GCCOL ! ;

: SUPER                       ( -- )
  VMI# @ MINUS GCPTR +! ;

: SUB                         ( -- )
  VMI# @ GCPTR +! ;




                                 ==>
```

Screen: 111

```
( Hi-res:  GCEMIT  GCTYPE       )

: GCEMIT                ( chr -- )
  DUP
  COND
    28 = (( DROP SUPER ))
    29 = (( DROP SUB   ))
    30 = (( DROP GCBKS ))
  NOCOND (GCEMIT)
  CONDEND ;

: GCTYPE         ( adr count -- )
  0 MAX -DUP
  IF O+S DO I C@ GCEMIT LOOP
  ELSE DROP
  ENDIF ;                        -->
```

Screen: 112

```
( Hi-res:  [GC"]  GC"               )

: (GC")                        ( -- )
  R COUNT DUP 1+ R> + >R
  GCTYPE ;


: GC"                          ( -- )
  34 STATE @
  IF
    COMPILE (GC") WORD
    HERE C@ 1+ ALLOT
  ELSE
    WORD HERE COUNT GCTYPE
  ENDIF ; IMMEDIATE
                                  ==>
```

Screen: 113

```
( Hi-res:  GCSPACE[S]  GCD.R    )

: GCSPACE                     ( -- )
  BL GCEMIT ;

: GCSPACES                  ( n -- )
  0 MAX -DUP
  IF 0
   DO GCSPACE
   LOOP
  ENDIF ;

: GCD.R                   ( d n -- )
  >R SWAP OVER DABS
  (# #S SIGN #) R> OVER -
  GCSPACES GCTYPE ;              -->
```

Screen: 114

```
( Hi-res:  GC.R  GC.  GCLEN    )

: GC.R                    ( n n -- )
  >R S->D R> GCD.R ;

: GC.                       ( n -- )
  0 GC.R GCSPACE ;

: GCLEN      ( adr cnt -- #chrs )
  0 <ROT O+S
  DO I C@ 28 -
     CASE 0 0 0
     NOCASE 1
     CASEND +
  LOOP ;
                               ==>
```

Screen: 115

```
( Hi-res:  VMI  GC.$            )

: VMI                       ( n -- )
  40 * VMI# ! ;

: GC$.                    ( adr -- )
  COUNT GCTYPE ;

: GCLS                       ( -- )
  88 @
  703 C@ 4 =
  IF 6400 ELSE 7680 ENDIF
  ERASE
  GCRGT @ 0 GCPOS ;

GCINIT                   BASE !
```

Screen: 116

Screen: 117

Screen: 118

Screen: 119

Screen: 120

```
( Double:  DVAR DCON D- D>R DR>)
BASE @ DCX
: DVARIABLE       ( cccc -- adr )
  VARIABLE , ;
: DCONSTANT        ( cccc -- d )
  <BUILDS , ,
  DOES> D@ ;
0. DCONSTANT 0.   1. DCONSTANT 1.
: D-              ( d d -- d )
  DMINUS D+ ;
                                ==>
```

Screen: 121

```
( Double:  D0= D= D0< D< D>     )
: D0=                ( d -- f )
  OR 0= ;
: D=               ( d d -- f )
  D- D0= ;
: D0<                ( d -- f )
  SWAP DROP 0< ;
: D<               ( d d -- f )
  D- D0< ;
: D>               ( d d -- f )
  2SWAP D< ;                    -->
```

Screen: 122

```
( Double:  DMIN  DMAX           )
: DMIN             ( d d -- d )
  2OVER 2OVER D>
  IF
    2SWAP
  ENDIF
  2DROP ;
: DMAX             ( d d -- d )
  2OVER 2OVER D<
  IF
    2SWAP
  ENDIF
  2DROP ;
                                ==>
```

Screen: 123

```
( Double:  D>R  DR>  D,  M+     )
: D>R                   ( d -- )
  R> (ROT SWAP >R >R >R ;
: DR>                   ( -- d )
  R> R> R> SWAP ROT >R ;
: D,                    ( d -- )
  , , ;
: M+                ( d n -- d )
  S->D D+ ;
                                -->
```

Screen: 124

```
( Double:  DU<                  )
: DU<
  DUP 4 PICK XOR 0<
  IF
    2DROP D0< NOT
  ELSE
    D- D0<
  ENDIF ;
                           BASE !
```

Screen: 125

Screen: 126

Screen: 127

Screen: 128

Screen: 129

Screen: 130

Screen: 131

Screen: 132

Screen: 133

Screen: 134

```
( Utils:  Initialization         )

BASE @ DCX

   '( XC! )( 21 KLOAD )
   '( HIDCHR )( 24 KLOAD )
   '( )BSCD )( 26 KLOAD )








                                 ==>
```

Screen: 135

```
( Utils:                           )














                                   -->
```

Screen: 136

```
( Utils:  XR/W                     )

: XR/W       ( #secs a blk# f -- )
  4 PICK 0
   DO
    3 PICK I B/BUF * +
    3 PICK I + 3 PICK R/W
   LOOP
   2DROP 2DROP ;






                                   ==>
```

Screen: 137

```
( Utils:  SMOVE                    )

: SMOVE        ( org des cnt -- )
  FLUSH MTB
  741 @ PAD DUP 1 AND - - 2DUP
  SWAP B/SCR * B/BUF * U<
  IF CR ." Too many: "
   B/BUF B/SCR * / U.
   ." max." DROP 2DROP
  ELSE DROP
   >R DCX MTB CR
   ." SMOVE from " OVER DUP 3 .R
   ."  thru " R + 1- 3 .R CR
   8 SPACES
   ." to " DUP DUP 3 .R
   ."  thru " R + 1- 3 .R     -->
```

Screen: 138

```
( Utils:  SMOVE                )
  SPACE Y/N
  IF
   CR ." Insert source" RETURN
   R B/SCR * PAD DUP 1 AND -
   4 ROLL B/SCR * OFFSET @ +
   1 XR/W
   CR ." Insert dest." RETURN
   R> B/SCR * PAD DUP 1 AND -
   ROT B/SCR * OFFSET @ +
   0 XR/W
  ELSE R> DROP 2DROP
   CR ." Smove aborted..." CR
  ENDIF
 ENDIF ;                      ==>
```

Screen: 139

```
( Utils:  LOADS  THRU            )
: LOADS             ( n cnt -- )
  O+S
  DO
   I LOAD ?EXIT
  LOOP ;
: THRU          ( n n -- n cnt )
  OVER - 1+ ;
                               -->
```

Screen: 140

```
( Utils:  SEC  MSEC              )
: SEC                   ( n -- )
  0 DO
   9300 0
   DO
   LOOP
  LOOP ;
: MSEC                  ( n -- )
  0 DO
   6 0
   DO
   LOOP NOOP
  LOOP ;
                               ==>
```

Screen: 141

```
( Utils:  H->L  L->H  H/L
HEX
CODE H->L              ( n -- n )
     B5 C, 01 C, 95 C, 00 C,
     94 C, 01 C, 4C C, NEXT , C;
CODE L->H              ( n -- n )
     B5 C, 00 C, 95 C, 01 C,
     94 C, 00 C, 4C C, NEXT , C;
CODE H/L             ( n -- n n )
     B5 C, 00 C, 94 C, 00 C,
     4C C, PUSH0A , C;
DCX                            -->
```

Screen: 142

```
( Utils:  BIT  ?BIT  TBIT       )
HEX
CODE BIT                ( b -- n )
 B4 C, 00 C, C8 C, A9 C, 00 C,
 95 C, 00 C, 95 C, 01 C, 38 C,
 36 C, 00 C, 36 C, 01 C, 18 C,
 88 C, D0 C, F8 C, 4C C, NEXT ,
C;
: ?BIT BIT AND 0# ; ( n b -- f )
: TBIT BIT XOR ;     ( n b -- n )
: SBIT BIT OR ;      ( n b -- n )
: RBIT               ( n b -- n )
  FFFF SWAP TBIT AND ;        ==>
```

Screen: 143

```
( Utils:  STICK                 )
HEX
HERE DUP 2DUP   0 , 1 , -1 , 0 ,
CODE STICK           ( n -- h v )
  B4 C, 00 C, B9 C, 78 C, 02 C,
  48 C, CA C, CA C, 29 C, 03 C,
  0A C, A8 C, B9 C,     , 95 C,
  02 C, C8 C, B9 C,     , 95 C,
  03 C, 68 C, 4A C, 4A C, 29 C,
  03 C, 0A C, A8 C, B9 C,     ,
  95 C, 00 C, C8 C, B9 C,     ,
  95 C, 01 C, 4C C, ' SWAP ,
CURRENT @ CONTEXT !            -->
```

Screen: 144

```
( Utils:  STRIG  PADDLE         )
HEX


CODE PADDLE                ( n -- n )
  B4 C, 00 C,  B9 C, 270 ,
  4C C, PUT0A , C;

CODE STRIG                 ( n -- f )
  B4 C, 00 C, B9 C, 284 ,
  49 C, 01 C, 4C C, PUT0A , C;

CODE PTRIG                 ( n -- f )
  B4 C, 00 C, B9 C, 27C ,
  49 C, 01 C, 4C C, PUT0A , C;
                                 ==>
```

Screen: 145

```
( Utils:  ATRACT  NXTATR        )

DCX

: ATTRACT                  ( f -- )
  IF 255 ELSE 0 ENDIF 77 C! ;

: NXTATR
  255 20 C! ;                ( -- )
( Changes user clock )

: HLDATR
  0 20 C! ;                  ( -- )
( Changes user clock )

                                 -->
```

Screen: 146

```
( Utils:  16TIME                )
HEX

CODE 16TIME
  CA C, CA C,
  A5 C, 13 C,  95 C, 01 C,
  A5 C, 14 C,  95 C, 00 C,
  D0 C, 04 C,
  A5 C, 13 C,  95 C, 01 C,
  4C C, NEXT , C;





                                 ==>
```

Screen: 147

```
( Utils:  8RND  16RND  CHOOSE  )
HEX

CODE 8RND                  ( -- b )
  AD C, D20A ,
  4C C, PUSH0A ,
C;

CODE 16RND                 ( -- n )
  AD C, D20A , 48 C, 68 C, 48 C,
  68 C, 48 C, AD  C, D20A ,
  4C C, PUSH ,  C;

: CHOOSE                   ( n -- n )
  16RND U* SWAP DROP ;
                                 -->
```

Screen: 148

```
( Utils:  CSHUFL  SHUFL         )
DCX
: CSHUFL                   ( a n -- )
  1- 0 SWAP
  DO
    DUP I CHOOSE + OVER I +
    2DUP C@ SWAP C@
    ROT C! SWAP C!
  -1 +LOOP DROP ;

: SHUFL                    ( a n -- )
  1- 0 SWAP
  DO DUP I CHOOSE 2* +
    OVER I 2* +
    2DUP @ SWAP @ ROT ! SWAP !
  -1 +LOOP DROP ;                ==>
```

Screen: 149

```
( Utils:  H.  A.                )

: A.                       ( a -- )
  C@ 127 AND
  DUP 32 < OVER
  124 > OR
  IF DROP 46 ENDIF
  SPEMIT ;

'( H. --> )(  )

: H.                       ( d -- )
  BASE @ HEX SWAP
  0 <# # # # #> TYPE
  BASE ! ;
                                 -->
```

Screen: 150

```
( Utils:  DUMP                     )
DCX

: DUMP                 ( a n -- )
  O+S
  DO
   CR I H->L H. I H.
   2 SPACES I 8 O+S 2DUP
   DO
    I C@ H. SPACE
   LOOP CR 7 SPACES
   DO
    I A. 2 SPACES
   LOOP ?EXIT
  8 /LOOP
  CR ;                         ==>
```

Screen: 151

```
( Utils:  BLKOP  -- system      )
HEX

CODE BLKOP  ( adr cnt byte -- )
  A9 C, 03 C, 20 C, SETUP ,
  HERE         C4 C, C4 C, D0 C,
  07 C, C6 C, C5 C, 10 C, 03 C,
  4C C, NEXT ,       B1 C, C6 C,
  A5 C, C2 C, 91 C, C6 C, C8 C,
  D0 C, EC C, E6 C, C7 C, 4C C,
  , DCX
C;



                               -->
```

Screen: 152

```
( Utils:  BXOR                     )
HEX
CODE BXOR     ( adr cnt byte -- )
  A9 C, 45 C,
  8D C, ' BLKOP 12 + ,
  4C C, ' BLKOP ,  C;

CODE BAND     ( adr cnt byte -- )
  A9 C, 25 C,
  8D C, ' BLKOP 12 + ,
  4C C, ' BLKOP ,  C;

CODE BOR      ( adr cnt byte -- )
  A9 C, 05 C,
  8D C, ' BLKOP 12 + ,
  4C C, ' BLKOP , C;  BASE !
```

Screen: 153

Screen: 154

Screen: 155

```
Screen: 156
( Strings:  -TEXT                 )
BASE @ DCX
: -TEXT                 ( a u a -- )
  2DUP + SWAP
  DO
    DROP 1+
    DUP 1- C@
    I C@ - DUP
    IF
      DUP ABS
      / LEAVE
    ENDIF
  LOOP
  SWAP DROP DUP
  IF 1 SWAP +- ENDIF ;
                                ==>

Screen: 157
( Strings:  -NUMBER               )

0 VARIABLE NFLG

: -NUMBER            ( addr -- d )
  BEGIN DUP C@ BL = DUP + NOT
  UNTIL 0 NFLG ! 0 0 ROT DUP 1+
  C@ 45 = DUP >R + -1
  BEGIN DPL ! (NUMBER) DUP C@
  DUP BL <> SWAP 0# AND
  WHILE DUP C@ 46 - NFLG !
  0 REPEAT DROP R> IF DMINUS
  ENDIF NFLG @
  IF 2DROP 0 0 ENDIF
  NFLG @ NOT NFLG ! ;
                                -->

Screen: 158
( Strings:  UMOVE , $!            )


FORTH DEFINITIONS

: UMOVE                 ( a a n -- )
  <ROT OVER OVER U<
  IF
    ROT <CMOVE
  ELSE
    ROT CMOVE
  ENDIF ;

: $!
  OVER C@ 1+ UMOVE ;
                                ==>

Screen: 159
( Strings:  $CON , $VAR , ["]  )

: $CONSTANT            ( $ ccc -- )
  PAD 512 + SWAP OVER $!
  0 VARIABLE -2 ALLOT
  HERE $! HERE C@ 1+ ALLOT ;

: $VARIABLE          ( len ccc -- )
  0 VARIABLE
  1- ALLOT ;

: (")                      ( -- $ )
  R DUP C@ 1+ R> + >R ;


                                -->

Screen: 160
( Strings:   "                    )

: "
  34 ( Ascii quote )
  STATE @
  IF                    ( cccc" -- )
    COMPILE (") WORD
    HERE C@ 1+ ALLOT
  ELSE
    WORD HERE        ( cccc" -- $ )
    PAD $! PAD
  ENDIF ;

  IMMEDIATE

                                ==>

Screen: 161
( Strings:  $. , $XCHG            )

: $.                      ( $ -- )
  DUP C@ 0>
  IF
    COUNT TYPE
  ELSE
    DROP
  ENDIF ;


: $XCHG                ( $1 $2 -- )
  DUP PAD 256 + $!
  OVER SWAP $!
  PAD 256 + SWAP $! ;
                                -->
```

```
Screen: 162
( Strings:  $+ , LEFT$            )

: $+                 ( $1 $2 -- $ )
  SWAP PAD 256 +
  >R R $!
  DUP C@ SWAP 1+
  R C@ 1+ R +
  3 PICK UMOVE
  R C@ + 255 MIN
  R C! R> PAD $! PAD ;

: LEFT$                 ( $ N -- $ )
  SWAP PAD <ROT PAD $!
  OVER C@ MIN
  OVER C! ;
                                 ==>
```

```
Screen: 163
( Strings:  RIGHT$ , MID$         )

: RIGHT$               ( $ n -- $ )
  SWAP PAD <ROT PAD $!
  OVER <ROT OVER C@
  DUP 4 PICK +
  <ROT MIN DUP
  <ROT 1- -
  SWAP ROT OVER OVER
  C! 1+ SWAP CMOVE ;

: MID$        ( $ start len -- $ )
  3 PICK C@ 1+ ROT -
  0 MAX ROT SWAP
  RIGHT$ SWAP OVER
  C@ MIN OVER C! ;               -->
```

```
Screen: 164
( Strings:  LEN , ASC , $COMP  )

: LEN               ( $ -- length )
  C@ ;

: ASC                     ( $ -- c )
  1+ C@ ;

: $COMPARE          ( $1 $2 -- f )
  2DUP C@ SWAP C@ SWAP
  2DUP MIN <ROT - >R
  ROT 1+ <ROT SWAP 1+
  -TEXT -DUP 0=
  IF R> DUP IF 1 SWAP +- ENDIF
  ELSE R> DROP ENDIF ;
                                 ==>
```

```
Screen: 165
( Strings:  $( , $= , $) , SV$   )

: $(                  ( $1 $2 -- f )
  $COMPARE 0< ;

: $=                  ( $1 $2 -- f )
  $COMPARE 0= ;

: $)                  ( $1 $2 -- f )
  $COMPARE 0> ;

: SAVE$                   ( $ -- $ )
  PAD 512 + SWAP
  OVER $! ;
                                 -->
```

```
Screen: 166
( Strings:  INSTR                  )

0 VARIABLE INCNT

: INSTR              ( $1 $2 -- n )
  0 INCNT ! 1+ SWAP DUP
  >R OVER 1- C@ >R 1+
  DUP 1- C@ R - 1+ 0 MAX
  OVER + SWAP R> <ROT
  DO
    2DUP I -TEXT 0=
    IF
      I J - INCNT ! LEAVE
    ENDIF
  LOOP
  2DROP R> DROP INCNT @ ;        ==>
```

```
Screen: 167
( Strings:  CHR$ , DVAL , VAL  )

: CHR$                    ( c -- $ )
  1 PAD C!
  PAD 1+ C!
  PAD ;

: DVAL                    ( $ -- d )
  PAD $! PAD
  DUP C@ OVER 1+ +
  0 SWAP C!
  -NUMBER ;

: VAL                     ( $ -- n )
  DVAL DROP ;
                                 -->
```

Screen: 168

```
( Strings:  DSTR$ , STR[ING]$  )

: DSTR$                    ( d -- $ )
  DUP <ROT DABS
  <# #S SIGN #>
  SWAP 1- DUP
  <ROT C! PAD $! PAD ;

: STR$                     ( d -- $ )
  S->D DSTR$ ;

: STRING$                ( n $ -- $ )
  1+ C@ OVER
  PAD C! PAD
  1+ <ROT FILL PAD ;
                                  ==>
```

Screen: 169

```
( Strings:  $-TB , #IN$ , IN$  )

: $-TB                     ( $ -- $ )
  DUP DUP 1+ SWAP C@
  -TRAILING SWAP DROP
  OVER C! ;

: #IN$                     ( n -- $ )
  -DUP 0= IF 255 ENDIF
  PAD 1+ SWAP EXPECT PAD
  BEGIN 1+ DUP C@ 0= UNTIL
  PAD 1+ - PAD C! PAD ;

: IN$                       ( -- $ )
  0 #IN$ ;
                          BASE !
```

Screen: 170

```
CONTENTS OF THIS DISK:

TRANSIENTS:                36 LOAD
ARRAYS & THEIR COUSINS:    42 LOAD
KEYSTROKE WORDS:           48 LOAD
SCREEN CODE CONVERSION:    52 LOAD
CASE STATEMENTS:           56 LOAD
valFORTH EDITOR 1.1:       64 LOAD
HIGH-RES TEXT:            106 LOAD
DOUBLE NUMBER XTNSIONS:   120 LOAD
MISCELLANEOUS UTILS:      134 LOAD
STRING WORDS:             156 LOAD
```

Screen: 171

Screen: 172

Screen: 173

```
Screen: 174
```

```
Screen: 175
```

```
Screen: 176
( Error messages                         )

Stack empty

Dictionary full

Wrong addressing mode

Is not unique

Value error

Disk address error

Stack full
```

```
Screen: 177
Disk Error!

Dictionary too big
```

```
Screen: 178
( Error messages                         )

Use only in Definitions

Execution only

Conditionals not paired

Definition not finished

In protected dictionary

Use only when loading

Off current screen
```

```
Screen: 179
Declare VOCABULARY
```

VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141

valFORTH T.M.

SOFTWARE SYSTEM
for ATARI*

valDOS

valFORTH T.M.
SOFTWARE SYSTEM

valDOS

Stephen Maguire

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

valDOS

Table of Contents

LXI. Strolling Through valDOS
A brief look at the features of the valDOS system.

LXII. valDOS Command Descriptions
Detailed descriptions of all the valDOS commands.

LXIII. valDOS System Glossary
a) The internal workings of valDOS 1
b) The system glossary 2
c) The command system glossary 10
d) Error messages and meanings 12

LXIV. valDOS File Editor
User's manual for the valDOS file editor.

LXV. valDOS I Supplied Source listing

LXVI. valDOS II Supplied Source Listing

Strolling Through valDOS

Until now, one of the major drawbacks of FORTHs for Atari was that you could could either save code on screens or in DOS files, but not both. The demand to have both brought on the development of this two-disk package -- valDOS.

With valDOS you can load FORTH screens just as before, or you can load FORTH source from standard Atari DOS files. But this package allows much more than that. A complete file editor is supplied which can be used to edit FORTH (or assembly, Pascal, etc.) source code and save it in DOS format.

To use this package to its fullest, at least 32K of memory should be available. If you have less than 32K, don't worry, it just means that you can't load all of the commands at once. The package is broken into six major parts:

1) valDOS (approximately 3.6K compiled)
2) valDOS extensions
3) Basic DOS commands (keyboard entry only)
4) DOS command extensions
5) Formatter/copiers
6) File editor

We shall begin this stroll by taking a look at the first five parts. Before starting, duplicate both master disks (we'll be modifying the valDOS II copy). Insert your valDOS I disk in drive one and list screen 170 by typing 170 LIST . Look for the line that says DOS COMMAND EXTENSIONS and load them in. This should take a few minutes. There are so many commands with so many variations on each that only the highlights of the package will be discussed here. Each of the commands is explained in detail in section LXII.

Insert your copy of the valDOS II diskette. Type DIR and press the return key. You should see the following on your display:

```
Files on: valDOS -- disk II

  NAME     EXT   SIZE   SEC   ATTR
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

 ASSM     .4TH    33     4     L
 XBOUT    .4TH    10    37     L
 DUCLISTS.4TH     7    47     L
 FILEIT   .4TH     8    54     L

209 sectors free.
```

The DIR command simply displays the directory of the disk. There are four files on this disk. ASSM.4TH is the standard valFORTH assembler with a correction for the "absolute,Y" instruction which assembled incorrectly when the "absolute" address was in the zero page. XBOOT.4TH is a routine which allows dictionaries greater than 32K to be SAVEd and AUTOed. DOCLISTS.4TH is the routine used to list six screens/page for valFORTH documentation. And finally, FILEIT.4TH is a routine that transfers FORTH screens to DOS files.

If you have a two drive system, DIR 2 will give the directory of the disk in drive two. The DIR command will also accept file specifications. Try the following:

```
DIR  ASSM.4TH
DIR  X*
DIR  ????I*
DIR  MYFILE
```

In these examples, there are the two characters "?" and "*" which need some explanation. Simply put, "?" will match any single letter when compare with a filename in the directory. "*" will match any group of letters to the right from its position to the end of the filename. In the second example above, all files starting with "X" are listed. In the third example, all files whose fifth letter is "I" are listed (the first four can be anything).

For advanced FORTH programmers, the SECtor column of the listing is the first sector of the file. This is a DOS sector (FORTH sectors are offset by -1, i.e., FORTH sectors are numbered starting at zero while Atari has numbered DOS sectors beginning with one; hence, DOS sector 4 is FORTH sector 3). The last column indicates the attributes of the files. All of the files on this disk are locked and therefore have the "L" attribute.

Notice that unlike most FORTH words, DIR expects its arguments on the right. Although valDOS system itself expects all parameters to be passed on the stack, the valDOS user commands all expect their arguments on the right, as is usual with DOS's. The only restriction on this method of input is that no blanks may appear within the command list.

Let's make a copy of one of the files. The COPY command does this nicely for us:

```
COPY  FILE1=FILEIT.4TH
```

This command has the basic format "new=old". The COPY command can also append two or more files together and save this new file into another file:

```
COPY  FILE2=FILEIT.4TH,DOCLISTS.4TH
```

FILE2 should now contain FILEIT.4TH with DOCLISTS.4TH appended to it. It is possible to string as many files together as desired. We're not quite through yet. If we were to enter:

```
COPY  FILE2=XBOOT.4TH
```

FILE2 would contain an exact copy of XBOOT.4TH. However, if we typed:

```
COPY  FILE2/A=XBOOT.4TH
```

XBOOT.4TH would be appended to whatever was already in FILE2, namely FILE1T.4TH + DOCLISTS.4TH. Of course, also legal is:

```
COPY  FILE2/A=XBOOT.4TH,ASSM.4TH
```

and all of the original files would be contained in FILE2. Unfortunately, FILE2 is FORTH source, but does not have the extension 4TH (it is not needed, but desirable). The RENAME command can rememdy this:

```
RENAME  FILE2.4TH=FILE2
```

Like the COPY command, the format for RENAME is "new=old". For some, typing a long name like RENAME might be a chore. Since we are are in FORTH, we can easily customize:

```
: REN   RENAME ;
```

Now that we have some more files to work with, type

```
    DIR  F*
and DIR  F*.
```

Notice the difference? In the first example, the remainder of the filename is wild, while in the second example, the files must end with a null extension.

If we want to keep the file FILE1 from being modified in any way, we can lock it using the LOCK command:

```
LOCK  FILE1
DIR
```

The LOCK (and so UNLOCK and KILL) command can take several arguments, separated by commas:

```
LOCK  FILE1,MYFILE,ASSM.4TH,F*.4TH
```

In this case, LOCK would lock FILE1, report MYFILE as being non-existent, lock ASSM.4TH, and then issue a verification prompt before locking any file that matches the specification F*.4TH. Try the following two examples:

```
LOCK  *.4TH
LOCK  *.4TH/N
```

In the first form, a verification prompt is issued for each file before it is locked, while in the second form, the verification message is not displayed. The /N switch stands for "No ask" or "No verify".

Both KILL and UNLOCK have exactly the same argument list as LOCK. Use caution when using the /N switch with the KILL command.

If we want to get a listing of a text file, we use the PRINT command. For example:

```
PRINT  FILE-1T.4TH
```

we get:

File: D1:FILEIT.4TH

```
( Routine: FILE-IT
  The following routine will
  transfer a specified range of
  FORTH screens to a file on
  a DOS formatted disk.
  Format:  FILE-IT 1st,last,filename
           FILE-IT 10,20,MYFILE
  Note that DOS commands tend to be
  long because of error checking and
  parameter parsing.                  )
: FILE-IT DOS              ( -- )
  GETARGS ?WRGARG 44 GETARG ?WRGARG
  GETVAL SWAP 44 GETARG ?WRGARG
  GETVAL SWAP <ROT OVER - 1+
  PAD DUP 1 AND - BUFBOT OVER -
  B/BUF / 3 PICK B/SCR * OVER >
  IF
    CR ." Too many screens, "
    B/SCR / . ." max." CR
    2DROP 2DROP
  ELSE
    DROP <ROT O+S
    DO
      16 0
      DO
        I J (LINE)
        -TRAILING >R OVER
        R CMOVE R> +
        155 OVER C! 1+
      LOOP
    LOOP
    PAD DUP 1 AND - SWAP OVER -
    FLUSH INSDST ROT DUP (ENTER) 0=
    IF FLEXST DSKERR = ?SYSERR ENDIF
    (OPEN) ?SYSERR SWAP DROP >R
    R (WRITE) ?SYSERR
    R (ENDF) ?SYSERR
    R> (CLOSE)
  ENDIF
  CR ;
FORTH
```

The PRINT command has several options. If the switch /N (no line numbers) is present, the file is displayed without line numbers. If an optional starting line number is supplied, printing begins at that line:

```
PRINT FILEIT.4TH/N,17
```

This will print the file starting with line 17, and with no line numbers. Note that the listing may be aborted at any time by pressing one of the yellow console keys.

The PRINT command is ideal for displaying text files, but is utterly useless for listing a binary or data file. For this reason, the command FDUMP (file DUMP) has been supplied. Let's try FDUMP on FILEIT.4TH:

```
FDUMP FILEIT.4TH

File: D1:FILEIT.4TH

0000  9B 28 20 52 6F 75 74 69  .( Routi
0008  6E 65 3A 20 46 49 4C 45  ne: FILE
0010  2D 49 54 9B 20 9B 20 20  -IT. .
0018  54 68 65 20 66 6F 6C 6C  The foll
0020  ...
```

Like PRINT, FDUMP may be aborted by pressing one of the yellow console keys. If the filename has the "wide" switch /W appended to it, the file is dumped with 16 bytes/line instead of eight as above. This format is more appropriate when sent to a printer. FDUMP always dumps in hexadecimal.

Let's compile a routine from a DOS file. The file DOCLISTS.4TH is the program used to print 6 screens/page for documentation. To load this file, we simply enter:

```
FLOAD DOCLISTS.4TH
```

and the routine DOCLISTS should now be in the dictionary. There is a nice feature to the FLOAD command:

```
FORGET DOCLISTS
ON ECHO
FLOAD DOCLISTS.4TH
```

Notice that this time the file was echoed to the display as it was being loaded. Like the PRINT command, FLOAD can take an optional starting line so that loading can begin mid-file. Holding down a console key will abort a load once the current definition is compiled.

If you have a source file which is physically contained two or more files, they can be linked together using the FLOAD command. For example, PART1.4TH could end with an

FLOAD PART2.4TH, and part2 could end with FLOAD PART3.4TH, etc. In this way, a multi-part file could be loaded. Although this method works, it should be avoided for several reasons. One reason is that if any of the files are renamed, or if they are moved to a different drive, each of the original files may have to be edited to change the FLOADs. A second and more complicated reason is that for each file FLOADed this way, a separate file buffer needs to be allocated. The default number of file buffers is four, therefore no more than four files can be chained.

The solution to this problem is to create a file which contains nothing but FLOADs:

File: ALLPARTS.4TH

```
0001 FLOAD PART1.4TH  ( load in the ??? routines )
0002 FLOAD PART2.4TH  ( load in the ??? routines )
0003 FLOAD PART3.4TH  ( ... )
```

This method requires exactly two file buffers and allows filenames to be changed easily, if desired.

This was just a brief stroll through the valDOS package. There are many powerful commands left to explore. The following section on command words explains each command in detail. Read through this section carefully and out the commands on a test disk. And then there is the File Editor, with its own set of documentation, in section LXIV.

Have fun.

File: D2:VERIFY.4TH

```
( Routine: Write w/o verify

  The following routine allows write
  operations to the disk without
  read verification. This speeds up
  disk access by many times. Note
  that once this routine is loaded,
  it may not be forgotten!

  Format:  ON  VERIFY
               OFF VERIFY
                                                  )

BASE @ HEX
ASSEMBLER

LABEL -DSK
   AD C, 02 C, 03 C, C9 C, 52 C,
   D0 C, 05 C, A9 C, 40 C, 4C C,
   HERE 4 + ,  A9 C, 80 C, 8D C,
   03 C, 03 C, A9 C, 31 C, 8D C,
   00 C, 03 C, A9 C, 07 C, 8D C,
   06 C, 03 C, A9 C, 80 C,
   ( or 00 C, for Percom?) 8D C,
   08 C, 03 C, A9 C, 00 C,
   ( or 01 C, for Percom?) 8D C,
   09 C, 03 C, 20 C, 59 C, E4 C,
   60 C,


: VERIFY                     ( f -- )
  0# 7 * 50 +
  [ ' -DISK 7 + ]
  LITERAL C! ;

-DSK ' -DISK 27 + !

BASE !
```

Command Words

Introduction and Conventions

The valFORTH Disk Operating System can be broken into two distinct categories. The first contains the system words which are for use within running programs and are rarely typed directly at the keyboard. The second category contains "command" words which were designed to be executed only at the keyboard. Typical command words are those that list the directory of a disk or delete a file from the disk.

Commands words differ from normal FORTH words in that all necessary arguments are entered following the command word. For example, to remove a file from a directory, we would type:

```
KILL UNWANTED.FIL
```

instead of the usual FORTH-like:

```
" UNWANTED.FIL" KILL
```

which will not work as-is. The only restriction placed on this method of input is that absolutely no blanks must appear within the command list since the blank serves to indicate the end of that list. Thus,

```
KILL FILE1.FILE2,FILE3
```

would properly kill the three files specified while

```
KILL FILE1, FILE2, FILE3
```

would kill the first file and then abort with an error.

In the command descriptions that follow, any portion of the command format enclosed by the braces "{" and "}" is optional and need not be entered.

Additionally, some of the commands may be aborted by pressing one of the yellow console keys found on the far right of the keyboard. Those commands which have this feature are indicated by the sentence:

"This command is interruptable"

All commands and arguments must be entered in upper case.

CLOSE

Release file buffer and update file.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

```
CLOSE  {filenum}
```

OPERATION:

The CLOSE command flushes the file buffer (different from the FORTH disk buffers), if updated, associated with the specified file number. The disk buffer is then released for a subsequent open (see OPEN). Any future references to the specified file number are ignored until another OPEN command re-assigns it. If a file-number is not specified, or if it is zero, all open files are closed.

EXAMPLES:

CLOSE

Close all open files.

CLOSE 0

Close all open files.

CLOSE 2

Close file number two.

NOTES:

COPY

Transfer the contents of one or more files to another file.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

```
COPY  outfile{/A}=infile1{,infile2{,...}}
```

OPERATION:

The contents of "infile1" are transferred to "outfile." If it does not already exist, "outfile" is created. If the /A switch is present, the input file is instead appended to the output file. All additional input files are appended to the current output file in the order in which they appear. Single drive users should also see FMOVE.

EXAMPLES:

COPY MYFILE.BAK=MYFILE

Transfer contents of MYFILE to MYFILE.BAK on the default drive unit. (see SETUNIT)

COPY D2:PART1/A=PART2

Append the file PART2 found on the default unit to file PART1 found on unit two.

COPY ALLPARTS=PART1,D3:PART2,PART3

Transfer the contents of PART1 to ALLPARTS, then append PART2 on unit three to the new file ALLPARTS, and finally, append PART3 to ALLPARTS.

NOTES:

In the event that an error occurs, the output file may be left open and should be closed using the CLOSE command.

DIR

Display list of files on disk.

STATUS:

User memory at PAD is untouched.
This command is interruptable.

COMMAND FORMATS:

DIR {filespec}
DIR unit

OPERATION:

The DIR command lists all files within the directory on the default drive unit unless an optional file specification or drive number is specified. If the optional filespec is specified, it must resolve to a legal filename or else an error will result. Likewise, if the optional unit specification is supplied, it must be a number from 1 to 4 inclusive. The DIR command also displays the current number of free sectors. Note that the size of a file displayed in the listing is not necessarily accurate unless that file is closed. However, the number of free sectors on disk is always accurate.

EXAMPLES:

DIR	list information about all files found on the default unit.
DIR MYFILE	list information on MYFILE found on the default unit.
DIR D2:*.4TH	list information on all files with the extension 4TH found on unit two.
DIR 3	list information about all files found on unit three.

NOTES:

If the DIR command is given within a file that is to be loaded, a filespec or unit must be specified. Thus the first example above must be: DIR *

DISKCOPYn

Duplicate an existing diskette.

STATUS:

PAD is modified.

COMMAND FORMAT:

DISKCOPY1	single-drive copy
DISKCOPY2	multi-drive copy

OPERATION:

DISKCOPY1 is for users with only one disk drive. DISKCOPY2 is for users with two or more drives.

EXAMPLES:

DISKCOPY1

Copy a disk using only drive one. The user is prompted to insert the source diskette, and then the destination diskette. This is repeated until the entire source diskette is duplicated.

DISKCOPY2

The diskette in unit one is copied to the diskette in unit two.

NOTES:

EDIT

Edit a (FORTH) source file.

STATUS:

PAD is modified.

COMMAND FORMAT:

EDIT infile{,outfile}

OPERATION:

The input file is read into memory beginning at PAD where the editor is used to modify it. Upon leaving the editor, the modified file is written to the output file, if specified, otherwise it is written back to the input file. See the valDOS File Editor documentation in section LXIV for further information.

EXAMPLES:

EDIT MYFILE

Edit the file MYFILE on the default drive unit and write the modified version back to MYFILE.

EDIT D1:PACMAN.4TH,D2:PACMAN.4TH

Edit the file PACMAN.4TH on unit one and write the resultant file into file PACMAN.4TH on unit two.

NOTES:

ENDFIL

Endfile at current file cursor position.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

ENDFIL filenum

OPERATION:

The current file cursor position within the specified file is marked as the new end of that file. All data after that point in the file is lost and any disk space used by lost data is reclaimed.

EXAMPLE:

ENDFIL 2

End-file file number two.

NOTES:

Care should be taken if this command is used on a file that is open under more than one file number.

EOF

Move the file cursor to the end of the file.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

EOF filenum

OPERATION:

The cursor of the specified file is repositioned at the end of the file. The file must already be open.

EXAMPLE:

EOF 1

Position the cursor of file one at the end of that file.

NOTES:

ENTER

Enter (create) a filename in a disk directory.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

ENTER filename

OPERATION:

The specified filename is entered into the indicated directory. The filename must not already exist or an error will result. The file is created, but is not opened. Although this command is usually called CREATE, that word already exists in FORTH.

EXAMPLES:

ENTER MYFILE

Enter the filename MYFILE into the directory on the default drive unit.

ENTER 02:INVADERS.4TH

Enter the filename INVADERS.4TH into the directory on unit two.

NOTES:

FDUMP

Perform a hex/ASCII dump of a file.

STATUS:

User memory at PAD is untouched.
This command is interruptable.

COMMAND FORMAT:

FDUMP filename{/W}

OPERATION:

The specified file is displayed as a sequence of hex numbers and ASCII equivalents. This is typically used for looking at machine language programs stored on disk. Normal output is 8 bytes per line, however, if the /W (for "wide") switch is present, 16 bytes per line are displayed, which is more suitable for printed output.

EXAMPLES:

FDUMP MYFILE.OBJ

Dump the file MYFILE.OBJ to the current output device. Eight bytes/line are displayed.

FDUMP MYFILE.OBJ/W

Dump the file MYFILE.OBJ to the current output device. 16 bytes/line are displayed.

NOTES:

FILE-IT

Transform FORTH screen format to DOS file format

STATUS:

PAD is used.

COMMAND FORMAT:

```
FILE-IT  scr1,scr2,filename
```

OPERATION:

The screens from scr1 to scr2, inclusive, are read into free memory. The DOS disk is then swapped into the drive and the screens are written to the specified filename.

EXAMPLES:

```
FILE-IT  50,60,MYCODE.4TH
```

Screens 50 through 60 are read into free memory. The user is then prompted to insert the DOS-format disk into the drive. The data is next written to the file MYCODE.4TH.

NOTES:

FLOAD

Compile a FORTH source file from disk.

STATUS:

PAD is moved by compilation as usual.
This command is interruptable.

COMMAND FORMAT:

```
FLOAD  filename{/C}
FLOAD  filename{,linenum}
```

OPERATION:

The FLOAD command sends the FORTH source code contained in the specified file to the valFORTH compiler. If the /C (for "continue") switch is given, loading begins at the beginning of the last line edited in the file editor. This allows load errors to be fixed and compilation to be continued mid-file. If the optional line number is present, loading proceeds from that line of the file.

EXAMPLES:

FLOAD MYGAME.4TH

Load the DOS file MYGAME.4TH from the default drive unit and compile it.

FLOAD D2:CYCLOPS/C

Load the DOS file CYCLOPS from unit two starting with the last line edited in the file editor.

FLOAD FORTRAN.4TH,50

Load the DOS file FORTRAN.4TH from the default unit beginning with the 50th line in the file.

NOTES:

It is possible to LOAD a screen from a file being FLOADed and vice versa. Usually, LOADing and FLOADing should done from different units. Also see ECHO command.

FMOVE

Single drive interdisk file transfer

STATUS:

Memory at PAD is used.

COMMAND FORMAT:

FMOVE outfile=infile

OPERATION:

The FMOVE command is for those users who have access to only one disk drive. It is used to transfer a DOS file from one disk to another. If infile is too large to fit in available free memory, multiple disk swaps must be made. FMOVE will prompt for all necessary inputs, and swaps.

EXAMPLES:

FMOVE MYFILE.BAK=MYFILE

Transfer contents of MYFILE to MYFILE.BAK. MYFILE is read into free memory, source and destination disks are then swapped, and the data stored in free memory is written to MYFILE.BAK.

NOTES:
Multi-drive users can use the COPY command for this purpose.

FORMAT

Format a diskette for use.

STATUS:

User memory between PAD and BUFBOT is untouched.

COMMAND FORMAT:

FORMAT {unit}

OPERATION:

The FORMAT command initializes a disk for use with valDOS. If a unit number is supplied, formatting will be attempted on that unit, otherwise the default unit is assumed. The command will issue verification prompts and will also allow sectors to be locked so that no file will ever occupy those sectors. This feature allows mixing FORTH screens (virtual memory) and files on a single disk. The valDOS II disk is an example of this. This command also allows the newly formatted disk to be named.

EXAMPLES:

FORMAT

Format the default drive unit.

FORMAT 2

Format unit two.

NOTES:

This command replaces the one found on the valFORTH 1.1 disk. See NAMEDISK.

FSPACE

Move the file cursor relative to its current position.

STATUS:

User memory between PAD and BUFBOT is untouched.

COMMAND FORMAT:

```
FSPACE  filenum,count
```

OPERATION:

The cursor of the specified file is moved by the signed offset specified by count. The file cursor may be moved up to 32767 bytes in either direction, with a negative count spacing toward the beginning of the file. If the count is zero, the file cursor is left untouched.

EXAMPLES:

FSPACE 1,123

Move the cursor of file one 123 bytes toward the end of the file.

FSPACE 2,-345

Move the cursor of file two 345 bytes toward the start of the file.

NOTES:

Spacing backward is generally much slower than spacing forward due to the manner in which data is stored on disk. For this reason, backward spacing should be avoided.

KILL

Remove files and release disk space.

STATUS:

User memory between PAD and BUFBOT is untouched.

COMMAND FORMAT:

```
KILL  filespec1{/N}{,filespec2{/N}{,...}}
```

OPERATION:

The KILL command removes the specified files from the specified unit. If no unit is given in the file specification, the default unit is assumed. If the filespec is unambiguous (i.e., no wild cards), no verify prompt is issued. If the filespec is ambiguous, a verify prompt is issued for every file about to be deleted unless the /N switch is present.

EXAMPLES:

KILL THIS

Delete file THIS on the default drive unit.

KILL PART1,PART2,PART3,D3:PART?

Remove files PART1, PART2, and PART3 from the default unit, and all files found on unit three with five letter names that begin with PART.

KILL *.BAS,*.BAK/N,D2:MYFILE,TEST.*/N

Remove all files on the default unit with the extension BAS giving verify prompts. Remove all files with the extension BAK without verify prompts. Delete MYFILE on unit two, and all files named TEST on the default unit.

NOTES:

LOCK

Write and modify-protect files on disk

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

```
LOCK  filespec1{/N}{,filespec2{/N}{,...}}
```

OPERATION:

The LOCK command protects the specified files from being modified in any way. If the filespec is unambiguous (i.e., no wild cards), no verify prompt is issued. If the filespec is ambiguous, a verify prompt is issued for every file about to be locked unless the /N switch is present.

EXAMPLES:

LOCK D2:*

Lock all files on unit two, with prompts.

LOCK FILE?/N,PART1,PART2

Lock all files with five letter names that start with FILE, without prompts. Lock PART1 and PART2.

LOCK *.4TH,D2:MYFILE,PROG*/N

Lock all files with the extension 4TH giving verify prompts. Lock MYFILE on unit two, and all files beginning with PROG on the default unit, without prompts.

NOTES:

NAMEDISK

Name a diskette.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

NAMEDISK {unit}

OPERATION:

The valDOS file system allows disks to named for identification. Currently, this name is displayed only in directory listings, but is available for user programs. The NAMEDISK command displays the name of the disk in the specified unit and prompts for the new name to be entered. Disknames may be up to 20 characters long, and any character may be included within that name.

EXAMPLE:

NAMEDISK 3

Rename the disk in unit three.

NOTES:

Disks named in valDOS will function properly in other DOS's for the Atari computers.

OPEN

Open a file for access.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

OPEN filename

OPERATION:

The OPEN command assigns a buffer area and a file access number to the specified file.

EXAMPLE:

OPEN TEST.4TH

Opens the file TEST.4TH on the default unit.

NOTES:

Files may be multiply open, but are logically different files as far as the DOS is concerned. If a file is opened more than once and an operation such as ENDFIL is given, it is possible that the other opens will contain data in their transfer buffers that technically no longer exists. Note also that disks should not be exchanged when there are files open on the disk.

OPEN?

List all files currently open.

STATUS:

User memory at PAD is untouched.

OPERATION:

All files currently open are displayed along with their associated file access numbers.

EXAMPLE:

```
OPEN?
```

NOTES:

PRINT

Display a text file on the current output device.

STATUS:

User memory at PAD is untouched.
This command is interruptable.

COMMAND FORMAT:

```
PRINT  filename{/N}{,linenum}
```

OPERATION:

The contents of the specified file are sent to the current output device. Each line of text is automatically numbered unless the /N switch is present. If the optional line number specification is supplied, printing will begin with that line of the file.

EXAMPLES:

```
PRINT  GALAXY.4TH
```

Print the file GALAXY.4TH with line numbers.

```
PRINT  EDITOR.4TH/N
```

Print the file EDITOR.4TH without line numbers.

```
PRINT  D2:MYFILE.TXT,56
```

Print the file MYFILE.TXT on unit two with line numbers, beginning with the 56th line of text in the file.

NOTES:

READ

Read a file into memory.

STATUS:

User memory between PAD and BUFBOT is untouched.

COMMAND FORMATS:

```
READ  filename,address{,count}
READ  filenum.address{,count}
```

OPERATION:

In the first form, the specified file is opened and the first "count" bytes are read into memory starting at the address specified. If the count is not specified, the entire file is read in. The file is left closed. In the second form of the command. the first "count" bytes of the already opened file are read into the specified address. The file is left open after the read is complete. Note that the address can be specified by a number or by a single word (such as PAD) which returns a number.

EXAMPLES:

READ DRIVER.OBJ.PAD

Read the entire file DRIVER.OBJ into the address specified by PAD. The file is then closed.

READ 1,40960,1000

Read the first 1000 bytes from the file specified by the file access number one into address 40960. The file is left open.

NOTES:

No check is made to see if the data is being read into memory occupied by the FORTH dictionary. DOS buffers, or video memory. It is up to the user to supply safe load addresses.

RENAME

Rename a file.

STATUS:

User memory at PAD is untouched.

COMMAND FORTMAT:

RENAME newname=oldname

OPERATION:

The specified file is given the specified new name. There must not alreadv exist a file with the same name as the specified new name or an error will result.

EXAMPLE:

RENAME MYFILE.4TH=MYFILE

The file MYFILE is renamed as MYFILE.4TH.

NOTES:

REWIND

Move the file cursor to the beginning of the file.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

REWIND filenum

OPERATION:

The cursor of the specified file is repositioned at the beginning of the file. The file must already be open.

EXAMPLE:

REWIND 1

Rewind file number one.

NOTES:

SETUNIT

Set the default drive unit.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

```
SETUNIT  unit
```

OPERATION:

Whenever a filename does not explicitly contain a drive specification, the new default drive unit will be assumed. Units are numbered from one to four.

EXAMPLE:

```
SETUNIT  2
```

Set the default unit to two.

NOTES:

UNLOCK

Unprotect a file so that it may be modified.

STATUS:

User memory at PAD is untouched.

COMMAND FORMAT:

```
UNLOCK   filespec1{/N}{,filespec2{/N}{,...}}
```

OPERATION:

The UNLOCK command removes protection from the specified files so that they may be killed or modified. If the filespec is unambiguous (i.e., no wild cards), no verify prompt is issued. If the filespec is ambiguous, a verify prompt is issued for every file about to be unlocked unless the /N switch is present.

EXAMPLES:

UNLOCK THIS

Unlock file THIS on the default drive unit.

UNLOCK PART1,PART2,PART3,D3:PART?

Unlock files PART1, PART2, and PART3 from the default unit, and all files found on unit three with five letter names that begin with PART.

UNLOCK *.BAS,*.BAK/N,D2:MYFILE,TEST.*/N

Unlock all files with the extension BAS giving verify prompts. Unlock all files with the extension BAK without verify prompts. Unlock MYFILE on unit two, and all files named TEST on the default unit.

NOTES:

WRITE

Write an area of memory to a file.

STATUS:

User memory at PAD is untouched.

COMMAND FORMATS:

```
WRITE  filename,address,count
WRITE  filenum,address,count
```

OPERATION:

In the first form, the first "count" bytes of memory at the specified address are written to the specified file. If the file does not already exist, it is created. The file is closed after the write. In the second form, the memory block is written to the already open file. The file is left open. The address can be specified by either a number or a single word (such as PAD) which returns a number.

EXAMPLES:

WRITE MYFILE,PAD,1000

Write the 1000 byte block of memory at the address specified by PAD to MYFILE.

WRITE 1,40960,256

Write the first 256 bytes of memory at address 40960 to the file associated with file access number one.

NOTES:

The valDOS System

The following set of words make up the heart of the valDOS system. Note that all of the following words are in a special vocabulary named DOS. Any word which uses one of these system operations must contain the word DOS in the definition, as it is necessary to inform the compiler where to look for these definitions. Since most of these commands are on the system level, a brief description of how the system works is appropriate.

Before any file can be manipulated, it must be opened for access. A file buffer and a status block are allocated for each open file. Additionally, each file has its own file cursor (which is positioned at the beginning of a file at open time). This cursor always points to a location within the file. If a byte is read from the file, the byte pointed to by this cursor is returned and the cursor is moved a byte deeper into the file. When a byte is written to the file, it replaces the byte pointed to by the file cursor, and the file cursor is then bumped.

Thus, if a 10K file is opened and five bytes are written to the file, the first five bytes will be changed, and the remainder of the file will be left untouched. If the file is then closed, the file will still be 10K long. Most BASICs, however, will "end-file" a file when it is closed, i.e., in this example, the file would be reduced to five bytes. By not implicitly end-filing a file when it is closed, greater flexibility is gained. Note that the valDOS commands (such as COPY and EDIT) do implicitly perform an end-file prior to closing a file.

The two most common operations are reading a file and writing a file. Usually when reading a file, the file is opened, read to the end (eof), and then closed. Generally when a file is written, it is opened, written, end-filed, and then closed.

valDOS System Words

DOS ---

This is the name of the vocabulary which contains all of the following DOS system words. DOS stands for Disk Operating System (not Software as some claim). Any word which contains one of the following words must contain this word within its definition: : myword DOS ... ; DOS is IMMEDIATE.

#ENTRIES --- n

A quan which contains the number of entries that matched the (wild) filename last checked by CHKDIR. CHKDIR returns this value automatically. This value can also be used to index the first "n" elements of the table pointed to by DIRTBL (below).

#FREE --- n

A quan which contains the number of free directory entries in the last directory scanned by CHKDIR.

#UNTRN --- n

A quan which contains the number of bytes left untransferred in the last block read/write operation. This value is only accurate immediately after the read/write operation returns control to the calling routine.

(?OPEN) $ --- {f 1}/0

This routine checks to see if the file specified by $ is open. If the operation is successful, a flag is returned along with the value one. If this flag is 0, the file is closed. If an error occurred, only a zero is returned. See DSKERR. This routine uses FNCON.

(OPEN) $ --- {addr fl# 1}/0

This routine opens the file specified by $. It is used to create the necessary data transfer path between the application and the DOS. If the open is successful, a file transfer buffer is allocated and is assigned a file access number. All subsequent operations upon the open file require that this file number be supplied. On a successful open, the file number and the transfer buffer address are returned. In most cases, the buffer address can simply be discarded, while the file number must be stored by the application. File numbers are always greater than zero. See DSKERR. This routine uses FNCON.

(CLOSE) fl# ---

The (CLOSE) operation closes the data transfer path associated with the specified file number. The file buffer is flushed to disk if updated, and is freed for a subsequent open operation. If "fl#" is zero, all open channels are closed from access. This routine generates no errors. Illegal file numbers are ignored.

(ENDF) fl# ---

Each open file has a file cursor which points to the next byte to be read or written. The "end file" operation marks the current byte as the end of the file. Thus if the file cursor points to the fifth byte of a 10K file, the current byte and all successive bytes are lost, and the disk space is reclaimed. This is typically used just before closing a file that has been written to. This ensures that no "stale" data remains. The command ENDFIL uses this routine.

(ENTER) $ --- f

The filename specified by $ is entered into the directory on the unit specified within the filename. If no unit is explicitly stated, the default unit (specified by DFLUNT) is assumed. A one is returned if the operation was successful, otherwise a zero is returned. See DSKERR. This routine uses FNCON. (This routine is usually called "create". That name is already used in FORTH, however.)

(KILL) $ -- f

The filename specified by $ is deleted from the directory on the unit specified within the filename. If no unit is explicitly stated, the default unit (specified by DFLUNT) is assumed. A one is returned if the operation was successful, otherwise a zero is returned. See DSKERR. This routine uses FNCON.

(LOCK) $ --- f

The filename specified by $ is locked so that it may not be written to, killed, or renamed. A one is returned if the operation was successful, otherwise a zero is returned. See DSKERR. This routine uses FNCON.

(RDB) fl# --- {b 1}/0

The "read byte" operation reads the next byte from the file whose access number is "fl#". If an error occurs a zero is returned, otherwise the byte along with a one is returned. If many bytes are to be read, (READ) should be used if possible as that routine is many times faster than (RDB). Note that if a read is attempted with the file cursor at the end of the file, an EOFERR error is generated. See DSKERR.

(READ) addr cnt fl# --- f

This operation reads the next "cnt" bytes of the file whose access number is "fl#" and stores them in memory beginning at "addr". If an error occurs a zero is returned, otherwise a one is returned. The quan #UNTRN contains the number of bytes left untransferred in the event of an error. See DSKERR.

(REN) $n $o --- f

This operation renames the file $o to $n. A one is returned if no error occurred, otherwise a zero is returned.

(SPACE) cnt fl# --- f

The space operation repositions the cursor of the file whose access number is "fl#" by the signed number "cnt". The value "cnt" must lie in the range of -32768 to 32767. Also note that the file cursor cannot be spaced past byte no. 65,535 of the file. If "cnt" is zero, the space operation is ignored. A one is returned if no error occurred, otherwise a zero is returned.

(UNLOCK) $ --- f

The filename specified by $ is unlocked so that it may be written to, killed, or renamed. A one is returned if the operation was successful. otherwise a zero is returned. See DSKERR. This routine uses FNCON.

(WIND) f fl# --- f

The (WIND) command is used to position the file cursor at beginning or end of the file. If "f" is one, the file cursor is rewound to the beginning of the file. This allows the file to be re-read. If "f" is zero, the file cursor is moved to the end of the file for subsequent writing, effecting an append operation. A one is returned if the operation was successful, otherwise a zero is returned. See DSKERR.

(WRB) b fl# --- f

The byte "b" is written to the file whose access number is "fl#". If many bytes are to be written to the file, (WRITE) below should be used instead, as it is many times faster than (WRB). A one is returned if the operation was successful, otherwise a zero is returned. See DSKERR.

(WRITE) addr cnt fl# --- f

The block of memory "cnt" bytes long beginning at memory location "addr" is written to the file whose access number is "fl#". A one is returned if the operation was successful, otherwise a zero is returned. See DSKERR.

?DOSERR f err# ---

This is one of three (see DOSERR, ?DSKERR) error routines available at the system level. If the flag "f" is zero, the system error "err#" is generated. Program control does not return to the word which contained ?DOSERR, but to the word which called this word. For example, if an application calls (OPEN) and a ?DOSERR within (OPEN) generates an error, ?DOSERR will not return to (OPEN), but to the application, passing a 0. If the ?DOSERR does not generate an error, program control will return to (OPEN). ?DOSERR stores the error value in the quan DSKERR. See the actual definition of (OPEN) for a good example of how this is used. ?DOSERR is essentially a SWAP 0= IF DOSERR ENDIF DROP

?DSKERR f ---

?DSKERR is used to propogate an error from one word to the next. It replaces the sequence DSKERR ?DOSERR. See ?DOSERR.

?WILD --- f

This is a quan which contains a one if the last filename converted by FNCON contains either of the wild card characters "?" or "*". A zero is returned if no wild cards appeared.

BUFBOT --- addr

This is a word which returns the lowest memory address used by the DOS file buffers.

CHKDIR --- n

The check-directory routine scans the directory on the unit specified within the last filename converted by FNCON for all occurrences of that filename. For every match found (multiple matches are due to wild cards) in the directory, a directory entry number is stored in the next available location of the memory block pointed to by DIRTBL. Thus if CHKDIR finds five occurrences of a filespec, the first five elements of DIRTBL contain the directory entry numbers for those five matches. These values can then be used in conjunction with the ENTRY command to access the files. CHKDIR returns the number of matches.

DFLUNT --- n

A quan which contains the number of the default unit. This drive number is assumed if no drive specification is contained within a filename converted by FNCON. Caution, this contains 0 if DOS drive 1 is the default, 1 if DOS drive 2 is the default, etc. (FORTH drive = DOS drive-1)

DIRFRE --- n

A quan which contains the entry number of the next free entry in the directory last scanned by the CHKDIR routine.

DIRTBL --- addr

A word which returns the starting address of a 64 byte memory block that contains directory entry numbers of all files that matched the last filename checked by the CHKDIR routine. See CHKDIR.

DIRUP ---

This marks the current directory as being updated so that it is written to disk upon the next DSKFLS command.

DOSERR err# --- 0

This is one of three (see ?DOSERR, ?DSKERR) error routines at the system level. DOSERR unconditionally generates the error whose error number is "err#". Program control continues two levels up (instead of the usual one) and a zero is returned. See ?DOSERR.

DSKERR --- n

A quan which contains the error number of the last DOS error that occurred. See List of Errors below.

DSKFLS ---

A routine which flushes the current directory and free space map if updated. All user defined commands should end with this command. See source listing for pre-defined commands.

ENTRY unit n --- addr

The entry command returns the address of the "nth" entry in the directory on the specified drive unit. The 16 byte entry has the following format:

```
addr+0:    status byte
           bit 7:  File deleted if set
               6:  File entry valid if set
               5:  File locked if set
               4:  File random if set
               3:  File in use if set (not used)
               2:  Unused
               1:  DOS format 2 if set
               0:  File open for output if set
addr+1:    length of file in sectors
addr+3:    first (DOS) sector of file
addr+5:    8 letter filename,
           left justified. blank filled
addr+13:   3 letter extension.
           left justified. blank filled
```

If any changes are made, executing DIRUP and DSKFLS will write those changes to disk.

FLBUF@ fl# --- addr

Whenever a file is opened, a 128 byte transfer buffer and a 16 byte status block are allocated. The FLBUF@ command returns the address of the file status block associated with "fl#". The 16 byte table contains the following information:

```
addr+0:     File status byte (see ENTRY).
addr+1:     Current size of file in sectors.
            If high bit is set, file is updated.
addr+3:     First (DOS) sector of file.
addr+5:     (FORTH) sector currently in file buffer.
addr+7:     Number of bytes into current sector.
addr+8:     Unit associated with the file.
addr+9:     Entry number in the directory.
addr+10:    Non-zero = current sector is updated.
addr+11:    (reserved for) Current random block.
addr+13:    (reserved for) Random block update flag.
addr+14:    Number of bytes into file (unsigned).
addr-128:   Address of 128 byte file buffer.
```

FLFLS fl# ---

This operation flushes the file buffer associated with the file access number "fl#".

FNCON $ --- f

The FNCON command takes the file specification $ and converts it to directory format (i.e., left justified and blank filled). It stores the 11 byte formatted filename at the memory location pointed to by FNFLD. All wild cards are converted to question marks, thus, "MY?FIL*" will become "MY?FIL?????". Additionally, if a drive specification is contained in the filespec, it is determined and stored in the quan UNIT. If no drive specification was supplied, the default unit (in DFLUNT) is stored in UNIT. FNCON will also parse out a single switch ("/s") where the character "s" is stored in the quan FNSWCH. If no switch is found, a zero is stored in FNSWCH. If any wild cards appear in the file specification, the quan ?WILD is set to one, otherwise zero is stored. If no errors were detected, a one is returned, otherwise zero.

FNFLD --- addr

A pointer to an eleven byte storage area which contains the filename last formatted by FNCON.

FNSWCH --- s

A quan which contains the ASCII value of the switch in the last filename converted by FNCON. If no switch was present, FNSWCH contains zero. See FNCON.

FSMAP --- addr

A word which returns the address of the current free space map.

FSMUP ---

The FSMUP command marks the current free space map as being updated.

MAXFL ---

A constant which contains the maximum number of files that can be open at any given time. This has a default value of four. If this constant is changed, valDOS must be completely reloaded.

TADR --- n

After a block read/write operation, this quan contains an address one byte higher in memory than that of the last byte transferred. This can be used to determine how much data was transferred in the event of an error.

UNIT --- n

A quan which contains the unit specification of the last filename converted by FNCON. Note that this is a FORTH unit (i.e., FORTH unit = DOS unit-1).

WRKSPC --- addr

A quan which points to a 128 byte scratch area used by many of the system words described above. This is free for user applications between valDOS system calls.

valDOS Command Support Words

?CMDERR n err# ---

Like ?DOSERR, but prints the error message, clears both stacks, and aborts program execution through QUIT, if the status flag "n" is zero. If "n" is true, ?CMDERR returns to the calling word.

?SYSERR n ---

Like ?DSKERR, but aborts through ?CMDERR if "n" is zero. If "n" is non-zero, ?SYSERR returns to the calling word.

?WRGARG n ---

This is an abbreviation for WRGARG ?CMDERR .

CONFN addr unit --- $

The CONFN routine takes the directory formatted filename at "addr" (usually FNFLD) and converts it to a string. The drive specification "unit" is attached to the beginning of the filename (i.e., unit = 0 would generate "D1:").

CMDERR err# ---

Like DOSERR, CMDERR unconditionally generates the error specified by "err#".

ECHO ON/OFF ---

When echo is ON, all files being FLOADed will be echoed to the current output device(s). When echo is OFF, no output is generated.

FWORD c ---

Read the next text characters from the input file whose file number is in the quan FLFL# (FLOAD initializes FLFL#) until a delimiter "c" is found, storing the packed character string beginning at the dictionary buffer HERE. FWORD leaves the character count in the first byte, followed by the characters, and ends with two or more blanks. Leading occurrences of "c" are ignored. Note that "c" may not be the return character (ATASCII 155).

GETVAL $ --- n

It $ is the name of a dictionary word, the word is executed and had better return a single value. If $ is not found in the dictionary, it is assumed to be a number and is converted leaving the number n. GETVAL is used so that addresses may be stated explicitly, or by reference (PAD, HERE, etc.).

GETARGS   --- {$ 1}/0

This routine converts the next non-blank set of characters in the input stream to a string. If a set of non-blank characters is found, it is returned along with a one. If no set is found, a zero is returned.

GETARG $ c --- {$1 $2 1}/{$ 0}

The string $ is divided into two parts, broken at the first occurrence of the character "c". If the character "c" is found in $, the leftmost portion of $ is returned 3rd on stack, the rightmost 2nd on stack, and 1 on top. If "c" is not in the string, the original string along with 0 is returned.

System Errors (all are CONSTANTs)

AMBNME --- n

Ambiguous filename. This error is generally issued when a filespec containing wild cards is passed to one of the primitive file operators like (OPEN) or (KILL).

BADFL# --- n

Bad file number. The error is generated when a file number does not lie in the range 1 ≤ fl# ≤ MAXFL (default of four).

BADFSM --- n

Bad free space map. This is issued when the number of free sectors does not match the true number of free sectors. If this error is reported, a new disk should be made and all files should be transfered to this new disk using COPY or FMOVE.

BADNME --- n

Bad filename. The filename passed to the system routine contained an illegal character, or was too long. File names can only contain the letters "A" through "Z" and the digits "0" through "9". Note that the first character of a filename must be a letter. The two wild card characters "?" and "*" are also allowed.

BADUNT --- n

Bad drive specification. A drive number was encountered that did not lie in the range 1 ≤ unit ≤ 4.

DIRFUL --- n

Directory is full. There is no more room in the directory. Kill some unwanted files and try the operation again.

DSKFUL --- n

Disk is full. There are no more free sectors on the disk. Kill some unwanted files and try the operation again.

EOFERR --- n

End of file has been reached. This error generally results from an attempt to read data past the end of file mark.

FLDNE --- n

File does not exist.

FLEXST --- n

File already exists.

FLNOPN --- n

File is not open. Use the OPEN command and open the file. This error should onlv occur when using the file primitives like (SPACE) and (WIND).

FLOPN --- n

File is open. Use the CLOSE command to close the appropriate file. See OPEN?

FLTBG --- n

File is too big. This error is usually reported by the file editor when there is not enough free memory to edit the file. Break the file into two parts, or use the FORGET command to free some memory.

FLWPRT --- n

file is write protected (locked). Use the UNLOCK command and perform the last operation again.

TMFOPN --- n

Too manv files open. For each file open, a file buffer and a file status block is allocated. There are a limited number of available buffers (determined by the constant MAXFL). If all buffers are being used and an open operation is attempted, this error will be generated. Use the CLOSE command to free a buffer.

WRGARG --- n

Bad/no argument list. This is generated when a DOS command expects a list of arguments and none is supplied. This is also generated when the wrong number or type of arguments is supplied.

A note on QUAN structures

The "quan" is a new FORTH data structure, developed at Valpar, and is used in this package. Quans were devised to cut down on wasted memory and runtime encountered when using the "variable" data structure. Quans work as follows: (Advanced users may want to follow along in the source code for these structures also.

Defining a quan:
QUAN BINGO

Note that quans do not take initial values. This form was chosen to allow for simpler upgrading to target-compiled code later on.

Giving a quan a value:
1234 TO BINGO

Note that since TO is immediate, "TO BINGO" compiles to only 2 bytes instead of the 4 bytes that would be required if BINGO were a variable (i.e., BINGO !).

Getting the value back from a quan:
BINGO

Simply saying the name of the quan will leave its value on the stack, in this case 1234. In this way, quans act like constants. BINGO above also compiles to only 2 bytes instead of the 4 bytes required to fetch it if it were a variable (i.e., BINGO @).

Getting the address of the data in the quan:
AT BINGO

This will leave the address of the first byte of data in BINGO on the stack, or compile the address as a literal if encountered during compilation. (AT is immediate.) This is useful for a variety of purposes in general programming and in interfacing to machine language routines.

Advanced users:

The FORTH 83 Standard appears to lean toward "non-state-smart" words, which is proper for target-compiled applications. We expect to support both "state-smart" and "non-state-smart" versions of various words, as appropriate for different users.

Note that while

```
15 AT BINGO +!     and     15 BINGO + TO BINGO
```

accomplish the same task and take the same amount of memory, the first version is faster by one primitive nest.

The most significant internal feature of a quan is that it has 3 cfa's instead of just the one common to most FORTH words.

This initial four byte disadvantage is overcome at the second use of a quan, and so poses essentially no problem. When a quan is not preceded by "TO" or "AT", the first cfa (quan@) is compiled in. If the quan is preceded by "TO", the second cfa (quan!) is compiled in. And if the quan is preceded by "AT". the third cfa ('quan) is compiled into the dictionary.

valDOS File Editor

Version 1.0
Oct. 1982

The FORTH language is a very powerful addition to the Atari home computer. Programs which are impossible to write in BASIC (usually because of limitations in speed and flexibility) can almost always be written in FORTH. Even when one has mastered the BASIC language, making corrections or additions to programs can be tedious. The video editor described here removes this problem from the FORTH environment. Similar to the MEMO PAD function in the Atari operating system, this editor makes it possible to insert and delete entire lines of code, insert and delete single characters, toggle between insert and replace modes, move entire blocks of text, and much more.

Overview

This editor is a powerful extension to the valFORTH system designed specifically for the Atari 400/800 series of microcomputers. The main purpose for this editor is to give the FORTH programmer an easy method of text entry to DOS file for subsequent compilation. For those already familiar with the valFORTH 1.1 screen editor, this editor is very similar in function. In fact, all of the commands found in that editor (except ctrl-A) are supported in the file editor. More importantly, many additional capabilities have been added to this editor. They are:

1) Tab stops can be set/reset
2) Splice (unsplit) is now supported
3) Global pattern searches
4) A repeat function which repeats the next command/key typed until a console key is pressed
5) File merge (i.e., reading a file into a file)
6) True single-/mult- line scrolling either forward or backward
7) Input and output files may be different

The editor has four basic modes of operation:

1) It allows entering of new text into a file as though typing on a regular typwriter.

2) It allows simple modification of any text with a powerful set of single stroke editing commands.

3) It pinpoints exactly where a compilation error has occurred and sets up the editor for immediate correction and recompilation.

The set of single stroke editing commands is a superset of the functions found the in MEMO PAD function of the standard Atari operating system. In addition to cursor movement, single character insertion/deletion, and line insertion/deletion, the editor supports a clear-to-end-of-line function, a split command which separates a single line into two lines, its complement splice (unsplit), global searches, and many other features usually found only in higher quality word processors, and almost never in file editors.

Also provided is a visible edit storage buffer which allows the user to move, replace, and insert up to 320 lines of text at a time. This feature alone allows the FORTH programmer to easily reorganize source code with the added benefit of knowing that re-typing mistakes are avoided. Usage has shown that once edit-buffer management is learned, significant typing and programming time can be saved.

For those times when not programming, the editor can double as a simple word processor for writing letters and filling other documentation needs. Perhaps the best method for learning how to use this powerful editor is to enter the edit mode and try each of the following commands as they are encountered in the reading.

NOTE:

This editor can be used to enter assembly language source, Pascal, or any other text oriented data. The only limitation upon this is that no lines may be longer than 38 characters in length. Additionally, this editor can edit files created from other sources; however, only the first 38 characters of a line will be retained.

Loading and Entering the Editor

To load the editor, first load valDOS as described in "Strolling Through valDOS". Next, insert the valDOS II disk and list screen 170. This should tell you exactly which screen to load. The edit mode is initiated using the EDIT command. This command has the following format:

EDIT infile{,outfile}

The DOS file "infile" is loaded for editing. When all changes have been made and the file is saved, the modified text is written to the file "outfile", if supplied, otherwise it is written back to "infile".

Insert a copy of your valDOS II disk and type:

EDIT FILEIT.4TH,MYFILE

The editor will display some information which can be ignored for the time being, and then it will wait for the return key to be typed. After pressing the return key, the display should look like fig. 1.

The top window, composed of a single line, indicates which file is currently being edited. Also shown is the size of the edit buffer (decribed later). In this example, the buffer is five lines in length. This window is known as the heading window.

The second window (the text window) contains 16 lines of text within the specified file. This window is 38 characters wide and 16 lines high. The white cursor (indicated by the symbol "█") will be in the upper-lefthand corner of the display awaiting editing commands.

The final five-line window found at the bottom of the display is known as the buffer window. This is used for advanced editing and is described in greater detail in the section entitled "Buffer Management."

```
.------------------------------------------.
|    File: MYFILE             #Bufs: 5     |
'------------------------------------------'

.------------------------------------------.
|  @                                        |
|  ( Routine:  FILE-IT                      |
|                                           |
|    The following routine will             |
|    transfer a specified range of          |
|    FORTH screens to a file on             |
|    a DOS formatted disk.                  |
|                                           |
|    Format:  FILE-IT 1st,last,filename     |
|             FILE-IT 10,20.MYFILE          |
|                                           |
|    Note that DOS commands tend to be      |
|    long because of error checking and     |
|    parameter parsing.                  )  |
|                                           |
|                                           |
'------------------------------------------'

.------------------------------------------.
|                                           |
|                                           |
|                                           |
|                                           |
|                                           |
'------------------------------------------'
```

Fig. 1

LL   re-edit last file   (---)

This command is used to re-edit the "Last" file edited. It functions identically to the EDIT command, except that no file names need to be specified.

Example: LL <ret>   (re-edit MYFILE)

WHERE   find location of error   (---)

If, when compiling code, a compilation error occurs, the WHERE command will enter the edit mode and position the cursor over the last letter of the offending word. The word can then be fixed and the file saved for subsequent compilation using the "FLOAD filename/C" command.

#BUFS   set buffer length   (#lines ---)

The #BUFS command allows the user to specify the length (in terms of number of lines) of the special edit storage buffer. The power of the edit buffer lies in the number of lines that can be stored in it. Although the default value is five, practice shows that at least 16 lines should be set aside for this buffer. The maximum number of lines allowable is 320 which is enough to hold 20 full screens simultaneously.

.INFO   **display file information**   (---)

If an error occurs and an edited file is not saved to disk, the .INFO command will supply all the necessary information to save the file using the WRITE command.

The following sections give a detailed description of all commands which the video editor recognizes. A quick reference command list can be found following these descriptions.

Cursor Movement

When the edit mode is first entered via the EDIT command, a cursor is placed in the upper lefthand corner of the screen. It should appear as a white block and may enclose a black letter. Whenever any key is typed and it is not recognized as an editor command, it is placed in the text window where the cursor appears. Likewise, any line functions (such as delete line) work on the line where the cursor is found.

ctrl ↑ , ctrl ↓ , ctrl ← , ctrl → — move-cursor commands

To change the current edit line or character, one of four commands may be given. These are known as cursor commands. They are the four keys with arrows on them. These keys move the cursor in the direction specified by the arrow on the particular key pressed. There are times, however, when this is not the case.

Similarly, if the cursor is positioned on the leftmost edge and the "cursor-left" command is given, the cursor will "wrap" to the rightmost character. Issuing "cursor-right" will wrap to the left edge.

RETURN — next-line command

The RETURN key positions the cursor on the first character of the next line. If RETURN is pressed when the cursor is on the last line of the file, a line is inserted at the end of the file.

TAB — tabulate command

The TAB key is used to tabulate to the next TAB stop to the right of the current cursor character.

ctrl TAB clear TAB stop

Clear the tab stop at the current cursor location. The default tab stops can be reset by issuing the RT subcommand.

shift TAB set TAB stop

Set a tab stop at the current cursor location. The default tab stops can be reset by issuing the RT subcommand.

ctrl L continue search

Search for the next occurrence of the pattern set up using the PS subcommand. Patterns can be up to 30 characters in length.

ctrl V enter subcommand mode

The puts the editor into the subcommand mode. See the section entitled "Subcommands" for a list of available commands.

NOTE:

Many commands in the editor will "mark" the file as updated so that any changes made can be preserved on disk. As simple cursor movement does not change the text window in any way, these commands never mark the file.

Editing Commands

Editing commands are those commands which modify the text in some predefined manner and mark the file as updated for later saving.

ctrl INS — character insert command

When the "insert-character" command is given, a blank character is inserted at the current cursor location. The current character and all characters to the right are pushed to the right by one character position. The last character of the line "falls off" the end and is lost. The inserted blank then becomes the current cursor character. This is the logical complement to the "delete-character" command described below.

ctrl DEL — delete character command

When the "delete-character" command is issued, the current cursor character is removed, and all characters to the right of the current cursor character are moved left one position, thus giving a "sqeeze" effect. This is normally called "closing" a line. The rightmost character on the line (which was vacated) is replaced with a blank. This serves as the logical complement to the "insert-command" described above.

shift INS — line insert command

The "line-insert" command inserts a blank line between the current cursor line and the line immediately above it. If this command is accidentally typed, the "oops" command (ctrl-O) described later can be used to recover from the mistake. Also see the "from buffer" command described in the section on buffer management for a similar command. This command serves as the logical complement to the "line-delete" command described below.

shift DEL — line delete command

The "line-delete" command deletes the current cursor line. If this command is accidentally issued, recovery can be made by issuing the "oops" command (ctrl-O) described later. Also see the "to-buffer" command described in the section on buffer management for a similar command. The "delete-line" command serves as the logical complement to the "line-insert" command.

ctrl H — erase to end of line

The "Hack" command performs a clear-to-end-of-line function. The current cursor character and all characters to the right of it on the current line are blank filled. All characters blanked are lost. The "oops" command described later can be used to recover from an accidentally hacked line.

ctrl I — insert/replace toggle

In normal operation, any key typed which is not recognized by the editor as a control command will replace the current cursor character with itself. This is the standard replace mode. Normally, if one wanted to insert a character at the current cursor location, the insert character command would have to be issued before any text could be entered. If inserting many characters, this is cumbersome.

When active, the insert submode automatically makes room for any new characters or words and frees the user from having to worry about this. When the editor is called up via the EDIT command, the insert mode is deactivated. Issuing the insert toggle command will activate it and the cursor will blink, indicating that the insert mode is on. Issuing the command a second time will deactivate the insert mode and restore the editor to the replace mode. Note that while in the insert mode, all edit commands (except BACKS, below) function as before.

BACKS delete previous character

The BACKS key behaves in two different ways, depending upon whether the editor is in the insert mode or in the replace mode. When issued while in the replace mode, the cursor is backed up one position and the new current character is replaced with a blank. If the cursor is at the beginning of the line, the cursor does not move, but the cursor character is still replaced with a blank.

If the editor is in the insert mode, the cursor backs up one position, then deletes the new current cursor character and then closes the line. If the cursor is at the beginning of the line, the cursor remains in the same position, the cursor character is deleted and the line closed.

NOTE:

As all of the above commands modify the file in some manner, the file is marked as having been changed. This is to ensure that all changes made are eventually saved on disk. The "quit" command described later allows one to abort the edit session so that major mistakes need not be saved.

Buffer Management

Much of the utility of the file editor lies in its ability to temporarily save text in a visible buffer. To aid the user, it is possible to temporarily send text to the buffer and to later retrieve it. This storage buffer can hold as many as 320 lines of text simultaneously. This buffer is viewed through a 5 line "peephole" visible as the last window on the screen. Using this buffer, it is possible to duplicate, move, and easily reorganize text, in addition to temporarily saving a line that is about to be edited so that the original form can be viewed or restored if necessary. The following section will explain exactly how to accomplish each of these actions.

ctrl T — to buffer command

The "to-buffer" command deletes the current cursor line, but unlike the "delete-line" command where the line is lost, this command moves the "peephole" down and copies the line to the bottom line of the visible buffer window. This line is the current buffer line. The buffer is rolled upon each occurrence of this command so that it may be used repeatedly without the loss of stored text.

For example, if the cursor is positioned on ninth line of the display shown in figure 1 and the "to-buffer" command is issued twice, the final result will be as shown in figure 2.

ctrl F — from buffer command

The "from-buffer" command does exactly the opposite of the "to-buffer" command described above. It takes the current buffer line and inserts it between the current cursor line and the line above it. The cursor line and all lines below it are moved down one line. If the cursor were placed on line 14 of the above screen display and the "from-buffer" command were issued twice, the display shown in figure 3 would result.

```
         .-----------------------------------------.
         :    File: MYFILE              #Bufs: 5    :
         '-----------------------------------------'

         .-----------------------------------------.
         :                                          :
         :  ( Routine:  FILE-IT                     :
         :                                          :
         :    The following routine will            :
         :    transfer a specified range of         :
         :    FORTH screens to a file on            :
         :    a DOS formatted disk.                 :
         :                                          :
current  :  @                                       :
         :    Note that DOS commands tend to be     :
         :    long because of error checking and    :
         :    parameter parsing.                  ) :
         :                                          :
         :                                          :
         :  : FILE-IT DOS              ( -- )       :
         :    GETARGS ?WRGARG 44 GETARG ?WRGARG     :
         '-----------------------------------------'

         .-----------------------------------------.
         :                                          :
         :                                          :
         :                                          :
         :                                          :
         :    Format:  FILE-IT 1st.last,filename    :
current  :             FILE-IT 10,20,MYFILE         :
         '-----------------------------------------'
```

fig. 2

```
.-----------------------------------------.
|    File: MYFILE            #Bufs: 5     |
'-----------------------------------------'

.-----------------------------------------.
|                                         |
|  ( Routine:  FILE-IT                    |
|                                         |
|    The following routine will           |
|    transfer a specified range of        |
|    FORTH screens to a file on           |
|    a DOS formatted disk.                |
|                                         |
|                                         |
|    Note that DOS commands tend to be    |
|    long because of error checking and   |
|    parameter parsing.                  )|
|                                         |
|  @ Format:  FILE-IT 1st,last,filename   |
|             FILE-IT 10.20,MYFILE        |
|  : FILE-IT DOS               ( -- )     |
'-----------------------------------------'

.-----------------------------------------.
|    Format:  FILE-IT 1st,last,filename   |
|             FILE-IT 10.20,MYFILE        |
|                                         |
|                                         |
|                                         |
'-----------------------------------------'
```

fig. 3

ctrl K — copy to buffer command

The "copy-to-buffer" command takes the current cursor line and duplicates it, sending the copy to the buffer. The cursor is then moved down one line. This commands functions identically to the "to-buffer" command described above, except that the current cursor line is NOT deleted from the text window.

ctrl U — copy from buffer

The "copy-from-buffer" command replaces the current cursor line with the current buffer line. This command functions identically to the "from-buffer" command described above, except that the buffer line is not inserted into the text window, it merely replaces the current cursor line. The "oops" command described below can be used to recover from accidental usage of this command.

ctrl R roll buffer

The "roll-buffer" command moves the buffer "peephole" down one line and redisplays the visible window. If the buffer were the minimum five lines in length, the bottom four lines in the window would move up a line and the top line would "wrap" to the bottom and become the current buffer line. If there were more than five buffer lines, the bottom four lines would move up a line, the topmost line would be pushed up behind the peephole, and a new buffer line coming up from below the peephole would be displayed and made current. For example, if the buffer were five lines long and contained:

```
          .--------------------------------.
          | ( Who?   )                     |
          | ( What?  )                     |
          | ( When?  )                     |
          | ( Where? )                     |
Current:  | ( Why?   )                     |
          '--------------------------------'
```

fig. 5

the "roll-buffer" command gives:

```
          .--------------------------------.
          | ( What?  )                     |
          | ( When?  )                     |
          | ( Where? )                     |
          | ( Why?   )                     |
Current:  | ( Who?   )                     |
          '--------------------------------'
```

fig. 6

ctrl B back-roll-buffer command

The "back-roll-buffer" does exactly the opposite of the "roll-buffer" command described above. For example, if given the buffer in figure 6 above, the "back-roll" command would give the buffer shown in figure 5.

ctrl C clear buffer line command

The "clear-buffer-line" command clears the current buffer line and then "back-rolls" the buffer so that successive clears can be used to erase the entire buffer.

NOTE:

Any of the above commands which change the text window will mark the file as updated. Those commands which alter only the buffer window (such as the "roll" command) will not change the status of the current screen.

Scrolling

ctrl X previous line command

The "previous-line" command scrolls the text window up a line within the file.

ctrl E next line command

The "next-line" command moves the text window down a line within the file.

ctrl P previous page command

The "previous-page" command scrolls the text window up 16 lines within the file.

ctrl N next page command

The "next-page" command moves the text window down 16 lines within the file.

ctrl S save command

The "save" command saves any changes made to the current file and exits the edit mode.

ctrl Q quit command

The "quit" command aborts the edit session "forgetting" any changes made to the text file in memory. The "quit" command is usually used when either the wrong file has been called up, or if it becomes desirable to start over and re-edit.

Special Commands

There are four special commands in this editor which allow greater flexibility in programming on the valFORTH system:

ESCAPE special key command

The "special-key" command instructs the video editor to ignore any command function of the next key typed and force a character to the screen. For example, normally when "ctrl →" is typed, the cursor is moved right. By typing "ESCAPE ctrl →" the cursor is not moved -- rather, the right-arrow is displayed.

ctrl J split line command

Often times, for formatting reasons, it is necessary to "split" a line into two lines. The split line command takes all characters to the left of the cursor and creates the first line, and with the remaining characters of the original line, a second line is created. Graphically, this looks like:

```
before:   | The quick@brown fox jumped.    |


after:    | The quick@                     |
          |           brown fox jumped.    |
```

ctrl G splice (unsplit) command

The "splice-command" performs just the opposite operation of the "split-command" above.

ctrl Y repeat command

The "repeat-command" repeats the next command or character typed until a predefined stop condition occurrs, or until a console key is pressed. This is used mostly with the previous/next page commands for continuous scrolling.

ctrl O oops command

Occassionally, a line is inserted or deleted accidentally, half a line cleared by mistake, or some other major editing blunder is made. As the name implies, the "oops" command corrects most of these major editing errors. The "oops" command can be used to recover from the following commands:

1)	insert line command	(shift INS)
2)	delete line comand	(shift DEL)
3)	hack command	(ctrl H)
4)	to buffer command	(ctrl T)
5)	from buffer command	(ctrl F)
6)	copy from buffer command	(ctrl U)
7)	split line command	(ctrl J)
8)	splice command	(ctrl G)

Subcommands

The subcommand is entered by typing ctrl-V. The display will be cleared and a prompt (": ") will be issued. The following commands may be typed in response.

ST <return>	Make the current line the start of the file. (i.e., hack off the beginning)
EN <return>	Make the current line the end of the file. (i.e., hack off the end)
FL <return>	Position the cursor on the first line of the file.
LL <return>	Position the cursor on the last line of the file.
RT <return>	Reset the TAB stops to their original settings.
PS <return>	Enter the pattern search submode. The user will be prompted to enter the search string. The ctrl-L command will continue the search.
IF filename <return>	Insert the specified file into the file just after the current cursor line. This is useful for pulling subroutines from another file. This can be stopped at any time by pressing a console key.

Editor Command Summary

Below is a quick reference list of all the commands which the video editor recognizes.

Entering the Edit Mode: (executed outside of the edit mode)

EDIT	infile(,outfile) Enter the edit mode and edit "infile". If "outfile" is specified, send edited text to "outfile", otherwise send it "infile".	(---)
LL	Re-edit the last file edited.	(---)
WHERE	Enter the edit mode and position the cursor over the word that caused a compilation error.	(---)
#BUFS	Sets the length (in lines) of the storage buffer. The default is five.	(#lines ---)
.INFO	Display memory allocation of the current file so that it may be saved using the WRITE command. (used in case of a save error)	(---)

Cursor Movement: (issued within the edit mode)

ctrl ↑	Move cursor up one line, scrolling the file down one line if necessary.
ctrl ↓	Move cursor down one line, scrolling the file up one line if necessary.
ctrl ←	Move cursor left one character, wrapping to the right edge if moved off the left.
ctrl →	Move cursor right one character, wrapping to the left edge if moved off the right.
RETURN	Position the cursor at the beginning of the next line. Insert line if at the end of the file.
TAB	Advance to next tabular column.

ctrl TAB	Clear tab stop at current cursor location.
shift TAB	Set tab stop at current cursor location.

Editing Commands: (issued within the edit mode)

ctrl INS	Insert one blank at cursor location, losing the last character on the line.
ctrl DEL	Delete character under cursor, closing the line.
shift INS	Insert blank line above current line.
shift DEL	Delete current cursor line, closing the file.
ctrl M	Insert blank line below current line.
ctrl I	Toggle insert-mode/replace-mode. (see full description of ctrl-I).
BACKS	Delete last character typed, if on the same line as the cursor.
ctrl H	Erase to end of line (Hack).
ctrl V	Enter the subcommand mode. (see below)

Buffer Management: (issued within the edit mode)

ctrl T	Delete current cursor line sending it TO the edit buffer for later use.
ctrl F	Take the current buffer line and insert it above the current cursor line.
ctrl K	Kopy current cursor line sending it to the edit buffer for later use.
ctrl U	Take the current buffer line and copy it to the current cursor line. Unkopy
ctrl R	Roll the buffer making the next buffer line current.
ctrl B	Roll the buffer Backwards making the previous buffer line on the screen current.
ctrl C	Clear the current buffer line and performs a ctrl-B.

Note: the current buffer line is last line visible in the buffer window.

Scrolling/Saving: (issued within the edit mode)

ctrl X	Scroll the edit window up a line within the file.
ctrl E	Scroll the edit window down a line within the file.
ctrl P	Scroll the edit window up 16 lines within the file.
ctrl N	Scroll the edit window down 16 lines within the file.
ctrl S	Save the changes made to the current file and exit the edit mode.
ctrl Q	Quit the edit session forgetting all changes made to the current file.

Special Keys: (issued within the edit mode)

ESC	Do not interpret the next key typed as any of the commands above. Send it directly to the screen instead.
ctrl J	Split the current line into two lines at the point where the cursor is.
ctrl G	Splice (unsplit) the current line and the line above it.
ctrl O	Corrects any major editing blunders. Oops!
ctrl L	Continue searching for the pattern entered in the subcommand mode. Look
ctrl Y	Enter the repeat mode. The next command or character typed will be repeated until a stop condition is met, or until a console key is pressed. Used mostly with ctrl-P, ctrl-N, and the cursor commands.

Subcommands: (entered in the subcommand mode)

The subcommand is entered by typing ctrl-V. The display will be cleared and a prompt (": ") will be issued. The following commands may be typed in response.

ST <return>	Make the current line the start of the file. (i.e., hack off the beginning)

EN <return> Make the current line the end of the file. (i.e., hack off the end)

FL <return> Position the cursor on the first line of the file.

LL <return> Position the cursor on the last line of the file.

RT <return> Reset the TAB stops to their original settings.

PS <return> Enter the pattern search submode. The user will be prompted to enter the search string. The ctrl-L command will continue the search.

IF filename <return>
Insert the specified file into the file just after the current cursor line. This is useful for pulling subroutines from another file. This can be stopped at any time by pressing a console key.

valDOS I

Supplied Source Listing

LXV.

```
Screen:  1Ø
( valDOS:  quans and constants )
' ( QUAN    ) ( 15Ø LOAD )
' ( CARRAY ) ( 16Ø LOAD )
VOCABULARY DOS IMMEDIATE
DOS DEFINITIONS

 LABEL FNFLD          11 ALLOT
 QUAN FNCNT           Ø TO FNCNT
 QUAN FNSEP           Ø TO FNSEP
 QUAN FNSWCH          Ø TO FNSWCH
 QUAN UNIT            Ø TO UNIT
 QUAN DFLUNT          Ø TO DFLUNT
 QUAN FNSEC           Ø TO FNSEC
 QUAN ?WILD           Ø TO ?WILD
 QUAN #ENTRIES        Ø TO #ENTRIES
 QUAN #FREE           Ø TO #FREE   -->
```

```
Screen:  11
( valDOS:  quans and constants )

 QUAN DSKERR          Ø TO DSKERR
 QUAN DIRBLK          Ø TO DIRBLK
 QUAN FSMBLK          Ø TO FSMBLK
 QUAN ?FSMUP          Ø TO ?FSMUP
 QUAN ?SAME
 QUAN DIRFRE
 QUAN FL#
 QUAN TADR
 QUAN #UNTRN
 QUAN N1$
 QUAN N2$
 QUAN DIRLOC
 QUAN WRKSPC
 QUAN DOSTMP                       -->
```

```
Screen:  12
( valDOS:  quans and constants )

VECT RDMOPN
ASSIGN NOOP TO RDMOPN

VECT RDMCLS
ASSIGN NOOP TO RDMCLS

VECT RDMSPC
ASSIGN NOOP TO RDMSPC

VECT RDMENDF
ASSIGN NOOP TO RDMENDF

VECT RDMWND
ASSIGN NOOP TO RDMWND              -->
```

```
Screen:  13
( valDOS:  arrays                        )

  4 CONSTANT MAXFL
128 CONSTANT 128

( addresses of file buffers )
MAXFL 1+ ARRAY FLBUF
Ø FLBUF MAXFL 1+ 2* ERASE

( map of buffers in use )
MAXFL 2* 2 + ARRAY OPNSTT
Ø OPNSTT MAXFL 4 * 2 + ERASE

( the filename "Dx:*" )
LABEL ALLNMS
Ø4 C, 68 C, Ø C, 58 C, 42 C, -->
```

```
Screen:  14
( valDOS:  error codes                   )

 1 CONSTANT BADNME
 2 CONSTANT BADUNT
 3 CONSTANT BADFSM
 4 CONSTANT FLEXST
 5 CONSTANT DIRFUL
 6 CONSTANT DSKFUL
 7 CONSTANT AMBNME
 8 CONSTANT FLDNE
 9 CONSTANT TMFOPN
1Ø CONSTANT EOFERR
11 CONSTANT FLNOPN
12 CONSTANT BADFL#
13 CONSTANT FLWPRT
14 CONSTANT WRGARG                 -->
```

```
Screen:  15
( valDOS:  error codes                   )
15 CONSTANT FLOPN
16 CONSTANT FLTBG

HERE 1 AND ALLOT  ( even addr )
144 MAXFL * 192 + ALLOT
HERE TO WRKSPC 128 ALLOT
: TOPOM 741 @ 1- ;

: FSMAP                      ( -- a )
  WRKSPC 128 - ;

: DIRTBL FSMAP 64 - ;        ( -- a )

: BUFBOT                     ( -- a )
  DIRTBL 144 MAXFL * - ;        -->
```

Screen: 16

```
( valDOS:  error routines        )
: DOSERR                    ( # -- )
  TO DSKERR R> DROP Ø ;
: ?DOSERR                 ( f # -- )
  SWAP Ø=
  IF
    R> DROP DOSERR
  ENDIF
  DROP ;
: ?DSKERR                   ( f -- )
  Ø= IF R> DROP
  DSKERR DOSERR ENDIF ;
                                -->
```

Screen: 17

```
( valDOS:  filename conversion )
: GETSWCH           ( a c -- a c )
  Ø TO FNSWCH DUP 2 >
  IF 2DUP + 2- C@ 47 ( "/" ) =
    IF
      2- 2DUP + 1+ C@
      TO FNSWCH
    ENDIF
  ENDIF ;
: GETUNIT           ( a c -- a c )
  DFLUNT TO UNIT DUP 2 >
  IF OVER @ 14916 ( "D:") =
   IF
     2- SWAP 2+ SWAP           -->
```

Screen: 18

```
( valDOS:  filename conversion )
   ELSE DUP 3 >
    IF OVER C@ 68 ( "D") =
     3 PICK 2+ C@ 58 = AND
     IF OVER 1+ C@ 49 ( "1") -
     DUP Ø< NOT OVER 3 > NOT AND
      IF
       TO UNIT 3 - SWAP 3 + SWAP
      ELSE
       2DROP DROP BADUNT DOSERR
      ENDIF
     ENDIF
    ENDIF
   ENDIF
  ENDIF 1 ;                    -->
```

Screen: 19

```
( valDOS:  filename conversion )
: FNCON                 ( $ -- f )
  Ø TO DSKERR
  COUNT GETSWCH GETUNIT ?DSKERR
  8 TO FNCNT 46 TO FNSEP
  FNFLD 11 BLANKS FNFLD
  BEGIN
    3 PICK C@ DUP FNSEP =
    IF
      2DROP FNFLD 7 +
      DUP 1+ 3 BLANKS
      3 TO FNCNT Ø TO FNSEP
    ELSE
      DUP 42 ( "*" ) =
                                -->
```

Screen: 20

```
( valDOS:  filename conversion )
      IF
        OVER FNFLD 11 + OVER -
        63 ( "?") FILL DROP
      ELSE
        DUP 48 ( "Ø") <  OVER
        90 ( "Z") > OR  OVER
        57 ( "9") >   3 PICK
        65 ( "A") < AND OR
        OVER 63 ( "?") <> AND
        IF 2DROP 2DROP
          BADNME DOSERR
        ENDIF
        OVER C!
      ENDIF                     -->
```

Screen: 21

```
( valDOS:  filename conversion )
      FNCNT 1- DUP TO FNCNT Ø<
      IF DROP 2DROP
        BADNME DOSERR
      ENDIF
    ENDIF
    1+ ROT 1+ ROT 1- ROT
    OVER Ø=
  UNTIL
  2DROP DROP FNFLD C@ 63 >=
  BADNME ?DOSERR Ø TO ?WILD
  11 Ø
  DO
    FNFLD I + C@ 63 =
    IF 1 TO ?WILD ENDIF
  LOOP 1 ;                      -->
```

Screen: 22

```
( valDOS:  alias definitions    )

: 3PICK                       ( -- n )
  3 PICK ;

: FLBUF@                      ( n -- a )
  FLBUF @ ;

: FLINFO                      ( -- a )
  FL# FLBUF@ ;

: 72Ø* 72Ø * ;
: 256* 256 * ;
: 128- 128 - ;
: 135- 135 - ;
: 4* 2* 2* ;                       -->
```

Screen: 23

```
( valDOS:  get & put directory )

: DIRGET              ( sector -- )
  OFFSET @ OVER DRØ BLOCK
  TO DIRLOC OFFSET ! TO DIRBLK ;

: DIRFLS                      ( -- )
  FLUSH Ø TO DIRBLK ;

: DIRUP                       ( -- )
  UPDATE ;

: ENTRY                 ( unit # -- )
  1- 8 /MOD 36Ø +
  ROT 72Ø* + DIRGET
  16 * DIRLOC + ;                  -->
```

Screen: 24

```
( valDOS:  get free space map  )

: FSMGET                   ( unit -- )
  72Ø* 359 +
  DUP FSMBLK <>
  IF
    ?FSMUP
    IF
      FSMAP FSMBLK Ø R/W
    ENDIF
    FSMAP OVER 1 R/W
    TO FSMBLK Ø TO ?FSMUP
  ELSE
    DROP
  ENDIF ;
                                   -->
```

Screen: 25

```
( valDOS:  put free space map  )

: FSMFLS                        ( -- )
  ?FSMUP
  IF
    FSMAP FSMBLK Ø R/W
  ENDIF
  Ø TO ?FSMUP
  Ø TO FSMBLK ;

: FSMUP                         ( -- )
  1 TO ?FSMUP ;

: DSKFLS                        ( -- )
  DIRFLS FSMFLS ;
                                   -->
```

Screen: 26

```
( valDOS:  check directory      )

: CHKDIR                      ( -- # )
  Ø TO #ENTRIES Ø TO #FREE
  65 TO DIRFRE Ø 368 36Ø
  DO
   I UNIT 72Ø* + DIRGET
   DIRLOC 128 + DIRLOC
   DO
    1+ 1 TO ?SAME
    I C@ 195 AND 66 =
    IF 11 Ø
     DO
      J I + 5 + C@ FNFLD I +
      C@ DUP 63 ( "?") =
                                   -->
```

Screen: 27

```
( valDOS:  check directory      )

      IF DROP DUP ENDIF <>
      IF Ø TO ?SAME LEAVE ENDIF
     LOOP ?SAME
     IF
       DUP #ENTRIES DIRTBL + C!
       1 AT #ENTRIES +!
     ENDIF
    ELSE
     DUP DIRFRE MIN TO DIRFRE
     1 AT #FREE +!
    ENDIF
   16 /LOOP
  LOOP
  DROP #ENTRIES ;                  -->
```

Screen: 28

```
( valDOS:  allocate a sector    )

: ALTSEC        ( unit - [# t]/f )
  FSMGET
  FSMAP 3 + @ DUP
  IF
   1- FSMAP 3 + !
   Ø -1 FSMAP 1Ø + 9Ø O+S
   DO
    I C@
    IF
     SWAP DROP DUP LEAVE
     128 I C@
     BEGIN DUP 128 AND Ø=
     WHILE 2* ROT 1+ ROT 2/ ROT
     REPEAT                    -->
```

Screen: 29

```
( valDOS:  allocate a sector    )

     DROP I SWAP TOGGLE
    ELSE
     8 +
    ENDIF
   1 /LOOP
   SWAP Ø=
   IF
     DROP BADFSM DOSERR
   ENDIF
   FSMUP 1
  ENDIF ;



                               -->
```

Screen: 30

```
( valDOS:  release a sector    )

: RELSEC         ( unit # -- )
  SWAP FSMGET
  1+ 8 /MOD FSMAP 1Ø + +
  SWAP 128
  BEGIN
    OVER
  WHILE
    2/ SWAP 1- SWAP
  REPEAT
  SWAP DROP OVER C@
  OR SWAP C!
  FSMAP 3 + DUP @ 1+ SWAP !
  FSMUP ;
                               -->
```

Screen: 31

```
( valDOS:  find free buffer    )

: NXTOPN              ( -- a )
  MAXFL 4* Ø
  DO
   I -144 * DIRTBL 16 - + I' Ø
   DO
     I OPNSTT @ OVER =
     IF DROP Ø LEAVE ENDIF
   LOOP -DUP
   IF DUP I' Ø
     DO I OPNSTT @ Ø=
       IF I OPNSTT ! LEAVE ENDIF
     LOOP LEAVE
   ENDIF
  LOOP ;                       -->
```

Screen: 32

```
( valDOS:  flush file buffer    )

: FLFLS              ( fl# -- )
  FLBUF@ DUP 1Ø + C@
  IF
    DUP 8 + C@ 72Ø*
    OVER 5 + @ +
    OVER 128- SWAP Ø R/W
    1Ø + Ø SWAP C!
  ELSE
    DROP
  ENDIF ;




                               -->
```

Screen: 33

```
( valDOS:  [OPEN]              )

: (OPEN)      ( $ -- [a # 1]/Ø )
  FNCON ?DSKERR
  ?WILD NOT AMBNME ?DOSERR
  CHKDIR FLDNE ?DOSERR
  Ø MAXFL 1+ 1
  DO
    I FLBUF@ Ø=
    IF I + LEAVE ENDIF
  LOOP
  -DUP TMFOPN ?DOSERR
  >R NXTOPN DUP R FLBUF !
  DUP 16 ERASE
  UNIT DIRTBL C@ ENTRY
                               -->
```

```
Screen:  34
( valDOS:  [OPEN]                          )

  OVER 5 CMOVE
  3 + DUP @ 1- OVER 2+ !
  5 + UNIT OVER C! 1+
  DIRTBL C@ SWAP C!
  R FLBUF@ 128- DUP
  UNIT 720* OVER 133 +
  @ + 1 R/W R>
  RDMOPN 1 ;
                                         -->
```

```
Screen:  35
( valDOS:  [CLOSE]                         )

: (CLOSE)                   ( fl# -- )
  -DUP
  IF DUP ELSE MAXFL 1+ 1 ENDIF
  DO
   I DUP TO FL# FLBUF@
   IF
     RDMCLS FL# FLFLS
     FLINFO 2+ C@ 128 AND
     IF
       FLINFO DUP 8 + C@
       OVER 9 + C@ ENTRY 1+
       SWAP 1+ @ 32767 AND
       SWAP ! DIRUP DIRFLS
     ENDIF                               -->
```

```
Screen:  36
( valDOS:  [CLOSE]                         )

       FLINFO 0 FL# FLBUF !
       0 OPNSTT
       BEGIN
         DUP @ 3PICK <>
       WHILE
         2+
       REPEAT
       SWAP DROP 0 SWAP !
     ENDIF
  LOOP ;
                                         -->
```

```
Screen:  37
( valDOS:  [READ]                          )

: (READ)     ( adr cnt fl# -- f )
  TO FL# TO #UNTRN TO TADR
  FL# 1 < FL# MAXFL > OR
  NOT BADFL# ?DOSERR
  FLINFO FLNOPN ?DOSERR
  BEGIN
    FLINFO 7 + DUP C@
    OVER 8 - C@ =
    IF
     DUP 10 - DUP C@ 3 AND
     256* SWAP 1+ C@ + -DUP 0=
     IF DROP EOFERR DOSERR ENDIF
     1- FL# FLFLS OVER 135-
     OVER 4 PICK 1+ C@                   -->
```

```
Screen:  38
( valDOS:  [READ]                          )

     720* + 1 R/W
     OVER 2- ! 0 OVER C!
    ENDIF
    DUP 8 - C@ OVER C@ -
    #UNTRN OVER U<
    IF DROP #UNTRN ENDIF
    DUP 3PICK 7 + DUP @
    ROT + SWAP !
    OVER 135- 3PICK C@ +
    TADR DUP 4 PICK + TO TADR
    3PICK CMOVE #UNTRN OVER -
    TO #UNTRN OVER C@ +
    SWAP C! #UNTRN 0=
  UNTIL 1 ;                              -->
```

```
Screen:  39
( valDOS:  [WRITE]                         )

: (WRITE)    ( adr cnt fl# -- f )
  TO FL# TO #UNTRN TO TADR
  FL# 1 < FL# MAXFL > OR
  NOT BADFL# ?DOSERR
  FLINFO -DUP FLNOPN ?DOSERR
  C@ 32 AND NOT FLWPRT ?DOSERR
  BEGIN
    FLINFO 7 + DUP C@ 125 =
    IF
      DUP 10 - DUP
      C@ 3 AND 256* SWAP 1+
      C@ + -DUP 0=
      IF DUP 1+ C@ ALTSEC 0=
                                         -->
```

Screen: 40

```
( valDOS:  [WRITE]                    )
        IF
          DROP DSKFUL DOSERR
        ENDIF
        DUP 1+ 255 AND
        OVER 1+ 32767 AND 256
        / 4 PICK 2+ C@ 1- 4*
        >R R OR 4 PICK 10 - C!
        3PICK 9 - C! 1 3PICK
        3 + C! FL# FLFLS OVER
        135- 128 ERASE R>
        3PICK 10 - C! OVER 6 -
        DUP @ 1+ 32768 OR SWAP !
                                 -->
```

Screen: 41

```
( valDOS:  [WRITE]                    )
      ELSE
        FL# FLFLS 1- OVER
        135- OVER 4 PICK
        1+ C@ 720* + 1 R/W
      ENDIF
      OVER 2- ! 0 OVER C!
    ENDIF
    125 OVER C@ -
    #UNTRN OVER U<
    IF DROP #UNTRN ENDIF
    DUP 3PICK 7 + DUP
    @ ROT + SWAP !
    OVER 135- 3PICK C@ +
                                 -->
```

Screen: 42

```
( valDOS:  [WRITE]  [WRB]        )
    TADR DUP 4 PICK + TO TADR
    SWAP 3PICK CMOVE
    OVER C@ OVER + DUP
    4 PICK C! 3PICK 8 -
    DUP C@ ROT MAX SWAP C!
    #UNTRN SWAP - TO #UNTRN
    1 SWAP 3 + C!
    #UNTRN 0=
  UNTIL 1 ;

: (WRB)            ( b fl# -- f )
  SWAP WRKSPC C!
  WRKSPC 1 ROT (WRITE) ;
                                 -->
```

Screen: 43

```
( valDOS:  [RDB]                     )
: (RDB)         ( fl# -- [b 1]/0 )
  DUP TO FL#
  1 < FL# MAXFL > OR
  NOT BADFL# ?DOSERR
  FLINFO FLNOPN ?DOSERR
  FLINFO 7 +
  DUP C@ OVER 8 - C@ =
  IF
    DUP 10 - DUP
    C@ 3 AND 256*
    SWAP 1+ C@ + -DUP 0=
    IF
      DROP EOFERR DOSERR
    ENDIF
                                 -->
```

Screen: 44

```
( valDOS:  [RDB]                     )
    1- FL# FLFLS
    OVER 135- OVER
    4 PICK 1+ C@ 720* +
    1 R/W
    OVER 2- ! 0 OVER C!
  ENDIF
  DUP C@ SWAP
  OVER 1+ OVER C!
  1 OVER 7 + +!
  135- + C@ 1 ;

FORTH DEFINITIONS
```

Screen: 45

Screen: 46

Screen: 47

Screen: 48

Screen: 49

Screen: 50

```
( valDOS:  [WIND]                    )
'( DOS DOS )(  )
'( FNFLD )( 10 LOAD )

DOS DEFINITIONS

: (WIND)            ( f fl# -- f )
  TO FL# TO DOSTMP
  FL# 1 < FL# MAXFL > OR
  NOT BADFL# ?DOSERR RDMWND
  FLINFO -DUP FLNOPN ?DOSERR
  DOSTMP
  IF ( rewinding )
    FL# FLFLS DUP 3 + @ 1-
    DUP 3PICK 5 + @ <>
                                   -->
```

Screen: 51

```
( valDOS:  [WIND]                    )

    IF
      DUP 3PICK 5 + ! OVER 8 +
      C@ 720* + OVER 128-
      SWAP 1 R/W
    ELSE DROP ENDIF
    0 OVER 7 + C!
    0 SWAP 14 + !
  ELSE  ( to end of file )
    DROP
    BEGIN
      WRKSPC 128 FL# (READ) 0=
    UNTIL
    DSKERR EOFERR = ?DSKERR
  ENDIF 1 ;                        -->
```

```
Screen:  52
( valDOS:  [ENTER]                )
: (ENTER)              ( $ -- f )
  FNCON ?DSKERR
  ?WILD NOT AMBNME ?DOSERR
  CHKDIR Ø= FLEXST ?DOSERR
  #FREE DIRFUL ?DOSERR
  UNIT ALTSEC DSKFUL ?DOSERR
  DUP 1+ TO FNSEC
  WRKSPC 128 ERASE
  DIRFRE 1- 4* WRKSPC
  125 + C! WRKSPC
  SWAP UNIT 720* + Ø R/W
  UNIT DIRFRE ENTRY
  66 OVER C! 1+ 1 OVER ! 2+
                                   -->
```

```
Screen:  53
( valDOS:  [ENTER]                )
  FNSEC OVER ! 2+
  FNFLD SWAP 11 CMOVE
  DIRUP FSMUP 1 ;
                                   -->
```

```
Screen:  54
( valDOS:  [SPACE]                )
: (SPACE)         ( cnt fl# -- f )
  TO FL#
  FL# 1 < FL# MAXFL > OR
  IF DROP BADFL# DOSERR ENDIF
  FLINFO Ø=
  IF DROP FLNOPN DOSERR ENDIF
  RDMSPC DUP Ø<  ( backwards )
  IF
    FLINFO 7 + DUP C@
    3PICK + DUP Ø<
    IF
      DROP 7 + @ +
      Ø MAX 1 FL# (WIND) Ø=
                                   -->
```

```
Screen:  55
( valDOS:  [SPACE]                )
      IF
        DROP DSKERR DOSERR
      ENDIF
    ELSE
      ROT DROP SWAP DUP
      C@ MINUS OVER 7 + +!
      Ø SWAP C!
    ENDIF
  ENDIF
  128 /MOD
  BEGIN DUP    ( space forward )
  WHILE
    WRKSPC 128 FL# (READ) Ø=
                                   -->
```

```
Screen:  56
( valDOS:  [SPACE]                )
    IF
      2DROP DSKERR DOSERR
    ENDIF
    1-
  REPEAT
  DROP WRKSPC SWAP
  FL# (READ) ?DSKERR
  1 ;
                                   -->
```

```
Screen:  57
( valDOS:  [?OPEN]                )
: (?OPEN)           ( $ - [f 1]/Ø )
  FNCON ?DSKERR
  ?WILD NOT AMBNME ?DOSERR
  CHKDIR FLDNE ?DOSERR
  UNIT DIRTBL C@ 256* +
  Ø MAXFL 1+ 1
  DO
    I FLBUF@ -DUP
    IF 8 + @ 3PICK =
      IF 1+ LEAVE ENDIF
    ENDIF
  LOOP
  SWAP DROP 1 ;
                                   -->
```

Screen: 58

```
( valDOS:  [REN]                        )

: (REN)              ( $n $o -- f )
  TO N1$ TO N2$
  N2$ FNCON ?DSKERR
  ?WILD NOT AMBNME ?DOSERR
  CHKDIR Ø= FLEXST ?DOSERR
  FNFLD WRKSPC 11 CMOVE
  UNIT TO DOSTMP
  N1$ FNCON ?DSKERR
  ?WILD NOT AMBNME ?DOSERR
  DOSTMP UNIT = BADUNT ?DOSERR
  CHKDIR FLDNE ?DOSERR
  UNIT DIRTBL C@ ENTRY C@
  32 AND Ø= FLWPRT ?DOSERR
                                  -->
```

Screen: 59

```
( valDOS:  [REN]  [KILL]                )

  WRKSPC UNIT
  DIRTBL C@ ENTRY 5 +
  11 CMOVE DIRUP 1 ;

: (KILL)                ( $ -- f )
  (?OPEN) ?DSKERR
  NOT FLOPN ?DOSERR
  UNIT DIRTBL C@ ENTRY
  DUP C@ 32 AND
  IF DROP FLWPRT DOSERR ENDIF
  DUP 3 + @ 1-
  BEGIN
    WRKSPC OVER
    UNIT 720* + 1 R/W             -->
```

Screen: 60

```
( valDOS:  [KILL]                       )

    UNIT SWAP RELSEC
    WRKSPC 125 + DUP
    1+ C@ DUP ROT C@
    3 AND DUP <ROT
    256* + 1- <ROT OR Ø=
  UNTIL
  DROP 128 SWAP C!
  DIRUP 1 ;






                                  -->
```

Screen: 61

```
( valDOS:  [LOCK]  [UNLOCK]             )

: (DOLCK)              ( $ opt -- f )
  TO DOSTMP (?OPEN) ?DSKERR
  NOT FLOPN ?DOSERR
  DOSTMP UNIT DIRTBL C@
  ENTRY DUP C@ ROT
  IF 32 OR ELSE 223 AND ENDIF
  SWAP C! DIRUP 1 ;

: (LOCK)                  ( $ -- f )
  1 (DOLCK) ?DSKERR 1 ;

: (UNLOCK)                ( $ -- f )
  Ø (DOLCK) ?DSKERR 1 ;
                                  -->
```

Screen: 62

```
( valDOS:  [ENDF]                       )

: (ENDF)                 ( f1# -- f )
  TO FL#
  FL# 1 < FL# MAXFL > OR
  NOT BADFL# ?DOSERR
  FLINFO -DUP FLNOPN ?DOSERR
  DUP C@ 32 AND
  IF DROP FLWPRT DOSERR ENDIF
  DUP 7 + C@ OVER 1- C!
  DUP 2- DUP C@ OVER 1- C@
  3 AND ROT DUP 11 + C@
  1- 4* SWAP 1- ! 1 4
  PICK 1Ø + C!
  ROT 8 + C@ <ROT
                                  -->
```

Screen: 63

```
( valDOS:  [ENDF]                       )

  BEGIN
    2DUP OR
  WHILE
    256* + 1- OVER 720*
    OVER + WRKSPC SWAP 1 R/W
    OVER SWAP RELSEC
    WRKSPC 126 + DUP C@
    SWAP 1- C@ 3 AND
    FLINFO 1+ DUP @
    1- 32768 OR SWAP !
  REPEAT
  2DROP DROP RDMENDF 1 ;

FORTH DEFINITIONS
```

Screen: 64

Screen: 65

Screen: 66

Screen: 67

Screen: 68

Screen: 69

Screen: 70

```
( valDOS:  quans and vects      )
' ( DOS DOS ) (  )
' ( (WIND) ) ( 50 LOAD )
' ( $. ) ( : $. COUNT TYPE ; )
' ( +Y/N ) ( 142 LOAD )

  QUAN FLFL#  QUAN FLCNT
  QUAN ED#IN  0 TO ED#IN
  LABEL FLNME 16 ALLOT
  VECT *ECHO
  ASSIGN DROP TO *ECHO

: ECHO                       ( f -- )
  IF ASSIGN EMIT
  ELSE ASSIGN DROP
  ENDIF TO *ECHO ;                -->
```

Screen: 71

```
( valDOS:  argument evaluation )

: GETVAL DOS              ( $ -- v )
  WRKSPC 34 BLANKS
  WRKSPC OVER C@ 1+ CMOVE
  WRKSPC LATEST (FIND)
  IF
    DROP CFA EXECUTE
  ELSE
    WRKSPC NUMBER DROP
  ENDIF ;

: GETARGS                   ( -- $ )
  32 WORD HERE 1 OVER @ 1 =
  IF 2DROP 0 ENDIF 0 84 C@
  40 * 85 @ + 88 @ + C! ;      -->
```

Screen: 72

```
( valDOS:  argument evaluation )

: GETARG   ( delm -- $ [$ 1]/0 )
  DOS OVER COUNT
  BEGIN
    DUP 0# 3PICK C@
    5 PICK <> AND
  WHILE 1- SWAP 1+ SWAP
  REPEAT
  ROT DROP -DUP
  IF
    DUP 1- 3PICK C! ROT
    DUP C@ ROT - OVER C! 1
  ELSE
  DROP 0 ENDIF ;
                                 -->
```

Screen: 73

```
( valDOS:  error routines          )

: CMDERR DOS            ( err -- )
  CR DUP TO DSKERR 4 /MOD
  696 + WRKSPC SWAP 1 R/W
  32 * WRKSPC + 32 -TRAILING
  TYPE CR SP! DSKFLS QUIT ;

: ?CMDERR             ( f err -- )
  SWAP 0= IF CMDERR ENDIF DROP ;

: ?WRGARG DOS              ( f -- )
  WRGARG ?CMDERR ;

: ?SYSERR DOS              ( f -- )
  DSKERR ?CMDERR ;                -->
```

Screen: 74

```
( valDOS:  CONFN                   )

LABEL FNAME 16 ALLOT

: CONFN DOS            ( a u -- $ )
  FNAME 1+ 68 OVER C!
  1+ SWAP 49 + OVER C!
  1+ 58 OVER C!
  1+ OVER 8 -TRAILING
  >R OVER R CMOVE R> +
  46 OVER C! 1+ SWAP 8 +
  3 -TRAILING  >R OVER R
  CMOVE R + R> 0= -
  FNAME 1+ - FNAME
  >R R C! R> ;
                                  -->
```

Screen: 75

```
( valDOS:  FLOAD support           )

: (SKIP) DOS           ( n fl# -- )
  FNSWCH 67 ( "C") =
  IF
    ED#IN OVER (SPACE) ?SYSERR
  ENDIF
  SWAP
  BEGIN -DUP
  WHILE 1-
    BEGIN
      OVER (RDB) ?SYSERR
      155 =
    UNTIL
  REPEAT
  DROP ;                          -->
```

```
Screen:  76
( valDOS:  FLOAD support          )

: NXTCHR DOS         ( -- [c 1]/0 )
  ?TERMINAL 0= STATE @ 0#
  OR EOFERR ?DOSERR
  FLFL# (RDB) ?DSKERR
  1 AT FLCNT +!  DUP *ECHO
  DUP 155 =
  IF
    DROP BL
    FLCNT 0 TO FLCNT 39 =
    IF
      DROP [ LATEST PFA CFA , ]
      ?DSKERR
    ENDIF
  ENDIF 1 ;                       -->
```

```
Screen:  77
( valDOS:  FLOAD support          )

: FWORD DOS                ( c -- )
  HERE 34 BLANKS HERE 1+ DUP
  BEGIN
    DROP NXTCHR DUP
    DSKERR EOFERR = OR ?SYSERR
    0= IF 0 ENDIF
    DUP 4 PICK <>
  UNTIL
  BEGIN
    OVER C! 1+
    TOPOM 32 - OVER U>
    FLTBG ?CMDERR
    NXTCHR DUP
                                  -->
```

```
Screen:  78
( valDOS:  FLOAD                  )

    DSKERR EOFERR = OR ?SYSERR
    0= IF 0 ENDIF
    DUP 4 PICK = OVER 0= OR
  UNTIL
  ROT 2DROP
  HERE 1+ - HERE C! ;

: FLOAD DOS                 ( -- )
  GETARGS ?WRGARG 44 GETARG
  IF SWAP GETVAL 1- 0 MAX
  ELSE 0 ENDIF
  SWAP FLFL# >R
  ' WORD DUP @ >R 2+ @ >R
  [ ' BRANCH CFA ] LITERAL     -->
```

```
Screen:  79
( valDOS:  FLOAD                  )

   ' WORD ! ' FWORD ' WORD 2+ -
   ' WORD 2+ !  BLK @ >R
  FLCNT >R 0 BLK !
  DUP FLNME 16 CMOVE
  (OPEN) ?SYSERR
  TO FLFL# DROP
  FLFL# (SKIP) CR INTERPRET
  FLFL# (CLOSE) DIRFLS
  R> TO FLCNT R> BLK !
  ' WORD R> OVER
  2+ ! R> SWAP !
  R> TO FLFL# ;

                                  -->
```

```
Screen:  80
( valDOS:  LOAD redefined         )

: FIXWORD DOS
  0 ' BLK CFA ' WORD !
  ' @ CFA ' WORD 2+ ! ;

' FIXWORD CFA ' QUIT !

: LOAD DOS                ( n -- )
  ' WORD D@ >R >R FIXWORD DROP
  LOAD R> R> ' WORD D! ;

: (?LOADING)
  0= ' WORD @ ' BRANCH CFA <>
  AND ;  ' (?LOADING) CFA
  ' ?LOADING 4 + !              -->
```

```
Screen:  81
( valDOS:  DIR                    )

: DIR DOS                   ( -- )
  DFLUNT 49 +  ALLNMS 2+ C!
  GETARGS 0= IF ALLNMS ENDIF
  DUP 1+ C@ 248 AND 48 =
  IF
    1+ C@ 15 AND 48 + ALLNMS
    2+ C! ALLNMS
  ENDIF
  FNCON ?SYSERR UNIT FSMGET
  CR CR ." Files on: "
  FSMAP 104 +
  BEGIN
    DUP C@ -DUP
  WHILE                           -->
```

```
Screen:  82
( valDOS:  DIR                         )
      EMIT 1+
   REPEAT
   DROP CR CR
      ."    NAME     EXT    SIZE    S
EC   ATTR  " CR
      ." +-+-+-+-+-+-+-+-+-+-+-+-+-+
-+-+-+-+-+"
   CR CR CHKDIR DUP
   BEGIN
      DUP Ø# ?TERMINAL Ø= AND
   WHILE
      2DUP - UNIT SWAP DIRTBL +
      C@ ENTRY SPACE DUP 5 + 8
      TYPE DUP 13 + C@ 32 <>
      IF 46 EMIT                   -->
```

```
Screen:  83
( valDOS:  DIR                         )
      ELSE SPACE ENDIF
      DUP 13 + 3 TYPE
      DUP 1+ @ 7 .R
      DUP 3 + @ 6 .R
      4 SPACES C@ DUP 32 AND
      IF 76 ( "L") EMIT ENDIF
      16 AND IF 82 ( "R") EMIT
      ENDIF 1- CR
   REPEAT
   CR FSMAP 3 + @ .
   ." sectors free." CR CR
   DSKFLS 2DROP ;
                                   -->
```

```
Screen:  84
( valDOS:  OPEN?                       )
: OPEN? DOS                     ( -- )
   CR Ø OUT ! MAXFL 1+ 1
   DO I FLBUF@ -DUP
      IF 8 + DUP C@
         SWAP 1+ C@
         I Ø <# # # # #> TYPE
         OVER TO UNIT ENTRY
         5 + UNIT CONFN
         2 SPACES $. CR
      ENDIF
   LOOP
   DIRFLS OUT @ Ø=
   IF ." No files open" CR ENDIF
   CR ;                            -->
```

```
Screen:  85
( valDOS:  COPY                        )
: COPY DOS                      ( -- )
   GETARGS ?WRGARG
   61 GETARG ?WRGARG
   DUP (ENTER) Ø=
   IF
     DSKERR FLEXST = ?SYSERR
   ENDIF
   DUP (?OPEN) ?SYSERR
   NOT FLOPN ?CMDERR
   (OPEN) ?SYSERR
   <ROT DROP OVER
   FLBUF@ C@ 32 AND
   Ø= FLWPRT ?CMDERR
                                   -->
```

```
Screen:  86
( valDOS:  COPY                        )
   FNSWCH 65 ( "A") =
   IF
     Ø 3PICK (WIND) ?SYSERR
   ENDIF
   BEGIN
     Ø >R 44 ( ",") GETARG Ø=
     IF DUP R> 1+ >R ENDIF
     (OPEN) ?SYSERR SWAP DROP
     BEGIN
       WRKSPC 128 3PICK (READ)
       DROP #UNTRN 128 <>
     WHILE
       WRKSPC 128 #UNTRN -
                                   -->
```

```
Screen:  87
( valDOS:  COPY  RENAME                )
       5 PICK (WRITE) ?SYSERR
     REPEAT
     (CLOSE) R>
   UNTIL
   CR CR DROP DUP
   (ENDF) ?SYSERR
   (CLOSE) DSKFLS ;
: RENAME DOS                    ( -- )
   GETARGS ?WRGARG
   61 GETARG ?WRGARG
   SWAP (REN) ?SYSERR
   CR ." File renamed" CR CR
   DSKFLS ;                        -->
```

```
Screen:  88
( valDOS:  system words          )

QUAN WLDFLG       QUAN SWITCH
VECT DISKOP       VECT $OUT
QUAN TEMP

: (CMDPAR) DOS                  ( -- )
  GETARGS ?WRGARG
  BEGIN
    Ø >R 44 ( ",") GETARG Ø=
    IF DUP R> 1+ >R ENDIF
    DUP TO N1$ FNCON FNSWCH
    TO SWITCH ?WILD TO WLDFLG
    IF CHKDIR DUP DUP Ø=
      IF
        FNFLD UNIT CONFN        -->
```

```
Screen:  89
( valDOS:  system words          )

        CR Ø OUT ! $.
        18 OUT @ - SPACES
        ." is non-existent"
      ENDIF
      BEGIN -DUP
      WHILE
        DIRTBL OVER - 3PICK +
        C@ UNIT SWAP ENTRY 5 +
        UNIT CONFN 1 WLDFLG
        SWITCH 78 ( "N") <> *
        IF DROP CR $OUT SPACE
          DUP $. ."  ? " +Y/N
        ENDIF
          Ø OUT !               -->
```

```
Screen:  9Ø
( valDOS:  system words          )

        IF DUP DISKOP Ø= SWAP CR
          $. 18 OUT @ - SPACES
          IF
            DSKERR FLOPN =
            IF
              ." is open"
            ELSE
              DSKERR FLWPRT =
              IF
                ." is locked"
              ELSE ." is ???"
              ENDIF
            ENDIF
          ELSE                  -->
```

```
Screen:  91
( valDOS:  system words          )

          ." is " $OUT ." ed"
        ENDIF
      ELSE
        DROP
      ENDIF 1-
    REPEAT
    DROP
  ELSE
    CR Ø OUT ! N1$ $.
    18 OUT @ - SPACES
    ." is illegal"
  ENDIF R>
UNTIL
DROP CR CR DSKFLS ;             -->
```

```
Screen:  92
( valDOS:  KILL  LOCK            )

: $KILL ." kill" ;

: KILL DOS                      ( -- )
  ASSIGN (KILL) TO DISKOP
  ASSIGN $KILL TO $OUT
  (CMDPAR) ;


: $LOCK ." lock" ;

: LOCK DOS                      ( -- )
  ASSIGN (LOCK) TO DISKOP
  ASSIGN $LOCK TO $OUT
  (CMDPAR) ;                    -->
```

```
Screen:  93
( valDOS:  UNLOCK  SETUNIT       )

: $UNLOCK ." unlock" ;

: UNLOCK DOS                    ( -- )
  ASSIGN (UNLOCK) TO DISKOP
  ASSIGN $UNLOCK TO $OUT
  (CMDPAR) ;


: SETUNIT DOS                   ( -- )
  GETARGS ?WRGARG GETVAL
  DUP 1 >= OVER 4 <= AND
  BADUNT ?CMDERR
  1- TO DFLUNT CR ;
                                -->
```

```
Screen:  94
( valDOS:  PRINT                        )

: PRINT DOS                     ( -- )
  GETARGS ?WRGARG 44 GETARG
  IF SWAP GETVAL 1- 0 MAX
  ELSE 0 ENDIF SWAP
  (OPEN) ?SYSERR SWAP DROP
  DUP <ROT OVER TO TEMP (SKIP)
  FNFLD UNIT CONFN
  CR CR ." File: " $. CR CR
  FNSWCH 78 <> DUP TO FNSWCH
  TEMP 1+ * TO TEMP
  BEGIN
    0 ?TERMINAL FNSWCH * 0=
    IF DROP DUP (RDB) ENDIF
                                    -->
```

```
Screen:  95
( valDOS:  PRINT                        )

  WHILE
    FNSWCH 1 =
    IF 0 TO FNSWCH TEMP
      IF
        TEMP 0
        <# # # # # #> TYPE
        SPACE 1 AT TEMP +!
      ENDIF
    ENDIF
    DUP EMIT 155 = TO FNSWCH
  REPEAT
  (CLOSE)
  CR DSKFLS ;
                                    -->
```

```
Screen:  96
( valDOS:  read/write utility  )

: FLOPEN DOS                  ( $ -- )
  DUP 1+ C@ DUP 240 AND 48 =
  IF
    15 AND SWAP DROP 0 TO TEMP
  ELSE
    DROP TEMP 2 AND
    IF DUP (ENTER) 0=
      IF
        DSKERR FLEXST = ?SYSERR
      ENDIF
    ENDIF
    (OPEN) ?SYSERR SWAP DROP
  ENDIF ;
                                    -->
```

```
Screen:  97
( valDOS:  READ                         )

: READ DOS                      ( -- )
  65535 GETARGS ?WRGARG
  44 GETARG ?WRGARG
  SWAP 44 GETARG
  IF
    SWAP 4 ROLL DROP
    GETVAL <ROT
  ENDIF
  GETVAL <ROT
  1 TO TEMP FLOPEN (READ)
  0= DSKERR EOFERR <>
  AND NOT ?SYSERR
  TEMP IF FL# (CLOSE) ENDIF
  CR DSKFLS ;                       -->
```

```
Screen:  98
( valDOS:  WRITE                        )

: WRITE DOS                     ( -- )
  GETARGS ?WRGARG
  44 GETARG ?WRGARG
  SWAP 44 GETARG ?WRGARG
  GETVAL SWAP GETVAL ROT
  2 TO TEMP FLOPEN (WRITE) 0=
  IF
    DSKERR FLWPRT = ?SYSERR
    FL# (CLOSE) FLWPRT CMDERR
  ENDIF
  TEMP IF
    FL# (ENDF) ?SYSERR
    FL# (CLOSE)
  ENDIF CR DSKFLS ;                 -->
```

```
Screen:  99
( valDOS:  CLOSE                        )

: CLOSE DOS                     ( -- )
  0 GETARGS
  IF
    SWAP DROP GETVAL
  ENDIF
  (CLOSE) CR DSKFLS ;
```

Screen: 100

Screen: 101

Screen: 102

Screen: 103

Screen: 104

Screen: 105

Screen: 106

Screen: 107

Screen: 108

Screen: 109

Screen: 110

```
( Utils:  OPEN  ENTER                  )

( FLFL# )( 70 LOAD )

: OPEN DOS                             ( -- )
  GETARGS ?WRGARG
  (OPEN) ?SYSERR
  CR ." File access #" .
  DROP CR DSKFLS ;

: ENTER DOS                            ( -- )
  GETARGS ?WRGARG
  (ENTER) ?SYSERR
  CR DSKFLS ;

                                           -->
```

Screen: 111

```
( Utils:  FSPACE  ENDFIL               )

: FSPACE DOS                           ( -- )
  GETARGS ?WRGARG
  44 GETARG ?WRGARG
  GETVAL SWAP GETVAL
  SWAP (SPACE) ?SYSERR
  CR DSKFLS ;

: ENDFIL DOS                           ( -- )
  GETARGS ?WRGARG
  GETVAL (ENDF) ?SYSERR
  CR DSKFLS ;


                                           -->
```

Screen: 112

```
( Utils:  FDUMP system words    )

: PSPACE                     ( -- )
  PFLAG @ DUP 254 AND
  PFLAG ! SPACE PFLAG ! ;

: P.                        ( c -- )
  127 AND DUP
  32 < OVER          ( ctrl? )
  124 > OR           ( clear...?)
  IF
    DROP 46 ( ".")
  ENDIF
  EMIT ;

                                 -->
```

Screen: 113

```
( Utils:  FDUMP                 )

: FDUMP DOS                  ( -- )
  GETARGS ?WRGARG
  (OPEN) ?SYSERR
  FNFLD UNIT CONFN
  CR CR ." File: " $. CR
  8 FNSWCH 87 = 1+ *
  TO FNSWCH SWAP DROP
  BASE @ HEX SWAP Ø
  BEGIN
    WRKSPC 128 4 PICK (READ)
    >R 128 #UNTRN - WRKSPC
    BEGIN
      OVER Ø> ?TERMINAL Ø= AND
                                 -->
```

Screen: 114

```
( Utils:  FDUMP                 )

    WHILE
      3 PICK FNSWCH MOD Ø=
      IF
        CR Ø TO TEMP
        Ø OUT ! 3 PICK Ø
        <# # # # # #>
        TYPE SPACE PSPACE
      ENDIF
      DUP C@ Ø <# # # #> TYPE
      SPACE 1+ ROT 1+
      ROT 1- ROT 1 AT TEMP +!
      OVER Ø= 4 PICK
      FNSWCH MOD Ø= OR
                                 -->
```

Screen: 115

```
( Utils:  FDUMP                 )

      IF
        FNSWCH 16 = 24 * 3Ø +
        OUT @ - SPACES PSPACE
        DUP TEMP - TEMP Ø
        DO DUP I + C@ P.
        LOOP DROP
      ENDIF
    REPEAT
    2DROP R> NOT
    ?TERMINAL OR
  UNTIL
  DROP (CLOSE)
  CR BASE ! DSKFLS ;
                                 -->
```

Screen: 116

```
( Utils:  REWIND  EOF           )

: REWIND DOS                 ( -- )
  GETARGS ?WRGARG
  GETVAL 1 SWAP (WIND)
  ?SYSERR CR DSKFLS ;

: EOF DOS                    ( -- )
  GETARGS ?WRGARG
  GETVAL Ø SWAP (WIND)
  ?SYSERR CR DSKFLS ;




                                 -->
```

Screen: 117

```
( Utils:  NAMEDISK              )

: NAMEDISK DOS               ( -- )
  DFLUNT 1+ GETARGS
  IF SWAP DROP GETVAL ENDIF
  DUP 1 < OVER 4 > OR
  NOT BADUNT ?CMDERR
  1- FSMGET FSMAP 104 +
  CR ." Old: " DUP
  BEGIN DUP C@ -DUP
  WHILE EMIT 1+
  REPEAT
  DROP DUP 2Ø ERASE CR
  ." New: " 2Ø EXPECT CR CR
  1 TO ?FSMUP DSKFLS ;
```

Screen: 118

Screen: 119

Screen: 120

```
( Fmtrs:  system words          )
' ( FLFL# ) ( 70 LOAD )

: INSIN CR
  ." Insert source: <START> "
  BEGIN
    ?TERMINAL 1 =
  UNTIL
  46 EMIT ;

: INSDST CR
  ." Insert dest:  <SELECT> "
  BEGIN
    ?TERMINAL 2 =
  UNTIL
  46 EMIT ;                     -->
```

Screen: 121

```
( Fmtrs:  FMOVE                 )

: FMOVE DOS                ( -- )
  GETARGS ?WRGARG
  61 GETARG ?WRGARG
  0 (CLOSE) SWAP INSIN
  (OPEN) ?SYSERR <ROT DROP
  DSKFLS INSDST DUP (ENTER) 0=
  IF
    DSKERR FLEXST = ?SYSERR
  ENDIF
  (OPEN) ?SYSERR SWAP DROP
  BEGIN
    INSIN PAD DUP 1 AND +
    TOPOM OVER - 4 PICK
    (READ) INSDST PAD DUP     -->
```

Screen: 122

```
( Fmtrs:  FMOVE                 )

    1 AND + TADR OVER -
    4 PICK (WRITE) DSKFLS 0=
    IF DROP (CLOSE) (CLOSE)
       DSKERR CMDERR
    ENDIF 0=
  UNTIL
  DUP (ENDF) <ROT (CLOSE)
  (CLOSE) ?SYSERR CR CR DSKFLS ;

: GETNUM                 ( -- n )
  HERE 1+ 10 EXPECT HERE 1+
  BEGIN DUP C@ WHILE 1+ REPEAT
  BL SWAP C! HERE NUMBER DROP ;
                                -->
```

Screen: 123

```
( Fmtrs:  system words          )

: LOCKOUT DOS       ( u s cnt -- )
  ROT FSMGET O+S
  DO
    I 1+ 8 /MOD FSMAP 10 + +
    SWAP 128
    BEGIN OVER
    WHILE 2/ SWAP 1- SWAP
    REPEAT
    SWAP DROP DUP 255 XOR SWAP
    3PICK C@ AND 0# FSMAP 3 +
    DUP @ ROT - SWAP !
    OVER C@ AND SWAP C!
  LOOP
  1 TO ?FSMUP ;                 -->
```

Screen: 124

```
( Fmtrs:  FORMAT                    )

: FORMAT DOS                  ( -- )
  DFLUNT 1+ GETARGS
  IF GETVAL SWAP DROP ENDIF
  DUP DUP 1- TO UNIT CR CR
  ." Format unit " . ." ? " +Y/N
  IF
    DUP WRKSPC 772 ! (FMT)
    DROP WRKSPC 128 ERASE
    2 WRKSPC C!
    WRKSPC 1+ 7Ø7 OVER ! 716
    SWAP 2+ ! 15 WRKSPC 1Ø + C!
    WRKSPC 11 + 89 255 FILL
    1- 72Ø* 359 +
    WRKSPC OVER Ø R/W          -->
```

Screen: 125

```
( Fmtrs:  FORMAT                    )

    UNIT 359 9 LOCKOUT CR CR
    ." Diskname: "
    FSMAP 1Ø4 + 2Ø EXPECT CR CR
    ." Lock out error screens? "
    +Y/N
    IF
      UNIT 695 24 LOCKOUT
    ENDIF
    BEGIN CR CR
      ." Lock out sectors? "
      +Y/N DUP
      IF
        UNIT CR
        ." First sector: "     -->
```

Screen: 126

```
( Fmtrs:  FORMAT                    )

        GETNUM CR
        ." # to lock out: "
        GETNUM LOCKOUT
      ENDIF
    Ø= UNTIL
  ENDIF
  DROP DSKFLS CR CR ;




                                   -->
```

Screen: 127

```
( Fmtrs:  system words              )

Ø VARIABLE SEC/PAS
Ø VARIABLE SECNT

: AXLN  DOS            ( system )
  4 PICK Ø
  DO 3PICK I 128 * +
     3PICK I + 3 PICK R/W
  LOOP 2DROP 2DROP ;

: DCSTP DOS                   ( -- )
  DSKFLS TOPOM PAD DUP 1 AND
  - - Ø 128 U/ SWAP DROP
  SEC/PAS ! Ø SECNT ! ;
                                 -->
```

Screen: 128

```
( Fmtrs:  DISKCOPY1                 )

: DISKCOPY1                   ( -- )
  DCSTP
  BEGIN
    CR INSIN
    72Ø SECNT @ - SEC/PAS @ MIN
    DUP >R PAD DUP 1 AND - SECNT
    @ 2DUP 5 PICK <ROT 1 AXLN
    INSDST Ø AXLN CR
    R> SECNT +! SECNT @ DUP .
    ." sectors copied" 72Ø =
  UNTIL
  MTB CR ;

                                 -->
```

Screen: 129

```
( Fmtrs:  DISKCOPY2                 )

: DISKCOPY2                   ( -- )
  DCSTP
  CR ." Insert source in drive 1
" CR ." Insert dest.  in drive 2
" CR ." Press START to copy"
  WAIT
  BEGIN
   72Ø SECNT @ - SEC/PAS @ MIN
   DUP >R PAD DUP 1 AND - SECNT
   @ 2DUP 5 PICK <ROT
   1 AXLN 72Ø + Ø AXLN
   R> SECNT +! SECNT @ 72Ø =
  UNTIL
  MTB CR ;
```

Screen: 130
Ø
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 131
Ø
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 132
Ø
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 133
Ø
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 134
Ø
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 135
Ø
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 136
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 137
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 138
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 139
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 14Ø
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 141
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 142

```
( +Y/N routine                         )
( This Y/N routine accepts upper
  or lower case and echos it.  )

BASE @ HEX

: +Y/N                          ( -- f )
  KEY DUP DF AND
  DUP 59 <>
  ØBRANCH [ ØØ14 , ]
  DUP 4E <>
  ØBRANCH [ ØØØ8 , ]
  2DROP BRANCH [ FFDA , ]
  SWAP EMIT 59 = ;

BASE !
```

Screen: 143

Screen: 144

Screen: 145

Screen: 146

Screen: 147

Screen: 148

Screen: 149

Screen: 150

```
( Quan:   ASSIGN                    )


' ( CFALIT
: ASSIGN [COMPILE] CFALIT ;
IMMEDIATE --> ) (  )

: ASSIGN                  ( -- cfa )
  STATE @
  [COMPILE] [
  [COMPILE] ' CFA SWAP
  IF ]
  ENDIF [COMPILE] LITERAL ;
  IMMEDIATE
                                  -->
```

Screen: 151

```
( Quan:   TO   AT                   )

: TO
  -FIND 0= 0 ?ERROR DROP
  STATE @
  IF ,
  ELSE EXECUTE
  ENDIF ; IMMEDIATE

: AT
  -FIND 0= 0 ?ERROR DROP
  2+ STATE @
  IF ,
  ELSE EXECUTE
  ENDIF ; IMMEDIATE
  ( corrected )                   -->
```

Screen: 152

```
( Quan:   [2@6]   [2!4]             )

ASSEMBLER HEX

LABEL (2@6)
  A0 C, 06 C, B1 C, W  C, 48 C,
  C8 C, B1 C, W  C, 4C C, PUSH ,

LABEL (2!4)
  A0 C, 04 C, B5 C, 00 C, 91 C,
  W  C, C8 C, B5 C, 01 C, 91 C,
  W  C, 4C C, POP ,



                                  -->
```

Screen: 153

```
( Quan:   [2V6]                     )

LABEL (2V6)
  A0 C, 07 C, B1 C, W  C, 48 C,
  88 C, B1 C, W  C, 85 C, W  C,
  68 C, 85 C, W 1+ C,
  A0 C, 00 C, 4C C, W 1- ,








                                  -->
```

Screen: 154

```
( Quan:  patch for CREATE       )

DCX

: (PTCH)                  ( system )
  SWAP >R R = 251 R = 249 R> =
  OR OR ;

: PTCH                    ( system )
  IF [ ' (PTCH) CFA ] LITERAL
  ELSE [ ' = CFA ] LITERAL
  ENDIF
  [ ' CREATE 63 + ] LITERAL ! ;


                                -->
```

Screen: 155

```
( Quan:  QUAN  VECT             )

: QUAN
  ON PTCH LABEL -2 ALLOT
  (2@6) , (2!4) ,
  [ ' VARIABLE 4 + ] LITERAL ,
  2 ALLOT OFF PTCH ;

: VECT
  ON PTCH LABEL -2 ALLOT
  (2V6) , (2!4) ,
  [ ' VARIABLE 4 + ] LITERAL ,
  [ ' NOOP CFA ] LITERAL ,
  OFF PTCH ;
```

Screen: 156

Screen: 157

Screen: 158

Screen: 159

Screen: 160

```
( Utils:  CARRAY  ARRAY         )
BASE @ HEX
: CARRAY          ( cccc, n -- )
  CREATE SMUDGE ( cccc: n -- a )
    ALLOT
  ;CODE CA C, CA C, 18 C,
  A5 C, W C,  69 C, Ø2 C, 95 C,
  ØØ C, 98 C, 65 C, W 1+ C,
  95 C, Ø1 C, 4C C,
  ' + ( CFA @ ) , C;
: ARRAY           ( cccc, n -- )
  CREATE SMUDGE ( cccc: n -- a )
    2* ALLOT
  ;CODE 16 C, ØØ C, 36 C, Ø1 C,
   4C C, ' CARRAY Ø8 + , C;  -->
```

Screen: 161

```
( Utils:  CTABLE  TABLE         )
: CTABLE          ( cccc, -- )
  CREATE SMUDGE ( cccc: n -- a )
  ;CODE
  4C C, ' CARRAY Ø8 + , C;
: TABLE           ( cccc, -- )
  CREATE SMUDGE ( cccc: n -- a )
  ;CODE
  4C C, ' ARRAY ØA + , C;
                                -->
```

Screen: 162

```
( Utils:  2CARRAY  2ARRAY       )
: 2CARRAY      ( cccc, n n -- )
  <BUILDS      ( cccc: n n -- a )
   SWAP DUP , * ALLOT
  DOES>
    DUP >R @ * + R> + 2+ ;
: 2ARRAY       ( cccc, n n -- )
  <BUILDS      ( cccc: n n -- a )
   SWAP DUP , * 2* ALLOT
  DOES>
    DUP >R @ * + 2* R> + 2+ ;
                                -->
```

Screen: 163

```
( Utils:  XC!  X!               )
: XC!    ( nØ...nm cnt addr -- )
  OVER 1- + >R Ø
  DO J I - C!
  LOOP R> DROP ;
: X!     ( nØ...nm cnt addr -- )
  OVER 1- 2* + >R Ø
  DO J I 2* - !
  LOOP R> DROP ;
( Caution: Remember limitation
( on stack size of 3Ø values
( because of OS conflict. )
                                -->
```

Screen: 164

```
( Utils:  CVECTOR  VECTOR       )
: CVECTOR        ( cccc, cnt -- )
  CREATE SMUDGE ( cccc: n -- a )
  HERE OVER ALLOT XC!
  ;CODE
  4C C, ' CARRAY Ø8 + , C;
: VECTOR         ( cccc, cnt -- )
  CREATE SMUDGE ( cccc: n -- a )
  HERE OVER 2* ALLOT X!
  ;CODE
  4C C, ' ARRAY  ØA + , C;
BASE !
```

Screen: 165

Screen: 166

Screen: 167

Screen: 168

Screen: 169

Screen: 170

```
CONTENTS OF THIS DISK:

Disk Operating System:    10 LOAD
DOS system extensions:    50 LOAD
BASIC DOS COMMANDS:       70 LOAD
DOS COMMAND EXTENSIONS: 110 LOAD

(note the packages lower on the
 screen will load all packages
 listed above themselves.)

DISK FORMATTER/COPIERS: 120 LOAD
QUAN STRUCTURES:        150 LOAD
ARRAYS & THEIR COUSINS: 160 LOAD

valDOS FILE EDITOR:    valDOS II
```

Screen: 171

Screen: 172

Screen: 173

Screen: 174

```
( valDOS error messages                              )
Illegal filename
Bad/Mismatched unit(s)
Bad free space map
File already exists
Directory is full
Disk is full
Filename is ambiguous
File does not exist
No room for buffer
End of file encountered
File is not open
Illegal file number
File is locked
Bad argument list
File is open
```

Screen: 175

```
File is too big
File is random
File is not random
No room for random map
Random map is bad
File is a device file
File is not a device file
Illegal access to device file
```

Screen: 176

```
( Error messages                                      )

Stack empty

Dictionary full

Wrong addressing mode

Is not unique

Value error

Disk address error

Stack full
```

Screen: 177

```
Disk Error!

Dictionary too big
```

Screen: 178

```
( Error messages                    )
Use only in Definitions
Execution only
Conditionals not paired
Definition not finished
In protected dictionary
Use only when loading
Off current screen
```

Screen: 179

```
Declare VOCABULARY
```

valDOS II

Supplied Source Listing

LXVI.

```
Screen:  50
( Case Statements:  CASE        )
BASE @ DCX
'( PERMANENT PERMANENT )(  )
: (CASE)
  R C@ MIN -1 MAX 2*
  R 3 + + @EX
  R C@ 2* 5 + R> + >R ;
'( TRANSIENT TRANSIENT )(  )
: CASE
  ?COMP COMPILE (CASE)
  HERE 0 C,
  COMPILE NOOP 6 ;    IMMEDIATE
'( PERMANENT PERMANENT )(  ) -->
```

```
Screen:  51
( Case statements:  CASE        )
: NOCASE
  6 ?PAIRS 7 ;         IMMEDIATE
: CASEND
  DUP 6 =
  IF DROP COMPILE NOOP
  ELSE 7 ?PAIRS
  ENDIF
  HERE 2- @ OVER 1+ !
  HERE OVER -
  5 - 2/ SWAP C! ;     IMMEDIATE
'( PERMANENT PERMANENT )(  )
                               -->
```

```
Screen:  52
( Case statements:  SEL         )
: (SEL)
  R 1+ DUP 2+ DUP R C@
  2* 2* + R> DROP DUP >R SWAP
  DO I @ 3 PICK =
    IF I 2+ SWAP DROP LEAVE THEN
  4 /LOOP SWAP DROP @EX ;
'( TRANSIENT TRANSIENT )(  )
: SEL    ?COMP
  ?LOADING COMPILE (SEL) HERE
  0 C, COMPILE NOOP [COMPILE] [
  8 ; IMMEDIATE
'( PERMANENT PERMANENT )(  ) -->
```

```
Screen:  53
( Case statements:  SEL         )
: NOSEL
  8 ?PAIRS [COMPILE] ' CFA
  OVER 1+ ! 8 ;          IMMEDIATE
: ->
  SWAP 8 ?PAIRS , DUP C@ 1+
  OVER C! [COMPILE] '
  CFA , 8 ;              IMMEDIATE
: SELEND
  8 ?PAIRS
  DROP [COMPILE] ] ;     IMMEDIATE
                               -->
```

```
Screen:  54
( Case statements:  COND        )
'( TRANSIENT TRANSIENT )(  )
: COND
  0 COMPILE DUP ;        IMMEDIATE
: <<
  1+ [COMPILE] IF
  COMPILE DROP ;         IMMEDIATE
: >>
  [COMPILE] ELSE COMPILE
  DUP ROT ;              IMMEDIATE
'( PERMANENT PERMANENT )(  ) -->
```

```
Screen:  55
( Case statements:  COND        )
: NOCOND
  COMPILE 2DROP ;        IMMEDIATE
: CONDEND
  0 DO
    [COMPILE] ENDIF
  LOOP ;                 IMMEDIATE
                               -->
```

```
Screen:  56
 ( Case statements:  CASE:        )

 ' ( PERMANENT PERMANENT ) (   )

 : CASE:
   <BUILDS
     SMUDGE !CSP
     [COMPILE] ]
   DOES>
     SWAP 2* + @EX ;


 BASE !
```

```
Screen:  57
```

```
Screen:  58
```

```
Screen:  59
```

```
Screen:  6Ø
 ( Screen code conversion words )

 BASE @ HEX

 CODE >BSCD              ( a a n -- )
   A9 C, Ø3 C, 2Ø C, SETUP ,
   HERE  C4 C, C2 C, DØ C, Ø7 C,
   C6 C, C3 C, 1Ø C, Ø3 C, 4C C,
   NEXT ,       B1 C, C6 C, 48 C,
   29 C, 7F C, C9 C, 6Ø C, BØ C,
   ØD C, C9 C, 2Ø C, BØ C, Ø6 C,
   18 C, 69 C, 4Ø C, 4C C, HERE
 2 ALLOT 38 C, E9 C, 2Ø C, HERE
 SWAP !  91 C, C4 C, 68 C, 29 C,
   8Ø C, 11 C, C4 C, 91 C, C4 C,
                             -->
```

```
Screen:  61
 ( Screen code conversion words )

   C8 C, DØ C, D3 C, E6 C, C7 C,
   E6 C, C5 C, 4C C, ,
 C;

 CODE BSCD>              ( a a n -- )
   A9 C, Ø3 C, 2Ø C, SETUP ,
   HERE  C4 C, C2 C, DØ C, Ø7 C,
   C6 C, C3 C, 1Ø C, Ø3 C, 4C C,
   NEXT ,       B1 C, C6 C, 48 C,
   29 C, 7F C, C9 C, 6Ø C, BØ C,
   ØD C, C9 C, 4Ø C, BØ C, Ø6 C,
   18 C, 69 C, 2Ø C, 4C C, HERE
 2 ALLOT 38 C, E9 C, 4Ø C, HERE
                             -->
```

Screen: 62

```
( Screen code conversion words )

SWAP !  91 C, C4 C, 68 C, 29 C,
  80 C, 11 C, C4 C, 91 C, C4 C,
  C8 C, D0 C, D3 C, E6 C, C7 C,
  E6 C, C5 C, 4C C, ,
C;

: >SCD                      ( c -- sc )
  SP@ DUP 1 >BSCD ;

: SCD>                      ( sc -- c )
  SP@ DUP 1 BSCD> ;


BASE !
```

Screen: 63

Screen: 64

Screen: 65

Screen: 66

Screen: 67

Screen: 68

Screen: 69

Screen: 70
```
( Utils:  HIDCHR  NOKEY  CURSOR)

BASE @ DCX

'( CASE )( 5Ø LOAD )

: HIDCHR                        ( -- )
  65535 94 ! ;

: NOKEY                         ( -- )
  255 764 C! ;

: CURSOR                        ( f -- )
  Ø= 752 C!
  28 EMIT 29 EMIT ;
                                   -->
```

Screen: 71
```
( Utils:  INKEY$                   )

: (INKEY$)                      ( c -- )
  7Ø2 C! NOKEY Ø ;

: INKEY$                        ( -- c )
  764 C@
  COND
    252 = << 128 (INKEY$) >>
     191 > <<   Ø >>
    188 = <<   Ø >>
    124 = <<  64 (INKEY$) >>
     6Ø = <<   Ø (INKEY$) >>
     39 = <<   Ø NOKEY    >>
  NOCOND KEY
  CONDEND ;                        -->
```

Screen: 72
```
( Utils:  -Y/N                     )

( This Y/N routine accepts upper
  or lower case.  )
HEX

: -Y/N                          ( -- f )
  KEY DF AND
  DUP 59 <>
  ØBRANCH [ ØØ14 , ]
  DUP 4E <>
  ØBRANCH [ ØØØ8 , ]
  2DROP BRANCH [ FFDA , ]
  59 = ;

DCX                                -->
```

Screen: 73
```
( Utils:  Y/N  -RETURN  RETURN )

: Y/N                           ( -- f )
  ."  <Y/N> " -Y/N DUP
  IF 89 ELSE 78 ENDIF
  EMIT SPACE ;

: -RETURN                       ( -- )
  BEGIN
    KEY 155 =
  UNTIL ;

: RETURN                        ( -- )
  ."  <RETURN> " -RETURN ;

BASE !
```

```
Screen: 1ØØ
 ( valDOS file editor 1.Ø          )
 '( FLFL# )( 16Ø LOAD ;S )
 BASE @ DCX
 VOCABULARY EDITOR IMMEDIATE
 EDITOR DEFINITIONS
 '( >BSCD  )( 6Ø LOAD )
 '( HIDCHR )( 7Ø LOAD )
 QUAN XLOC      Ø TO XLOC
 QUAN YLOC      Ø TO YLOC
 QUAN INSRT     Ø TO INSRT
 QUAN LSTCHR    Ø TO LSTCHR
 QUAN ?BUFSM    Ø TO ?BUFSM     -->
```

```
Screen: 1Ø1
 ( valDOS file editor 1.Ø          )
 QUAN ?PADSM    Ø TO ?PADSM
 QUAN ?MULTI    Ø TO ?MULTI
 QUAN ?MARK     Ø TO ?MARK
 QUAN ?UPDAT    Ø TO ?UPDAT
 QUAN TXTBOT    Ø TO TXTBOT
 QUAN TXTEND    Ø TO TXTEND
 QUAN DSPTOP    Ø TO DSPTOP
 QUAN DSPBOT    Ø TO DSPBOT
 QUAN LNCNT     Ø TO LNCNT
 QUAN CURLN     Ø TO CURLN
 QUAN EDFL#     Ø TO EDFL#
 QUAN MEMTOP    Ø TO MEMTOP
 QUAN PTFLG     Ø TO PTFLG
 QUAN ?2BIG     Ø TO ?2BIG      -->
```

```
Screen: 1Ø2
 ( valDOS file editor 1.Ø          )
 LABEL OOPSLN 38 ALLOT
 OOPSLN 38 ERASE
 LABEL EDN1$ 16 ALLOT
 LABEL EDN2$ 16 ALLOT
 LABEL SRCH$ 34 ALLOT Ø SRCH$ C!
 LABEL EDWRK 4Ø ALLOT
 LABEL TABS 4 ALLOT 32 C,
 TABS 4 68 FILL
 36 TABS 4 + C!
 37 CONSTANT 37
 16 CONSTANT #LNS
 4Ø CONSTANT LLEN
 5 LLEN * CONSTANT BLEN         -->
```

```
Screen: 1Ø3
 ( valDOS file editor 1.Ø          )
 HEX
 LABEL TOS1+
  F6 C, ØØ C, DØ C, Ø2 C, F6 C,
  Ø1 C, 6Ø C,
 LABEL TOS1-
  B5 C, ØØ C, DØ C, Ø2 C, D6 C,
  Ø1 C, D6 C, ØØ C, 6Ø C,
 CODE PRVLN               ( a -- a )
  2Ø C, TOS1- , A1 C, ØØ C, FØ C,
  Ø9 C, 2Ø C, TOS1- , A1 C, ØØ C,
  C9 C, 9B C,   DØ C, F3 C, 2Ø C,
  TOS1+ , 4C C, NEXT , C;       -->
```

```
Screen: 1Ø4
 ( valDOS file editor 1.Ø          )
 CODE NXTLN               ( a -- a )
  A1 C,   ØØ C, FØ C, Ø7 C, 2Ø C,
  TOS1+ , C9 C, 9B C, DØ C, F5 C,
  4C C, NEXT , C;
 CODE CRAM              ( A A -- L )
   A9 C, Ø2 C, 2Ø C, SETUP ,
   AØ C, 26 C, 88 C, FØ C, Ø4 C,
   B1 C, N 2+ C, FØ C, -7 C,
   C8 C, 84 C, XSAVE C, A9 C,
   DB C, 91 C, N C, 88 C, 3Ø C,
   Ø7 C, B1 C, N 2+ C, 91 C,
   N  C, 4C C, HERE 8 - , A4 C,
                                -->
```

```
Screen: 1Ø5
 ( valDOS file editor 1.Ø          )
   XSAVE C, C8 C, 98 C,
   4C C, PUSHØA ,
 C;
 CODE UNCRAM            ( a a -- a )
   A9 C, Ø2 C, 2Ø C, SETUP ,
   B1 C, N 2+ C, FØ C, ØD C,
   B1 C, N 2+ C, C9 C, 9B C,
   FØ C, Ø6 C, 91 C, N C,
   C8 C, 4C C, HERE ØA - ,
   C8 C, 18 C, 98 C, 65 C,
   N 2+ C, 48 C, A9 C, ØØ C,
   65 C, N 3 + C, 4C C, PUSH ,
 C;                             -->
```

Screen: 1Ø6

```
( valDOS file editor 1.Ø        )
CODE ?ENUF              ( a -- a cnt )
 84 C, N  C, A1 C, ØØ C, C9 C,
 9B C, DØ C, ØA C, E6 C, N  C,
 A9 C, 1Ø C, C5 C, N  C, FØ C,
 ØE C, DØ C, Ø3 C, ØA C, FØ C,
 Ø9 C, F6 C, ØØ C, DØ C, Ø2 C,
 F6 C, Ø1 C, 4C C, ' ?ENUF 2+ ,
 A5 C, N  C, 4C C, PUSHØA ,
C;
DCX
                                -->
```

Screen: 1Ø7

```
( valDOS file editor 1.Ø        )
: LMOVE                 ( f t -- )
  4Ø CMOVE ;
: BOL                   ( -- a )
  88 @ YLOC 1+ LLEN * + 2+ ;
: SBL                   ( -- a )
  88 @ LLEN #LNS 1+ * + ;
: PBL                   ( -- a )
  PAD 128 + ;
: PBLL                  ( -- a )
  PBL BLEN + LLEN - ;           -->
```

Screen: 1Ø8

```
( valDOS file editor 1.Ø        )
: TOPLN                 ( -- a )
  88 @ LLEN + 2+ ;
: BOTLN                 ( -- a )
  88 @ #LNS LLEN * + 2+ ;
: CURLOC                ( -- a )
  BOL XLOC + ;
: CSHOW                 ( -- a )
  CURLOC DUP C@
  128 OR SWAP C! ;
                                -->
```

Screen: 1Ø9

```
( valDOS file editor 1.Ø        )
: CBLANK                ( -- )
  CURLOC DUP C@
  127 AND SWAP C! ;
: NORPT Ø TO ?MULTI ;   ( -- )
: CRWT                  ( -- )
  CR ." C/R to continue"
  253 EMIT
  BEGIN
    KEY 155 =
  UNTIL
  88 @ 4Ø + 64Ø ERASE ;
                                -->
```

Screen: 11Ø

```
( valDOS file editor 1.Ø        )
: OOPSV                 ( -- )
  BOL OOPSLN 38 CMOVE ;
: ?FULL                 ( -- )
  MEMTOP 1- TXTEND U<
  DUP TO ?2BIG
  IF
    15ØØ Ø
    DO
      Ø 53279 C!
    LOOP
  ENDIF ;
                                -->
```

Screen: 111

```
( valDOS file editor 1.Ø        )
: SAVLN                 ( -- )
  ?MARK ?2BIG NOT AND
  IF Ø TO ?MARK BOL PAD CRAM
    DUP LNCNT <>
    IF CURLN NXTLN DUP 3 PICK
      LNCNT - DUP >R +
      TXTEND 3 PICK - R>
      DUP AT TXTEND +! Ø<
      IF CMOVE
      ELSE <CMOVE ENDIF ?FULL
    ENDIF
    PAD CURLN ROT BSCD>
  ENDIF ;
                                -->
```

Screen: 112
```
( valDOS file editor 1.Ø          )
: DISPLAY                    ( -- )
  TOPLN 2- DSPTOP
  16 Ø
  DO
    PAD 4Ø BLANKS DUP
    PAD 2+ UNCRAM DUP ROT =
    IF
      OVER 4Ø 64 FILL
    ELSE
      PAD 3 PICK LLEN >BSCD
    ENDIF
    SWAP 4Ø + SWAP
  LOOP
  2DROP 1 752 C! ;              -->
```

Screen: 113
```
( valDOS file editor 1.Ø          )
: GETLN                      ( -- )
  CURLN DUP NXTLN
  SWAP - TO LNCNT ;
: GC     GETLN CSHOW ;       ( -- )
: DC     DISPLAY CSHOW ;     ( -- )
: CS     CBLANK SAVLN ;      ( -- )
: HOME               ( dtop -- )
  DUP TO DSPTOP TO CURLN
  Ø TO XLOC Ø TO YLOC
  DISPLAY GC ;                  -->
```

Screen: 114
```
( valDOS file editor 1.Ø          )
: ROLLDN                     ( -- )
  DSPTOP DUP
  ?ENUF 2DROP 2- @
  IF
    CS DSPTOP PRVLN
    DUP TO DSPTOP
    PAD 4Ø BLANKS PAD 2+
    UNCRAM DROP PAD
    TOPLN 2- DUP DUP 4Ø +
    6ØØ <CMOVE LLEN >BSCD
    CURLN PRVLN TO CURLN GC
  ELSE
    NORPT
  ENDIF ;                       -->
```

Screen: 115
```
( valDOS file editor 1.Ø          )
: ROLLUP                     ( -- )
  DSPTOP CS
  ?ENUF SWAP 1+ TO DSPBOT
  16 <> DSPBOT C@ Ø= OR NOT
  IF
    CURLN NXTLN TO CURLN
    TOPLN 2- DUP 4Ø + SWAP 6ØØ
    CMOVE PAD 4Ø BLANKS
    DSPBOT PAD 2+ UNCRAM DROP
    PAD BOTLN 2- LLEN >BSCD
    DSPTOP NXTLN TO DSPTOP
  ELSE
    NORPT
  ENDIF GC ;                    -->
```

Screen: 116
```
( valDOS file editor 1.Ø          )
: UPCUR                      ( -- )
  ?MULTI Ø= YLOC OR
  IF YLOC
    IF
      CS CURLN PRVLN TO CURLN
      -1 AT YLOC +! GC
    ELSE
      ROLLDN
    ENDIF
  ELSE
    NORPT
  ENDIF ;
                                -->
```

Screen: 117
```
( valDOS file editor 1.Ø          )
: DNCUR                      ( -- )
  ?MULTI Ø= YLOC 15 <> OR
  IF YLOC #LNS 1- =
    IF
      ROLLUP
    ELSE
      CURLOC 4Ø + C@ 64 <>
      IF
        CS CURLN NXTLN TO CURLN
        1 AT YLOC +! GETLN
      ENDIF
    ENDIF
  ELSE NORPT
  ENDIF CSHOW ;                 -->
```

Screen: 118

```
( valDOS file editor 1.0          )
: LFCUR                        ( -- )
  XLOC 1- DUP 0<
  IF UPCUR DROP 37 ENDIF
  CBLANK TO XLOC CSHOW ?MULTI
  IF XLOC TO ?MULTI ENDIF ;
: RTCUR                        ( -- )
  XLOC 1+ DUP 37 >
  IF DROP 0 DNCUR ENDIF
  CBLANK TO XLOC CSHOW ?MULTI
  IF XLOC 37 <> TO ?MULTI THEN ;
: EDMRK
  1 TO ?MARK 1 TO ?UPDAT ;    -->
```

Screen: 119

```
( valDOS file editor 1.0          )
: INTGL                        ( -- )
  INSRT NOT TO INSRT NORPT ;
: CLREOL                       ( -- )
  CBLANK OOPSV CURLOC 38 XLOC -
  ERASE CSHOW EDMRK NORPT ;
HEX
CODE 2^                ( n -- 2^n )
  B4 C, 00 C, C8 C, A9 C, 00 C,
  95 C, 00 C, 95 C, 01 C, 38 C,
  36 C, 00 C, 36 C, 01 C, 18 C,
  88 C, D0 C, F8 C, 4C C, NEXT ,
  C; DCX                        -->
```

Screen: 120

```
( valDOS file editor 1.0          )
: BYTINS                       ( -- )
  CBLANK XLOC 37 <
  IF
    CURLOC DUP 1+
    37 XLOC - <CMOVE
  ENDIF
  0 CURLOC C!
  CSHOW EDMRK ;
                                -->
```

Screen: 121

```
( valDOS file editor 1.0          )
: BYTDEL                       ( -- )
  CBLANK
  XLOC 37 <
  IF
    CURLOC DUP 1+ SWAP
    37 XLOC - CMOVE
  ENDIF
  0 CURLOC
  37 XLOC - + C!
  CSHOW EDMRK ;
                                -->
```

Screen: 122

```
( valDOS file editor 1.0          )
: LNINS                        ( -- )
  ?2BIG NOT
  IF
    CS YLOC 15 <
    IF BOL 2- DUP LLEN + #LNS 1-
       YLOC - LLEN * <CMOVE
    ENDIF
    BOL 2- 40 ERASE CURLN
    DUP TXTEND OVER - 1+
    OVER 1+ SWAP <CMOVE 155
    SWAP C! 1 AT TXTEND +!
    GC 1 TO ?UPDAT
  ENDIF
  ?FULL ;                       -->
```

Screen: 123

```
( valDOS file editor 1.0          )
: BONXT                        ( -- )
  CS 0 TO XLOC
  CURLN NXTLN C@ 0= PTFLG OR
  ?2BIG NOT AND
  IF
    CURLN NXTLN
    DUP DUP 1+ TXTEND 3 PICK
    - 2+ <CMOVE 155 SWAP C!
    1 AT TXTEND +!
    DISPLAY ?FULL
    1 TO ?UPDAT
  ENDIF
  DNCUR
  0 TO PTFLG ;                  -->
```

```
Screen: 124
( valDOS file editor 1.Ø       )
: LNDEL                       ( -- )
  YLOC CURLOC 4Ø + C@
  64 <> OR
  IF CBLANK OOPSV
    CURLN DUP NXTLN 2DUP
    SWAP TXTEND 3 PICK - 2+
    CMOVE - AT TXTEND +!
    Ø TO ?MARK GETLN DISPLAY
    CURLOC C@ 64 =
    IF UPCUR NORPT ENDIF
    CSHOW 1 TO ?UPDAT
  ELSE
    NORPT
  ENDIF ;                       -->
```

```
Screen: 125
( valDOS file editor 1.Ø       )
: BFSHW                       ( -- )
  PBLL LLEN 2* 2* -
  SBL LLEN 5 * CMOVE ;
: BFROT                       ( -- )
  PBL DUP BLEN + LMOVE
  PBL DUP LLEN + SWAP
  BLEN LLEN - CMOVE
  PBLL LLEN + PBLL LMOVE
  BFSHW ;
: BFCPY                       ( -- )
  CBLANK BFROT BOL PBLL 2+
  38 CMOVE BFSHW CSHOW ;        -->
```

```
Screen: 126
( valDOS file editor 1.Ø       )
: <BFROT                      ( -- )
  PBLL DUP LLEN + LMOVE
  PBL DUP LLEN +
  BLEN LLEN - <CMOVE
  PBL DUP BLEN +
  SWAP LMOVE BFSHW ;
: BFCLR                       ( -- )
  PBLL LLEN ERASE
  <BFROT ;
: BFLN>                       ( -- )
  LNINS PBLL 2+ BOL 38 CMOVE
  CSHOW <BFROT EDMRK ;          -->
```

```
Screen: 127
( valDOS file editor 1.Ø       )
: >BFNXT BFCPY DNCUR ;        ( -- )
: >BFLN BFCPY LNDEL ;         ( -- )
: BFRPL                       ( -- )
  CBLANK OOPSV
  PBLL 2+ BOL 38 CMOVE EDMRK
  <BFROT CSHOW EDMRK ;
: TABSTP                      ( -- )
  XLOC 8 /MOD TABS +
  SWAP 2^ OVER C@ OR
  SWAP C! NORPT ;
                                -->
```

```
Screen: 128
( valDOS file editor 1.Ø       )
: TABCLR                      ( -- )
  XLOC 8 /MOD TABS +
  SWAP 2^ 255 XOR OVER
  C@ AND SWAP C! NORPT ;
: TAB CBLANK                  ( -- )
  38
  BEGIN
    1- 1 AT XLOC +! XLOC
    38 MOD 8 /MOD TABS +
    C@ SWAP 2^ AND OVER Ø= OR
  UNTIL DROP XLOC
  38 /MOD SWAP TO XLOC
  IF DNCUR ENDIF CSHOW ;        -->
```

```
Screen: 129
( valDOS file editor 1.Ø       )
: RUB                         ( -- )
  XLOC
  IF LFCUR Ø CURLOC C!
     CSHOW EDMRK
  ENDIF
  INSRT IF BYTDEL ENDIF NORPT ;
: BOTSCR                      ( -- )
  15 XLOC - -DUP
  IF Ø DO DNCUR LOOP ENDIF ;
: OOPS                        ( -- )
  LNINS OOPSLN BOL 38 CMOVE
  EDMRK CSHOW NORPT ;           -->
```

```
Screen: 130
( valDOS file editor 1.Ø         )

: PTCHR                      ( -- )
  LSTCHR 13 = ?MULTI 1 <> AND
  IF
    1 TO PTFLG BONXT
  ELSE EDMRK
    INSRT IF BYTINS ENDIF
    LSTCHR >SCD CURLOC C!
    XLOC 37 =
    IF 1 TO PTFLG BONXT
    ELSE RTCUR ENDIF CSHOW
  ENDIF ;

: PRVSCR                     ( -- )
  #LNS Ø DO ROLLDN LOOP ;     -->
```

```
Screen: 131
( valDOS file editor 1.Ø         )

: NXTSCR                     ( -- )
  #LNS Ø DO ROLLUP LOOP ;

: SPLCHR                     ( -- )
  Ø ?MULTI Ø=
  IF 1+ 37 88 @ 25 + C! ENDIF
  TO ?MULTI ;

: MULTI                      ( -- )
  Ø ?MULTI Ø=
  IF 2+ 57 88 @ 25 + C! ENDIF
  TO ?MULTI ;


                                -->
```

```
Screen: 132
( valDOS file editor 1.Ø         )
: SPLT                       ( -- )
  CBLANK BOL EDWRK 38 CMOVE
  XLOC CLREOL 1 TO PTFLG BONXT
  CBLANK TO XLOC EDWRK XLOC +
  BOL XLOC + 38 XLOC - CMOVE
  EDMRK CSHOW ;

: -SPLT                      ( -- )
  CURLN TXTBOT <>
  IF CBLANK BOL EDWRK 38 CMOVE
    LNDEL UPCUR CBLANK OOPSV
    EDWRK XLOC + BOL XLOC +
    38 XLOC - CMOVE CSHOW EDMRK
    EDWRK OOPSLN XLOC CMOVE
  ENDIF NORPT ;               -->
```

```
Screen: 133
( valDOS file editor 1.Ø         )

FORTH DEFINITIONS

: .INFO EDITOR               ( -- )
  CR ." File start addr: "
  TXTBOT DUP U.
  CR ." File end addr: "
  TXTEND 2- DUP U.
  CR ." File count: "
  SWAP - U.
  CR ." Bytes free: "
  MEMTOP TXTEND -
  Ø MAX U. CR ;

EDITOR DEFINITIONS              -->
```

```
Screen: 134
( valDOS file editor 1.Ø         )

: WIPEIT                     ( -- )
  88 @ 4Ø + 64Ø ERASE
  12 84 C! Ø 752 C! CR ;

: FLFL                       ( -- )
  TXTBOT HOME ;

: FLLL                       ( -- )
  TXTEND 2- 15 Ø
  DO PRVLN LOOP HOME BOTSCR ;

: FLEN                       ( -- )
  CURLN NXTLN DUP
  2+ TO TXTEND Ø SWAP ! DC ; -->
```

```
Screen: 135
( valDOS file editor 1.Ø         )
: SRCH
  SRCH$ C@
  IF CS CURLN XLOC + 1+ TXTEND
   OVER - SRCH$ COUNT MATCH SWAP
    IF CURLN XLOC + + DUP
      PRVLN DUP TO CURLN -
      TO XLOC Ø TO YLOC CURLN
      6 Ø DO
        DUP PRVLN SWAP OVER
        <> AT YLOC +!
      LOOP TO DSPTOP GETLN DC
    ELSE DROP WIPEIT CR
      ." Not found" CRWT FLFL
    ENDIF
  ENDIF ;                       -->
```

```
Screen: 136
( valDOS file editor 1.0        )

: GET$                 ( delm -- $ )
  >R PAD 2+
  BEGIN DUP C@ R =
  WHILE 1+ REPEAT DUP
  BEGIN DUP C@ DUP R <>
    SWAP 0# AND
  WHILE 1+ REPEAT OVER - OVER
  1- C! 1- R> DROP ;

: STSRCH                      ( -- )
  ." ? " PAD 31 EXPECT
  CR 88 @ 564 + SRCH$ 1+ CRAM
  1- SRCH$ C! SRCH$ 1+ DUP
  33 BSCD> SRCH ;               -->
```

```
Screen: 137
( valDOS file editor 1.0        )

: INFL DOS                    ( -- )
  INSRT 0 TO INSRT
  BL GET$ (OPEN)
  IF DC LNINS
    BEGIN
      ?TERMINAL ?2BIG OR NOT DUP
      IF DROP DUP (RDB) ENDIF
    WHILE
      DUP 155 =
      IF DROP 13 ENDIF
      TO LSTCHR PTCHR
    REPEAT (CLOSE) DROP
  ELSE CR
    ." Unable to load file"  -->
```

```
Screen: 138
( valDOS file editor 1.0        )
    CRWT WIPEIT DC
  ENDIF TO INSRT ;

: FLST                        ( -- )
  CURLN TXTBOT 2DUP
  TXTEND CURLN - 1+ CMOVE
  - MINUS AT TXTEND +! FLFL ;

: TABRST                      ( -- )
  TABS 4 68 FILL
  36 TABS 4 + C! DC ;

: BADSUB                      ( -- )
  CR ." Bad subcommand"
  CRWT WIPEIT DC ;              -->
```

```
Screen: 139
( valDOS file editor 1.0        )

: SUBCMD                      ( -- )
  CS WIPEIT
  ." : " PAD 32 ERASE
  PAD 31 EXPECT CR PAD @
  SEL ( ST) 21587 -> FLST
      ( EN) 20037 -> FLEN
      ( FL) 19526 -> FLFL
      ( LL) 19532 -> FLLL
      ( IF) 17993 -> INFL
      ( PS) 21328 -> STSRCH
      ( RT) 21586 -> TABRST
                0 -> DC
  NOSEL BADSUB
  SELEND ;                      -->
```

```
Screen: 140
( valDOS file editor 1.0        )

: EDTABT                      ( -- )
  CS WIPEIT ." Abort? "
  +Y/N 19 * DUP TO LSTCHR 0=
  IF DC ELSE ?UPDAT 0=
  IF CURLN TXTBOT - TO ED#IN
  ELSE TXTBOT DUP TO DSPTOP
  TO CURLN 0 TO XLOC 0 TO YLOC
  ENDIF ENDIF NORPT ;

: EXIT  DOS                   ( -- )
  CS WIPEIT 8 84 C!
  .INFO CR ." Save '"
  EDN2$ COUNT TYPE ." ' ? "
  +Y/N 19 * DUP TO LSTCHR      -->
```

```
Screen: 141
( valDOS file editor 1.0        )
  IF EDN2$ (OPEN) 0=
    IF 0 DSKERR FLDNE =
      IF DROP EDN2$ (ENTER)
         DROP EDN2$ (OPEN)
      ENDIF
      0= IF DC ;S ENDIF
    ENDIF
    SWAP DROP TO EDFL#
    TXTBOT TXTEND 2- OVER -
    EDFL# (WRITE) DROP EDFL#
    (ENDF) DROP EDFL# (CLOSE)
    CURLN TXTBOT - TO ED#IN
  ELSE
    DC
  ENDIF ;                       -->
```

Screen: 142

```
( valDOS file editor 1.Ø        )

: CONTROL                ( n -- )
  SEL
     19 -> EXIT     17 -> EDTABT
     28 -> UPCUR    29 -> DNCUR
     3Ø -> LFCUR    31 -> RTCUR
    126 -> RUB     127 -> TAB
      9 -> INTGL   155 -> BONXT
    255 -> BYTINS  254 -> BYTDEL
    157 -> LNINS   156 -> LNDEL
     18 -> BFROT     2 -> <BFROT
      3 -> BFCLR    11 -> >BFNXT
     2Ø -> >BFLN     6 -> BFLN>
     16 -> PRVSCR   14 -> NXTSCR
     15 -> OOPS                 -->
```

Screen: 143

```
( valDOS file editor 1.Ø        )

     27 -> SPLCHR    8 -> CLREOL
     21 -> BFRPL    25 -> MULTI
     24 -> ROLLDN    5 -> ROLLUP
     22 -> SUBCMD   12 -> SRCH
     1Ø -> SPLT      7 -> -SPLT
    159 -> TABSTP 158 -> TABCLR
  NOSEL PTCHR SELEND ;

: 2STROKE                ( -- )
  ?MULTI 1 =
  IF DROP PTCHR
  ELSE
    ?MULTI 2 =
    IF                          -->
```

Screen: 144

```
( valDOS file editor 1.Ø        )

    BEGIN
      8 53279 C! DUP CONTROL
      ?MULTI NOT ?TERMINAL OR
    UNTIL
    DROP ENDIF
  ENDIF
  Ø TO LSTCHR NORPT
  Ø 88 @ 25 + C! ;

: EDTSTP DOS          ( $o $i -- )
  DECIMAL 1 PFLAG !
  PBL BLEN + LLEN + 2+
  TO TXTBOT
  EDN1$ (OPEN) ?SYSERR          -->
```

Screen: 145

```
( valDOS file editor 1.Ø        )

  SWAP DROP TO EDFL#
  Ø TXTBOT 2- ! 155 TXTBOT C!
  TXTBOT MEMTOP OVER - DUP
  <ROT EDFL# (READ)
  IF EDFL# (CLOSE)
    FLTBG CMDERR
  ENDIF
  DSKERR EOFERR = ?SYSERR
  EDFL# (CLOSE)
  #UNTRN - DUP Ø= + TXTBOT +
  DUP 2+ TO TXTEND Ø SWAP !
  Ø TO ED#IN Ø TO ?UPDAT ;

                                -->
```

Screen: 146

```
( valDOS file editor 1.Ø        )

: PROCESS DOS            ( -- )

  Ø GR. 1 752 C! CLS
  112 56Ø @ 6 + C!
  112 56Ø @ 23 + C!
  ."  File: " EDN2$ $.
  28 85 !
  ." #Bufs: " BLEN LLEN / .
  GETLN DC

  PAD ?PADSM OVER TO ?PADSM =
  PBL @ ?BUFSM = AND NOT
  IF PBL BLEN ERASE ENDIF
  BFSHW                         -->
```

Screen: 147

```
( valDOS file editor 1.Ø        )

  BEGIN
    INKEY$ DUP TO LSTCHR -DUP
    IF
      ?MULTI
      IF 2STROKE
      ELSE CONTROL ENDIF
    ELSE
      INSRT
      IF CBLANK ( PAUSE)
        2DUP DROP DROP CSHOW
      ENDIF
    ENDIF
    LSTCHR 19 =
  UNTIL                         -->
```

Screen: 148

```
( valDOS file editor 1.0         )

  0 767 C! 0 752 C!
  2 560 @ 6 + C!
  2 560 @ 23 + C!
  CLS CR
  ." Last edit in: " EDN2$
  $. CR .INFO CR
  PBL @ TO ?BUFSM
  DSKFLS ;
                                 -->
```

Screen: 149

```
( valDOS file editor 1.0         )

FORTH DEFINITIONS

: EDIT DOS                  ( -- )
  GETARGS ?WRGARG
  TOPOM 40 - EDITOR TO MEMTOP
  DOS 44 ( ",") GETARG 0=
  IF DUP ENDIF TO N1$ TO N2$
  0 (CLOSE) N1$ (?OPEN)
  IF
    0= FLOPN ?CMDERR
  ELSE
    DSKERR FLDNE = ?SYSERR
                                 -->
```

Screen: 150

```
( valDOS file editor 1.0         )

    CR N1$ $.
   ."  does not exist, create? "
    +Y/N NOT CR IF ;S ENDIF
    N1$ (ENTER) ?SYSERR
  ENDIF
  UNIT DIRTBL C@ ENTRY 1+ @
  N2$ (?OPEN)
  IF
    0= FLOPN ?CMDERR
    UNIT DIRTBL C@ ENTRY 1+ @
  ELSE
    DSKERR FLDNE = ?SYSERR 0
  ENDIF -
                                 -->
```

Screen: 151

```
( valDOS file editor 1.0         )

  UNIT FSMGET
  FSMAP 3 + @ SWAP - DUP 20 <
  IF CR
    ." Warning, disk space " 0>
    IF ." low"
    ELSE ." empty" ENDIF
    ." , edit?"
    +Y/N NOT CR IF ;S ENDIF
  ELSE DROP ENDIF
  UNIT DIRTBL C@ ENTRY C@ 32 AND
  IF CR
   ." File is locked, edit? "
   +Y/N NOT CR IF ;S ENDIF
  ENDIF                          -->
```

Screen: 152

```
( valDOS file editor 1.0         )

  N1$ EDITOR EDN1$ OVER
  C@ 1+ 16 MIN CMOVE
  DOS N2$ EDITOR  EDN2$ OVER
  C@ 1+ 16 MIN CMOVE
  EDTSTP 0 TO INSRT
  TXTBOT DUP TO DSPTOP TO CURLN
  0 TO XLOC 0 TO YLOC GETLN
  CR .INFO CRWT PROCESS ;

FORTH
                                 -->
```

Screen: 153

```
( valDOS file editor 1.0         )

: LL EDITOR                 ( -- )
  EDN2$ EDN1$ 16 CMOVE
  PAD ?PADSM <>
  IF 0 TO XLOC 0 TO YLOC PBL
  BLEN + LLEN + 2+ DUP TO TXTBOT
  DUP TO DSPTOP TO CURLN
  GETLN ENDIF EDTSTP PROCESS ;

: WHERE DOS                 ( -- )
  FLFL# FLBUF@ -DUP 0=
  IF
    CR ." No error on record..."
    CR QUIT
  ENDIF                          -->
```

Screen: 154

```
( valDOS file editor 1.Ø         )

  Ø (CLOSE) 14 + @
  FLNME EDITOR EDN1$ 16 CMOVE
  DOS FLNME EDITOR EDN2$
  16 CMOVE EDTSTP
  1- TXTBOT + DUP PRVLN
  SWAP OVER - 1- TO XLOC
  DUP TO CURLN Ø TO YLOC 6 Ø
  DO
    DUP PRVLN
    SWAP OVER <> AT YLOC +!
  LOOP
  TO DSPTOP
  1 TO INSRT
  PROCESS ;                        -->
```

Screen: 155

```
( valDOS file editor 1.Ø         )

: #BUFS EDITOR              ( n -- )
  5 MAX 32Ø MIN LLEN *
  ' BLEN ! Ø TO ?PADSM
  PBL BLEN + LLEN + 2+
  DUP TO TXTBOT DUP
  TO DSPTOP TO CURLN
  Ø TO XLOC Ø TO YLOC ;

FORTH

BASE !
```

Screen: 156

Screen: 157

Screen: 158

Screen: 159

Screen: 160

```
( file editor load message      )

CLS
CR CR CR
." valDOS and the basic DOS" CR
." commands must be loaded"  CR
." before the file editor"   CR
." is compiled."             CR
CR
." Insert the valDOS I disk" CR
." and load the necessary"   CR
." routines."                CR
CR CR
FLUSH
```

Screen: 161

Screen: 162

Screen: 163

Screen: 164

Screen: 165

Screen: 166

```
 6
```

Screen: 167

Screen: 168

Screen: 169

Screen: 170

```
CONTENTS OF THIS DISK:

CASE STATEMENTS:            5Ø LOAD
SCREEN CODE CONVERSION:     6Ø LOAD
KEYSTROKE WORDS:            7Ø LOAD
DOS FILE EDITOR 1.Ø:       1ØØ LOAD

 **** CAUTION **** CAUTION ****

 This is a DOS format disk
 with screens 5Ø-79 and 1ØØ-179
 locked out for FORTH source
 code. Do not store FORTH
 code on screens that are not
 locked out!
```

Screen: 171

Screen: 172

Screen: 173

Screen: 174

```
( valDOS error messages             )
Illegal filename
Bad/Mismatched unit(s)
Bad free space map
File already exists
Directory is full
Disk is full
Filename is ambiguous
File does not exist
No room for buffer
End of file encountered
File is not open
Illegal file number
File is locked
Bad argument list
File is open
```

Screen: 175

```
File is too big
File is random
File is not random
No room for random map
Random map is bad
File is a device file
File is not a device file
Illegal access to device file
```

Screen: 176

```
( Error messages                        )

Stack empty

Dictionary full

Wrong addressing mode

Is not unique

Value error

Disk address error

Stack full
```

Screen: 177

```
Disk Error!

Dictionary too big
```

```
Screen: 178
( Error messages                    )
Use only in Definitions
Execution only
Conditionals not paired
Definition not finished
In protected dictionary
Use only when loading
Off current screen
```

```
Screen: 179
Declare VOCABULARY
```

VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141

valFORTH

T.M.

SOFTWARE SYSTEM

for ATARI*

PLAYER-MISSILE GRAPHICS

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

valFORTH T.M.
SOFTWARE SYSTEM

PLAYER-MISSILE GRAPHICS

Stephen Maguire

valFORTH T.M.

PLAYER-MISSILE GRAPHICS

Version 1.0
April 1982

The following is a description of commands used in creating seemingly difficult video displays using players and missiles. Used alone or in combination with the other available systems by Valpar International, it is possible to obtain graphic displays which compare with those of the best arcade games. The use of players and missiles (also called "player/missiles") allows the beginner to create high quality moving video displays.

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

PLAYER/MISSILE GRAPHICS PACKAGE

XXI. PLAYER/MISSILE GRAPHICS

a) STROLLING THROUGH PLAYER/MISSILE GRAPHICS 1
b) PLAYER/MISSILE GLOSSARY
   1) ENABLING PLAYER/MISSILE GRAPHICS 6
   2) CREATING PLAYERS AND MISSILES 10
   3) MOVING/PLACING PLAYERS AND MISSILES 12
   4) SETTING PLAYER/MISSILE BOUNDARIES 14
   5) COLLISIONS BETWEEN PLAYERS AND MISSILES 16

XXII. CHARACTER EDITOR
User's manual for the character set editor.

XXIII. SOUND EDITOR
Description of the audio-palette sound editor.

XXIV. PLAYER/MISSILE SUPPLIED SOURCE

As knowledge of the internal workings of player/missile graphics is not necessary to use this valFORTH package effectively, the internal workings are not explained in this manual. However, for the serious programmer trying to optimize his/her program in every way, an understanding of these internal workings could at times improve code efficiency and/or speed of execution. For a complete explanation of player/missile graphics at the nut-and-bolt level, see the series of articles by Dave and Sandy Small in Creative Computing.

STROLLING THROUGH PLAYER/MISSILE GRAPHICS

One of the biggest differences between the Atari graphic capabilities and those of most other computers is the Atari's ability to use players and missiles. This discussion will not explain the internal workings of player/missile graphics on the Atari; rather, it will explain how to use the basic commands in this valFORTH package. Before we proceed, please load the player/missile graphic routines from the Player/Missile disk. The directory on screen 170 will show what screen to load. Also, if you have the valFORTH Editor/Utilities package, load in the high speed STICK command found in the Miscellaneous Utilities; otherwise, load in the slower version on your Player/Missile disk. (Check the directory for its location).

To start with, let's get a simple player up on the screen to experiment with. First we must initialize the player/missile graphic system and design the player's image. This is simple:

```
1  PMINIT                  ( Initialize for single
                             resolution players )

2  BASE  !                 ( Change to binary for ease )

LABEL CROSS                ( Give the player image a name )
00011000  C,
00011000  C,
00011000  C,
11111111  C,               ( A large plus sign )
11111111  C,
00011000  C,
00011000  C,
00011000  C,

DECIMAL                    ( Now back into base 10 )

PMCLR                      ( Clear player/missile memory )

ON PLAYERS                 ( Turn on the players )

CROSS  8  180  50  0  BLOPLY      ( Build a player )
```

You should now see the cross in the upper right-hand corner of the video screen. Now let's take a look at this and see how it works.

First, players are initialized using the PMINIT command. Players can be in either a single or double resolution mode (double res players are twice as tall). "1 PMINIT" is used for single res players. If we had wanted double res players, we would have used "2 PMINIT".

Next, the player image is created. Since it is much easier to make player images as 1's and 0's, we use binary (base two) number entry. Before we design the image, it must be given a name. The LABEL command does this nicely for us.

This image is named CROSS. All that need be done now is to draw the picture. Notice how easy it is to see the image when using base two. Of course, we could have stayed in base 10 and still designed the image, but this is usually more difficult. The word C, after each number simply tells FORTH to store that number in the dictionary. Once the picture is designed, we return to decimal for ease.

Both the PMCLR and ON PLAYERS commands are fairly self-descriptive: PMCLR erases all players and missiles so that no random trash appears when the PLAYERS are turned ON. Next, the BLDPLY (build player) command takes the image named CROSS which is 8 bytes tall and assigns it to player 0 at horizontal location 180 and vertical location 50 on the display. Of course, we could have built player 1, 2, or 3 instead.

The cross should be black. Suppose we wanted a blue or green cross instead. This can be done using the PMCOL (player/missile color) command. Try this:

```
0  9  8  PMCOL            ( player  hue  lum  PMCOL )
```

The cross should now appear blue. This command assigns a BLUE (9) hue with a luminance of 8 to player 0. If the color commands are loaded from the valFORTH disk,

```
0  BLUE  8  PMCOL
```

could have been used with the same results. Try changing the color of the player to GREEN (12) or PINK (4). Note that the default colors for players 2 and 3 make them invisible: Their colors should be set immediately upon being built.

Now that we have a player on the screen, let's move it around. We use the PLYMV (player move) command for this. PLYMV needs to know which player to move (there could be as many as five), how far to move it in the horizontal direction, and how far to move it in the vertical direction. Try this:

```
1  1  0  PLYMV          ( horz  vert  player  PLYMV )
```

This moves player 0 down 1 line and right one horizontal position, thus giving the effect of a diagonal move towards the lower right-hand corner. Try these as well:

```
 1   0  0  PLYMV        ( move right one position )
-5   0  0  PLYMV        ( move left five positions )
 0  20  0  PLYMV        ( move down 20 lines )
 0 -15  0  PLYMV        ( move up 15 lines )
-5   2  0  PLYMV        ( move left five, and down two )
```

That's all there is to moving a player. Positive horizontal offsets move the player right, and negative values move the player left. Likewise, positive vertical offsets move the player down while negative ones move the player up. The following program can be typed in and you will have a joystick controlled player:

```
: JOY
  BEGIN
    0 STICK              ( STICK leaves two offsets )
    0 PLYMV              ( for PLYMV to use. )
    ?TERMINAL
  UNTIL ;
```

```
JOY <ret>
```

Move the player with stick 0, the left-most stick port. Press any console button to exit the program.

Currently, if the player is moved off any edge, it "wraps" to the opposite side. In other words, we have an "unbound" player. This is rarely desirable. Normally, we want to restrict player movement to certain boundaries. The PLYMV command has a built in boundary check routine specifically for this reason. Right now, new boundaries are set so wrapping occurs. Let's set some boundaries:

```
60  150  50  200  0  PLYBND
```

This sets the boundaries of player zero to 75 on the left, 150 on the right, 50 on the top, and 200 on the bottom. Type JOY again to verify that you can no longer move freely about the display. Try different boundary settings and experiment to get the feel of the command. Boundary checking can be disabled for any or all of the edges. Setting the left or upper boundary to 0 will disable the check on that edge, likewise, 255 in either the right or lower boundary will do the same.

Let's build another player in the lower right-hand corner of the screen. This time, instead of designing the player ourself, let's borrow the image from the standard Atari character set stored in ROM. The image of the digit zero starts at address 57472. The other numbers follow zero. Try this:

```
57472  16  160  150  1  BLDPLY
```

You should now see the numbers 0 and 1 on your screen. This command builds player 1 with the image at address 57472 that is 16 bytes tall and puts it at horizontal position 160 and vertical position 150. Give this player a color if you want.

Until now, we have been using normal size players. It is possible to make the two players on the display different widths using the PLYWID command. PLYWID expects a width specification of 0 or 2 (normal), 1 (double), or 3 (quadruple). Its command form is:

```
width   player   PLYWID
```

Thus,

```
3  1  PLYWID
```

should make player one four times its original size. The same can be done with player zero:

```
3  0  PLYWID
```

Type JOY again and notice that the width has no effect on movement whatsoever. Also notice that player one is unaffected by movement of player zero.

Now that we have two players on the screen, let's interface both of them to the joystick. Type in the following program:

```
: JOY2
  BEGIN
    0 STICK          ( Record stick movement )
    2DUP             ( Make a copy )
    0 PLYMV          ( Move player 0 )
    SWAP             ( Rotate stick 90 degrees )
    1 PLYMV          ( Move player 1 )
    ?TERMINAL
  UNTIL ;

JOY2 <ret>
```

Notice that when you push the stick up, player zero goes up, but player one moves left. The SWAP instruction exchanges the vertical and horizontal offsets from STICK before moving player one. If we were to take the SWAP out, the players would move identically.

In many applications, it is necessary to know when a player has hit another player or some background image. Fortunately, the Atari computer automatically makes this information available. An entire collection of valFORTH words allows checking of all collisions possible. The most general word is ?COL which simply returns a true flag if anything has hit anything else. Here is an example:

```
: BUMP
  BEGIN
    HITCLR
    0 STICK
    0 PLYMV
    ?COL
    IF
      CR ." oops!"
    ENDIF
    ?TERMINAL
  UNTIL ;

BUMP <ret>
```

Move the player around and watch the results. Every time you hit any letters or player one, the word "oops!" should be printed out. This program is quite simple. First, the HITCLR command is issued which erases any old collision information. If this command were omitted, the first time a collision occurred, "oops!" would be continuously printed out. Next the joystick is read and the player moved. If the player touches anything when moved, the collision registers are set. ?COL reads these registers and leaves a true flag if the player has hit something, and the IF statement will then print out "oops!".

Type JOY again and notice that the width has no effect on movement whatsoever. Also notice that player one is unaffected by movement of player zero.

Now that we have two players on the screen, let's interface both of them to the joystick. Type in the following program:

```
: JOY2
  BEGIN
    0 STICK              ( Record stick movement )
    2DUP                 ( Make a copy )
    0 PLYMV              ( Move player 0 )
    SWAP                 ( Rotate stick 90 degrees )
    1 PLYMV              ( Move player 1 )
    ?TERMINAL
  UNTIL ;

JOY2 <ret>
```

Notice that when you push the stick up, player zero goes up, but player one moves left. The SWAP instruction exchanges the vertical and horizontal offsets from STICK before moving player one. If we were to take the SWAP out, the players would move identically.

In many applications, it is necessary to know when a player has hit another player or some background image. Fortunately, the Atari computer automatically makes this information available. An entire collection of valFORTH words allows checking of all collisions possible. The most general word is ?COL which simply returns a true flag if anything has hit anything else. Here is an example:

```
: BUMP
  BEGIN
    HITCLR
    0 STICK
    0 PLYMV
    ?COL
    IF
      CR ." oops!"
    ENDIF
    ?TERMINAL
  UNTIL ;

BUMP <ret>
```

Move the player around and watch the results. Every time you hit any letters or player one, the word "oops!" should be printed out. This program is quite simple. First, the HITCLR command is issued which erases any old collision information. If this command were omitted, the first time a collision occurred, "oops!" would be continuously printed out. Next the joystick is read and the player moved. If the player touches anything when moved, the collision registers are set. ?COL reads these registers and leaves a true flag if the player has hit something, and the IF statement will then print out "oops!".

Using other commands found in the glossary, we can tell specifically what the player has hit. For example, the ?PXPF command checks to see if a specific player has hit a playfield, and if so, it returns information indicating which playfield.

Although this discussion was limited to using players, the routines for missiles function similarly and can be found in the following glossary. Two player/missile example programs can be found on your Player/Missile disk. These demonstrate how short player/missile routines can be.

PLAYER/MISSILE GLOSSARY

Enabling Player-Missile Graphics

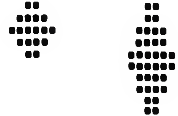
To make use of players and missiles, the video processor must be activated. Players can be several sizes, they can have different overlap priority schemes, and they can have different colors. The following collection of "words" makes this setup task quite simple. Note: Players and missiles are numbered 0 through 3. The fifth player is numbered as four.

(PMINIT) (addr res ---)

The (PMINIT) command (or PMINIT below) must be used to initialize the player missile routines before any other player missile command may be used. (PMINIT) expects both the address of player/missile memory and a 1 or a 2 indicating whether single or double resolution is desired.

NOTE: The difference between single and double resolution is shown graphically below:

Player as defined in memory:	single res on screen:	double res on screen:
00011000 00111100 01111110 00111100 00011000		



PMINIT (res ---)

The PMINIT command functions identically to the (PMINIT) command above, except that no address need be given. PMINIT calculates an address based on the current graphic mode. It uses the first unused 2K block of memory below the highest free memory (i.e., below the display list). This should only be used while first learning the system, after that, (PMINIT) should be used to optimize memory utilization. Note that the variable PMBAS contains the calculated address upon return.

PMBAS (--- addr)

A variable containing the address of player/missile memory. This value must lie on a 2K boundary if single resolution players are used and on a 1K boundary if double resolution players are used. This is set using the (PMINIT) command and is automatically set by the PMINIT command described above. This value should never be set directly, but can be read at any time.

PLAYERS (ON/OFF ---)

If the flag found on the top of the stack equates to TRUE or ON, then the player/missiles are activated. This does not clear out player missile memory; therefore, the PMCLR command described below is usually used prior to enabling the players and missiles to ensure that no random trash appears on the screen.

If the flag found on the top of the stack equates to FALSE or OFF, then the player/missile graphic mode is de-activated. Turning players off does not clear player-missile memory; therefore, a subsequent ON PLAYERS command would redisplay any previously defined players and missiles. If players are already disabled, the command is ignored.

5THPLY (flag ---)

In many applications it is desirable to combine the four missiles and simulate a fifth player, thus giving five players (numbered 0-4), and no missiles. If the flag on the stack is non-zero, then the fifth player mode will be initiated; otherwise, the missile mode will be re-activated.

Normally, missiles take on the color of their corresponding players; however, when a fifth player is asked for, all missiles take on the common color of playfield #3. In addition, it also allows the fifth player to be treated exactly as any other player would be treated. Bear in mind that although it is called a "fifth" player, its reference number is four (4). The fifth player is "built" with missile zero on the right, and missile three on the left:

|m3|m2|m1|m0| = fifth player

(Note: For convenience, the words ON and OFF have been defined to allow niceties such as:

```
ON 5THPLY
OFF 5THPLY
```

These two words are recognized by all words that require an ON/OFF type indication.)

PLYCLR (pl# ---)

Few applications use all available players. To keep these unused players from displaying trash, they can be cleared of all data by using the PLYCLR command. The PLYCLR command expects the player number on the top of the stack and fills the specified player with zeroes. This command can be used to "turn off" players which are no longer needed.

MSLCLR (ml# ---)

The MSLCLR command is very much like the PLYCLR command, described above, except that it clears the specified missile. In addition, this can be used when the fifth player is activated to erase parts of the fifth player for special effects.

PMCLR (---)

This command clears all players and all missiles. This is generally used just prior to activating the player-missile graphic mode to ensure that no random trash is placed on the video screen. PMCLR expects no values on the stack, nor does it leave any.

MCPLY (F ---)

The MCPLY (Multi-Color Player) command expects one value on the top of the stack. If this value is 0 or OFF, then the multi-color player mode is disabled. If this value is 1 or ON, this command instructs the video processor to logically "or" the bits of the colors of player zero with player one, and also of player two with player three. In other words, when players 0 and 1 overlap (or players 2 and 3), a third color (determined by the colors of the overlapping players) will be assigned to the overlapped region rather than assigning one of the players a higher priority. Since players must be one color, this allows for multi-colored players. For example:

```
  Player 0          Player 1          MCPlayer
 Pink color        Blue color         Pink/blue
   ( 4 )             ( 8 )            ( 4 OR 8
                                      = green )

                     BBBB               BBBB
                   BBBBBBBB           BBBBBBBB
 PPPPPPPP                             PPPPPPPP
 PPPPPPPP          BB    BB           PGGPPGGP
 PPPPPPPP                             PPPPPPPP
  PP  PP                               PP  PP
   PPPP                                 PPPP
```

NOTE: The lums of the two players are also OR'd.

PRIOR (n ---)

The PRIOR command expects one value on the top of the stack. This value must be 8, 4, 2, or 1, otherwise unpredictable video displays may occur. PRIOR instructs the video processor as to what has higher priority for a video location on the screen. For example, it will determine whether a plane (a player) will pass in front of a building (a playfield), or whether the plane will pass behind the building. Objects with higher priorities will appear to pass in front of those with lower priorities. The following table shows the available priority settings:

n=8	n=4	n=2	n=1
PF0	PF0	PL0	PL0
PF1	PF1	PL1	PL1
PL0	PF2	PF0	PL2
PL1	PF3*	PF1	PL3
PL2	PL0	PF2	PF0
PL3	PL1	PF3*	PF1
PF2	PL2	PL2	PF2
PF3*	PL3	PL3	PF3*
BAK	BAK	BAK	BAK

* PF3 and PL4 share the same priority

Objects higher on the list will appear to pass in front of objects lower on the list.

CREATING PLAYERS AND MISSILES

Once the player/missile graphics system has been activated and the priorities set, all that need be done is to create the players themselves. Normally, this would be quite difficult to do; however, using the commands and designing techniques described below, this task is made very simple. There are really only three things to do in the creation of a player: setting the width size, setting the color, and creating the picture.

PLYWID (width pl# ---)

The PLYWID command sets the specified player to the desired width. Players are numbered 0, 1, 2, 3, or in the case of the fifth player, 4. Legal widths are:

```
image:     10111101

0  =  normal width:      ■ ■■■■ ■
1  =  double width:      ■■  ■■■■■■■■  ■■
2  =  normal width:      ■ ■■■■ ■
3  =  quad. width:       ■■■■    ■■■■■■■■■■■■■■■■    ■■■■
```

Any other value may cause strange results.

MSLWID (size ml# ---)

The MSLWID command is identical to the PLYWID command described above except that it is used to set the size of the missiles. The same size values apply also. The MSLWID command should only be used when in the missile mode (i.e., with the fifth player deactivated).

PMCOL (pl# hue lum ---)

To set the color (hue and lum) of a player, the PMCOL (Player-Missile-Color) command is used. It sets the specified player to the hue and lumina desired. Note that there is no corresponding command to set the colors of missiles as missiles take on the colors of their respective players. To set the color of the 5th player, "pl#" should be 4. If the color words on the valFORTH 1.1 disk are loaded, they can be used to set player colors:

```
0 BLUE 8 PMCOL
```

This sets player #0 to a medium blue color.

BLDPLY (addr len horz vert pl# ---)

The BLDPLY command is probably the most useful of all the commands in this graphic package. It takes an easily predefined picture that resides in memory at address "addr" whose length is "len" and converts it to the specified player "pl#". It then positions the player at the coordinates (horz,vert). The player is then ready to be moved about the screen using the PLYMV command described below.

As an example, a player in the form of an arrow pointing upward will be created, assuming that priorities and such have already been taken care of. Practice has proven that the following method is easiest for creating players:

```
2 BASE  !           ( put into binary mode   )

LABEL PICTURE       ( the image is named PICTURE )
  00011000 C,
  00111100 C,
  01111110 C,
  11011011 C,
  00011000 C,
  00011000 C,
  00011000 C,
  00011000 C,
DECIMAL

1 PMINIT            ( initialize for single resolution )
PICTURE  8  80  40  0  BLDPLY
```

Takes the image at location PICTURE which is 8 bytes long, and builds player #0 at location (80,40).

BLDMSL (addr len horz vert ml# ---)

The BLDPLY command described above does just about everything necessary to create a high-resolution player. The BLDMSL command functions identically to the BLDPLY command except that it is used for setting up missiles (which are in effect just skinny players). The method for creating players can be used for creating missiles as well. Note that if the fifth player mode is activated, the BLDPLY command must be used to create the player.

Building missiles takes a bit more care than building players. Players occupy separate memory, while the four missiles share the same memory. Each missile is two bits wide; all four together are exactly a byte wide. Missile memory is shared with the two lowest bits devoted to missile zero, and the two highest bits devoted to missile three:

| m3 | m3 | m2 | m2 | m1 | m1 | m0 | m0 |

All players with the same shape can use the same image without any problem since they all are a full byte wide. Missiles, however, cannot use the same shape since their images must be ORed into missile memory. This means that the missile images must be in the proper bit columns. For example, the same image for separate missiles could be:

11000000	00110000	00001100	00000011
11000000	00110000	00001100	00000011
11000000	00110000	00001100	00000011
msl#3	msl#2	msl#1	msl#0

PUTTING PLAYERS AND MISSILES IN THEIR PLACE

Generally, once a player or missile has been created and put to the video screen, it is moved around. This can be accomplished very easily with the next set of words. Interfacing a movable player with the joystick can improve just about any program which requires input. As a result, it usually gives the program a more professional appearance.

PLYLOC (pl# --- horz vert)

The PLYLOC command (PLaYer LOCation) returns the vertical and horizontal positions of the specified player. This is normally used when a joystick/button setup is being utilized -- i.e., when a joystick is moving a player and the button is used to pinpoint where the player is. A program which draws lines between two dots could use this. The joystick is used to move the player to the desired spot on the screen. Pressing the button tells the program that a selected spot has been made. Once a second spot has been selected, the program then draws a line between them.

MSLLOC (ml# --- horz vert)

The MSLLOC command performs the same function as the PLYLOC command described above except that it is used to find locations of missiles instead of players. Note that using MSLLOC on a fifth player gives meaningless results.

PLYMV (horz vert pl# ---)

The PLaYer MoVe command moves the specified player the direction specified by "vert" and "horz". If "vert" or "horz" is negative, the player is moved up or left respectively, otherwise it is moved down or right unless they happen to be zero in which case nothing happens. The following examples clarify this:

```
 0 -5  0 PLYMV    ( Move player 0 up 5 lines )
-1 -1  3 PLYMV    ( Move player 3 left and up one line )
 3 -1  2 PLYMV    ( Move player 2 up one dot and right 3 )
```

MSLMV (horz vert ml# ---)

The MSLMV is identical in function as the PLYMV command described above except that it is used to move missiles about the video screen.

PLYPUT (horz vert pl# ---)

The PLYPUT command positions player "pl#" to the location (horz,vert) on the video screen.

PLYCHG (addr len pl# ---)

Oftentimes it is necessary to change the image of a player after it has been built. The PLYCHG command allows this to be easily done. The PLYCHG command takes the image with length "len" at address "addr" and assigns it to player "pl#". Note that if the new image is shorter than the previous one, part of the previous image will remain. This can be overcome by executing a PLYCLR command prior to PLYCHG.

PLYSEL (addr # pl# --)

The PLYSEL command is used to select image "#" out of a table of images of the same length and assigns that image to the specified player. PLYSEL is typically used to animate players. An example usage of this can be found in Player/Missile Example #2 found in the directory of the disk.

PLAYER/MISSILE BOUNDARIES

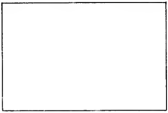
It is often desirable to put limitations on the movements of players and missiles. Boundaries can be set up for each player and missile independently and upon each move command, they will remain within those boundaries. Additionally, a boundary status byte for each player is available for scrutiny at any time. This section explains how this is used.

PLYBND (left right top bottom pl# --)

In most applications, the movements of players are kept within certain boundaries. The PLYBND command frees the user from having to worry about boundary checking. This command expects the player number and all four boundaries. Whenever a PLYMV is then used, the player is always kept within the set boundaries. Also, upon each move a boundary status byte is left in the c-array PLYSTT (see ?PLYSTT below). The edge boundaries of the screen are:

32 for single, 16 for double

48 for both resolutions



207 for both resolutions

223 for single, 111 for double

Note that in special cases the boundary checker will fail. If the left boundary is 0 and the player is at the boundary, any move left will not be checked as expected. For example, if it were moved left by one position (-1), the new horizontal position would be -1 or FFFF in hex. Since only 8 bit unsigned comparisons are made, the horizontal position appears to be 255 (FF hex). Post calculating boundary checking turns out to be more useful because it allows any or all edges to be unbounded. If an unbounded player is desired, use this:

```
0 255 0 255 pl# PLYBND
```

For an example of PLYBND, see the example program found in the directory on screen 170 of your disk.

MSLBND (left right top bottom ml# --)

The MSLBND command is the same as the PLYBND command above, except that it is used for missiles. Upon each move a boundary status byte is left in the array MSLSTT. See ?MSLSTT below.

?BND (--- n)

This command leaves the boundary check status of the last PLYMV or MSLMV performed. The value has the following form:

0	0	...	0	l	r	t	b
15	14		4	3	2	1	0

Only the lower four bits are of use. Each bit represents a different edge. If the bit is set, then the player or missile has attempted to move beyond that boundary. Note that only two of the four bits can be set at any time.

Note:

```
DECIMAL
...
?BND  3  AND
IF  hit-vertical-boundary ENDIF
?BND  12  AND
IF  hit-horizontal-boundary ENDIF
...
```

?PLYSTT (pl# --- val)

Given a player number, returns the boundary check byte of that player. This byte is the status byte for the most recent PLYMV of that player. See ?BND above for the description of the status byte.

?MSLSTT (ml# --- val)

Given a missile number, returns the boundary check byte of that missile. This byte is the status byte for the most recent MSLMV of that missile. See ?BND above for the description of the status byte.

CHECKING FOR INTERACTION BETWEEN PLAYERS

All the commands given so far allow the creation of any player or missile desired. But once that player is on the screen and moving around, it is often necessary to know when two or more objects (players, missiles, and playfields) touch or "crash" into each other. This remaining collection of commands allows checking of all possible "hit" combinations.

?COL (--- f)

The ?COL command is a very general collision detector. It does nothing more than indicate whether two or more objects have "crashed" -- it does not give any indication of what has collided. It leaves a 1 on the stack if a collision has taken place; otherwise it leaves a zero.

?MXPF (m1# --- n)

The ?MXPF command is a much more specific collision detection command. It stands for "?collision of Missile #X with any PlayField". It is used to check if a specific missile has hit any playfield. It returns a zero if no collision has taken place, and leaves an 8, 4, 2, 1, or combinations of these (e.g., 12 = 8+4) if a collision has occurred. Each of these four basic values represents a specific playfield:

3 ?MXPF (Has missile #3 hit any playfields?)

TOS	binary	meaning of val
0	0000	no collisions
1	0001	with pf#0
2	0010	with pf#1
3	0011	with pf#0,1
4	0100	with pf#2
5	0101	with pf#2,0
6	0110	with pf#2,1
7	0111	with pf#2,1,0
8	1000	with pf#3
9	1001	with pf#3,0
10	1010	with pf#3,1
11	1011	with pf#3,1,0
12	1100	with pf#3,2
13	1101	with pf#3,2,0
14	1110	with pf#3,2,1
15	1111	with pf#3,2,1,0

To test for a collision with one specific playfield, use one of the following:

1 ANO	(Leaves 1 if collision with pf#0, else 0)
2 AND	(" 1 " " pf#1, " 0)
4 ANO	(" 1 " " pf#2, " 0)
8 ANO	(" 1 " " pf#3, " 0)

?PXPF (pl# --- n)

The ?PXPF command (?collision of Player #X with any PlayField) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions with players and playfields instead of missiles and playfields.

?MXPL (ml# --- n)

The ?MXPL command (?collision of Missile #X with any Player) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions between missiles and players. Note that it is impossible for a missile to collide with a fifth player since it would be, in effect, colliding with itself.

?PXPL (pl# --- n)

The ?PXPL command (?collision of Player #X with any other players) behaves in exactly the same manner as the ?MXPF command above except that it tests for collisions between players. Note that it is impossible for a player to collide with itself.

HITCLR (---)

The HITCLR command clears all collision registers. In other words, it sets the collision monitor to a state which indicates that no collisions have occurred.

THE CHARACTER SET EDITOR

Character Sets

Whenever the computer has to display a character on the video screen, it must refer to a table which holds the shape definition for that character. By changing this table, new character sets can be formed.

The shape of a single character in the table (or character set) is made up of 8 bytes of data. A character is one byte wide and 8 bytes tall forming an 8 by 8 bit matrix. If a bit in this matrix is set (1), then a dot will appear on the screen. If a bit is reset (0), nothing is displayed. For example, the letter I could be defined as:

	00000000	$00 = 0
●●●●●●	01111110	$7E = 126
●●	00011000	$18 = 24
●●	00011000	$18 = 24
●●	00011000	$18 = 24
●●	00011000	$18 = 24
●●●●●●	01111110	$7E = 126
	00000000	$00 = 0

Thus, the sequence 0, 126, 24, 24, 24, 24, 126, 0, represents the letter I. The entire alphabet is constructed in this fashion. By selectively setting the bit pattern, custom made characters can be formed. This can find many uses. A British character set can be made by changing the one character "#" to the British monetary symbol. Likewise, a Japanese character set could be made by replacing the lowercase characters with Katakana letters.

Another use would be to design special symbol sets. For example, an entire set could be devoted to special mathematical symbols such as plus-minus signs, square-root signs integration signs, or vector signs. (Although this would be of little use in normal operation where character sets cannot be mixed on the same line, using the high resolution text output routines in the Editor/Utilities package. It becomes easy to mix character sets in this fashion.) Assuming the character sets were defined, it would be possible to have a Japanese quotation (in kana of course) embedded within the text of a mathematical explanation of some kind all on the same line!

A final use for custom character sets is for "map-making." Characters can be designed so that they can be pieced togehter to form a picture. An excellent example of this can be found in Cris Crawford's Eastern Front game available through the Atari Program Exchange. When done properly, the final "puzzle" will appear as though it is a complicated high resolution picture.

Now, on to the editor...

The Editor

The following description explains how to use the character editor found on the Player/Missile disk. This editor allows a character set to be designed and then saved on disk for later modification or use. A copy of the standard character has already been saved and can be located through the directory on screen 170.

After loading the character editor, it is executed by typing:

CHAR-EDIT <ret>

The screen has an 8 by 8 grid in the upper-lefthand corner. On the right side there is a command list, and at the bottom, a section is reserved to display the current character set.

The Commands:

I) The joystick
A joystick in port 0 (the leftmost port) is used to move the character cursor (the solid circle) within the 8 by 8 grid. The cursor indicates where the next change to the current character will be made.

II) The button
When pressed, the joystick button will toggle the bit under the character cursor in the 8 by 8 grid. If the bit is set (on), it will be reset. If the bit is reset (off), it will be set. The character will be updated in the character set found at the bottom of the screen.

III) "1" command
By pressing the "1" the current character is cleared in both the grid and in the character set at the bottom of the display. There is no verify prompt for this command.

IV) "2" command
By pressing the "2" key the current character and character set are cleared. User verification is required before any action is taken.

V) "3" command
By pressing the "3" key the current character is saved to disk. User verification is required with a yes/no response. If a yes response is given, a screen number is asked for and the current character set is saved on the specified screen. The current character is not destroyed upon a save.

VI) "4" command
By pressing the "4" key a character set is loading from disk, destroying the current character set. User verification is required with a yes/no response. If a yes response is given, a screen number is asked for and a character set loaded from the specified screen.

VII) "<-- " and "-->" commands

These two arrow keys move the character pointer through the character set to allow modification of any character in the current set.

VIII) Console key

Pressing any console key terminates the edit session and returns control to the FORTH system. The current character set is lost unless it is saved to disk prior to ending the session.

Loading Character Sets

The following three words allow easy use of custom character sets.

CHLOAD (addr scr# cnt ---)

The CHLOAD command takes the first "cnt" characters on screen "scr#" and stores them consecutively starting at address "addr". Each screen (in half-K mode) will only hold 64 character definitions. If "cnt" is greater than 64, CHLOAD will continue loading from the next screen. Many character sets could be loaded at one time by giving a very large "cnt" value. Besides being able to load a full set, the CHLOAD command allows the building of a new set from several other sets.

Note that if a 20 character/line mode is being used, "addr" should lie on a half-K boundary (only upper 7 bits significant). If a 40 character/line mode is being used, "addr" should lie on an 1K boundary (only upper 6 bits significant). Also note that PAD is modified by CHLOAD.

SPLCHR (addr ---)

The SPLCHR commands activates the character set at the address specified.

NMLCHR (---)

The NMLCHR command re-activates the normal character set.

AUDIO-PALETTE -- A SOUND EDITOR

Audio-Palette is a sound editor which generates all possible time-independent sounds that the Atari 400/800 microcomputer can produce. Each of the four channels are interfaced to one of the four joystick ports. The joysticks allow the setting of the pitch (horizontal) the distortion (vertical) of their corresponding channel. When the joystick button is pushed, the sound is made. To get a better idea of how this works, load the editor (see screen 170) and type:

AUDED <ret>

The screen should clear and a table of values should appear at the bottom of the display. In the upper lefthand corner of the screen, there should be four numerals (players) overlayed (one for each channel). Each of these players can be moved around the display by using a joystick in the appropriate port.

As a player is moved vertically, the distortion changes. As a player is moved horizontally, the pitch changes. By pressing the button, a sound will be made according to the current frequency (pitch), distortion, volume, and audio control settings. To increase the volume, the up-arrow is used. Any time the up-arrow is pressed, all channels whose corresponding joystick buttons are pressed will have their volumes increased. Likewise, the down-arrow will decrease the volumes.

Each bit of the audio control value performs some function in the sound generator. The bits are numbered 0 to 7. Pressing the keys 0 to 7 will toggle the corresponding bits in the audio control register. For a description of these bit settings, please refer to the explanation of SOUND in the valFORTH 1.1 package.

XXIV. PLAYER/MISSILE SUPPLIED SOURCE

```
Screen:  30
( PlyMsl:  arrays and variables)
BASE @
DCX '( ARRAY )( 80 KLOAD )
0 VARIABLE PMBAS
5    CARRAY PLYVRT
5    CARRAY PLYHRZ
5    CARRAY PLYLEN
5      ARRAY PLYADR
4    CARRAY MSLVRT
4    CARRAY MSLHRZ
4    CARRAY MSLLEN
4      ARRAY MSLADR
5      ARRAY PMADR
0 VARIABLE PMLEN
0 VARIABLE PMRES
0 VARIABLE MSLSZ                  ==>
```

```
Screen:  31
( PlyMsl:  arrays and variables)
0 VARIABLE BOUNDS 34 ALLOT
5    CARRAY PLYSTT
4    CARRAY MSLSTT
0 VARIABLE BNDCOL
2 VARIABLE 5THWID
CTABLE 5THDAT
2 C, 4 C, 2 C, 8 C,
HEX
CTABLE MSLDAT
FC C, F3 C, CF C, 3F C,
                                  -->
```

```
Screen:  32
( PlyMsl:  [PMINIT]              )
: (PMINIT)      ( addr  res  -- )
  SWAP PMBAS ! 1- DUP PMRES !
  NOT 10 * 0C OR
  22F C@ EF AND OR 22F C!
  PMBAS @ 180 PMRES @
  NOT 1+ >R
  R * + DUP 4 PMADR !
  80 R> * >R
  R + DUP 0 PMADR !
  R + DUP 1 PMADR !
  R + DUP 2 PMADR !
  R +     3 PMADR !
  R> PMLEN ! ;
                                  ==>
```

```
Screen:  33
( PlyMsl:  PMINIT  PLAYERS      )
: PMINIT                  ( res -- )
  2E6 C@ 8 - F8 AND
  OVER 1- 4 * + 100 *
  SWAP (PMINIT) ;
: PLAYERS                 ( f -- )
  IF
    PMBAS @ DUP
    PMRES @ 1+ (PMINIT)
    SP@ 1+ C@ SWAP
    DROP D407 C!
    SGRCTL @ 3 OR DUP
    SGRCTL ! D01D C!
  ELSE                              -->
```

```
Screen:  34
( PlyMsl:  5THPLY               )
    SGRCTL @ FC AND
    DUP SGRCTL ! D01D C!
    22F C@ E3 AND 22F C!
    D00D 5 ERASE
  ENDIF ;
: 5THPLY                  ( f -- )
  26F C@ SWAP
  IF 10 OR
  ELSE EF AND
  ENDIF
  26F C! ;
                                  ==>
```

```
Screen:  35
( PlyMsl:  PMCLR  PLYCLR        )
: PMCLR                    ( -- )
  4 PMADR @
  PMLEN @ 5 *
  0 FILL ;
: PLYCLR                ( pl# -- )
  PMADR @
  PMLEN @
  0 FILL ;
                                  -->
```

Screen: 36

```
( PlyMsl:  MSLCLR PRIOR          )
: MSLCLR                ( ml# -- )
  4 PMADR @ DUP
  PMLEN @ + SWAP
  DO
    DUP MSLDAT C@
      I C@ AND I C!
  LOOP
  DROP ;
: PRIOR                 ( n -- )
  26F C@ 0F0 AND
  OR 26F C! ;
                               ==>
```

Screen: 37

```
( PlyMsl:  PLYMV                 )
CODE PLYMV
 84 C, N 6 + C, B5 C, 00 C,
 0A C, A8 C, B9 C, 0 PMADR 1+ ,
 85 C, N 1+ C, B9 C, 0 PMADR ,
 85 C, N 1- C,  B9 C, 0 PLYADR ,
 85 C, N 2+  C, B9 C, 0 PLYADR
 1+  , 85 C, N 3 + C, B4 C, 0 C,
 B9 C, 0 PLYLEN , 85 C, N 4 + C,
 B9 C, 0 PLYHRZ , 18 C, 75 C,
 04 C, D9 C, BOUNDS , B0 C, 5 C,
 B9 C, BOUNDS , E6 C, N 6 + C,
 06 C, N 6 + C, D9 C, BOUNDS 5 +
 , F0 C, 07 C, 90 C, 05 C, B9 C,
 BOUNDS 5 + , E6 C, N 6 + C, -->
```

Screen: 38

```
( PlyMsl:  PLYMV                 )
99 C, 0 PLYHRZ , 95 C, 05 C,
B9 C, 0 PLYVRT , 85 C, N C,
18 C, 75 C, 2 C, 06 C, N 6 + C,
D9 C, BOUNDS A + , B0 C, 05 C,
B9 C, BOUNDS A + , E6 C, N 6 +
C, 6 C, N 6 + C, D9 C, BOUNDS
F + , F0 C, 07 C, 90 C, 05 C,
B9 C, BOUNDS F + , E6 C, N 6 +
C, 99 C, 0 PLYVRT , 95 C, 3 C,
38 C, E5 C, N C, B0 C, 05 C,
A5 C, N C, 38 C, F5 C, 03 C,
95 C, 02 C, C5 C, N 4 + C,
90 C, 02 C, A5 C, N 4 + C,
85 C, N 5 + C,
                               ==>
```

Screen: 39

```
( PlyMsl:  PLYMV                 )
A5 C, N C, D5 C, 03 C, 90 C,
08 C, 18 C, 65 C, N 4 + C, 38
C, E5 C, N 5 + C, 85 C, N C,
18 C, 65 C, N 1- C, 85 C, N C,
B5 C, 2 C, F0 C, 0B C, A0 C,
00 C, 98 C, 88 C, C8 C, 91 C,
 N C, C4 C, N 5 + C, D0 C,
F9 C, B5 C, 00 C, C9 C, 04 C,
D0 C, 14 C, B5 C, 05 C, A0 C,
04 C, HERE 88 C, 30 C, 0A C,
99 C, D004 , 18 C, 6D C, 5THWID
, 4C C, , 4C C, HERE 2 ALLOT
B5 C, 05 C, B4 C, 00 C, 99 C,
D000 , HERE SWAP ! B4 C, 00 C,
A5 C, N 6 + C,                 -->
```

Screen: 40

```
( PlyMsl:  PLYMV                 )
99 C, 0 PLYSTT , 8D C, BNDCOL ,
B5 C, 3 C, 18 C, 65 C, N 1- C,
85 C, N C, A0 C, 00 C,
B1 C, N 2+ C,
91 C, N C, C8 C, C4 C, N 4 + C,
D0 C, F7 C, E8 C, E8 C,
4C C, POPTWO ,    C;
                               ==>
```

Screen: 41

```
( PlyMsl:  MSLMV                 )
HEX
CODE MSLMV
 84 C, N 6 + C, B5 C, 0 C, 0A C,
 A8 C, AD C, 4 PMADR 1+ , 85 C,
 N 1+ C, AD C, 4 PMADR , 85 C,
 N 1- C, B9 C, 0 MSLADR , 85 C,
 N 2+ C, B9 C, 0 MSLADR 1+ ,
 85 C, N 3 + C, B4 C, 0 C, B9 C,
 0 MSLDAT , 85 C, N 7 + C, B9 C,
 0 MSLLEN , 85 C, N 4 + C, B9 C,
 0 MSLHRZ , 18 C, 75 C, 04 C,
 D9 C, BOUNDS 14 + , B0 C, 5 C,
 B9 C, BOUNDS 14 + , E6 C, N 6 +
                               -->
```

Screen: 42
```
( PlyMsl:  MSLMV                    )
C, 6 C, N 6 + C, D9 C, BOUNDS
18 + , F0 C, 07 C, 90 C,
05 C, B9 C, BOUNDS 18 + ,
E6 C, N 6 + C,
99 C, 0 MSLHRZ , 95 C, 05 C,
B9 C, 0 MSLVRT , 85 C, N  C,
18 C, 75 C, 02 C, 6 C, N 6 + C,
D9 C, BOUNDS 1C + , B0 C, 5 C,
B9 C, BOUNDS 1C + , E6 C, N 6 +
C, 06 C, N 6 + C, D9 C, BOUNDS
20 + , F0 C, 7 C, 90 C, 5 C,
B9 C, BOUNDS 20 + , E6 C, N 6 +
C, 99 C, 0 MSLVRT , 95 C, 3 C,
                                 ==>
```

Screen: 43
```
( PlyMsl:  MSLMV                    )
38 C, E5 C, N C, B0 C, 5 C, A5
C, N C, 38 C, F5 C, 3 C, 95 C,
2 C, C5 C, N 4 + C, 90 C, 2 C,
A5 C, N 4 + C, 85 C, N 5 + C,
A5 C, N C, D5 C, 3 C, 90 C,
8 C, 18 C, 65 C, N 4 + C, 38
C, E5 C, N 5 + C, 85 C, N C,
18 C, 65 C, N 1- C, 85 C, N C,
A0 C, FF C, C8 C, B1 C, N C,
25 C, N 7 + C, 91 C, N C, C4
C, N 5 + C, D0 C, F5 C, B5 C,
5 C, B4 C, 0 C, 99 C, D004 ,
                                 -->
```

Screen: 44
```
( PlyMsl:  MSLMV                    )
  B4 C, 0 C, A5 C, N 6 + C, 99
  C, 0 MSLSTT , 8D C,
  BNDCOL , B5 C, 3 C, 18 C,
  65 C, N 1- C, 85 C, N C,
  A0 C, 00 C, B1 C, N C,
  25 C, N 7 + C, 11 C, N 2+ C,
  91 C, N C, C8 C,
  C4 C, N 4 + C, D0 C, F3 C, E8
  C, E8 C, 4C C, POPTWO , C;
                                 ==>
```

Screen: 45
```
( PlyMsl:  BLDPLY  BLDMSL          )
: BLDPLY       ( a l h v pl# -- )
  >R                  R PLYVRT C!
  R PLYHRZ C!     R PLYLEN C!
  R PLYADR  !   ( R PLYCLR )
  0 0 R> PLYMV ;

: BLDMSL       ( a l h v pl# -- )
  >R                  R MSLVRT C!
  R MSLHRZ C!    R MSLLEN C!
  R MSLADR  ! ( R MSLCLR )
  0 0 R> MSLMV ;
                                 -->
```

Screen: 46
```
( PlyMsl:  PLYCHG PLYSEL PLYPUT)
: PLYCHG         ( a len pl# -- )
  >R R PLYLEN C!
  R PLYADR !
  0 0 R> PLYMV ;

: PLYSEL            ( a # pl# -- )
  >R R PLYLEN C@ * +
  R PLYLEN C@ R> PLYCHG ;

: PLYPUT            ( h v pl# -- )
  >R R PLYVRT C@ -
  SWAP R PLYHRZ C@ -
  SWAP R> PLYMV ;
                                 ==>
```

Screen: 47
```
( PlyMsl:  PLYWID                   )
CODE PLYWID
  B5 C, 00 C, C9 C, 04 C, F0 C,
  09 C, A8 C, B5 C, 02 C, 99 C,
  D008 , 4C C, HERE 2 ALLOT
  A8 C, A0 C, 04 C, 0A C, 0A C,
  15 C, 02 C, 88 C, D0 C, F9 C,
  8D C, MSLSZ , 8D C, D00C ,
  B4 C, 02 C, B9 C, 0 5THDAT ,
  85 C,  N C, 8D C, 5THWID ,
  AD C, 4 PLYHRZ , A0 C, 04 C,
  HERE 88 C, 30 C, 09 C, 99 C,
  D004 , 18 C, 65 C, N C, 4C C,
  , HERE SWAP ! 4C C, POPTWO ,
C;                               -->
```

Screen: 48

```
( PlyMsl:  MSLWID                    )
CODE MSLWID
  B4 C, 00 C, B9 C, 0 MSLDAT ,
  2D C, MSLSZ , HERE
  88 C, 30 C, 7 C, 16 C, 02 C,
  16 C, 02 C, 4C C, , 15 C,
  02 C, 8D C, MSLSZ , 8D C,
  D00C , 4C C, POPTWO ,
C;
                                  ==>
```

Screen: 49

```
( PlyMsl:  PLYLOC MSLLOC MCPLY )
CODE PLYLOC          ( pl# -- h v )
 94 C, 01 C, B4 C, 0 C,
 B9 C, 0 PLYHRZ , 95 C, 0 C,
 B9 C, 0 PLYVRT , 4C C, PUSH0A ,
CODE MSLLOC          ( ml# -- h v )
 94 C, 01 C, B4 C, 0 C,
 B9 C, 0 MSLHRZ , 95 C, 0 C,
 B9 C, 0 MSLVRT , 4C C, PUSH0A ,
: MCPLY                     ( f -- )
  26F C@ SWAP
  IF 20 OR ELSE DF AND ENDIF
  26F C! ;                        -->
```

Screen: 50

```
( PlyMsl:  ?COL HITCLR ?MXPF...)
CODE ?COL                  ( -- f )
 CA C, CA C, 98 C, A0 C, 0F C,
 19 C, D000 , 88 C, 10 C, FA C,
 C8 C, 94 C, 01 C, 95 C, 00 C,
 4C C, ' 0# ( CFA @ ) ,
C;
CODE ?MXPF              ( ml# -- n )
  B4 C, 00 C,  B9 C, D000 ,
  4C C, PUT0A ,  C;
CODE ?PXPF              ( pl# -- n )
  B4 C, 00 C,  B9 C, D004 ,
  4C C, PUT0A ,  C;               ==>
```

Screen: 51

```
( PlyMsl:  ?MXPL ?PXPL PLYBND  )
CODE ?MXPL              ( ml# -- n )
  B4 C, 00 C,  B9 C, D008 ,
  4C C, PUT0A ,  C;
CODE ?PXPL              ( pl# -- n )
  B4 C, 00 C,  B9 C, D00C ,
  4C C, PUT0A ,  C;
CODE HITCLR                  ( -- )
  8C C, D01E , 4C C, NEXT , C;
CODE ?BND               ( xl# -- n )
  AD C, BNDCOL ,
  4C C, PUSH0A ,  C;              -->
```

Screen: 52

```
( PlyMsl:  MSLBND  ?BND         )
CODE ?PLYSTT            ( pl# -- n )
  B4 C, 00 C, B9 C, 0 PLYSTT ,
  4C C, PUT0A ,  C;
CODE ?MSLSTT            ( ml# -- n )
  B4 C, 00 C, B9 C, 0 MSLSTT ,
  4C C, PUT0A ,  C;
: PLYBND        ( l r t b pl# -- )
  )R 4 ROLL )R
  (ROT SWAP R) R)
  BOUNDS + 14 O+S
  DO I C! 5 /LOOP ;               ==>
```

Screen: 53

```
( PlyMsl:  PMCOL                )
: MSLBND        ( l r t b ml# -- )
  )R 4 ROLL )R
  (ROT SWAP R) R)
  BOUNDS + 14 + 10 O+S
  DO I C! 4 /LOOP ;
: PMCOL         ( pl# col lum -- )
  SWAP 10 * +
  SWAP DUP 4 =
  IF
    DROP 2C7 C!
  ELSE
    2C0 + C!
  ENDIF ;                         -->
```

Screen: 54

```
( PlyMsl: initialization       )

DCX

BOUNDS        36   0 FILL
BOUNDS  5 +   5 255 FILL
BOUNDS 15 +   5 255 FILL
BOUNDS 24 +   4 255 FILL
BOUNDS 32 +   4 255 FILL

0 PLYSTT 5 ERASE
0 MSLSTT 4 ERASE

1 PMINIT      ( Set up defaults )

BASE !
```

Screen: 55

Screen: 56

Screen: 57

Screen: 58

Screen: 59

```
Screen: 60
( Audio Editor                    )
BASE @ DCX
'( PLYMV )( 15 KLOAD )
'( SOUND )( 83 KLOAD )
'( STICK )( 84 KLOAD )
VOCABULARY AUDPAL IMMEDIATE
AUDPAL DEFINITIONS
4 CARRAY PIT
4 CARRAY VOL
4 CARRAY DST
0 VARIABLE ACTL                ==>
```

```
Screen: 61
( Audio Editor                    )
HEX
 CTABLE TBL
 32 C, 1F C, 1E C, 1A C, 18 C,
 1D C, 1B C, 33 C, 0F C, 0E C,
DCX
: WPIT                ( pl# -- )
  10 OVER 20 + POS. PIT C@
  3 .R ;
: WDST                ( pl# -- )
  16 OVER 20 + POS. DST C@
  2 .R ;
                               -->
```

```
Screen: 62
( Audio Editor                    )
: WVOL                ( pl# -- )
  20 OVER 20 +
  POS. VOL C@ 2 .R ;
: WACTL                   ( -- )
  28 21 POS. BASE C@ ACTL C@
  DUP DUP 3 .R 2 BASE C!
  26 22 POS. 0
  (# # # # # # # # # #) TYPE
  FILTER! BASE C! ;
                               ==>
```

```
Screen: 63
( Audio Editor                    )
HEX
: SETP                    ( -- )
  2 PMINIT PMCLR 1 PRIOR
  0  3 ( RDORNG ) 6  PMCOL
  1  8 ( BLUE   ) 6  PMCOL
  2  4 ( PINK   ) 8  PMCOL
  3  1 ( GOLD   ) 6  PMCOL
  4 0
  DO
   1 I PLYWID
   E080 I 8 * + 8 37 15 I
   BLDPLY
  LOOP
  ON PLAYERS ;
DCX                            -->
```

```
Screen: 64
( Audio Editor                    )
: INIT                    ( -- )
  0 GR. 1 752 C! CLS 3 19 POS.
  ." Chan  Freq  Dist "
  ." Vol    AUDCTL"
  4 0
  DO
   8 I VOL C!
   0 I PIT C!
   0 I DST C!
   CR I 3 SPACES . I WPIT
   I WDST I WVOL
  LOOP
  0 ACTL ! WACTL SETP ;
                               ==>
```

```
Screen: 65
( Audio Editor                    )
: SND               ( pl# f -- )
  IF
   >R R R PIT C@ R DST C@
   R> VOL C@ SOUND
  ELSE
   XSND
  ENDIF ;
HEX
CODE DIG                ( n -- n )
  B5 C, 00 C, 94 C, 00 C,
  94 C, 01 C, 38 C, A8 C,
  36 C, 00 C, 36 C, 01 C,
  88 C, D0 C, F9 C, 4C C,
  NEXT , C; DCX                -->
```

Screen: 66

```
( Audio Editor                    )

: VOLUPD                ( n -- )
  4 0
  DO
   I STRIG
   IF
    DUP I VOL C@ + 0 MAX 15 MIN
    I VOL C! I WVOL
   ENDIF
  LOOP
  DROP ;



                                 ==>
```

Screen: 67

```
( Audio Editor                    )

: AKEY            ( -- n tf / ff )
  0 764 C@ DUP 255 <>
  IF
   255 764 C!
   10 0
   DO
    DUP I TBL C@ =
    IF
     DROP NOT I SWAP 0 LEAVE
    ENDIF
   LOOP
  ENDIF
  DROP ;
                                 -->
```

Screen: 68

```
( Audio Editor                    )

: ?AKEY                    ( -- )
  AKEY
  IF
   DUP 8 <
   IF
    ACTL C@ SWAP 1+ DIG XOR
    ACTL C! WACTL
   ELSE
    9 = 2* 1- VOLUPD
   ENDIF
  ENDIF ;


                                 ==>
```

Screen: 69

```
( Audio Editor                    )

: PDADJ       ( hrz vrt pl# -- )
  >R -DUP
  IF 2* R DST C@ +
   0 MAX 14 MIN R DST C!
   R WDST
  ENDIF
  -DUP
  IF I PIT C@ +
   0 MAX 255 MIN R PIT C!
   R WPIT
  ENDIF
  R> DROP ;

                                 -->
```

Screen: 70

```
( Audio Editor                    )

: DIGMV                  ( pl# -- )
  >R R PIT C@ 2/ 55 +
  R DST C@ 4 * 21 +
  R> PLYPUT ;









                                 ==>
```

Screen: 71

```
( Audio Editor      AUDED         )

FORTH DEFINITIONS

: AUDED                    ( -- )
  AUDPAL INIT
  BEGIN 4 0
   DO
    I STICK I PDADJ
    I DIGMV I I STRIG SND
   LOOP
   ?AKEY ?TERMINAL
  UNTIL
  OFF PLAYERS 0 752 C!
  0 0 POS. XSND4 ;
BASE ! FORTH
```

Screen: 72
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 73
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 74
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 75
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 76
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 77
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 78
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 79
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 80
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 81
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 82
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 83
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 84

Screen: 85

Screen: 86

Screen: 87

Screen: 88

Screen: 89

```
Screen: 90
( Charedit: var defs           )
BASE @ DCX
'( POS. )( : POS. 84 C! 85 ! ; )
'( STICK )( 84 KLOAD )
VOCABULARY CHREDT IMMEDIATE
CHREDT DEFINITIONS
 0 VARIABLE HORZ
 0 VARIABLE VERT
 0 VARIABLE CHAR#
 0 VARIABLE CURLOC
 0 VARIABLE DEFLOC
 0 VARIABLE TPTR
 0 VARIABLE CSET-LOC          ==)
```

```
Screen: 91
( Charedit                     )
: POSCUR               ( n n -- )
  SWAP CURLOC @
  DUP C@ 84 -
  SWAP C! 40 * + 203 +
  88 @ + DUP C@
  84 + OVER C!
  CURLOC ! ;
: CLICK                   ( -- )
  0 53279 C!
  8 53279 C! ;
                              --)
```

```
Screen: 92
( Charedit                     )
HEX
: ANTIC                 ( f -- )
  22F C@ SWAP
  IF 20 OR ELSE DF AND ENDIF
  22F C! ;
CODE CHSB0            ( b -- n )
 B4 C, 00 C, C8 C, A9 C, 00 C,
 95 C, 00 C, 95 C, 01 C, 38 C,
 36 C, 00 C, 36 C, 01 C, 18 C,
 88 C, D0 C, F8 C, 4C C, NEXT ,
C;
: CHSB1             ( n b -- f )
  CHSB0 AND 0# ; DCX          ==)
```

```
Screen: 93
( Charedit                     )
'( NFLG --) )(  )
0 VARIABLE NFLG
: -NUMBER            ( addr -- d )
  BEGIN DUP C@ BL = DUP + NOT
  UNTIL 0 NFLG ! 0 0 ROT DUP 1+
  C@ 45 = DUP )R + -1
  BEGIN DPL ! (NUMBER) DUP C@
  DUP BL () SWAP 0# AND
  WHILE DUP C@ 46 - NFLG !
  0 REPEAT DROP R) IF DMINUS
  ENDIF NFLG @ IF 2DROP ENDIF
  NFLG @ NOT NFLG ! ;
                              --)
```

```
Screen: 94
( Charedit                     )
: DSPCHR                  ( -- )
  88 @ 203 + CURLOC ! DUP 320 +
  SWAP
  DO
   I 8 0 DO
    0 OVER C@ 7 I - CHSB1
    IF 128 + ENDIF
    CURLOC @ C! 1 CURLOC +!
   LOOP
   DROP 32 CURLOC +! 40
  +LOOP 0 0 VERT ! HORZ ! 88 @
  203 + DUP DUP CURLOC ! C@
  84 + SWAP C! ;
                              ==)
```

```
Screen: 95
( Charedit                     )
: GRAFC                   ( -- n )
  88 @ 882 + ;
: GR8                     ( -- n )
  88 @ 802 + ;
: SCR/W              ( n n n -- )
  SWAP B/SCR * OFFSET @ +
  DUP 4 + SWAP
  DO
   2DUP I SWAP R/W
   SWAP 128 + SWAP
  LOOP
  2DROP ;                     --)
```

```
Screen:  96
 ( Charedit                          )
 HEX
 CODE CHSB2                 ( n -- n )
   B5 C, 00 C, 94 C, 00 C,
   94 C, 01 C, 38 C, A8 C,
   36 C, 00 C, 36 C, 01 C,
   88 C, D0 C, F9 C, 4C C,
   NEXT , C;
 DCX

 : MPTRR                          ( -- )
   TPTR @ 0 OVER C! 1+ DUP
   GR8 2- 33 + U>
   IF 32 - ENDIF
   DUP TPTR ! 93 SWAP C! CLICK ;
                                    ==>
```

```
Screen:  97
 ( Charedit                          )

 : MPTRL                          ( -- )
   TPTR @ 0 OVER C! 1-
   DUP GR8 U<
   IF
    32 +
   ENDIF
   DUP TPTR !
   93 SWAP C!
   CLICK ;




                                    -->
```

```
Screen:  98
 ( Charedit                          )

 HEX
 : DBMAKE                         ( -- )
   OFF ANTIC 58 @ 300 - DUP
   58 ! FF00 AND DUP 230 !
   DUP 3 70 FILL
   3 + DUP 42 SWAP C!
   1+   DUP 58  @ SWAP !
   2+   DUP 15 2 FILL
   15 + DUP 12 F FILL
   12 + DUP 41 SWAP C!
   1  + 230 @ SWAP !
   ON ANTIC ;
 DCX
                                    ==>
```

```
Screen:  99
 ( Charedit                          )

 : PTCST                   ( scr# -- )
   PAD CSET-LOC !
   GRAFC DUP 320 + SWAP
   2 0 DO
    32 0 DO
     DUP DUP 320 + SWAP DO
      I C@ CSET-LOC @ C!
      1 CSET-LOC +!
     40 /LOOP
    1+ LOOP
   DROP
  LOOP
  PAD SWAP 0 SCR/W ;
                                    -->
```

```
Screen: 100
 ( Charedit                          )

 : GTCST                   ( scr# -- )
   GRAFC PAD ROT 1 SCR/W
   PAD CSET-LOC ! 2 0
   DO
    32 0 DO
     DUP DUP 320 + SWAP DO
      CSET-LOC @ C@ I C!
      1 CSET-LOC +!
     40 /LOOP
    1+ LOOP
   288 + LOOP DROP GRAFC DUP
   DEFLOC ! DSPCHR 0 CHAR# !
   GR8 DUP 0 TPTR @ C! 12 14 POS.
   0 . 93 SWAP C! TPTR ! ;      ==>
```

```
Screen: 101
 ( Charedit                          )

 : GETSCR                  ( -- scr# )
   BEGIN
    18 14 POS. ." Screen #: "
    PAD  5 EXPECT PAD 1- -NUMBER
    DROP 128 17 C! 1 752 C!
    18 14 POS. 16 SPACES NFLG @
    IF
     DUP 1 ( OVER 179 ) OR
     ?1K IF OVER 89 ) OR ENDIF
     IF DROP 0 ELSE 1 ENDIF
    ELSE DROP 0
    ENDIF
   UNTIL
   DUP 13 15 POS. 3 .R ;       -->
```

```
Screen: 102
( Charedit                              )

: VFIO                         ( -- f )
  KEY 89 = 18 14 POS.
  18 SPACES ;

: SVCST                        ( -- )
  18 14 POS. ." Save this set?"
  VFIO
  IF GETSCR PTCST ENDIF ;

: LDCST                        ( -- )
  18 14 POS. ." Load new set?"
  VFIO
  IF GETSCR GTCST ENDIF ;
                                     ==>

Screen: 103
( Charedit                              )

: MVRHT                        ( -- )
  CHAR# @ DUP 63 <>
  IF
   31 =
   IF 289 ELSE 1 ENDIF
   DEFLOC +!
   1 CHAR# +! DEFLOC
   @ DSPCHR MPTRR
   12 14 POS.
   CHAR# ?
  ELSE
   DROP
  ENDIF ;
                                     -->

Screen: 104
( Charedit                              )

: MVLFT                        ( -- )
  CHAR# @ -DUP
  IF
   32 =
   IF -289 ELSE -1 ENDIF
   DEFLOC +! -1 CHAR# +!
   DEFLOC @ DSPCHR MPTRL
   12 14 POS. CHAR# ?
  ENDIF ;




                                     ==>

Screen: 105
( Charedit                              )

: CLRCHR                       ( -- )
  DEFLOC @  8 0
  DO DUP I 40 * + 0 SWAP C! LOOP
  DROP 88 @ 203 +   8 0
  DO
    DUP I 40 * +   8 0
    DO
      DUP I + 0 SWAP C!
    LOOP DROP
  LOOP DROP
  0 VERT ! 0 HORZ !
  88 @ 203 + DUP C@
  84 + SWAP DUP
  CURLOC ! C! ;                      -->

Screen: 106
( Charedit                              )

: CLRCST                       ( -- )
  18 14 POS. ." Clear this set?"
  KEY 89 =
  IF
   GRAFC DUP DUP 680 + SWAP
   DO
    0 I C!
   LOOP
   CLRCHR 0 CHAR# ! DEFLOC !
   12 14 POS. CHAR# ?
   GR8 0 TPTR @ C! 93 OVER
   C! TPTR !
  ENDIF
  18 14 POS. 15 SPACES ;             ==>

Screen: 107
( Charedit                              )

HEX

: CKOPT                        ( -- )
  2FC C@ FF 2FC C!
  DUP 1F = IF CLRCHR ENDIF
  DUP 1E = IF CLRCST ENDIF
  DUP 18 = IF LDCST  ENDIF
  DUP 1A = IF SVCST  ENDIF
  DUP 06 = IF MVLFT  ENDIF
      07 = IF MVRHT  ENDIF ;



DCX                                  -->
```

Screen: 108
```
( Charedit                          )
: CKBTN                      ( -- )
  644 C@ NOT
  IF
   CLICK
   CURLOC @ DUP C@ 8 CHSB2 XOR
   SWAP C! DEFLOC @ VERT @
   40 * + DUP C@ 7 HORZ @
   - 1+ CHSB2 XOR SWAP C!
   2000 0 DO LOOP
  ENDIF ;
                                 ==>
```

Screen: 109
```
( Charedit                          )
: CKSTK                      ( -- )
  0 STICK 2DUP OR
  IF
   VERT @ + 0 MAX 7 MIN VERT !
   HORZ @ + 0 MAX 7 MIN HORZ !
   VERT @ HORZ @ POSCUR
   2000 0 DO LOOP
  ELSE
   2DROP
  ENDIF ;
: CHECK                      ( -- )
  CKSTK CKBTN CKOPT ;
                                 -->
```

Screen: 110
```
( Charedit                          )
: STPSCR                     ( -- )
  CR 4 SPACES
  ." * * * CHARACTER-EDIT * * *"
  CR CR CR ."  01234567" CR
  8 0 DO I . CR LOOP
  18 4 POS.
  ."       Options:"
  18 6 POS.
  ." (1) Clear Character"
  18 8 POS.
  ." (2) Clear this set"
  18 10 POS.
  ." (3) Save this set"
                                 ==>
```

Screen: 111
```
( Charedit                          )
  18 12 POS.
  ." (4) Load a new set"
  2 14 POS. ." Character 0"
  2 15 POS. ." Load/Save: "
  2 17 POS.
  ." Use '" 30 SPEMIT
  ." ' and '" 31 SPEMIT ." ' to"
  CR
  ." through the character set."
  0 0 POS. ;
                                 -->
```

Screen: 112
```
( Charedit                          )
FORTH DEFINITIONS
: CHAR-EDIT                  ( -- )
  CHREDT ( enter vocabulary )
  0 GR. 1 752 C!
  CLS DBMAKE
  88 @ 1300 ERASE
  GRAFC DEFLOC !
  GR8 DUP TPTR !
  93 SWAP C!
  STPSCR
  88 @ 203 + DUP CURLOC !
  84 SWAP C!
                                 ==>
```

Screen: 113
```
( Charedit                          )
  0 HORZ !
  0 VERT !
  0 CHAR# !

  DCX
  BEGIN
    CHECK
    1 752 C! 128 17 C!
    ?TERMINAL
  UNTIL
  0 GR. ;

BASE !  FORTH
```

Screen: 114

Screen: 115

Screen: 116

Screen: 117

Screen: 118

Screen: 119

Screen: 120

```
( Character words:  CHLOAD      )

BASE @ DCX

: CHLOAD    ( addr scr# cnt -- )
  8 * DUP <ROT
  128 /MOD SWAP 0# +
  >R B/SCR * R> 0
  DO
    PAD 128 I * +
    OVER I + 1 R/W
  LOOP
  DROP
  PAD <ROT CMOVE ;

                                   ==>
```

Screen: 121

```
( Character words:  NML/SPLCHR )


: SPLCHR               ( CHBAS -- )
  SP@ 1+ C@
  SWAP DROP 756 C! ;


: NMLCHR                     ( -- )
  57344 SPLCHR  ;


BASE !
```

Screen: 122

Screen: 123

Screen: 124

Screen: 125

Screen: 126
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 127
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 128
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 129
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Screen: 130
0
1
2
3
4
5
6
7 (Standard Character set)
8
9
10
11
12
13
14
15

Screen: 131
0
1
2
3
4
5
6
7 (Standard Character set)
8
9
10
11
12
13
14
15

```
Screen: 132

( PM example #2 ship images )
```

```
Screen: 133
```

```
Screen: 134
```

```
Screen: 135
```

```
Screen: 136
```

```
Screen: 137
```

Screen: 138

Screen: 139

Screen: 140

```
( Player/Missile example 1      )
'( PLYMV )( 15 KLOAD )
BASE @ 2 BASE !

1 VARIABLE HBALL
1 VARIABLE VBALL

LABEL IMAGE
  011100 C,
  111110 C,
  111110 C,
  111110 C,     ( A BIG BALL )
  111110 C,
  111110 C,
  011100 C,
                        DECIMAL    ==>
```

Screen: 141

```
( Player/Missile example 1      )

: BOP 0 53279 C! 8 53279 C! ;

: MOVE-BALL
  BEGIN
    HBALL @ VBALL @ 0 PLYMV
    0 PLYSTT C@ DUP 3 AND
    IF VBALL @ MINUS VBALL ! BOP
    ENDIF
    3 >
    IF HBALL @ MINUS HBALL ! BOP
    ENDIF
    50 0 DO LOOP  ( Wait... )
    ?TERMINAL
  UNTIL ;                       -->
```

Screen: 142

```
( Player/Missile example 1      )

: BOUNCE
  CLS
  1 PMINIT
  PMCLR
  1 PRIOR
  ON PLAYERS
  47 200 32 217 0 PLYBND
  0 9 ( BLUE ) 8 PMCOL
  IMAGE 7 100 75 0 BLDPLY

  ." Press START to stop... "
  MOVE-BALL
  OFF PLAYERS ;
                           BASE !
```

Screen: 143

Screen: 144

Screen: 145

Screen: 146

Screen: 147

Screen: 148

Screen: 149

Screen: 150

```
( Player/Missile example 2     )
BASE @ DCX
'( CHLOAD )( 60 KLOAD )
'( PLYMV  )( 15 KLOAD )
'( STICK  )( 84 KLOAD )
: FLY
  BEGIN
    75 0 DO LOOP       ( wait )
    PAD                ( addr )
    0 PLYLOC SWAP DROP
    8 / 11 SWAP -
    11 MIN 0 MAX       ( image# )
    0 PLYSEL           ( pl#0 )
    0 STICK 0 PLYMV
    ?TERMINAL
  UNTIL ;                       ==>
```

Screen: 151

```
( Player/Missile example 2     )

: SHIP
  2 PMINIT
  1 PRIOR
  PMCLR
  0 9 ( BLUE ) 8 PMCOL
  PAD 132 15 CHLOAD
  PAD 8 50 50 0 BLDPLY
  50 200 10 110 0 PLYBND
  CLS
  ." Move player with stick 0."
  CR
  ." Press START to stop... "
  ON PLAYERS FLY OFF PLAYERS ;
BASE !                          -->
```

Screen: 152

Screen: 153

Screen: 154

Screen: 155

Screen: 156

Screen: 157

Screen: 158

Screen: 159

Screen: 160

```
( Utils:  CARRAY  ARRAY          )
BASE @ HEX
: CARRAY          ( cccc, n -- )
  CREATE SMUDGE ( cccc: n -- a )
    ALLOT
  ;CODE CA C, CA C, 18 C,
  A5 C, W C,  69 C, 02 C, 95 C,
  00 C, 98 C, 65 C, W 1+ C,
  95 C, 01 C, 4C C,
  ' + ( CFA @ ) , C;

: ARRAY           ( cccc, n -- )
  CREATE SMUDGE ( cccc: n -- a )
    2* ALLOT
  ;CODE 16 C, 00 C, 36 C, 01 C,
   4C C, ' CARRAY 08 + , C;  ==>
```

Screen: 161

```
( Utils:  CTABLE  TABLE          )

: CTABLE          ( cccc, -- )
  CREATE SMUDGE ( cccc: n -- a )
  ;CODE
  4C C, ' CARRAY 08 + , C;

: TABLE           ( cccc, -- )
  CREATE SMUDGE ( cccc: n -- a )
  ;CODE
  4C C, ' ARRAY 0A + , C;




                                -->
```

Screen: 162

```
( Utils:  2CARRAY  2ARRAY      )

: 2CARRAY      ( cccc, n n -- )
 <BUILDS       ( cccc: n n -- a )
  SWAP DUP , * ALLOT
 DOES>
   DUP >R @ * + R> + 2+ ;

: 2ARRAY       ( cccc, n n -- )
 <BUILDS       ( cccc: n n -- a )
  SWAP DUP , * 2* ALLOT
 DOES>
   DUP >R @ * + 2* R> + 2+ ;


                              ==>
```

Screen: 163

```
( Utils:  XC!  X!              )

: XC!    ( n0...nm cnt addr -- )
  OVER 1- + >R 0
  DO J I - C!
  LOOP R> DROP ;

: X!     ( n0...nm cnt addr -- )
  OVER 1- 2* + >R 0
  DO J I 2* - !
  LOOP R> DROP ;

( Caution: Remember limitation
( on stack size of 30 values
( because of OS conflict. )
                              -->
```

Screen: 164

```
( Utils:  CVECTOR  VECTOR      )

: CVECTOR        ( cccc, cnt -- )
  CREATE SMUDGE ( cccc: n -- a )
  HERE OVER ALLOT XC!
  ;CODE
  4C C, ' CARRAY 08 + , C;

: VECTOR         ( cccc, cnt -- )
  CREATE SMUDGE ( cccc: n -- a )
  HERE OVER 2* ALLOT X!
  ;CODE
  4C C, ' ARRAY  0A + , C;


                        BASE !
```

Screen: 165

Screen: 166

```
( Sound:  SOUND  SO.  FILTER!  )

BASE @ HEX
0 VARIABLE AUDCTL

: SOUND  ( ch# freq dist vol --)
  3 DUP D20F C! 232 C!
  SWAP 10 * + ROT 2*
  D200 + ROT OVER C! 1+ C!
  AUDCTL C@ D208 C! ;

: SO. SOUND ;

: FILTER!              ( b -- )
  DUP D208 C! AUDCTL ! ;
                              ==>
```

Screen: 167

```
( Sound:  XSND  XSND4          )


: XSND            ( voice# -- )
  2* D201 +
  0 SWAP C! ;


: XSND4                 ( -- )
  D200 8 0 FILL
  0 FILTER! ;


'( POS. )( : POS. 54 C! 55 ! ; )

BASE !
```

Screen: 168

```
( Utils:  STICK                    )
BASE @ HEX
LABEL STKARY
  0 , -1 , 1 , 0 ,

: STICK                ( n -- n n )
  278 + C@ 0F XOR
  DUP 2/ 2/ 3 AND
  2* STKARY + @
  SWAP 3 AND
  2* STKARY + @ ;

CODE STRIG                ( n -- f )
  B4 C, 00 C,  B9 C, 284 ,
  49 C, 01 C, 4C C, PUT0A , C;
BASE !
```

Screen: 169

Screen: 170

```
CONTENTS OF THIS DISK:

PLAYER/MISSILES:             30 LOAD
AUDIO EDITOR:                60 LOAD
CHARACTER EDITOR:            90 LOAD
CHARACTER SET WORDS:        120 LOAD

STANDARD CHARACTER SET      130 LIST
SPACE SHIP IMAGES           132 LIST

PM EX. #1  ( BOUNCE )       140 LOAD
PM EX. #2  (  SHIP  )       150 LOAD

ARRAYS ( FOR ALL    )       160 LOAD
SOUNDS ( FOR AUDED )        166 LOAD
STICK                       168 LOAD
```

Screen: 171

Screen: 172

Screen: 173

Screen: 174

Screen: 175

Screen: 176

```
( Error messages                    )

Stack empty

Dictionary full

Wrong addressing mode

Is not unique

Value error

Disk address error

Stack full
```

Screen: 177

```
Disk Error!

Dictionary too big
```

Screen: 178

```
( Error messages                    )

Use only in Definitions

Execution only

Conditionals not paired

Definition not finished

In protected dictionary

Use only when loading

Off current screen
```

Screen: 179

```
Declare VOCABULARY
```

VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141

valFORTH T.M.

SOFTWARE SYSTEM
for ATARI*

DISPLAY FORMATTER

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

valFORTH

T.M.

DISPLAY FORMATTER

Version 1.1
March 1982

The following is a description of commands used in creating video display lists on the Atari 400/800 series microcomputers. Creating custom display lists allows for innovative graphic layouts of games, simulations, or business applications which utilize both hi-resolution graphics and text simultaneously.

valFORTH

T.M.

SOFTWARE SYSTEM

DISPLAY FORMATTER

Stephen Maguire

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

An indepth explanation of display lists was written by Dave and Sandy Small in a series of articles found in Creative Computing. We suggest that this be read to get the most out of this valFORTH package.

STROLLING THROUGH THE DISPLAY FORMATTER

In Atari Basic there are many different graphic modes. Some of these are text modes, some are graphics modes, and some are mixed. These different graphic modes are based upon display lists. A display list is a list of display instructions which tell the video processor whether a particular portion of the screen is to be high resolution graphics or normal text. Any given section of the display can actually take on one of 18 different characteristics.

Let's take a look at the display list for a graphic 0 display: (These values are in base 16)

```
BC20        70 }
            70 }      24 blank scan lines
            70 }
            42 }
            40 }      DM jump to BC40
            BC }
             2 }
             2 }
             . }
             . }      23 graphic 0 lines
             . }
             2 }
             2 }
            41 }
            20 }      jump to BC20
            BC }

BC40                  start of display memory
```

Each opcode 70 instructs the video processor to display 8 blank scan lines. Opcode 2 produces one standard graphic 0 text line. Opcode 42 is a modified 2 instruction. In addition to creating a standard text line, it also informs the video processor where the display memory is located (the address is found in the next two bytes). At the end of the list there is a three byte jump instruction which transfers display list interpretation to the address specified in the next two bytes of the list. Each of the graphic settings have a similar list. This valFORTH package allows you to design your own lists. Let's make one now.

Look in the directory (screen 170) and load in the display formatter. Most of the formatter words begin with DB (for display block). To initialize the system type:

```
DBINIT
HEX
```

This initializes the system and puts it into the more useful hexadecimal mode. Graphic mode B is a high resolution graphic mode with a four line text window at the bottom of the display. Let's make a display with a four line text window at the top of the screen followed by the high resolution graphics plate. First, we need 24 blank scan lines at the top:

```
70 DBM
70 DBM
70 DBM
```

The DBM command stands for "Display-Block Make." It takes the opcode on top of the stack and tacks it onto the end of the display list currently being created. Additionally, it enters an address into the array DBLST which points to the first byte of memory used by that display block. There is a plural form of the DBM command:

```
3   70 DBMS
```

This adds 3 opcode 70's to the current display list. Now let's add the four line text window. Recall that a normal text line has an opcode of two:

```
4   2 DBMS
```

Note that the display memory jump described earlier is automatically inserted into the display list. Now we need to define the high resolution portion of the display. A standard graphic B line has an opcode of $F (15 in decimal). Let's create 20 graphic 8 lines (20 in base 16 is 14).

```
14   F DBMS
```

This list is good enough for now. To verify that it has been entered properly, type:

```
DMPLST
```

You should get something like:

BLK	ADDR	BYTE	MODS
---	----	----	----
0	A100	70	
1	A100	70	
2	A100	70	
3	A100	2	J A100
4	A128	2	
5	A150	2	
6	A178	2	
7	A1A0	F	
8	A1C8	F	
9	A1F0	F	
A	A218	F	
B	A240	F	
C	A268	F	
D	A290	F	
E	A2B8	F	
F	A2E0	F	
10	A308	F	
11	A330	F	
12	A358	F	
13	A380	F	
14	A3A8	F	
15	A3D0	F	
16	A3F8	F	
17	A420	F	
18	A448	F	
19	A470	F	
1A	A498	F	

Note the automatic insertion of the display memory jump in block three. Display memory cannot cross a 4K memory boundary without a display memory jump. As each display block is added, a check is made to detect any 4K memory crossings caused by the display block. If the block does cross, a display memory jump is automatically inserted into the list to account for it.

Now that we have a display list, let's enable it. There are several ways to activate a list. For now type:

MIXED CLS

This MIXED command enables the new display list and also re-directs output to the display memory specified by the list. This allows for interactive display list creation. There should be a recognizably different display. Hold down the RETURN key and watch how the "ok" message is displayed as the cursor moves down the screen. You should see "ok"'s on the text lines, but in the high resolution lines, it shoud look quite different. You can type in a high resolution mode because the Atari operating system does not know that the display list has been changed. To return to a normal display, the GR. command is used:

```
0 GR.
```

Dump the display list again using the DMPLST command. Let's put some text lines in at block B. To do this type:

```
   B DBPTR
10 2 DBMS
```

The DBPTR command positions the display list pointer to the specified block. That block then becomes the end of the list. After that, we add 16 (10 hex) graphic 0 lines. Dump the list again and verify that this is indeed what was accomplished. To view this new display, type:

```
MIXED CLS
```

Hold down the RETURN key again. Notice what happens as the cursor passes through the high resolution section and then back into the second text section. Type DMPLST again while in this mode and notice that everything works the same, the data is simply displayed differently. To get out, type "0 GR.".

Besides adding display blocks onto the end of a display list, the display formatter allows display blocks to be inserted and deleted as well. Block two has an opcode 70 which produces 8 blank scan lines on the video screen. By deleting this block from the list, the entire display will shift upwards by 8 lines. This is accomplished using the DBDEL command:

```
2 DBDEL
```

Dump the list and verify that the block has indeed been deleted. Enable the list using "MIXED CLS". Note that the first text line appears much higher than usual on the video screen. While still in this display, execute:

```
4 6 DBDELS
```

This will delete the four display blocks starting at block six. In this case, the four high resolution display lines are deleted. Type "MIXED CLS" and watch the screen shrink slightly as the display blocks are extracted.

Display blocks can be inserted using the DBIN command. When a DBIN command is executed, the specified opcode is inserted into the specified block. The opcode previously in that block and all opcodes following are pushed back by one block. As an example, we will insert opcode 70 (8 blank scan lines) at block five. This will do it:

```
70 5 DBIN
```

"MIXED CLS" will activate the new list. Press the RETURN key a few times and notice how the output routines seem to ignore the blank scan lines. The DBINS command is a plural form of the DBIN command. Let's insert a different opcode other than 2 or $F. Opcode 6 is a mode which displays colored characters which are much larger than normal. This will insert three opcode 6's at block 9:

```
3  6  9  DBINS
```

Activate this new list in the normal way and experiment with it. The following section describes all of the available opcodes. Experiment with these as you read about them and you should have no problem understanding any of them.

This brief explanation of display list formatting should show the power available to the programmer who wants to get that unique display. There are many more commands available for use. These are explained thoroughly in the glossary at the end of the next section.

DISPLAY LIST INSTRUCTIONS

There are four basic display list instructions. Those that produce blank scan lines, the display list jump, the jump on vertical blank, and the display block instructions. This is a description of these four basic instructions.

Blank Scan Lines

Byte form:

I	x	x	x	0	0	0	0

n n = 0 to 7

This opcode produces n+1 blank scan lines of color BAK. No video memory is used by this instruction.

If the I bit is set, a display list interrupt (DLI) will occur upon interpretation by Antic (the video processor).

The 8 legal values are:

\$00 = 0	1 blank scan line	(128 with I bit set)
\$10 = 16	2 blank scan lines	(144)
\$20 = 32	3 blank scan lines	(160)
\$30 = 48	4 blank scan lines	(176)
\$40 = 64	5 blank scan lines	(192)
\$50 = 80	6 blank scan lines	(208)
\$60 = 96	7 blank scan lines	(224)
\$70 = 112	8 blank scan lines	(240)

Display List Jump

Byte form:

I	0	0	0	0	0	0	1

This command instructs Antic to search for the next display list instruction specified by the address contained in the next two bytes of the display list. The low byte of the address is found lower in memory. This command is used primarily to continue a display list across a 1K memory boundary (Antic will not handle this properly). This is the only instruction not supported by the display formatter since its occurrence is rare. It is explained here for completeness sake and its use is absolutely forbidden. Future releases may have this implemented.

If the I bit is set, a display list interrupt will occur upon interpretation by Antic.

Legal form:

\$01 addr-low addr-hi	Transfer display list interpretation to addr.

Jump On Vertical Blank

Byte form:

I	1	x	x	0	0	0	1

This three byte opcode instructs Antic to transfer display list interpretation to the address specified by the following two bytes (low byte of address first) and to pause until vertical blank occurs. Since display list processing halts, any remaining portion of the video display takes on the color of BAK. This command is not to be entered by the user. The display formatter automatically adds this to the end of the display list whenever it is moved or activated.

If the I bit is set, a display list interrupt will occur upon interpretation by Antic.

Legal form:

\$41	addr-low	addr-hi	Transfer display list interpretation to addr. (65)

Display Block Opcodes

Byte form:

I	J	V	H	x	x	x	x

n = 2 to \$F (15)   n

There are 14 display modes. Six are character modes, eight are graphic modes. Each of these modes varies greatly and will be discussed individually. But first, the four status bits I, J, V, and H, will be discussed as they function similarly for all display modes.

If the I bit is set, a display list interrupt will occur upon interpretation by Antic.

If set, the J bit instructs Antic to perform a display memory jump. Antic expects the next two bytes in the display list to point to the new display memory location. The first display block instruction should always have this bit set. Also, Antic cannot properly retrieve data from display memory across 4K boundaries. Thus, if the display memory must cross a 4K boundary, a display memory jump must be used. Note that the display formatter automatically takes care of these two problems for the user.

If set, the V bit informs Antic that the current display block is to be vertically scrolled upward according to the value in VSCROL (address \$D405). Note that vertical scrolling is accomplished only if two or more consecutive display blocks have this bit set.

If set, the H bit informs Antic that the current display block is to be horizontally scrolled right according to the value in HSCROL (address $D4D4). Note that for horizontally scrolled display blocks, extra bytes of memory are needed. The exact number of bytes varies for different screen (playfield) widths. Use the following calculation:

```
# extra  =  X / n

where: X = the number of characters/display block
       n = 4 for a narrow playfield
         = 5 for a standard playfield
```

There are no extra bytes for the wide playfield setting.

For example, a 40 character/line display block in the standard width would use a total of 40 + 40/5 or 48 characters. Note that only one of these extra bytes is actually used for the display.

The Character Modes

There are 6 character modes (opcodes 2 thru 7). All character modes work in the same way, i.e., the values in display memory are indices to a large "n" by 8 byte array. In some of these modes, the highest one or two bits are used to specify a color with only the remaining lower bits used for indexing. The following table gives information about each of the modes:

Antic mode	2	3	4	5	6	7
Basic mode	0	---	---	---	1	2
# color *	1.5	1.5	5	5	5	5
Chars/line narrow wid	32	32	32	32	16	16
Chars/line normal wid	40	40	40	40	20	20
Chars/line wide screen	48	48	48	48	24	24
Scan lines/ pixel	8	10	8	16	8	16
Bits/pixel	1	1	2	2	1	1
Color clocks per pixel	.5	.5	1	1	1	1

Colors:

mode 2: Takes the color of PF2 with the lum of PF1
(Artifacting/bleed very noticeable)
mode 3: Same as above
mode 4: Two bits/pixel in character definitions
00 = BAK   01 = PF0   10 = PF1
11 = PF2 if bit 7 of index = 0, else PF3
mode 5: Same as 4 above
mode 6: Most significant two bits of index
0 = PF0   1 = PF1 etc.
mode 7: Same as 6 above

The Graphic Modes

There are 8 graphic modes. Unlike character modes, the values in display memory are not indices into an array of character definitions, but rather are the definitions themselves. Oepending on the graphic mode, these values give different results. The following table gives various information about each mode.

Antic mode	8	9	A	B	C	0	E	F*
Basic mode	3	4	5	6	---	7	---	8
# colors	4	2	4	2	2	4	4	1.5
bytes/line narrow wid	8	8	16	16	16	32	32	32
bytes/line normal wid	10	10	20	20	20	40	40	40
bytes/line wide screen	12	12	24	24	24	48	48	48
Pixels per normal wid	40	80	80	160	160	160	160	320
Scan lines/pixel	8	4	4	2	1	2	1	1
Bits/pixel	2	1	2	1	1	2	2	1
Color clocks per pixel	4	2	2	1	1	1	1	.5

*Mode F values differ when in GTIA modes

Colors:

```
mode 8:  Two bits/pixel, 4 pixels/byte
         00 = BAK     01 = PF0     10 = PF1     11 = PF2
mode 9:  One bit/pixel, 8 pixels/byte
         0 = BAK     1 = PF0
mode A:  Same as mode 8 above
mode B:  Same as mode 9 above
mode C:  Same as mode 9 above
mode 0:  Same as mode 8 above
mode E:  Same as mode 8 above
mode F:  Take the color of PF2 and lum of PF1
         (if not in a GTIA mode)
```

GLOSSARY

(DBINIT) (dmem dlist ---)

The (DBINIT) routine initializes the display formatter. It expects two addresses on the stack. The address on top of the stack is used as the target address for the display list. The address found second on the stack is the target address for display memory. The display list is actually created in a c-array named DSPLST. Note that while building the list, no check is made to ensure that the display list does not cross a 1K memory boundary.

DBINIT (---)

Like the (DBINIT) command above, this initializes the display formatter. But unlike (DBINIT), this expects no arguments. Instead, these values are calculated automatically. The display memory address is top of memory minus 1F00 hex. This is enough for a full graphics 8 screen. The display list address is 256 bytes below the display memory address. Note that this is very memory wasteful, and should only be used while still learning the system. After that, (DBINIT) should be used.

DBPTR (block# ---)

This command instructs the display formatter to create the next display block in the specified "block#" of the current display list. To begin creating a new display list, use:

```
0 DBPTR
```

DBM (antic-mode ---)

The DBM command adds "antic-mode" to the end of the current display list. For example, to create a video display with a single line at the top of the screen, the following would be executed:

```
0  DBPTR     (new list)
2  DBM       (A graphic 0 line)
```

(Note: Antic mode 2 is a BASIC graphics 0 line.)

DBMS (#times antic-mode ---)

The DBMS command performs a multiple DBM. For example, to create a full graphics 0 screen, the following two commands must be performed:

```
   0 DBPTR     (new list)
24 2 DBMS      (24 graphic 0 lines)
```

This would create a full graphics 8 screen:

```
     0 DBPTR
192 15 DBMS     (Antic 15 = graphic 8)
```

(192 graphic 8 lines fill one video screen)

DBMS (cont'd)

Mixed lists are also possible:

```
    0 DBPTR
160 15 DBMS
  4  2 DBMS
```

This would create a screen of 160 graphic 8 lines with four text lines at the bottom.

DBIN (antic-mode block# ---)

Oftentimes, it is desirable to slightly change the existing display list to obtain special effects midway through a running program. The DBIN command allows insertion of new display blocks within the current display list. This command inserts "antic-mode" into the block specified by "block#". Whatever was in the block "block#" and following is pushed back one block. For example:

	Display list
block #0	2
1	2
2	8

with the above display list, a

```
15 1 DBIN
```

would give the following display list.

	Display list
block # 0	2
1	15
2	2
3	8

The DBIN allows the user to create new display lists without the need to duplicate already existing display list sections.

DBINS (#times antic-mode block# ---)

This command repeats "antic-mode block# DBIN" the specified number of times.

DBDEL (block# ---)

The DBDEL command serves as the logical complement to the DBIN command. Thus, after inserting a temporary display block, the DBDEL command may be used to delete that display block once it is no longer needed:

	Display list
block # 0	2
1	15
2	2
3	8

1 DBDEL would give:

	Display list
block # D	2
1	2
2	8

Note: Deleting non-existing display blocks gives unexpected results.

DBDELS (#times block# ---)

This command performs "block# DBDEL" the specified number of times. This serves as the logical complement to the DBINS command.

DBDELL (---)

This form of the DBDEL command deletes the last display block created using the DBM command. For example:

	Display list
block # 0	2
1	15
2	2
3	8

DBDELL would give:

	Display list
block # 0	2
1	15
2	2

The main use for the DBDELL command is for "backing up" and re-entering a display block when an error has been made while creating a display list directly at the keyboard. The DBDELL command can be used successively for deleting a section of display blocks at the end of the current display list. There is no plural command for DBDELL command as its use is rather limited.

?ANTMOD (block# --- antic-mode)

Occasionally, it is desirable to know what antic-mode is being used for a particular display block (such as for a text output routine -- text should not be output on a hi-resolution line, for example). This command returns the antic-mode of the specified block.

DBMOD (modifier block# ---)

When creating display lists, it is possible to give extra meaning to a particular block or section of blocks in the list. This is accomplished by using one or more of the three available antic-modifiers: vertical scroll modifier (VRTMOD), horizontal scroll modifier (HRZMOD), and the display-list interrupt (INTMOD). The following are examples of each:

```
VRTMOD 0 DBMOD
HRZMOD 3 DBMOD
INTMOD 5 DBMOD
```

DBMOD (cont'd)

There are several methods in which to put more than one modifier on a given display block. For example, each of the following would give the same final result:

```
VRTMOD 20 DBMOD
HRZMOD 20 DBMOD
```

or

```
VRTMOD HRZMOD + 2D DBMOD
```

To attach all three modifiers, the best method is:

```
VRTMOD HRZMOD INTMOD + + 20 DBMOD
```

It should also be noted that it is possible to create modified display blocks, thus reducing the need for the DBMOD command:

```
HRZMOD 2 + DBM
```

This would create one graphic 0 line with a horizontal modifier. It is also easy to obtain 16 lines of hi-resolution graphics with both horizontal and vertical scroll modifiers:

```
16 VRTMOD HRZMOD 15 + + DBMS
```

CAUTION: VRTMOD and HRZMOD can only be used on antic-modes 2 through 15 ($2-$F).

(Note: There is one additional modifier, JMPMOD; however its use is absolutely forbidden! This has been defined as it will be implemented in the next release.)

DBMODL (modifier ---)

This command modifies the last display-block in the display list.

?DBMODS (block# --- modifiers)

This returns the modifiers on the specified display block. For example:

```
VRTMOD 2 + 0 DBM
         0 ?DBMODS
```

would give VRTMOD. Also:

```
VRTMOD HRZMOD 2 + + 0 DBM
                0 ?DBMODS
```

would give VRTMOD + HRZMOD. To test for VRTMOD, the following method must be used:

```
0 ?DBMODS
VRTMOD AND
```

The last line leaves only the vertical modifier, if present, or leaves 0 indicating no vertical modifier.

DBREM (block# ---)

The DBREM command removes all modifiers from the specified display block. Care should be taken when stripping modifiers, as stripping a horizontal modifier (if present) will change the size of the video memory.

DBREMS (#times block# ---)

This performs "block# DBREM" the specified number of times.

DBREML (---)

This removes the modifiers from the last display block in the current display list.

?DBVAL (block# --- info)

The ?DBVAL command returns all information about the display block specified, i.e., the antic mode and any modifiers. This information is returned as one value.

DBWID (width ---)

The DBWID command is used to set the desired playfield width so that the address array DBLST gives the proper values. Legal settings are: 1 - narrow, 2 - normal, and 3 - wide.

USRDSP (---)

Once a display list has been created, USRDSP activates the new list.

MIXED (---)

The MIXED command performs a USRDSP and instructs the Atari operating system to re-direct all output to the video display memory specified by the newly created display list.

DMPLST (---)

The DMPLST command instructs the display list assembler to give a complete, informative listing of the display list last created.

DBADR (block# --- address)

The DBADR command is one of the most useful commands to the programmer. Given a display block number, it returns the address of the first byte of that display block. This is extremely useful for determining where output text or graphic displays should be located.

DMCLR (---)

The DMCLR command clears the display memory pointed to by the display list currently being created. It [illegible] of memory.

In addition, there are various variables available to the programmer:

DSPEND	Points to the end of the current display list. It is an offset from 0 DSPLST.
DSPBLK	Contains the number of the next display block to be created.
DMLOC	Points to the beginning of display memory.
LSTLOC	Contains the address of where the display list is to reside in memory.
DBLST	Is an array of addresses used by DBADR.
DSPLST	Is a byte array containing the display list currently being created. DSPEND above points to the end of the list in this array.

XXXII. DISPLAY FORMATTER SUPPLIED SOURCE LISTING

Screen: 30

```
( Graph Sys:  tables            )
'( TRANSIENT TRANSIENT )(  )
BASE @ DECIMAL
'( CTABLE )( 45 KLOAD )

LABEL BLKNML
   0 C,  0 C, 40 C, 40 C, 40 C,
  40 C, 20 C, 20 C, 10 C, 10 C,
  20 C, 20 C, 20 C, 40 C, 40 C,
  40 C,

CTABLE HSOFS
  50 C,  4 C,  5 C, 50 C,

TABLE BLKOFS
  50 , -5 , 50 , 5 ,              ==>
```

Screen: 31

```
( Graph Sys:  variables         )

CTABLE BYTBLK 16 ALLOT
BLKNML 0 BYTBLK 16 CMOVE

 255    CARRAY DSPLST
 255     ARRAY DBLST
   5 VARIABLE HS#/
   0 VARIABLE DSPBLK
   0 VARIABLE PGECRS
   0 VARIABLE DSPEND
   0 VARIABLE NWLST
   6 VARIABLE SDTMP
   0 VARIABLE DBCNT
   0 VARIABLE DBVRT
                                  -->
```

Screen: 32

```
( Graph Sys:  constants         )
HEX
    0 VARIABLE DMLOC
    0 VARIABLE LSTLOC

   10 CONSTANT HRZMOD
   20 CONSTANT VRTMOD
   40 CONSTANT JMPMOD
   80 CONSTANT INTMOD

DECIMAL

  11    CARRAY JMPDAT
  10    CARRAY JMPSTT

   0 JMPDAT 11 ERASE              ==>
```

Screen: 33

```
( Graph Sys:  [DBINIT]  DBINIT )


: (DBINIT)        ( DM  LIST  -- )
  LSTLOC !     DUP
  DMLOC !      0 DBLST  !
  0 DSPEND !   0 PGECRS !
  0 DSPBLK !   1 NWLST  !
  0 JMPDAT 11 ERASE ;

: DBINIT                    ( -- )
  106 C@ 256 * 7936 -
  DUP 256 -
  (DBINIT) ;

DBINIT                          -->
```

Screen: 34

```
( Graph Sys:  HINYB  EODB       )
DECIMAL

: HINYB              ( nmmm -- n )
  61440 AND 4096 / 16 + 15 AND ;

: EODB                  ( n -- a )
  31 AND DUP 15 AND  ( Find end)
  BYTBLK C@ SWAP        ( of disp )
  15 >                  ( block )
  IF               ( Horz. scroll )
    DUP HS#/ @ / +
  ENDIF
  DSPBLK @         ( Update the  )
  DBLST @          ( addr list )
  + 1- ;                          ==>
```

Screen: 35

```
( Graph Sys:  DBPTR             )

: DBPTR                ( blk# -- )
  DMLOC @ 0 DBLST !
  0 PGECRS !      0 NWLST !
  DUP DSPBLK !   DUP DSPEND !
  0 JMPDAT C@ -DUP
  IF 1+ 1 DO
     I JMPDAT C@ OVER <
     IF
       2 DSPEND +!
     ELSE
       I 1- 0 JMPDAT C! LEAVE
     ENDIF
   LOOP
  ENDIF DROP                      -->
```

```
Screen:  36
( Graph Sys:  JMPINS  DBPTR     )

  DSPEND @ DSPBLK @ =
  IF
    1 NWLST !
  ELSE
    DSPBLK @ DBLST @
    1- HINYB PGECRS !
  ENDIF ;

: JMPINS                      ( n -- )
  DUP JMPMOD OR
  DSPEND @ SWAP OVER
  DSPLST C! 1+ NWLST @
  IF
    0 NWLST !                        ==>
```

```
Screen:  37
( Graph Sys:  JMPINS               )

    DMLOC @ DUP DUP 40 +
    HINYB SWAP HINYB <>
    IF
      HINYB 1+ 4096 *
    ENDIF
  ELSE
    PGECRS @ 4096 *
  ENDIF
  DUP ROT DSPLST !
  3 DSPEND +!
  DSPBLK @ 0 JMPDAT C@ 1+
  DUP 0 JMPDAT C!
  JMPDAT C! ;
                                     -->
```

```
Screen:  38
( Graph Sys:  DBCRT                )

: DBCRT                       ( n -- )
  DUP EODB DUP
  HINYB PGECRS @
  OVER PGECRS ! <>
  IF
    DROP JMPINS
    DSPBLK @ DBLST ! EODB
  ELSE
    SWAP DSPEND @
    DSPLST C!
    1 DSPEND +!
  ENDIF
  1+ DSPBLK 1 OVER +!
  @ DBLST ! ;                        ==>
```

```
Screen:  39
( Graph Sys:  DBM                  )

: DBM                         ( n -- )
  DUP 15 AND
  IF               ( antic instr.? )
    191 AND        ( strip jump    )
    DBCRT
  ELSE                  ( scan lines )
    DSPEND @
    DSPLST C!
    1 DSPEND +!
    DSPBLK @ DUP DBLST @
    SWAP 1+ DBLST !
    1 DSPBLK +!
  ENDIF ;
                                     -->
```

```
Screen:  40
( Graph Sys:  LSTSV                )

: LSTSV                ( blk# -- a )

  DSPEND @             ( pt to blk )
  DSPLST 65 OVER C!
  SWAP DBPTR

  DSPEND @ DSPLST  ( save list )
  DUP SDTMP @ + ROT
  >R OVER R> -
  ABS 1+ <CMOVE

  DSPEND @ DSPLST  ( leave save)
  SDTMP @ + ;      ( address )
                                     ==>
```

```
Screen:  41
( Graph Sys:  LSTRST               )

: LSTRST                      ( a -- )
  BEGIN
    DUP C@ DUP 65 <>     ( eolst?)
  WHILE
    DUP 15 AND 0#
    OVER 64 AND 0# AND
    IF                   ( jump? )
      191 AND DBM 3 +
    ELSE                ( normal )
      DBM 1+
    ENDIF
  REPEAT
  2DROP ;
                                     -->
```

Screen: 42

```
( Graph Sys:  DBIN DBDEL DBMOD )

: DBIN                    ( n n -- )
  LSTSV SWAP DBM LSTRST ;

: DBDEL                   ( n -- )
  DUP 1+ LSTSV SWAP
  DBPTR LSTRST ;

: DBMOD                   ( n n -- )
  DUP 1+ LSTSV
  SWAP DBPTR SWAP
  DSPEND @ DSPLST C@ OR
  DBM LSTRST ;

                                ==>
```

Screen: 43

```
( DBREM  DBDELL  DBMODL DBREML )

: DBREM                   ( n -- )
  DUP 1+ LSTSV
  SWAP DBPTR
  DSPEND @ DSPLST C@
  15 AND DBM LSTRST ;

: DBDELL                    ( -- )
  DSPBLK @ 1- DBPTR ;

: DBMODL                    ( -- )
  DSPBLK @ 1- DBMOD ;

: DBREML                    ( -- )
  DSPBLK @ 1- DBREM ;      -->
```

Screen: 44

```
( Graph Sys:  ?DBVAL             )

: ?DBVAL                  ( n -- )
  DSPBLK C@ 0 JMPDAT C@
  ROT DBPTR
  DSPEND @ DSPLST C@
  <ROT 0 JMPDAT C!
  DBPTR ;

                                ==>
```

Screen: 45

```
( Graph Sys:  ?ANTMOD  ?DBMODS )

: ?ANTMOD                 ( n -- )
  ?DBVAL DUP 15 AND 0=
  IF 127 ELSE 15 ENDIF
  AND ;

: ?DBMODS                 ( n -- )
  ?DBVAL DUP 15 AND 0=
  IF
    DROP 0
  ELSE
    240 AND
  ENDIF ;

                                -->
```

Screen: 46

```
( Graph Sys:  DBMS  DBDELS       )

: DBMS                  ( # n -- )
  SWAP 0
  DO
    DUP DBM
  LOOP
  DROP ;

: DBDELS                ( # n -- )
  SWAP 0
  DO
    DUP DBDEL
  LOOP
  DROP ;
                                ==>
```

Screen: 47

```
( Graph Sys:  DBREMS             )

: DBREMS                ( # n -- )
  SWAP 0
  DO
    DUP DBREM 1+
  LOOP
  DROP ;

                                -->
```

```
Screen: 48
( Graph Sys:  DBINS                      )

: DBINS                ( # n n -- )
  ROT DUP DBCNT !
  6 + SDTMP @ SWAP SDTMP !
  (ROT DUP LSTSV
  SWAP DBPTR SWAP
  BEGIN
    DBCNT @        ( count zero? )
  WHILE
    DUP DBM       ( keep creating)
    -1 DBCNT +! ( dec. counter )
  REPEAT
  DROP LSTRST   ( unsave list )
  SDTMP ! ;
                                 ==>
```

```
Screen: 49
( Graph Sys:  SCRWID  LSTMV      )

: SCRWID              ( width -- )
  559 C@ 252 AND
  OR 559 C! ;

: LSTMV                     ( -- )
  LSTLOC @ DSPEND @
  DSPLST 65
  OVER C! 1+ !
  0 DSPLST LSTLOC @
  DSPEND @ CMOVE ;



                                 -->
```

```
Screen: 50
( Graph Sys:  USRDSP MIXED       )

: USRDSP                    ( -- )
  LSTMV
  559 C@ 3 AND
  0 SCRWID
  LSTLOC @ 560 !
  SCRWID  ;

: MIXED                     ( -- )
  DMLOC @ 88 !
  USRDSP ;




                                 ==>
```

```
Screen: 51
( Graph Sys:  DMPLST                  )

: DMPLST                      ( -- )
  CR DSPBLK @ -DUP
  IF
    CR ."  BLK     ADDR"
    ."      BYTE    MODS"
    CR ."  ---     ----"
    ."      ----    ----"
    0 DO
      CR I 3 .R
      I DBLST @ 9 U.R
      I ?ANTMOD  8 U.R
      I ?DBMODS  -DUP
      IF
        3 SPACES                  -->
```

```
Screen: 52
( Graph Sys:  DMPLST                  )

        DUP 128 AND
        IF ." I" ENDIF
        DUP 32 AND
        IF ." V" ENDIF
        DUP 16 AND
        IF ." H" ENDIF
            64 AND
        IF ." J " I DBLST @ U.
        ENDIF
      ENDIF ?EXIT
    LOOP
  ELSE
    ." No display list"
  ENDIF CR ;                      ==>
```

```
Screen: 53
( Graph Sys: DMCHG DMCLR DBADR )

: DMCLR                       ( -- )
  DMLOC @
  106 @ 256 *
  OVER -
  ERASE ;

: DBADR               ( blk# -- a )
  DBLST @ ;






                                  -->
```

Screen: 54

```
( VAL-FORTH GRAPHIC SYSTEM 1.1 )

: DBWID                    ( width -- )
  0 LSTSV SWAP BLKNML
  [ 0 BYTBLK ] LITERAL
  16 CMOVE
  BLKOFS @ [ 0 BYTBLK 16
  OVER + ] LITERAL LITERAL
  DO
    I C@ DUP 3 PICK / + I C!
  1 /LOOP
  HSOFS C@ HS#/ !
  LSTRST ;

'( PERMANENT PERMANENT )(  )
BASE !
```

Screen: 55

Screen: 56

Screen: 57

Screen: 58

Screen: 59

```
Screen: 60
( Transients:  setup            )
BASE @ DCX

HERE


741 @ 4000 - DP !
( SUGGESTED PLACEMENT OF TAREA )


HERE CONSTANT TAREA
    0 VARIABLE TP
    1 VARIABLE TPFLAG
      VARIABLE OLDDP

                              ==>
```

Screen: 63

```
Screen: 61
( Xsients: TRANSIENT PERMANENT )

: TRANSIENT                ( -- )
  TPFLAG @ NOT
  IF HERE OLDDP ! TP @ DP !
     1 TPFLAG !
  ENDIF ;

: PERMANENT                ( -- )
  TPFLAG @
  IF HERE TP ! OLDDP @ DP !
     0 TPFLAG !
  ENDIF ;


                              -->
```

Screen: 64

```
Screen: 62
( Transients:  DISPOSE          )
: DISPOSE   PERMANENT
  CR ." Disposing..." VOC-LINK
  BEGIN DUP    0 53279 C!
   BEGIN @ DUP TAREA U<
   UNTIL DUP ROT ! DUP 0=
  UNTIL DROP VOC-LINK @
  BEGIN DUP 4 -
   BEGIN DUP    0 53279 C!
    BEGIN PFA LFA @ DUP TAREA U<
    UNTIL
     DUP ROT PFA LFA ! DUP 0=
   UNTIL DROP @ DUP 0=
  UNTIL DROP [COMPILE] FORTH
  DEFINITIONS ." Done" CR ;
  PERMANENT             BASE !
```

Screen: 65

Screens 66 thru 89 are blank

```
Screen:  90
( Utils:  CARRAY  ARRAY          )
BASE @ HEX
: CARRAY          ( cccc, n -- )
  CREATE SMUDGE ( cccc: n -- a )
    ALLOT
  ;CODE CA C, CA C, 18 C,
  A5 C, W C,  69 C, 02 C, 95 C,
  00 C, 98 C, 65 C, W 1+ C,
  95 C, 01 C, 4C C,
  ' + ( CFA @ ) , C;

: ARRAY           ( cccc, n -- )
  CREATE SMUDGE ( cccc: n -- a )
    2* ALLOT
  ;CODE 16 C, 00 C, 36 C, 01 C,
   4C C, ' CARRAY 08 + , C;  ==>
```

```
Screen:  91
( Utils:  CTABLE  TABLE          )

: CTABLE          ( cccc, -- )
  CREATE SMUDGE ( cccc: n -- a )
  ;CODE
  4C C, ' CARRAY 08 + , C;

: TABLE           ( cccc, -- )
  CREATE SMUDGE ( cccc: n -- a )
  ;CODE
  4C C, ' ARRAY 0A + , C;





                                 -->
```

```
Screen:  92
( Utils:  2CARRAY  2ARRAY        )

: 2CARRAY        ( cccc, n n -- )
  <BUILDS        ( cccc: n n -- a )
   SWAP DUP , * ALLOT
  DOES>
    DUP >R @ * + R> + 2+ ;

: 2ARRAY         ( cccc, n n -- )
  <BUILDS        ( cccc: n n -- a )
   SWAP DUP , * 2* ALLOT
  DOES>
    DUP >R @ * + 2* R> + 2+ ;



                                 ==>
```

```
Screen:  93
( Utils:  XC!  X!                )

: XC!     ( n0...nm cnt addr -- )
  OVER 1- + >R 0
  DO J I - C!
  LOOP R> DROP ;

: X!      ( n0...nm cnt addr -- )
  OVER 1- 2* + >R 0
  DO J I 2* - !
  LOOP R> DROP ;

( Caution: Remember limitation
( on stack size of 30 values
( because of OS conflict. )
                                 -->
```

```
Screen:  94
( Utils:  CVECTOR  VECTOR        )

: CVECTOR        ( cccc, cnt -- )
  CREATE SMUDGE ( cccc: n -- a )
  HERE OVER ALLOT XC!
  ;CODE
  4C C, ' CARRAY 08 + , C;

: VECTOR         ( cccc, cnt -- )
  CREATE SMUDGE ( cccc: n -- a )
  HERE OVER 2* ALLOT X!
  ;CODE
  4C C, ' ARRAY  0A + , C;


                         BASE !
```

```
Screen:  95
```

Screens 96 thru 167 are blank

```
Screen: 168

Screen: 169

Screen: 170
CONTENTS OF THIS DISK:

DISPLAY FORMATTER:          30 LOAD
TRANSIENTS:                 60 LOAD
ARRAYS & THEIR COUSINS:     90 LOAD

Screen: 171

Screen: 172

Screen: 173
```

```
Screen: 174
```

```
Screen: 175
```

```
Screen: 176
( Error messages                              )

Stack empty

Dictionary full

Wrong addressing mode

Is not unique

Value error

Disk address error

Stack full
```

```
Screen: 177
Disk Error!

Dictionary too big
```

```
Screen: 178
( Error messages                              )

Use only in Definitions

Execution only

Conditionals not paired

Definition not finished

In protected dictionary

Use only when loading

Off current screen
```

```
Screen: 179
Declare VOCABULARY
```

VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
602-790-7141

valFORTH T.M.

SOFTWARE SYSTEM
for ATARI*

TURTLE & valGRAPHICS TM
AND ADVANCED
FLOATING POINT ROUTINES

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

valFORTH
T.M.
SOFTWARE SYSTEM

TURTLE & valGRAPHICS T.M. AND ADVANCED FLOATING POINT ROUTINES

DRAWLN ROUTINES: Stephen Maguire
ARMADILLD GRAPHICS: Evan Rosen, William Volk
QUAN STRUCTURES: Evan Rosen

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

valGRAPHICS 1.2 AND QTRIG USER'S MANUAL

Table of Contents

Page

XLI. valGRAPHICS

a) Overview 2
b) Strolling Through valGraphics
(Walking the Armadillo) 5
c) Strolling Through valGraphics, II
(Windows, Lines, and Labeling) 14

XLII. valGRAPHICS GLOSSARIES

a) General Functions 1
b) Windows and Coordinate Systems 7
c) Options 10
d) Screen Dump 14
e) Interfacing to Custom Display Lists 15
f) A Note on QUAN Structures 16

XLIII. QUICK TRIG

XLIV. valGRAPHICS SUPPLIED SOURCE LISTING

valGRAPHICS T.M.

Version 1.2
July 1982

Many different approaches to creating "computer graphics" are possible, and, indeed, many have been implemented. One of the most fruitful approaches, particularly for two-dimensional graphics work, is a system usually called "turtle graphics." The valGraphics package is a turtle-like system patterned after the ATARI PILOT turtle graphics rendition, though with many significant extensions.

Overview

Turtle geometry was originally developed at MIT by Dr. Seymour Papert and the LOGO group there. Since that time, a variety of different computer-based applications have been said to support "turtle graphics," though in general they differ in various ways from the initial LOGO formulation. The formulation adopted for this package follows the ATARI PILOT turtle graphics nomenclature where possible. In general, commands in this package are much smarter than their PILOT counterparts, but have also been made friendly so that in their default modes they are functionally similar to PILOT commands. Because this package differs from "true" turtle graphics in many ways, it is called "armadillo graphics."

Some minor changes have been made in the names of a few ATARI PILOT commands because of collisions with existing valFORTH names. Important variations are listed here:

PILOT	valGraphics
CLEAR	WIPE (CLEAR exists)
FILL	PHIL (FILL exists)
FILLTO	PHILTO (for consistency)
QUIT	(not needed and exists)
PEN ERASE	0 PEN (ERASE exists)
LOCATE	LOOK (LOCATE exists)

It should be mentioned that for this package, virtually the entire set of ATARI operating system graphics functions have been replaced by much faster (approximately 6 times) and much smarter graphic routines. Highlights of these improvements are:

* The PHIL and PHILTO functions allow filling to the left and/or right, filling across areas already filled, filling until a specific color is hit, filling until a specific color is not hit (i.e., re-filling), filling until hitting a set boundary regardless of what lies in the way, filling by either replacement or exclusive OR'ing, and filling into or out of corners without artifacting.

* The DRAW and DRAWTO functions allow drawing by either replacement or exclusive OR'ing, drawing until hitting a specified color, drawing until a specific color is not hit. Additionally, lines are more symmetric, and optional starting point plot is supported (the Atari routines never plot the first point of a line).

* All line drawing and fill routines allow plotting in wide and narrow screen width settings as well as normal ones, allowing true full screen graphics and memory conservative graphics, for the advanced programmer.

* Graphic mode 7+ ("7 and a half") is fully supported and is activated like any other graphic mode by using the GR. command and standard options. This mode is the four-color high-resolution graphic mode found in most of the better video games available for the Atari computers. (Antic mode 14.)

* Display windowing and clipping is full supported. Options include "machine specific" coordinates for speed and "scaled" coordinates for portability.

* TURNTWD (turn toward) and 2LNX (two line intersection) commands are available allowing simplified perspective drawing.

Although the Atari 400/800 computers have extensive graphic capabilities the need to keep the ROM operating system under 10K apparently forced Atari to omit the highest resolution color graphic mode (graphic mode 7+) and allowed only inefficient draw and fill routines to be implemented. Since this severely limited the usefulness of the computer for plotting, both of these problems have been corrected.

A New Graphic Mode

Many of the better video games for the Atari 400/800 computers use a color graphic mode not supported in BASIC. By redefining the GR. command, it was possible to implement this previously unavailable mode.

This new graphic mode, which in this package is called graphic mode 12, is similar to graphic mode seven. The difference is that a pixel (a single dot) in graphic 12 is half as tall as the same pixel in graphic mode 7. This mode is activated in the same manner as other graphic modes:

12 GR.

All options (i.e., split/full screen etc.) available for other modes will work with this new mode. In the split screen graphics 12 mode, there are 160 horizontal by 160 vertical pixel locations. In the full screen mode, there are 160 horizontal by 192 vertical locations. Note that to use this mode the valGraphics package must be loaded and the new draw routines must be used (the operating system routines fail in this mode).

Draw and Fill Routines

Because the line and fill routines in this package represent significant enhancement to the original operating system routines, an explanation of the why's and how's of this implementation is offered in the following.

It was first decided that the line-drawing routines must be speeded up so as to at least be in the class of routines of other 8 bit graphic machines. Because of the differing bit structures in the various graphics modes, these routines take up about 1000 bytes of memory. This was deemed a reasonable tradeoff. Since a complete rewrite had thus been elected, the opportunity was taken to expand the versatility of the routines, trading a small portion of the speed increase already gained. Several capabilities were deemed desirable and were implemented:

* As mentioned above, the draw routines work in graphics mode 12.
* Assuming that display memory has been properly laid out, the draw routines work in wide and narrow screen widths as well as the normal ones.
* The draw and fill functions, at user option, XOR rather than replace pixels in display memory so that new images can be written over background images. (Images are then erased by rewriting, restoring the background image.)
* The draw and fill functions can detect a variety of conditions so as to allow concepts like "draw until" and "draw until not" as well as "fill until" and "fill until not."
* The fill function allows the edge color and the surface color to be different, at user option, with the default setting that they are the same.
* The fill function allows filling to the left, right, or both simultaneously, at user option.
* Fills are able to start from and pass through corners without artifacting, at user option. (Implemented for vertical draw only.)
* Simple initialization of draw functions for custom display lists is provided.

These features were implemented and will be described shortly.

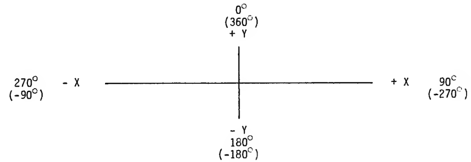
STROLLING THROUGH valGRAPHICS

Walking the Armadillo

To get started, insert your valGraphics disk and load in the armadillo package and all optional graphics packages ("+'s") (including the demos). It would even be a good idea to SAVE a copy of this system in case you crash later on. (Insert a formatted disk and type SAVE.) The load addresses may be found on screen 170 of the disk. Note that you do not need, and probably won't want, to first load the graphics package provided on your valFORTH 1.1 disk. Note also that as these packages load, some load comments may be reported as "xxxx Is not unique" and can be ignored. This message simply states that a word has just been defined with the same name as an already existing word.

When plotting in BASIC, location (0,0) is in the upper lefthand corner of the video display. All horizontal and vertical positions to the right and down are referenced with positive offsets from the (0,0) point. Armadillo graphics uses a somewhat different method to specify a location.

In armadillo graphics, the point (0,0) is located in the center of the display. Horizontal locations to right are referenced with positive offsets from this point, while locations to the left are referenced using negative offsets. Likewise, locations higher on the screen from the origin are referenced with positive vertical offsets while those lower on the screen are referenced using negative ones. Since this setup follows the standard cartesian coordinate system, function plotting is greatly simplified.



Let's take a look at the basic armadillo graphic commands. Type:

```
8 GR.     ON  ASPECT
```

This will put the system into graphic mode 8 with the armadillo positioned in the center of the display facing upward (0 degrees). The "dimensions" of the display are 320 pixels wide and 160 pixels high. The boundaries are set from -159 to 159 left-to-right, and 79 to -79 top-to-bottom. (The lowest line of pixels and the furthest right are excluded for code symmetry and shortness.) The command "ON ASPECT" will be explained later, but basically it ensures that squares will look like squares and not like rectangles (as in BASIC). First let's turn the armadillo to the right. To change its direction, we use the TURN command:

```
90  TURN
```

This command turns the armadillo clockwise by 90 degrees from its current direction. To draw a line (and move) the armadillo, the DRAW command can be used. Try this:

```
30  DRAW    90  TURN    50  DRAW
```

A short line should have been drawn toward the right -- 30 steps in the direction the armadillo was facing. The 90 TURN command was then used to aim the armadillo downward, and 50 steps in that direction were taken. The DRAW command moves the armadillo the specified number of steps in the direction that it is facing. Note that a negative step count tells the armadillo to draw in the direction opposite that in which it faces. It is also possible to move the armadillo to a specified point on the screen regardless of which direction it is facing. The DRAWTO command is used for this:

```
0  -60  DRAWTO    0  TURNTO
```

Although the armadillo was facing down, it moved directly to the point (0,-60). Note that although it moved diagonally, it still is facing directly downward (to 180 degrees). The TURNTO command is used to face the armadillo in the specified direction regardless of where it is currently facing. In this case, the armadillo is turned to face 0 degrees.

In addition to drawing lines as it moves, the armadillo can fill in areas of the display. The PHIL command is used for this purpose and functions very much like the DRAW command. (FILL is already defined and if used mistakenly for PHIL, the system will probably crash.) Try this:

```
20  PHIL
```

This commands the armadillo to take 20 steps in its current direction filling the surface area to its right as it goes (the area to the left can be filled also -- more on that later). Similar to the DRAWTO command, there is also a PHILTO command which works just like PHIL except that the armadillo moves to a specified point regardless of the direction it is currently facing. To PHIL to the origin (0,0), use:

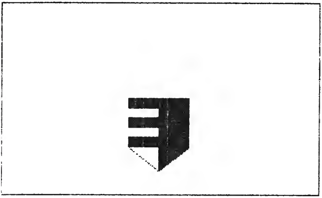
```
0  0  PHILTO
```

The PHILTO command should have filled straight up to the point (0,0).

So far, we have used the TURN, TURNTO, DRAW, DRAWTO, PHIL, and PHILTO commands. These are the basic "drawing" words, used constantly, when working with armadillo graphics. You will encounter times when you need to move the armadillo without drawing a line between its starting point and its destination point. There are four similar commands which allow this. The GO, GOTO, GO., and GOTO. All reposition the armadillo without drawing a line. The GO and GOTO commands function like DRAW and DRAWTO respectively; however, the armadillo is placed at the position where the last dot of the plotted line would have been and no line is drawn. GO. and GOTO. function like GO and GOTO; however, a single point is plotted at the destination point. Try this:

```
-30  0  GOTO.     180  TURNTO
10  PHIL     10  GO     10  PHIL     10  GO     10  PHIL
0  -60  DRAWTO
```

After entering the above, type FRAME to frame this picture. If all went well, your display should look like:



Let us now explore the new graphic 12 mode. In this mode, there are four colors numbered zero to three. When the armadillo is moved, there must be some way to specify which color to DRAW with. The PEN command is used for this purpose. Enter the graphic 12 mode by typing:

```
12 GR. ON ASPECT
```

The GR. command automatically sets the draw color to one (usually red). Let's draw some colored lines now:

```
10 DRAW                          (draw in color 1)
2 PEN        10 DRAW             (draw in color 2)
3 PEN        10 DRAW             (draw in color 3)
```

You should now have a vertical red, green, and blue line. Note that color 0 is black (actually background) and is used primarily for erasing lines. Besides setting the draw color, the PEN command also sets the PHIL color.

```
-50 0 GOTO    45 TURN
2 PEN     10 PHIL
```

After positioning the armadillo in a good position for filling, the draw and fill color is set to 2 (usually green) and 10 steps are taken. By using PHPEN command, it is possible to set the PHIL color to something other than the PEN color. PHPEN stands for "phil-pen" and is used in the same manner as the PEN command:

```
3 PHPEN    32 PHIL
```

Note how the edge line remains the color set by the last PEN command, while the PHIL command uses the color set by the last PEN or PHPEN (whichever came last) command. The PHIL color is always set by the PEN command for convenience. Experiment with this a bit.

To clean the current display, the WIPE command is used. Usually after wiping the display, the armadillo is repositioned to the center of the screen using either the CENTER or CENTERO command. The CENTER command simply does a "U 0 GOTO" while the CENTERO command does a "CENTER 0 TURNTO".

```
WIPE    CENTERO
```

Either right or left filling can be performed, as well as both simultaneously. The two commands RPHIL and LPHIL take an ON/OFF value and instruct the next PHIL or PHILTO command to take appropriate action. The default setting is "ON RPHIL" and "OFF LPHIL". <u>The command DINIT will return all settings to their default values. This is especially valuable when learning, as it is easy to get fouled up</u>. Type in the following set of commands and observe what happens:

```
50 50 PHILTO
CENTER    OFF RPHIL    ON LPHIL
-50 50 PHILTO
CENTER    ON RPHIL    1 PEN
40 PHIL
```

This demonstration first shows right filling, then left filling, and then simultaneous right/left filling. Note that although there was nothing on the screen to halt the right/left fills, they still stopped upon hitting the edge of the display. In fact, by using windows (which will be described later), fill boundaries can be set anywhere on the screen and fills will never occur outside of those boundaries. This is invaluable when trying to restrict drawing and filling to a select portion of the display.

Another unique feature of the fill routines is that they allow filling over any pseudo-background color (default is 0). The PHBAK command is used to specify this background color. Like the PEN and PHPEN commands, PHBAK accepts a color specification on the stack. WIPE uses the color specified by the last PHBAK command, and the fill routines recognize this as background to be filled over. Try this:

```
3  PHBAK     WIPE
CENTERO     0 PEN     1  PHPEN
50  50  PHILTO
```

For the time being, we will leave the background color blue and continue on. Next we are going to define a few words which will draw simple shapes. Bear in mind that when defining shape words, TURNTO, DRAWTO, and PHILTO should be avoided as they are absolute in nature. Typically, figures should be drawn relative to the armadillo's direction. Likewise, the armadillo should generally be returned to its original position and heading once the "canned" shape has been drawn. (For the curious, the words DX1 and DY1 return the x-y coordinates of the armadillo's current location. The word DAZM returns the directional angle of the armadillo.) We shall now define a word which will draw a square on the screen.

```
:  SQUARE                              ( #steps/side  ---  )
   DUPDRAW   90 TURN
   DUPDRAW   90 TURN
   DUPDRAW   90 TURN
      DRAW   90 TURN  ;

WIPE   CENTERO  2  PEN
20  SQUARE
45  TURN      20 SQUARE
```

There are several points to be mentioned here. First, because combinations of DUP with DRAW, GO, and GO. occur often, the words DUPDRAW, DUPGO, and DUPGO. have been defined to conserve memory.

Also notice that the squares drawn really have sides of equal length (in BASIC, the vertical legs would be much shorter). The armadillo package performs "aspect ratio" calculations which ensures that "equal" lines are drawn the same length regardless of their orientation to a fixed axis. These routines were enabled at the beginning of this stroll with the "ON ASPECT" command. Because these calculations do take time (approximately 3 milliseconds per draw), they can be turned off using the command:

```
OFF  ASPECT
```

Now, notice how in the last example, the second square was rotated 45 degrees from the first square. We can write simple words using this effect that look pretty snappy on the screen:

```
: FAN                           ( #steps/side ---  )
  20  0                         ( 20 squares for fan )
  DO
    18   TURN                   ( 360 degrees/20 = 18 )
    DUP   SQUARE
  LOOP
  DROP ;

WIPE    CENTER     30  FAN
```

This word draws 20 squares on the screen each offset from each other by 18 degrees. Try changing PEN colors and give different step sizes to FAN and watch the results. Each of the boxes drawn by FAN is the same size. We can write another simple word which will slightly increase the size of each box drawn and obtain a different effect:

```
: WHIRL                         (  #boxes --- )
  ( #boxes )  0
  DO
    1  3 /  SQUARE              (  increase size )
    5  TURN
  LOOP  ;
WIPE     CENTER    250  WHIRL
```

This word draws the specified number of boxes, each one rotated from the last by 5 degrees. After three boxes are drawn, the box size is increased. This is how the swirl effect is obtained. A slight variation of this is to change the PEN color before each square is drawn, but this is left to the reader.

Up to now, we have drawn lines from one point to another regardless of what the line replaces. This is standard for line drawing routines. In the valGraphics package, however, "draw until" is supported. In other words, lines can be drawn that will stop on the first occurrence of another line (actually, until the color specified by PHBAK or DRBAK, whichever came last). When the draw-until switch DRUNT is ON, all DRAW, DRAWTO, PHIL, and PHILTO commands will stop when the base line hits another line on the display. Here's an example:

```
ON  DRUNT    WIPE   CENTERO
40  SQUARE
```

Don't worry if only the two vertical sides of the square were drawn, this is normal. Since the draw routines in this package plot both the end point <u>and</u> the starting point, the end point of the first side stopped the line draw of the second side. In most cases, this is the desired function for DRAW, but while drawing-until (ON DRUNT), first point plotting is not desired. For this reason, it can be easily turned off using the DR1ST switch:

```
OFF  DR1ST    WIPE
40  SQUARE
```

With DRUNT still on, try the following example:

```
WIPE    3D  QCIRCLE
40  SQUARE
```

The QCIRCLE command draws a quick-and-dirty circle about the armadillo, with the value on top of stack taken as the approximate radius. Notice how the 40-step square turned out. Because the draw-until mode is on, each side is drawn until another line is encountered. Note, however, that even though the line was not drawn to the destination point, the armadillo was still positioned there. Because interesting results can be obtained by using this feature, the RELOC command must be used explicitly to reposition the armadillo to the last plotted point of the line. The following two definitions might come in handy:

```
: DRWUNTIL                    ( #steps --- )
  DILLO
  ON DRUNT
  DRAW DFLG
  IF RELOC ENDIF
  OFF DRUNT ;

: DRW2UNTIL                   ( x  y  --- )
  DILLO
  ON DRUNT
  DRAWTO DFLG
  IF RELOC ENDIF
  OFF DRUNT :
```

(DFLG is a flag set true only if the last DRAW or DRAWTO crossed the current window.)

These two commands will automatically reposition the armadillo at the end of the drawn line after each draw. One last point about draw-until -- occasionally it is desirable to know when a draw-until line was stopped by the draw-until function, rather than by reaching an end point or window boundary. The ?DRSTP word will return a one (1) if the last line was stopped, otherwise it will return zero. Try drawing a few lines and verifying this. ?DRSTP is in the DILLO vocabulary. (See the glossary.)

Up until now, when we filled areas or drew-until, both the fills and draws would stop when encountering a non-pseudo background color (set by PHBAK or DRBAK). Often, it is desirable to refill an area (i.e., fill until background is hit) or draw-until hitting the pseudo background color. There are two switches which can be turned ON or OFF as desired. The PHUNOT (fill until not) switch, when ON, fills until the color set by the last PHBAK command is not hit. This is the defaulat condition (i.e., fill until background is not hit). When OFF, the fill routines continue to fill until the pseudo background color is hit. Likewise, the DRUNDT (draw until not) switch, when ON, draws until the color set by the last PHBAK or DRBAK (whichever came last) command is hit. Let's take a look at this:

```
DINIT                              (Reset draw/fill switches)
2  PEN      3  PHPEN
WIPE    CENTERO      50  5D  PHILTO
1  PHPEN     180  TURNTO
OFF  PHUNOT              (fill while not background)
OFF  RPHIL      ON  LPHIL      6D  PHIL
```

As you may recall, the DINIT command initializes all eleven switch settings (five of which have yet to be introduced). Next a normal right fill (filling over background) is performed. The PHUNOT switch is then set for filling while not background and a left fill is performed. Notice that no filling occurred when the base fill-line extended out of the previously filled area.

DRUNOT works in the same manner. Execute the last example a second time, but turn both PHUNOT and DRUNOT off where previously just PHUNOT was turned off. Also turn DRUNT on. This time, no line should extend past the previously filled area. (Note that the base line of a fill responds to all the draw switches).

To finish off this first part of the stroll, the final five draw switches will be explained. Briefly, they are PH-DR which allows the base line of a fill to be drawn or not, DRXOR and PHXOR which allow lines and fills to be XOR'd into place, PHCRNR which enables/disables rudimentary corner check tests for filling, and PHUNT which allows filling to the edge regardless of what lies in the way.

The PH+DR switch is available because there are times when it is not desirable to actually draw the base line of a fill. This is the case when PAINTing (i.e., "shape filling," which is not supported but may be implemented). The default value for PH+DR is ON. When PH+DR is OFF , the pixels where the base line should be drawn are left untouched.

The DRXOR and PHXOR switches allow lines and fills to be XOR'd into place. This has the useful property that by simply redrawing or refilling the exact same line or shape the object will erase itself. For a good example of this, we can use graphic mode 8:

```
8 GR.  DINIT  250  WHIRL
1 PHBAK     ON  DRXOR     WIPE
```

Recall that the WIPE command uses the value set by the PHBAK command—in this case, one. WIPE is defined to use a multiple DRAW and therefore responds to most (but not all) of the draw switches. Because the WIPE is performed with the DRXOR mode on, the display is inverted. WIPE the display a second time to re-invert it. To erase the display, DRXOR must be turned off. Try this:

```
0  PHBAK     OFF  DRXOR     WIPE    ON     DRXOR
1  PEN     CENTERO     40   FAN
40  FAN                           ( one more time )
```

It is important to remember that lines drawn with pen zero have no effect in the DRXOR mode. Likewise, first point plot should generally be turned off when DRXOR is on otherwise endpoints will be lost. Now to demonstrate PHXOR and PHUNT try these examples:

```
DINIT          WIPE             (normal situation)
CENTERO     100  WHIRL     50  50  PHILTO
```

Now with PHUNT off:

```
CENTERO     OFF  PHUNT     50  50  PHILTO
```

The last fill command should have filled clear to the edge of the display, ignoring everything in its path. Using this with PHXOR, interesting results can be obtained:

```
WIPE      CENTERO        50  FAN
ON  PHXOR     50  50  PHILTO
```

Now, try:

```
CENTERO     50  50  PHILTO
```

By using windows (described later), the fill and draw commands can be restricted to selected areas of the display. In combination with windows, PHXOR can produce astounding visual effects (especially in GTIA modes).

The last remaining switch to be described is the PHCRNR switch. PHCRNR allows rudimentary corner checking for vertical fills. Because its use is specialized, PHCRNR is normally turned off. The following example will show its function:

```
DINIT      CENTERO     WIPE
50  GO      50  0  DRAWTO     0  0  DRAWTO    50 PHIL
```

Notice the artifact at the top corner. Now, turn PHCRNR on and perform the same example (less the DINIT command). This time, no artifact should have appeared. It is important to remember that these corner tests will not work with many diagonal fills, and completely fail when refilling an area. Also note that when first point plot is disabled, even vertical filling fails.

All of the basic armadillo commands have been explained and are summarized in the glossary and on the valGraphic Handy Reference Card. Although many commands have been discussed, there are many more left to talk about. These include the perspective drawing commands for three dimensional displays, and the complete set of window commands which will be described next.

(NOTE: In all of the above examples, WIPE has been used to clear the display. In many cases, the memory FILL command can be used instead: 88 @ n 0 FILL where n is the size of display memory in bytes. This method is much faster but cannot be used with windows.)

STROLLING THROUGH valGRAPHICS, PART II:

Windows, Lines and Labeling

Windows

Up until now we've been working in the base window that is set up when using the GR. command. Let's compose some other windows. Type:

```
12 GR.
FRAME
10 QUBE
-50 -10 30 -10 WINDOW
FRAME
DOT
```

We entered graphics 12, framed the base window, made a window whose left, right, top, and bottom edges were at -50, -10, 30, and -10 respectively, framed it, and then put a dot at the armadillo and found that it was at the center of the new window. Now type

```
25 QUBE
```

and note that the cube is clipped within the boundaries of the new window, not the old one. This could be very useful, say, in showing what was visible through a "real" window in a house that you had drawn, without going to a lot of extra trouble to restrict the image to the house's window. Now type

```
WIPE
CENTERO
25 QUBE
```

Nothing happens. This is because CENTERO centered the armadillo in the base window. We need to use a different word to re-center in the new window. Type

```
WCTRO 25 QUBE
```

That's more like it. WCTRO stands for "Window CenTeR 0 turnto," and there is also just a WCTR, for "Window CenTeR." Let's try some of the other tricks from before:

```
3 PHBAK
WIPE
WCTR
20 QUBE
FRAME
ON DRXOR
WIPE
```

Get the idea? When we did ON DRXOR, the draw routines, which are used by WIPE, started doing an XOR instead of a replace, with the same effect as we've seen before, but this time restricted to a smaller window. Type

```
DINIT
WIPE
DOT
```

to get things back to normal. Note that DINIT returns to the base window and so WIPE wipes the entire screen. The window in which we were just working is forgotten. (We'll discuss ways to remember it a little later.) DOT shows that the armadillo is back at the center. Now type

```
ON ASPECT
-50 -10 30 -10 WINDOW
FRAME
```

and you see that this "same" 40 by 40 window as before now looks much more nearly square. This illustrates that ASPECT works on windows as well as lines. With ASPECT on, what you give up in order to get better shapes is some information about what coordinates the top and bottom of the screen actually are, but for "hands on" use this is not much of a loss. Let's make two more of this type of window:

```
2 PEN
0 30 20 -30 WINDOW FRAME
DOT
```

OK, and then

```
-30 30 90 -90 WINDOW
```

Notice that this window is larger than the base window. Now type

```
DOT
FRAME
```

and notice the trash in the text window. If you choose to make a window larger than the base window, the system will not protect you; it assumes that you know what you're doing.

Type CLS once or twice to clear the screen. Then type

```
12 GR.
10 50 -10 -60 RELWND
FRAME
2 PEN
40 QUBE
```

Interesting. Now a 40 QUBE used to be much bigger; but because we typed RELWND instead of WINDOW, objects are drawn relative to the new window, as if it were the base window. Type

```
3 PHBAK
WIPE
FRAME
10 50 -10 -60 RELWND
FRAME
40 QUBE
```

Get the idea? Relative windows are useful for all sorts of tricks. Often, it would be helpful to be able to return to a window, and relative windows are the hardest to reconstruct. Try typing, on one line,

```
THISWND  LIVING-ROOM
```

(Defining words should always be followed by the name of the new word on the same line.)

By typing LIVING-ROOM later on we can return to this window, as a relative window, with no further work. To demonstrate, type

```
0 PHBAK
WIPE
BASWND
ON DRXOR
1 PHBAK
WIPE
-20 0 20 0 WINDOW WIPE
THISWND MY-ROOM
LIVING-ROOM
WIPE
50 QUBE
MY-ROOM
50 QUBE
```

Normal windows, created by WINDOW, of course can also be named more directly:

```
: window-name number number number number WINDOW ;
```

and you've got it.

Well, what else? Type

```
40 GR.
```

What have we here? 40 is 32 + 8 so we've entered 8 GR. without pre-erasing. (This is one of the standard GR. options, you'll recall.) Since 12 GR. and 8 GR. occupy exactly the same display memory, what we see is the 12 GR. image data interpreted as 8 GR. Four color 8 GR. This effect has been written up in various places, and here it is. You can come back and play with this sometime. Right now, type

```
DINIT
WIPE
DOT (you may not be able to see it on your screen
     without adjustment.)
0 -30 GOTO 60 DRAW
30 0 GOTO 270 TURNTO 60 DRAW
(Now it's more visible)
87 -31 GOTO.
(Move away)
0 0 TURNTWD    ("turn-toward")
100 DRAW
```

You'll notice that the line doesn't hit 0,0 exactly. This is because the armadillo's direction is only represented to the nearest degree. Still, this is good enough for most purposes.

Finally, let's draw the a rectangular solid in two-point perspective. (The procedure in this example is not necessarily the best one, but it illustrates several capabilities. You might want to have the debugging package loaded from the valFORTH 1.1 disk, and have the stack turned on. That way you can follow the action on the stack also.) First we set up a horizon and two vanishing points:

```
WIPE
-200 60 GOTO 90 TURNTO 500 DRAW
-100 60 NAMEPT VP1  (name the point on stack)
 100 60 NAMEPT VP2
```

Then we "construct" the solid

```
CENTER VP1 TURNTWD 40 DRAW
THISPT PT1   (name the present point)
CENTER VP2 TURNTWD 30 DRAW
THISPT PT2
CENTERO 20 DRAW
THISPT PT3
VP1 TURNTWD MAKLN (leave a "line" on the stack)
PT1 GOTO 0 TURNTO MAKLN (leave a second line)
2LNX   (find their intersection)
NAMEPT PT4 (and name the point)
PT4 DRAWTO PT3 DRAWTO
VP2 TURNTWD MAKLN (do it again)
PT2 GOTO 0 TURNTO MAKLN (second line)
2LNX (intersection)
2DUP GOTO (make a copy then go there)
PT3 DRAWTO PT2 GOTO 2DUP DRAWTO (put in 2 more lines)
VP1 TURNTWD MAKLN PT4 GOTO VP2 TURNTWD MAKLN 2LNX
DRAWTO DRAWTO (finished)
0 GR. VLIST (see the new words:  point names.)
40 GR.  (Still there.)
```

In addition to MAKLN there is also THISLN which name the line the armadillo lies on, and NAMELN which will name a line on the stack. Given two points on the stack, 2PT-LN will change the four values into three, suitable for use with NAMELN. Practice, and some study of the glossary, will help. The user should realize that points and lines can't be named very easily within a program, but only while the program is loading. Within a program, use the stack or array structures for saving points and lines.

This stroll is not meant to exhaust the possibilities of this package, but merely to indicate them. A clever programmer, for instance, would have little trouble in figuring out how to interface this package to a joystick to make a very versatile sketchpad.

Hmmmmm?

(intentionally left blank)

Turtle graphics, and so also valGRAPHICS, uses a coordinate system different from that used by the Atari OS. In valGRAPHICS, the center of the graphics display is the point o,o. "x" values are positive to the right and negative to the left, while "y" values are positive toward the top of the display and negative toward the bottom. The maximum values of x and y may vary between display modes, depending on various user options that are selected. In this regard, see examples in the "Strolling Through valGRAPHICS" sections of this package, and also the words MCOOR and SCOOR in the "Windows" section below. In the glossary that follows, all mention of coordinates will apply to valGRAPHICS coordinates rather than to Atari OS coordinates.

The DRAW, DRAWTO, PHIL, and PHILTO commands support a number of options with a fair degree of complexity and power when used fully. These commands and options are discussed as a group at the end of the glossary and summarized with a chart which also appears on the handy reference card. The functions discussed are necessarily complex; however, the command DINIT ("d-init") is provided so that the user may return the system to a "standard-option" status during experimentation and practice, or during actual program execution.

The term "pixel" stands for "picture element" and refers to the smallest "point" which may be drawn in a given graphics mode.

As usual, "color" specification numbers refer to color registers. The actual colors in the color registers may be changed by various means, including loading the COLOR COMMANDS package from the valFORTH 1.1 disk and using the SETCOLOR or SE. command.

On GTIA-equipped machines in 10 GR. there are nine colors available, because the four player/missile color registers are also used. Since these registers sit just below the playfield color registers in memory, they may be set by using negative "playfield" numbers when using SE. . For instance, -3 PINK 6 SE. will set player/missile 1 (= -3 + 4) to PINK 6.

On GTIA-equipped machines in 9 GR. the "color" set by the various color commands below, e.g. PEN, PHPEN, PHBAK, etc., is interpreted explicitly as luminance between 0 and 15. The hue is that of the background color register.

On GTIA-equipped machines in 11 GR. the "color" set by the various color commands below is interpreted explicitly as a hue between 0 and 15. The lum is that of the background color register.

The term "armadillo" rather than "turtle" will be used in this package.

DILLO (short for armadillo) is a vocabulary that branches from FORTH. All of the system words in this package have been put in the DILLO vocabulary to keep them out of the way during VLIST and other tasks. Some little-used words are also in DILLO, though advanced users may want to get at them. To enter the DILLO vocabulary simply type DILLO and these words will now be recognized by the system. Note that since the word : generally puts the system back into the FORTH vocabulary, DILLO may have to be used within a colon definition. See the source code for numerous examples of this. Words in the DILLO vocabulary are so specified in the glossary below. (The word DILLO is immediate.)

For clarity, some definitions may be repeated. Within this glossary, however, the same name indicates the same word.

General Functions:

GR. (n --)

Appears to function as always, but is now much more powerful:

* For n = 12, or 12 with higher bits set for the usual options, the mode known popularly as 7+ will be activated. This mode is set by Antic instruction 14 and its characteristics are listed on the handy reference card which accompanies this package.

* For n = 3 to 12, (possibly with higher bit options), the appropriate graphics mode will be set up, and all armadillo parameters will be initialized. Note, of course, that if your machine does not have a GTIA chip, then modes 9, 10, and 11 will not operate as they should.

* For n = 0 to 2, (plus higher bit options), the system will respond as usual.

* GR. initializes a number of system and user quantities. Data about pixel and display-memory dimensions are sent to appropriate addresses. A pen color register of 1 is set by 1 PEN, and the background color register for fill commands is set to 0 by 0 PHBAK. OFF ASPECT is executed.

PEN (n --)

This command is used to change the color that the armadillo draws with. PEN sets a new color register, n, to be used by the DRAW, DRAWTO, PHIL, and PHILTO commands.

PHPEN (n --)

This command is used to change the color that the armadillo fills with. PHPEN sets a new color register, n, to be used by the PHIL and PHILTO commands. Note that PEN also sets a new color register for PHIL and PHILTO, so the value used by PHIL and PHILTO will be determined by whichever command, PEN OR PHPEN, was done last.

DRCLR (-- b)

Returns the present color used by the armadillo for drawing. DRCLR is in the DILLO vocabulary.

PHCLR (-- b)

Returns the present color used for filling. PHCLR is in the DILLO vocabulary.

GO (n --)

GO moves the armadillo n units in the direction in which it is facing. No lines are drawn or points plotted.

DUPGO (n -- n)

Same as GO, but doesn't destroy stack argument.

GO. (n --)

GO. moves the armadillo n units in the direction in which it is facing and then pokes the pixel at its new location with the value set by the last PEN command.

DUPGO. (n -- n)

Same as GO., but doesn't destroy stack argument.

DOT (--)

DOT puts a dot of the present armadillo color, set by PEN, at the present armadillo position.

GOTO (x y --)

GOTO positions the armadillo at x,y. No lines are drawn or points plotted.

GOTO. (x y --)

GOTO. positions the armadillo at x,y and pokes the pixel at the new position according to the color register selected by the last PEN command.

CENTER (--)

Positions the armadillo at the point 0,0. The direction the armadillo is facing is unchanged.

CENTERO (--)

Positions the armadillo at the point 0,0 and turns it to face 0, i.e., straight up.

RELOC (--)

Positions the armadillo at the last point drawn by the system routines. This is a special purpose command and is used in conjunction with clipping in windows, and with the "draw-until" option, described elsewhere. RELOC is in the DILLO vocabulary.

ASPECT (ON or OFF --)

ON ASPECT will cause vertical components of subsequent graphics commands to be scaled to account for the fact that pixels are not square. Thus, circles will be rounder, squares will be squarer, and so on. Of course, shapes that previously fit on the screen may not fit any longer, as a result of the vertical expansion. OFF ASPECT will turn the compensation off for subsequent commands. OFF is the default mode, but this may be altered by changing "OFF ASPECT" to "ON ASPECT" at the end of the source code for GR.

LOOK (x y -- b)

This command returns the value of the pixel at location x,y. LOOK does not move the armadillo. For example, to find the color of the pixel under the armadillo, use the armadillo's coordinates: DX1 DY1 LOOK.

DX1 (-- n)

Returns the x coordinate of the armadillo.

DY1 (-- n)

Returns the y coordinate of the armadillo.

TURN (n --)

Changes the direction that the armadillo is facing by n degrees clockwise. Hence, if n is negative, the armadillo will turn counter-clockwise.

TURNTO (n --)

Turns the armadillo to a heading of n degrees from vertical. Hence, 0 TURNTO points the armadillo toward the top of the display, and 90 TURNTO points the armadillo toward the right edge of the display, and -90 TURNTO or 270 TURNTO both point the armadillo toward the left edge of the display.

TURNTWD (x y --)

Turns the armadillo so that it faces toward the point x,y. "Turn-toward."

DAZM (-- n)

Returns the direction, in degrees (0-359), in which the armadillo is facing. Stands for "dillo azimuth."

DINIT (--)

DINIT stands for "armaDillo INITialize." Use it to return all options to their default values and to center the armadillo in the display. Useful during practice and experimentation.

DRAW (n --)

Move the armadillo n units in the direction in which it is heading. Draw that portion of the line of travel of the armadillo, including the first point, that falls within the current window, using the current PEN value.

DRAWTO (x y --)

Move the armadillo to x y and draw that portion of the line of travel that falls within the current window, using the current PEN color register.

PHIL (n --)

Move the armadillo n spaces in the direction it is heading, and as in DRAW, color that portion of the path of travel with the PEN value. Also perform a fill to the right during the time that the armadillo is in the current window.

PHILTO (x y --)

Move the armadillo to the point x y. Then proceed as in PHIL.

Windows and Coordinate Systems

The following discussion is largely technical. Even so, it may be skimmed by the casual user, who can also get a "hands-on" feel for the operation of windows and coordinate systems by following the examples in the "Strolling Through valGRAPHICS" section of this package. The proliferation of quantities in this package is necessitated by allowing it to handle both "absolute" and "relative" windows at the same time. Because of the complex changes of coordinate system that this entails, a variety of different data are kept on system configuration. This process is transparent to the casual user but may be used with great power by the experienced programmer.

This package uses cartesian (rectangular) coordinate systems (CS's) throughout. For highest speed in graphics work, the graphics coordinate system should be in the same "scale" as the hardware. That is, moving one unit horizontally or vertically in the graphics CS should move the graphics cursor (in this case called the armadillo), one pixel. Doing this avoids additional, usually relatively slow, multiplication and division operations to make the graphics CS "fit" the hardware CS. However, sometimes the speed sacrifice is worthwhile in achieving a desired effect. Therefore, both types of CS are supported in this package. The default CS is of the first type, and it may also be called into play explicitly by the command "MCOOR" which stands for "machine coordinates." This mode is used for high-speed at some sacrifice of flexibility. The optional mode is called by "SCOOR" which stands for "scaled coordinates." Before executing SCOOR, the user may want to set up coordinate boundaries by using SET-SCALE, defined below. Moving between these two types of CS may also be handled automatically by the window routines discussed next. Because of automatic initialization routines in GR., the user may employ both machine and scaled CS's without ever calling them up explicitly. This happens through the commands WINDOW, which puts the system into the machine CS before interpreting its 4 stack arguments; and RELWND, which puts the system into scaled coordinates before interpreting its 4 stack arguments. (Clearly, RELWND must force the system into scaled coordinates, since it will be creating a window with the same numerical coordinates as the one RELWND works from, though the windows will generally be different sizes.) For some help in familiarization with these procedures, please refer to the examples in the "Strolling..." section.

A "window," for the purposes of this package, is a rectangular portion of the graphics display area. Windows are implemented to allow "clipping" as well as some additional scaling and distortion features. Clipping allows the armadillo to travel inside and outside the currently active window, while allowing drawing and filling only while the armadillo is within the boundaries of the window.

The current window's "physical" boundaries are kept in the system quans WNDLFT, WNDRGT, WNDTOP, and WNDBOT. (For an explanation of the QUAN structure, see the section on this topic.) The user does not generally access these quantities directly, but sometimes may want to do so for special effects. WNDLFT and WNDRGT are, respectively, the number of pixels from the left edge of the display to the left edge of the window, and the number of pixels from the left edge of the display to the right edge of the window. Similarly, WNDTOP and WNDBOT are referenced from the top of the display. Again, the user doesn't have to use these quantities; they are, however, the "bedrock" of the windowing process. These quans are in the DILLO vocabulary.

When a graphics-type GR. command is executed (3-12, see GR. above) a window, called the "base window," is set up which takes up the entire graphics display area. (The "physical" edges of the base window are stored in the system quans WNDL, WNDR, WNDT, and WNDB, which have meanings similar to WNDLFT, etc., above, and are likewise in the DILLO vocabulary.) The user may generally return to the base window at any time before leaving the graphics mode by executing BASWND. When the base window is made current by the user explicitly or by GR., the armadillo is placed at the point 0,0, i.e., the center of the window, and turned to 0 degrees, or straight up. The default "numerical" values of the window-boundaries are set so that they correspond to pixel counts vertically and horizontally. For instance, in 7 GR. the numerical boundaries would be +-79 horizontally (since there are 160 pixels across the display in that mode), and +-39 vertically (since the mode is 80 pixels high.) These values are stored in the system quans WNDW, WNDE, WNDN, and WNDS, which stand for "window-west," etc. These values may be altered by means described below (SET-SCALE), although the change will slow down the draw routines because of the extra transformation required when not working in the "natural" coordinates of the system.

After initializing to a graphics mode with GR., the user may use the various commands in this package to create graphics displays in the base window. However, additional flexibility is available to the user by defining new windows, as follows.

The command WINDOW is used to define temporarily a rectangular area of the display as the current window. This definition will last until the next window defining command e.g., WINDOW, BASWND, GR., DINIT, etc. WINDOW defines the window in the coordinate system of the base window. Indeed, WINDOW does BASWND before proceeding. (The base window is set up automatically by GR., or by DEFBAS when using a customized display list.) WINDOW expects four arguments on the stack, namely the left, right, top and botton edges of the new window, expressed in the coordinate system of the base window. (RELWND ("rel-wind"), defines a window relative to the current window, not the base window; its description otherwise parallels that of WINDOW.) When WINDOW is executed, a new window is made current, and all applicable internal quans are altered as appropriate. The armadillo is centered in the new window and turned to 0 degrees. The numerical boundaries of the new window will be, as stated before, WNDW, WNDE, WNDN, and WNDS.

(Advanced users: NOTE that, when in a GR. mode, decimal 88 @ will leave the address of the byte in the upper-left-hand corner of the display. Internal calculations are based on this location. In general, if the user wishes to redirect the graphics routines in this package to a display memory area in a non-GR. display mode, he or she need do two things: Store the appropriate value into memory location decimal 88, and then execute DEFBAS, described below, to establish a base window. Note, however, that if your display memory makes a discontinuous jump, as can occur for instance when crossing a 4K boundary, the graphics routines will not function properly.)

Additionally, the window-naming word, THISWND, is provided for ease simplicity in returning to a specific window.

Reference on clipping algorithms:
<u>A Practical Introduction to Computer Programs</u>, Ian O. Angell.

Windows and Coordinate Systems

WINDOW (left right top bottom --)

Sets a new window whose boundaries, expressed in the coordinate system of the base window (not the current window), are taken from the stack in the order indicated. The armadillo is centered in the new window and turned to a zero angle. Machine coordinates are activated. (See MCOOR).

RELWND (left right top bottom --)

Makes current a window whose edges are as indicated on stack in the coordinate system of the current window (not the base window). Scaled coordinates are activated. (See SCOOR).

WIPE (--)

Colors the entire current window according to the color register selected by the last PHBAK command. Note that since WIPE uses the system routine DRAWLN it will be affected by DRXOR. Hence if ON DRXOR has been executed last then WIPE will XOR all pixels in the entire current window with the value set by PHBAK, rather than replacing them with that value. This is useful for interesting and often eerie effects.

FRAME (--)

Draws a line around the current window according to the color register selected by the last PEN command.

BASWND (--)

Makes the base window (usually the full window first put up by a GR. command) current, centers the armadillo and turns it to 0 degrees.

THISWND xxx, (--)
xxx: (--)

Creates a word, xxx, which when executed makes current the window which was current at the time xxx was defined. Also centers the armadillo and turns it to 0 degrees, and restores XFORM to its state at the time xxx was defined. Located in the "Window Naming" package.

DEFBAS (left right top bottom --)

Advanced users. Used to set up a base window when not using GR.. The values indicated are the number of pixels from the left edge of the display (for left and right) and from the top edge of the display (for top and bottom). Before using this command, the value at decimal 88 should be set to point to the byte that represents the upper-left-hand corner of the display area to be used for graphics. DEFBAS is in the DILLO vocabulary.

SET-SCALE (horiz vert --)

Used to redefine the horizontal and vertical numerical boundaries of windows. After executing SET-SCALE, the SCOOR (stands for "scaled coordinates") command will set windows to range horizontally between +-horiz and vertically between +-vert. Note that the point 0,0 will remain the center point of windows. Since the command RELWND does SCOOR, relative windows will reflect use of SET-SCALE. SET-SCALE is in the DILLO vocabulary.

MCOOR (--)

Sets the horizontal and vertical numerical boundaries of windows to correspond to the number of pixels in each direction in the base window. "MCOOR" stands for "machine coordinates." It is not generally accessed directly by the user, with one exception: After having done a RELWND and returning to the base window by BASWND, an increase in speed may be had by executing MCOOR, if the user was using the default scale set automatically by GR. This is a fine point, but worth noting. MCOOR is in the DILLO vocabulary.

SCOOR (--)

Sets the horizontal and vertical numerical boundaries of windows to correspond to the default values set by GR. or by values set by SET-SCALE. "SCOOR" stands for "scaled coordinates." It is not generally accessed directly by the user. SCOOR is in the DILLO vocabulary.

:WCTR (--)

Centers the armadillo in the current window.

:WCTRO (--)

Centers the armadillo in the current window and turns it to 0 degrees.

Line-naming and line manipulation; point-naming

These packages support labeling various graphics "entities" for convenience in recalling them subsequently, for a variety of purposes.

Lines are stored internally as three-number quantities which are the (non-unique) A, B, and C parameters in standard algebraic line notation. (See the section on The Straight Line in Mathematical Handbook for Scientists and Engineers, 2nd Edition, by Korn and Korn. Point/slope representation is insufficient; point/azimuth representation would work but was not used because of some doubts concerning execution speed.) Labeling of lines is done principally for subsequent geometric-construction-type operations, like finding the intersection of two lines, or the point where the armadillo would intersect a given line.

```
NAMEPT  xxx,  ( x y -- )
        xxx:  ( -- x y )
```

Creates a word xxx. When xxx is executed, it returns x and y to the stack.

```
THISPT  xxx,  (----)
        xxx:  ( -- x y )
```

Creates a word xxx. When xxx is executed, it returns to the stack the x and y coordinates of the armadillo in the coordinate system of the window current at the time xxx was created.

```
2PT-LN  ( x1 y1 x2 y2 -- a b c )
```

Takes the coordinates of two points on the stack and leaves A, B, and C coefficients of the line connecting two points. "Two-point-line."

```
MAKLN       ( -- a b c )
```

Pushes to stack the A, B, C representation of the imaginary line on which the armadillo is sitting and along which it faces. Useful in finding where the armadillo would intersect a line along its current path. ("Make-line.")

```
NAMELN      xxx, ( a b c -- )
            xxx: ( -- a b c )
```

Creates the word xxx. When xxx is executed, it returns the values a b c to the stack.

```
THISLN      xxx, ( -- )
            xxx: ( -- a b c )
```

Creates the word xxx. When xxx is executed, it returns the A B and C values of the line that the armadillo was sitting on and facing along when xxx was created. ("This-line.")

```
2LNX  ( a1 b1 c1 a2 b2 c2 -- x y )
```

Given two lines on the stack in a b c form, 2LNX returns the point of intersection of the two lines. If the lines are parallel or if their point of intersection is very distant and would cause coordinate overflow, 2LNX will leave -1, -1. ("Two-line-intersection" or "Two-line-X.")

Options

The basic commands, followed by the commands that operate the "switches" on options, are described below.

DRAW (n --)

Standard option: Move the armadillo n units in the direction in which it is heading. Draw that portion of the line of travel of the armadillo, including the first point, that falls within the current window, using the current PEN value.

ON DRXOR: XOR pixels with the PEN color instead of overwriting them with the PEN color.

ON DRUNT: Stop on hitting a pixel of the value selected with the last DRBAK or PHBAK command, whichever was last.

OFF DRUNOT: DRUNOT makes a difference only when ON DRUNT has been executed. When DRUNOT is off and DRUNT is on, lines halt upon hitting a pixel of the last color set by DRBAK or PHBAK, whichever was executed last. When DRUNOT is on, which is the default case, and DRUNT is on also, lines will halt upon hitting a pixel not of the last color set by DRBAK or PHBAK, whichever was executed last.

OFF DR1ST: Don't draw the first point in a line. Useful when drawing connected lines after ON DRUNT so that the last point of a line won't be interpreted as the stop condition of the next line. See "Strolling..." for an example.

DRAWTO (x y --)

Standard option: Move the armadillo to x y and draw that portion of the line of travel that falls within the current window, using the current PEN color register.

ON DRXOR: XOR pixels with the PEN color instead of overwriting them with the PEN value.

ON DRUNT: Stop on encountering a pixel of the color selected with the last DRBAK or PHBAK command, whichever was last.

OFF DRUNOT: DRUNOT makes a difference only when ON DRUNT has been executed. When DRUNOT is off and DRUNT is on, lines halt upon hitting a pixel of the last color set by DRBAK or PHBAK, whichever was executed last. When DRUNOT is on, which is the default case, and DRUNT is on also, lines will halt upon hitting a pixel not of the last color set by DRBAK or PHBAK, whichever was executed last.

DRAWTO (cont'd)

OFF DR1ST: Don't draw the first point in a line. Useful when drawing connected lines after ON DRUNT so that the last point of a line won't be interpreted as the stop condition of the next line. See "Strolling..." for an example.

?DRSTP (-- f)

?DRSTP is a quan whose value is adjusted after each DRAW and DRAWTO. If ?DRSTP is true (non-zero) then the last DRAW or DRAWTO was terminated because ON DRUNT had been executed and the line-drawing routine encountered a pixel whose value was that selected by the last DRBAK or PHBAK command, whichever was last. ?DRSTP is useful in conjunction with RELOC.

RELOC (--)

Relocates the armadillo to the location of the last pixel drawn by the last DRAW or DRAWTO command. If no points were drawn by the last DRAW or DRAWTO command, (e.g., if the line fell entirely outside the current window) then the armadillo is not moved. RELOC is useful in conjunction with ON DRUNT. See example in "Strolling..." RELOC i in the DILLO vocabulary.

DRAWLN (column row --)

A system routine, not intended for general use. This high-speed routine replaces the DRAWTO routine in valFORTH 1.1, which used the same OS routine as the BASIC DRAWTO command. DRAWLN is in the DILLO vocabulary.

PHIL (n --)

Standard option: Move the armadillo n spaces in the direction it is heading, and ...

As in DRAW, color that portion of the path of travel with the PEN value. Also perform a fill to the right during the time that the armadillo is in the current window. The color of the fill is set either by the PEN value or the PHPEN value, whichever was declared last. The fill will always terminate on reaching the edge of the current window if it has not been terminated prior to this event. The fill will also terminate on reaching a pixel that is not background color. In the standard option, the command ON PHUNT ("phil until") has been executed so that the fill will stop on the pixel of color register set by PHBAK, and 0 PHBAK has been executed so that the actual background register, 0, will also be used as the phil "background" register. ON RPHIL and OFF LPHIL have been executed so that the fill will be toward the right only. ON PH+DR has also been executed so that the line of travel of the armadillo is drawn in addition to the fill operation.

Example Options:

1 PHBAK: The fill will now stop on reaching a pixel of color register 1 (in this example), or the edge of the window.

ON PHUNOT 2 PHBAK: The state of PHUNOT only matters if ON PHUNT has been executed. The effect of PHUNOT ("fill until not") is that the fill will now stop on reaching a pixel NOT of color register 2 (in this example), or on reaching the edge of the window.

OFF PHUNT: Turning off fill-until means that now the fill will ONLY stop on reaching the edge of the window.

OFF PH+DR: Turning off PH+DR means that now the routines will not draw the line the armadillo is moving along, and will just fill as indicated.

ON LPHIL: Now the routines will also fill to the left.

OFF RPHIL: Now the routines will not fill to the right.

ON PHXOR: Now the routines will XOR the pixels with the PEN or PHPEN value, whichever was last declared, rather than replacing them with it.

PHILTO (x y --)

Move the armadillo to the point x y. Then proceed as in PHIL.

Options:
(All words below take a flag stack argument, and leave none.)

Switch	Default	ON	OFF
RPHIL	on	Enables right fill with PHIL, PHILTO	Disables right fill with PHIL, PHILTO.
LPHIL	off	Enables left fill with PHIL, PHILTO.	Disables left fill with PHIL, PHILTO.
DRXOR	off	DRAW, DRAWTO will xor pixels with line color.	DRAW, DRAWTO will replace pxls with line color.
PHXOR	off	PHIL, PHILTO will xor pixels with fill color.	PHIL, PHILTO will replace pxls with fill color.
DRUNT	off	Enable draw-until functions.	Disable draw-until functions.
PHUNT	off	Fill to edge of window or to dest. pixel.	Fill until encountering halt pixel cond set by PHBAK, PHUNOT.
DRUNOT	on	With DRUNT on, DRAW, DRAWTO draw until hit color set by DRBAK, PHBAK.	With DRUNT on, DRAW, DRAWTO draw until hit not color set by DRBAK, PHBAK.
PHUNOT	on	With PHUNT on, PHIL, PHILTO fill until hitting color set by PHBAK.	With PHUNT on, PHIL, PHILTO fill until hitting not color set by PHBAK.
PH+DR	on	PHIL, PHILTO draw line as filling.	PHIL, PHILTO don't draw line as filling.
DR1ST	on	First point of lines is drawn.	First point of lines is not drawn.
PHCRNR	off	PHIL, PHILTO perform corner checking, armadillo must be moving vertically.	No corner checking.

DINIT sets all switches to their default values.

Screen Dump

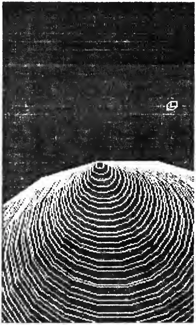
This graphics 8 screen-to-Epson/Graftrax dump routine was contributed by William Volk, who also collaborated on other parts of this package.

To dump graphics 8 screens (split or full), load this code and execute with GRDUMP. Some samples are shown below.

```
Scr #
( Dillo:  GRDUMP                     )

: DMPCOL DILLO            ( col -- )
   -1 WNDB DO
    DUP 88 @ +
    I BYT/LN * + C@
    EMIT
   -1 +LOOP
   DROP ;

: GRPLT DILLO                ( -- )
   ."          "
   27 EMIT 75 EMI1 WNDB
   WNDT    - 1+ EMIT Ø EMIT ;

                                 ==.
```

```
Scr #
( Dillo:  BOX-KITE

: BOX-KITE
   8 GR.
   5Ø Ø
   DO
     I TCIRCLE
   LOOP
   -6Ø 6Ø GOTO 5 QUBE
   ON DRXOR 1 PHBAK WIPE
   DINIT ;
```

```
Scr #
( Dillo:  GRDUMP                     )

: GRDUMP  DILLO
   ( turn off screen, on printer)
   PFLAG @ 2 PFLAG !
   ( set line/inch = 9 on Epson )
   27 EMIT 65 EMIT 8 EMIT
   ( dump the screen               )
   CR BYT/LN Ø
   DO
     GRPLT I DMPCOL CR
   LOOP
   27 EMIT 65 EMIT 12 EMIT
   CR CR PFLAG ! ;
```



Interfacing to Custom Display Lists

The advanced user wishing to interface valGraphics to a custom (non-GR.) display list should recognize that any area of display memory in which valGraphics will be required to draw must be continuous. Thus, for example, if a 4K memory boundary is crossed, necessitating a jump instruction in the display list, the user must ensure that display memory itself crosses the 4K boundary smoothly.

The location 88 decimal was used by the Atari OS to point to the byte in display memory corresponding to the upper left corner of the display, and has been adopted for the same purpose in this package. The first thing to do, then, is point 88 to the address in display memory that valGraphics should treat as the upper left corner of its drawing area.

The second step is to set up a base window, much as the GR. provided in this package does. Use the word DEFBAS to do this, as described in the glossary. Note that this word expects its arguments as numbers of pixels, and that "left" and "top" will usually be 0.

Finally, you need to tell the system what graphics mode you're drawing in. The word UGR. (for "user GR.") is provided for this purpose. Give it a number from 3 through 12, and it will set up quans like PX/BYT and so on. UGR. recognizes if you have set up for wide or narrow screen widths, also, and acts accordingly.

Do BASWND and the armadillo is centered, pointed up, and ready.

A note on QUAN structures

The "quan" is a new FORTH data structure, developed at Valpar, and being introduced in this package. Quans were devised to cut down on wasted memory and runtime encountered when using the "variable" data structure. Quans work as follows: (Advanced users may want to follow along in the source code for these structures also.)

Defining a quan:

QUAN BINGO

Note that quans do not take initial values. This form was chosen to allow for simpler upgrading to target-compiled code later on.

Giving a quan a value:

1234 TO BINGO

Note that since TO is immediate, "TO BINGO" compiles to only 2 bytes instead of the 4 bytes that would be required if BINGO were a variable (i.e., BINGO !).

Getting a value back from a quan:

BINGO

Simply saying the name of the quan will leave its value on the stack, in this case 1234. In this way, quans act like constants. BINGO above also compiles to only 2 bytes instead of the 4 required to fetch if it were a variable (i.e., BINGO @).

Getting the address of the data in the quan:

AT BINGO

This will leave the address of the first byte of data in BINGO on the stack, or compile the address as a literal if encountered during compilation. (AT is immediate.) This is useful for a variety of purposes in general programming and in interfacing to machine language routines.

Advanced users:

The FORTH 83 Standard appears to lean toward "non-state-smart" words, which is proper for target-compiled applications. We expect to support both "state-smart" and "non-state'smart" versions of various words, as appropriate for different users.

Note that while

15 AT BINGO +! and 15 BINGO + TO BINGO

accomplish the same task and take the same amount of memory, the first version is faster by one primitive nest.

The most significant internal feature of quan is that it has 3 cfa's instead of just the one common to most FORTH words. This initial 4 byte disadvantage is overcome at the second use of a quan, and so poses essentially no problem. CQUAN, 2QUAN, 3QUAN, etc., have also been implemented, and the user may have some fun puzzling these out before they are published elsewhere. Note that a 2quan takes 2 arguments from the stack when used with TO, and leaves 2 when used alone. When used with AT, a 2quan still leaves the address of the first byte of its "parameter field," as does QUAN. Also, when defining CQUAN it is probably a good idea to still allot 2 bytes for data, so that +! can be used without fear of negative stack arguments. Another new defining structure is called "FLAG." Flags have only two cfa's, dropping the one that supports the "AT" function. Flags keep only one byte of data, a flag; hence they are 3 bytes shorter than quans. Flags would not be used in this package enough to justify the additional code, but may be worthwhile in other applications.

Higher speed and cleaner array structures may also be implemented using the quan strategy, and may be included in a future release of our utilities-editor package. (This would be made available to current u/e owners at a price-difference-plus-handling charge.)

The word VECT has also been introduced in this package. It has two cfa's, and replaces the rather cumbersome variable-based vectoring procedure,

```
' SOMEWORD CFA  SOMEVARIABLE ! and
SOMEVARIABLE  @  EXECUTE
```

with the cleaner, faster, and memory-shorter

```
' SOMEWORD CFA TO SOMEVECT  and
  SOMEVECT
```

(intentionally left blank)

QUICK TRIG

Since floating point trigonometric operations on the Atari machines are rather slow and provide accuracy unnecessary for many applications, this package provides integer versions of sine, cosine, and arctangent functions that run much faster than their floating point cousins.

QSIN and QCOS expect scaled radian arguments in the range +-31416, (+-pi), with 10000 representing one radian.

Similarly, QATN returns scaled radian arguments in range +-15708 (+-pi/2). QATN accepts arguments in the full single number range, again interpreting 10000 as 1. This at first glance seems to be a significant limitation on QATN's input range but is circumvented by the existence of the more useful QATN2. QATN2 is a four-quadrant arctangent function. It accepts two stack arguments, which may be thought of as "delta-x" and "delta-y," and uses these arguments to construct a value to be used by QATN. QATN2 then performs sign corrections as necessary and returns a value in the range +-31416. QATN2 is what is actually used in graphics work, and is used in the word TURNTWD ("turn toward") elsewhere in this package.

For user convenience, the words ->QRD and ->QDG are used to convert from scaled-degree arguments to scaled radian arguments and back again.

16K/ (d -- n)

This is a special-purpose high-speed routine that may find other uses. It divides a double number by 16384 and leaves a single number result. Used to speed quick-trig functions.

QSIN (scaled-radians -- scaled-sine)

Takes a scaled-radian argument (range +-31416) and leaves the scaled sine in the range +-10000.

QCOS (scaled-radians -- scaled-cosine)

Takes a scaled-radian argument (range +-31416) and leaves the scaled cosine in the range +-10000.

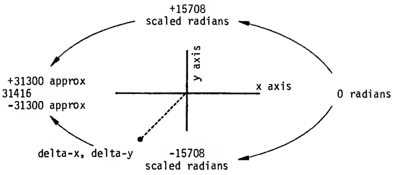
QATN (scaled-argument -- scaled-radians)

Takes a scaled-argument (range 0 to 10000) and leaves the scaled arctangent in the range +-15708 (+-pi/2, scaled).

QATN2 ("delta-x" "delta-y" -- scaled-radians)

Assuming that the "x" axis points toward zero radians (zero degrees) on one end and pi radians (180 degrees) on the other, QATN2 leaves the angle (range +-31416) between a line from the origin to the point (delta-x, delta-y).

Counterclockwise angles are positive.



In the illustration above, arctan2 of (delta-x, delta-y) would be approximately -2.1 radians, or -21000 as computed by QATN2.

->QRD (scaled-degrees -- scaled-radians)

Takes a scaled-degree argument (range +-18000) and converts it to a scaled-radian argument (range +- 31416).

->QDG (scaled radians -- scaled-degrees)

Takes a scaled-radian argument (range +-31416) and converts it to a scaled-degree argument. (range +-18000).

```
Screen:   1
( Dillo:  Drawline routine       )
HEX   DILLO DEFINITIONS
LABEL INCMOD                    ( sys )
  E6 C, C4 C, E6 C, C3 C, D0 C,
  02 C, E6 C, C2 C, A5 C, C2 C,
  F0 C, 02 C, 24 C, C4 C, 30 C,
  22 C, 85 C, C6 C, A5 C, C3 C,
  85 C, C7 C, 84 C, C5 C, A9 C,
  10 C, 85 C, C1 C, 06 C, C7 C,
  26 C, C6 C, 26 C, C5 C, 38 C,
  A5 C, C5 C, E5 C, C4 C, 90 C,
  04 C, 85 C, C5 C, E6 C, C7 C,
  C6 C, C1 C, D0 C, EB C, 60 C,
                               ==)
```

```
Screen:   2
( Dillo:  Drawline routine       )
  84 C, C6 C, 84 C, C7 C, A5 C,
  C3 C, 85 C, C5 C, E6 C, C7 C,
  38 C, A5 C, C5 C, E5 C, C4 C,
  85 C, C5 C, C5 C, C4 C, B0 C,
  F3 C, 60 C,
LABEL BUMPY                     ( sys )
  18 C, AD C, DRWRK ,      65 C,
  C8 C, 85 C, C8 C, AD C, DRWRK
  1+  , 65 C, C9 C, 85 C, C9 C,
  E6 C, 5A C, 2C C, DRWRK 1+  ,
  10 C, 04 C, C6 C, 5A C, C6 C,
  5A C, 60 C,                  ==)
```

```
Screen:   3
( Dillo:  Drawline routine       )
LABEL BUMPX                     ( sys )
  2C C, DRWRK 2+  , 30 C, 16 C,
  E8 C, 8A C, CD C, AT PX/BYT ,
  90 C, 08 C, A2 C, 00 C, E6 C,
  C8 C, D0 C, 02 C, E6 C, C9 C,
  E6 C, 5B C, D0 C, 02 C, E6 C,
  5C C, 60 C, CA C, 10 C, 0C C,
  AE C, AT PX/BYT , CA C, A5 C,
  C8 C, D0 C, 02 C, C6 C, C9 C,
  C6 C, C8 C, A5 C, 5B C, D0 C,
  02 C, C6 C, 5C C, C6 C, 5B C,
  60 C,
                               ==)
```

```
Screen:   4
( Dillo:  Drawline routine       )
LABEL PHTST                     ( sys )
  E6 C, C1 C, B1 C, C8 C, 3D C,
  MTBL2 , 8D C, DRWRK 4 + , AD
  C, AT PHUDAT , 3D C, MTBL2 ,
  CD C, DRWRK 4 + , D0 C, 07 C,
  2C C, DRWSTT 6 + , 30 C, 09 C,
  10 C, 05 C, 2C C, DRWSTT 6 + ,
  10 C, 02 C, C6 C, C1 C, 60 C,
                               -->
```

```
Screen:   5
( Dillo:  Drawline routine       )
LABEL (DOPHL)                   ( sys )
  A5 C, 5B C, 48 C, A5 C, 5C C,
  48 C, A5 C, C8 C, 48 C, A5 C,
  C9 C, 48 C, 8A C, 48 C, 84 C,
  C1 C, 20 C, PHTST , 20 C,
  BUMPX , 2C C, DRWSTT 4 + ,
  30 C, 36 C, 2C C, DRWSTT 7 + ,
  10 C, 31 C, A5 C, C8 C, 48 C,
  A5 C, C9 C, 48 C, 2C C, DRWRK
  3 + , 10 C, 08 C, 20 C,
  BUMPY , EE C, DRWRK 3 + ,
  F0 C, 0F C, 38 C, A5 C, C8 C,
  ED C, DRWRK ,      85 C, C8 C,
                               ==)
```

```
Screen:   6
( Dillo:  Drawline routine       )
  A5 C, C9 C, ED C, DRWRK 1+ ,
  85 C, C9 C, 20 C, PHTST ,
  68 C, 85 C, C9 C, 68 C, 85 C,
  C8 C, A5 C, C1 C, C9 C, 02 C,
  F0 C, 46 C, 38 C, A5 C, 5B C,
  ED C, AT WNDLFT , A5 C, 5C C,
  ED C, AT WNDLFT 1+ ,     30 C,
  39 C, 38 C, AD C, AT WNDRGT ,
  E5 C, 5B C, AD C, AT WNDRGT 1+
     , E5 C, 5C C, 30 C, 2C C,
  2C C, DRWSTT 4 + ,       30 C,
  09 C, 84 C, C1 C, 20 C, PHTST
     , A5 C, C1 C, D0 C, 1E C,
                               -->
```

Screen: 7

```
( Dillo:  Drawline routine      )
  B1 C, C8 C, 2C C, DRWSTT 5 + ,
  30 C, 03 C, 3D C, MTBL1 ,
  85 C, C1 C, AD C, AT PHDAT ,
  3D C, MTBL2 ,       45 C, C1 C,
  91 C, C8 C, 84 C, C1 C, 20 C,
  BUMPX , 4C C, HERE 44 - ,
  68 C, AA C, 68 C, 85 C, C9 C,
  68 C, 85 C, C8 C, 68 C, 85 C,
  5C C, 68 C, 85 C, 5B C, 60 C,
                                ==)
```

Screen: 8

```
( Dillo:  Drawline routine      )
LABEL (PHIL)               ( sys )
  2C C, DRWRK 5 + , 10 C, 20 C,
  AD C, DRWRK 2+  , 48 C, 2C C,
  DRWSTT 1+ , 10 C, 06 C, 8C C,
  DRWRK  2+ , 20 C, (DOPHL)     ,
  2C C, DRWSTT 2+ , 10 C, 08 C,
  A9 C, FF C, 8D C, DRWRK 2+  ,
  20 C, (DOPHL)   , 68 C, 8D C,
  DRWRK 2+  , 60 C,
                                ==)
```

Screen: 9

```
( Dillo:  Drawline routine      )
LABEL PIXEL                ( sys )
  B5 C, 00 C, 85 C, 5A C, 85 C,
  C4 C, B5 C, 02 C, 85 C, 5B C,
  B5 C, 03 C, 85 C, 5C C, 84 C,
  C2 C, 84 C, C3 C, A2 C, 08 C,
  A5 C, C4 C, 29 C, 01 C, F0 C,
  0E C, 18 C, AD C, AT BYT/LN ,
  65 C, C3 C, 85 C, C3 C, A9 C,
  00 C, 65 C, C2 C, 85 C, C2 C,
  46 C, C2 C, 66 C, C3 C, 66 C,
  C4 C, CA C, D0 C, E3 C, 18 C,
  A5 C, 58 C, 65 C, C4 C, 85 C,
                                --)
```

Screen: 10

```
( Dillo:  Drawline routine      )
  C8 C, A5 C, 59 C, 65 C, C3 C,
  85 C, C9 C, AD C, AT PX/BYT ,
  85 C, C4 C, A5 C, 5B C, 85 C,
  C3 C, A5 C, 5C C, 85 C, C2 C,
  20 C, INCMOD 8 + , 18 C, A5 C,
  C7 C, 65 C, C8 C, 85 C, C8 C,
  A5 C, C9 C, 69 C, 00 C, 85 C,
  C9 C, A6 C, C5 C, 60 C,
                                ==)
```

Screen: 11

```
( Dillo:  Drawline routine      )
LABEL DTST                 ( sys )
  E6 C, C1 C, AD C, AT DRUDAT ,
  3D C, MTBL2 , 8D C, DRWRK 4 +
  , B1 C, C8 C, 3D C, MTBL2 ,
  CD C, DRWRK 4 + , D0 C, 07 C,
  2C C, DRWSTT B + DUP  , 30 C,
  07 C, 10 C, 06 C, 2C C,       ,
  30 C, 01 C, 60 C, C6 C, C1 C,
  AD C, DRWRK F + , F0 C, 42 C,
  EE C, AT ?DRSTP , C8 C, 8C C,
  DRWRK E + , 88 C, AD C, DRWRK
  2+  , 48 C, 20 C, BUMPX ,
  B1 C, C8 C, 3D C, MTBL2 ,
                                --)
```

Screen: 12

```
( Dillo:  Drawline routine      )
  8D C, DRWRK 4 + DUP , AD C,
  AT DRUDAT    , 3D C, MTBL2 ,
  CD C, , D0 C, 07 C, 2C C,
  DRWSTT B + DUP  , 30 C, 0D C,
  10 C, 05 C, 2C C, , 10 C, 06
  C, CE C, DRWRK E + , CE C,
  AT ?DRSTP , 68 C, 48 C,
  49 C, FF C, 8D C,
  DRWRK 2+ DUP , 20 C, BUMPX   ,
  68 C, 8D C, , 60 C,
                                ==)
```

Screen: 13

```
( Dillo:  Drawline routine      )

CODE DRAWLN  DILLO     ( x y -- )
  86 C, D1 C, 8C C, DRWRK 1+ ,
  8C C, DRWRK 2+  , AD C,
  AT BYT/LN , 8D C, DRWRK DUP ,
  38 C, B5 C, 02 C, E5 C, 5B C,
  8D C, DUP 6 +   , B5 C, 03 C,
  E5 C, 5C C, 8D C, DUP 7 + ,
  10 C, 12 C, CE C, DUP 2+ ,
  38 C, 98 C, ED C, DUP 6 + ,
  8D C, DUP 6 + , 98 C, ED C,
  DUP 7 + , 8D C, DUP 7 + , AD
  C, DUP 6 + , 0D C, DUP 7 + ,
  8D C, DUP F + , 38 C, B5 C,
                              -->
```

Screen: 14

```
( Dillo:  Drawline routine      )

  00 C, E5 C, 5A C, 8D C, DUP
  8 + , 98 C, E9 C, 00 C, 8D C,
  DUP 9 + ,   10 C, 1A C, 38 C,
  98 C, ED C, DUP 8 + ,   8D C,
  DUP 8 + , 98 C, ED C, DUP 9 +
  DUP , 8D C, , CE C,  DUP 1+ ,
  38 C, 98 C, ED C, DUP , 8D C,
  DUP , A5 C, 5A C, 85 C, C4 C,
  20 C, PIXEL E + , 8C C, DUP
  A + , 84 C, C2 C, AD C, DUP
  8 + , 85 C, C3 C, AD C, DUP
  6 + , 85 C, C4 C, C5 C, C3 C,
  B0 C, 05 C, AD C, DUP 7 + ,
                              ==>
```

Screen: 15

```
( Dillo:  Drawline routine      )

  F0 C, 11 C, CE C, DUP A + ,
  A5 C, C4 C, 85 C, C3 C, AD C,
  DUP 7 + , 85 C, C2 C, AD C,
  DUP 8 + , 85 C, C4 C, 20 C,
  INCMOD ,  A5 C, C4 C, A5 C,
  C4 C, 8D C, DUP B + , 8D C,
  DUP C + ,   ) 84 C, C4 C, A9 C,
  FF C, 8D C, DUP 3 + , 8C C,
  AT ?DRSTP , 8C C, DUP D + ,
  2C C, DRWSTT A + , 10 C, 03 C,
  CE C, DUP D + , A5 C, C6 C,
  85 C, C2 C, A5 C, C7 C, 85 C,
  C3 C, 18 C, A5 C, C5 C, 65 C,
  ( WAS 46 )                  -->
```

Screen: 16

```
( Dillo:  Drawline routine      )

  C4 C, B0 C, 05 C, CD C, DUP
  C + DUP   , 90 C, 09 C, ED C,
, E6 C, C3 C, D0 C, 02 C, E6 C,
  C2 C, 85 C, C4 C, A9 C, FF C,
  8D C, DUP 5 + , 2C C, DUP D +
, 30 C, 36 C, 2C C, DRWSTT 9 + ,
  10 C, 0E C, 84 C, C1 C, 20 C,
  DTST , A5 C, C1 C, F0 C, 05 C,
  EE C, AT ?DRSTP , D0 C, 76 C,
  2C C, DRWSTT , 10 C, 08 C,
  20 C, (PHIL) , 2C C, DRWSTT
  3 + , 30 C, 16 C, B1 C, C8 C,
  2C C, DRWSTT 8 + , 30 C, 03 C,
                              ==>
```

Screen: 17

```
( Dillo:  Drawline routine      )

  3D C, MTBL1 , 85 C, C1 C,
  AD C, AT DRDAT , 3D C, MTBL2 ,
  45 C, C1 C, 91 C, C8 C, AD C,
  DUP A + , D0 C, 06 C, 20 C,
  BUMPY , 4C C, HERE 8 + , 20 C,
  BUMPX , 8C C, DUP 5 + , AD C,
  AT ?DRSTP , D0 C, 3D C, A5 C,
  C3 C, D0 C, 02 C, C6 C, C2 C,
  C6 C, C3 C, A5 C, C2 C, 05 C,
  C3 C, D0 C, A6 C, AD C, DUP
  A + , D0 C, 06 C, 20 C, BUMPX
, 4C C, HERE 5 + , 20 C, BUMPY ,
  8C C, DUP D + , CE C, B + ,
                              -->
```

Screen: 18

```
( Dillo:  Drawline routine      )

  F0 C, 03 C, 4C C, HERE 9A - ,
  A9 C, FF C, 4D C, DRWRK 1+ ,
  8D C, DRWRK 1+ , 20 C, BUMPY ,
  A9 C, FF C, 4D C, DRWRK 2+ ,
  8D C, DRWRK 2+ , 20 C, BUMPX ,
  A6 C, D1 C, 4C C,
  ASSEMBLER POPTWO ,

C;


DCX

                              ==>
```

```
Screen: 19
( Dillo:  PLOT                    )

: PLOT   DILLO        ( xa ya -- )
  [ DRWSTT 9 + ]L )R
  R @ 0 R ! )R
  2DUP 90 C! 91 ! DRAWLN
  R) R) ! ;
                                --)
```

```
Screen: 20
( Dillo:  [LOOK]                  )

HEX
CODE (LOOK) ( xa ya -- px# pxl )
  86 C, XSAVE C, 20 C, PIXEL ,
  B1 C, N 6 + C, 3D C, MTBL2 ,
  48 C, 8A C, A6 C, XSAVE C,
  95 C, 02 C, A9 C, 00 C,
  95 C, 03 C, 68 C, 95 C, 00 C,
  4C C, NEXT , C;
DCX
                                ==)
```

```
Screen: 21
( Dillo:  LOOK                    )

FORTH DEFINITIONS

: LOOK  DILLO      ( x y -- pen )
  91 @ 90 C@ 2SWAP XF/L
  (LOOK) SWAP 8 PX/BYT / )R R *
  8 - R) + SHIFT
  (ROT 90 C! 91 ! ;
```

```
Screen: 22
```

```
Screen: 23
```

```
Screen: 24
```

Screen: 25

Screen: 26

Screen: 27

Screen: 28

Screen: 29

Screen: 30

```
( Qtrig:  16K/                    )
HEX
CODE 16K/
   36 C, 03 C,
   36 C, 00 C, 36 C, 01 C,
   36 C, 03 C, 36 C, 00 C,
   36 C, 01 C, B5 C, 00 C, 95 C,
   02 C, B5 C, 01 C, 95 C,
   03 C, 4C C, POP , C;

DCX
```

==>

```
Screen: 31
( Qtrig:  [QS/C]                    )
: (QS/C)              ( rad -- n )
  DUP 670 >
  IF DUP 966 >
   IF 16384 10000 */
    DUP DUP 4 / M* 16K/
    11 ( TR4 ) OVER M* 16K/
    -208 ( TR3 ) + OVER M* 16K/
    2184 ( TR2 ) + OVER M* 16K/
    -10923 ( TR1 ) + M* 16K/
    OVER M* 16K/ +
    10000 M* 16K/ 1+
   ELSE 1-
   ENDIF
  ENDIF ;                           -->
```

```
Screen: 32
( Qtrig:  QSIN  QCOS                )
: QSIN                ( rad -- n )
  DUP 0< >R ABS DUP 15708 >
  IF 31416 SWAP - ENDIF
  (QS/C) R>
  IF MINUS ENDIF ;

: QCOS                ( rad -- n )
  15708 SWAP ABS DUP 15708 >
  DUP >R
  IF SWAP
  ENDIF - (QS/C) R>
  IF MINUS ENDIF ;
                                    ==>
```

```
Screen: 33
( Qtrig:  ->QRD  ->QDG              )
: ->QRD     ( scaled degrees -- )
                ( scaled radians )
  31416 18000 */ ;

: ->QDG     ( scaled radians -- )
                ( scaled degrees )
  18000 31416 */ ;

'( (QATN) ;S )( )

                                    -->
```

```
Screen: 34
( Qtrig:  [QATN]                    )
: (QATN)              ( n -- rad )
  0 >R
  DUP 2679 >
  IF DUP 17321 10000 */ 10000 -
    SWAP 17321 + 10000 SWAP */
    R> 1+ >R
  ENDIF
  DUP DUP 10000 */ DUP DUP
  -509 10000 */ -4708 +
  SWAP 2/ 7603 +
  */ 2/ OVER
  10000 */ +
  R>
  IF 5236 + ENDIF ;                 ==>
```

```
Screen: 35
( Qtrig:  QATN                      )
: QATN                  ( n -- n )
  DUP 0< >R ABS (QATN) R>
  IF MINUS ENDIF ;

                                    -->
```

```
Screen: 36
( Qtrig:  QATN2                     )
: QATN2               ( x y -- rad )
  SWAP 2DUP 0< >R 0< >R
  ABS SWAP ABS SWAP
  2DUP > DUP >R
  IF SWAP ENDIF
  10000 SWAP */ (QATN) R>
  IF 15708 SWAP -
  ENDIF I'
  IF 31416 I 0=
   IF SWAP ENDIF -
  ELSE I
   IF MINUS ENDIF
  ENDIF R> DROP R> DROP ;
```

Screen: 37

Screen: 38

Screen: 39

Screen: 40

```
( Dillo:  DILLO [ARMADILLO]      )

FORTH DEFINITIONS

'( QUAN )( 57 KLOAD )

VOCABULARY DILLO IMMEDIATE
   ( ARMADILLO )
                                 ==>
```

Screen: 41

```
( Dillo:  quans                  )

QUAN DAZM
QUAN DX1  QUAN DY1

DILLO DEFINITIONS

QUAN WNDLFT QUAN WNDRGT
QUAN WNDTOP QUAN WNDBOT
QUAN WNDL QUAN WNDR
QUAN WNDT QUAN WNDB
QUAN WNDW QUAN WNDE
QUAN WNDN QUAN WNDS
QUAN DX2  QUAN DY2
QUAN ?DOWN  QUAN DFLG
QUAN ?MCOOR QUAN ?XFM            -->
```

Screen: 42

```
( Dillo :  quans etc.            )

QUAN X1D
QUAN Y1D
QUAN X2D
QUAN Y2D
QUAN IX1
QUAN IY1
QUAN IX2
QUAN IY2
VECT XF/L  VECT <XF/L
VECT CASP
LABEL DRWRK 16 ALLOT
LABEL DRWSTT 16 ALLOT
DRWSTT 16 ERASE
                                 ==>
```

```
Screen: 43
( Dillo:  quans etc.            )

QUAN DRCLR
QUAN PHCLR
QUAN DRUCLR
QUAN PHUCLR
QUAN DRDAT
QUAN PHDAT
QUAN DRUDAT
QUAN PHUDAT
QUAN BYT/LN
QUAN PX/BYT
LABEL MTBL1 8 ALLOT
LABEL MTBL2 8 ALLOT

                                -->
```

```
Screen: 44
( Dillo:  quans etc.            )

QUAN ?DRSTP 0 TO ?DRSTP
QUAN COSFAC  1000 TO COSFAC
QUAN HSCL    1000 TO HSCL
QUAN VSCL    1000 TO VSCL
QUAN HABS QUAN VABS
QUAN HUSR    1000 TO HUSR
QUAN VUSR    1000 TO VUSR

                                ==>
```

```
Screen: 45
( Dillo:  SHIFT                 )

FORTH DEFINITIONS

HEX
CODE SHIFT
  B4 C, 00 C, 10 C, 0A C, 56 C,
  03 C, 76 C, 02 C, C8 C, D0 C,
  F9 C, 4C C, HERE C + ,  F0 C,
  08 C, 16 C, 02 C, 36 C, 03 C,
  88 C, 4C C, HERE 8 - ,
  4C C, POP , C;

                                -->
```

```
Screen: 46
( Dillo:  SWAP-  ]L             )

CODE SWAP-
  38 C, B5 C, 00 C, F5 C, 02 C,
  48 C, B5 C, 01 C, F5 C, 03 C,
  4C C, BINARY , C;

DCX

: ]L                      ( n -- )
  [COMPILE] ]
  [COMPILE] LITERAL ; IMMEDIATE


     ( DRAWLN : )
       1 LOAD                   ==>
```

```
Screen: 47
( Dillo:  phil/draw options     )

: RPHIL DILLO             ( f -- )
  MINUS [ DRWSTT 1+ ]L C! ;

: LPHIL DILLO             ( f -- )
  MINUS [ DRWSTT 2+ ]L C! ;

: PH+DR DILLO             ( f -- )
  NOT MINUS [ DRWSTT 3 + ]L C! ;

: PHUNT DILLO             ( f -- )
  NOT MINUS [ DRWSTT 4 + ]L C! ;

: PHXOR DILLO             ( f -- )
  MINUS [ DRWSTT 5 + ]L C! ; -->
```

```
Screen: 48
( Dillo:  phil/draw options     )

HEX

: PHUNOT DILLO            ( f -- )
  NOT MINUS [ DRWSTT 6 + ]L C! ;

: PHCRNR DILLO            ( f -- )
  MINUS [ DRWSTT 7 + ]L C! ;

: DRXOR DILLO             ( f -- )
  MINUS [ DRWSTT 8 + ]L C! ;

: DRUNT DILLO             ( f -- )
  MINUS [ DRWSTT 9 + ]L C! ;
                                ==>
```

```
Screen:  49
( Dillo:  DR1ST  DRUNOT          )
: DR1ST DILLO              ( f -- )
  NOT MINUS [ DRWSTT A + ]L C! ;
: DRUNOT DILLO             ( f -- )
  NOT MINUS [ DRWSTT B + ]L C! ;
DCX
                                 -->
```

```
Screen:  50
( Dillo:  DTBL                   )
DILLO DEFINITIONS
'( TABLE ==) )(  )
HEX
: DTBL
  CREATE SMUDGE
  ;CODE 16 C, 00 C, 36 C, 01 C,
  CA C, CA C, 18 C, A5 C,
   W C, 69 C, 02 C, 95 C, 00 C,
  98 C, 65 C,     W 1+ C, 95 C,
  01 C, 4C C, ' + , C;
DCX
                                 ==>
```

```
Screen:  51
( Dillo:  [ADJC]                 )
: (ADJC)            ( adr val -- )
  8 PX/BYT 8 MIN 0 MAX / SWAP
  8 3 PICK - SHIFT 255 AND DUP
  8 4 PICK
  DO
   3 PICK MINUS SHIFT
   SWAP O+S
   3 PICK
  +LOOP
  DROP SWAP DROP SWAP C! ;
                                 -->
```

```
Screen:  52
( Dillo:  ADJCLR  MSKTBL          )
: ADJCLR                    ( -- )
  AT DRDAT  DRCLR  (ADJC)
  AT PHDAT  PHCLR  (ADJC)
  AT DRUDAT DRUCLR (ADJC)
  AT PHUDAT PHUCLR (ADJC) ;
HEX
LABEL MSKTBL
  7F C, BF C, DF C, EF C,
  F7 C, FB C, FD C, FE C,
  3F C, CF C, F3 C, FC C,
  0F C, F0 C,
DCX
                                  ==>
```

```
Screen:  53
( Dillo:  reserved                )
                                  -->
```

```
Screen:  54
( Dillo:  reserved                )
                                  ==>
```

```
Screen:  55
( Dillo:  reserved                    )















                                   -->
```

```
Screen:  56
( Dillo:  reserved                    )















                                   ==>
```

```
Screen:  57
( Dillo:  reserved                    )















                                   -->
```

```
Screen:  58
( Dillo:  reserved                    )















                                   ==>
```

```
Screen:  59
( Dillo:  reserved                    )















                                   -->
```

```
Screen:  60
( Dillo:  POSIT  SCRWID               )

FORTH DEFINITIONS

: POSIT            ( xabs yabs -- )
  90 C! 91 ! ;


'( SCRWID' )(
: SCRWID
  559 C@ 252 AND OR 559 C! ;      )
                      ( width -- )(



                                   ==>
```

```
Screen: 61
( Turtle:  TRGTBL                    )
DILLO DEFINITIONS
'( TABLE TABLE TRGTBL                )
( DTBL TRGTBL )                   0 ,
   571 ,  1143 ,  1714 ,  2285 ,
  2855 ,  3425 ,  3993 ,  4560 ,
  5125 ,  5689 ,  6252 ,  6812 ,
  7370 ,  7927 ,  8480 ,  9031 ,
  9580 , 10125 , 10667 , 11206 ,
 11742 , 12274 , 12803 , 13327 ,
 13847 , 14364 , 14875 , 15383 ,
 15885 , 16383 , 16876 , 17363 ,
 17846 , 18323 , 18794 , 19259 ,
 19719 , 20173 , 20620 , 21062 ,
 21497 , 21925 , 22347 , 22761 ,
                                  --)
```

```
Screen: 62
( Turtle:  TRGTBL                    )

 23169 , 23570 , 23964 , 24350 ,
 24729 , 25100 , 25464 , 25820 ,
 26168 , 26509 , 26841 , 27165 ,
 27480 , 27787 , 28086 , 28377 ,
 28658 , 28931 , 29195 , 29450 ,
 29696 , 29934 , 30162 , 30381 ,
 30590 , 30790 , 30981 , 31163 ,
 31335 , 31497 , 31650 , 31793 ,
 31927 , 32050 , 32164 , 32269 ,
 32363 , 32448 , 32522 , 32587 ,
 32642 , 32687 , 32722 , 32747 ,
 32762 , 32767 ,
                                  ==)
```

```
Screen: 63
( Dillo:  AZMADJ                     )

: AZMADJ                      ( -- )
  DAZM ABS 360 )=
  IF DAZM 360 MOD TO DAZM ENDIF
  DAZM 0(
  IF 360 AT DAZM +! ENDIF ;
                                  --)
```

```
Screen: 64
( Dillo:  +-SIN,COS  #S/C           )

: +-SIN                  ( |n| -- n )
  DAZM 180 )
  IF MINUS ENDIF ;

: +-COS                  ( |n| -- n )
  DAZM 90 ) DAZM 270 ( AND
  IF MINUS ENDIF ;

: #S/C                   ( -- index )
  DAZM DUP 180 )
  IF 180 - ENDIF
  DUP 90 )
  IF 180 SWAP- ENDIF ;
                                  ==)
```

```
Screen: 65
( Dillo:  COSASP  *SIN  *COS         )

: COSASP        ( delY -- fixed )
  COSFAC 1000 */ ;

: *SIN              ( n -- n*"sin" )
  #S/C TRGTBL @ +-SIN
  32767 */ ;

: *COS              ( n -- n*"cos" )
  #S/C 90 SWAP- TRGTBL @ +-COS
  32767 */ CASP ;
                                  --)
```

```
Screen: 66
( Dillo:  ASPECT                     )

FORTH DEFINITIONS

: ASPECT DILLO          ( ON/OFF -- )
  IF   [ ' COSASP CFA ]L
  ELSE [ ' NOOP   CFA ]L
  ENDIF
  TO CASP ;

OFF ASPECT
                                  ==)
```

Screen: 67

```
( Dillo: [XX] [YY] DXF DXL )
DILLO DEFINITIONS

: (XX)              ( XnD -- xabs )
  HSCL + WNDRGT WNDLFT -
  HSCL 2* NOOP */ WNDLFT + ;

: (YY)              ( YnD -- yabs )
  VSCL SWAP- WNDBOT WNDTOP -
  VSCL 2* NOOP */ WNDTOP + ;

: DXF    ( XnD YnD -- xabs yabs )
  (YY) SWAP (XX) SWAP ;

: DXL    ( XnD YnD -- xabs yabs )
  )R HABS + R) VABS SWAP- ;  -->
```

Screen: 68

```
( Dillo: ([XX] ([YY] (DXF (DXL )

: ((XX)             ( xabs -- YnD )
  WNDLFT - HSCL 2* 1+
  WNDRGT WNDLFT - */ HSCL - ;

: ((YY)             ( yabs -- XnD )
  WNDTOP - VSCL 2* 1+
  WNDTOP WNDBOT - */ VSCL + ;

: (DXF     ( xabs yabs -- X# Y# )
  ((YY) SWAP ((XX) SWAP ;

: (DXL     ( xabs yabs -- X# Y# )
  )R HABS - R) VABS SWAP- ;
                              ==)
```

Screen: 69

```
( Dillo:  XFORM                )

FORTH DEFINITIONS

: XFORM DILLO              ( f -- )
  IF   [ '  DXF CFA ]L
       [ '  (DXF CFA ]L 1
  ELSE [ '  DXL CFA ]L
       [ '  (DXL CFA ]L 0
  ENDIF
  TO ?XFM TO (XF/L TO XF/L ;

OFF XFORM


                              -->
```

Screen: 70

```
( Dillo:  ?HZONE ?VZONE DLINE  )

DILLO DEFINITIONS

: ?HZONE              ( coor -- f )
  DUP WNDW (
  IF DROP -1 ELSE WNDE ) ENDIF ;

: ?VZONE              ( coor -- f )
  DUP WNDS (
  IF DROP -1 ELSE WNDN ) ENDIF ;

: DLINE                      ( -- )
  X1D Y1D XF/L POSIT
  X2D Y2D XF/L DRAWLN ;
                               ==)
```

Screen: 71

```
( Dillo:  HWSTP  VWSTP  ICLIP  )

: HWSTP  0)            ( f -- n n )
  IF WNDE ELSE WNDW ENDIF DUP ;

: VWSTP  0)            ( f -- n n )
  IF WNDN ELSE WNDS ENDIF DUP ;

: ICLIP                      ( -- )
  DX1 TO X1D  DY1 TO Y1D
  DX2 TO X2D  DY2 TO Y2D
  DX1 ?HZONE TO IX1
  DX2 ?HZONE TO IX2
  DY1 ?VZONE TO IY1
  DY2 ?VZONE TO IY2  ;
                               -->
```

Screen: 72

```
( Dillo:  CLIP                 )

: CLIP                       ( -- )
  1 ICLIP IX1 IY1 IX2 IY2
  OR OR OR DUP NOT TO DFLG
  IX1 IX2 () IX1 0= OR
  IY1 IY2 () IY1 0= OR AND AND
  IF IX1
   IF DY1 DY2 - IX1 HWSTP TO X1D
    DX2 - DX1 DX2 - */
    DY2 + DUP ?VZONE
    IF 2DROP 0
    ELSE TO Y1D 0 TO IY1
     1 TO DFLG
    ENDIF
   ENDIF                       ==)
```

Screen: 73
```
( Dillo:  CLIP                  )

    IY1
    IF DX1 DX2 - IY1 VWSTP TO Y1D
     DY2 - DY1 DY2 - */
     DX2 + DUP ?HZONE
     IF 2DROP 0
     ELSE TO X1D 1 TO DFLG
     ENDIF
    ENDIF DFLG OR

    IX2 OVER AND
    IF DY2 DY1 - IX2 HWSTP TO X2D
     DX1 - DX2 DX1 - */
     DY1 + DUP ?VZONE
                                 -->
```

Screen: 74
```
( Dillo:  CLIP                  )
    IF DROP
    ELSE TO Y2D 0 TO IY2
     1 TO DFLG
    ENDIF
   ENDIF
   IY2 AND
   IF DX2 DX1 - IY2 VWSTP TO Y2D
    DY1 - DY2 DY1 - */
    DX1 + DUP ?HZONE
    IF DROP
    ELSE TO X2D 1 TO DFLG
    ENDIF
   ENDIF
  ELSE DROP
  ENDIF ;                        ==>
```

Screen: 75
```
( Dillo:  DLN/UPD MOVE TURN[TO])
: DLN/UPD                  ( -- )
  DFLG  IF DLINE ENDIF
  DY2 TO DY1 DX2 TO DX1 ;

: MOVE                   ( n -- )
  DUP *SIN DX1 + TO DX2
  *COS DY1 + TO DY2 0 TO DFLG
  ?DOWN IF CLIP ENDIF DLN/UPD ;

FORTH DEFINITIONS
: TURN DILLO          ( azim -- )
  AT DAZM +! AZMADJ ;

: TURNTO DILLO        ( azim -- )
  TO DAZM AZMADJ ;             -->
```

Screen: 76
```
( Dillo:  GO[TO] DUPGO DRAW    )

: GO DILLO               ( n -- )
  ?DOWN SWAP 0 TO ?DOWN MOVE
  TO ?DOWN ;

: GOTO DILLO           ( x y -- )
  TO DY1 TO DX1 ;

: DUPGO                ( n -- n )
  DUP GO ;

: DRAW DILLO             ( n -- )
  ?DOWN SWAP 1 TO ?DOWN MOVE
  TO ?DOWN ;
                                ==>
```

Screen: 77
```
( Dillo:  DUPDRAW DRAWTO PHPEN )

: DUPDRAW              ( n -- n )
  DUP DRAW ;

: DRAWTO DILLO         ( x y -- )
  TO DY2 TO DX2 CLIP DLN/UPD ;

: PHPEN DILLO            ( b -- )
  TO PHCLR ADJCLR ;






                                -->
```

Screen: 78
```
( Dillo:  GO[TO].  DUPGO.  DOT )

: GOTO. DILLO          ( x y -- )
  2DUP 2DUP GOTO
  TO DY2 TO DX2 CLIP DFLG
  IF [ DRWSTT 9 + ]L >R R @
     0 R ! >R DRAWTO R> R> !
  ELSE 2DROP
  ENDIF ;

: GO. DILLO              ( n -- )
  GO DX1 DY1 GOTO. ;

: DUPGO.                 ( n -- )
  DUP GO. ;
                                ==>
```

```
Screen:  79
( Dillo:  DOT PEN DRBAK PHBAK  )

: DOT                          ( -- )
  0 GO. ;

: PEN DILLO                  ( b -- )
  DUP TO DRCLR PHPEN ;

: DRBAK  DILLO          ( color -- )
  TO DRUCLR ADJCLR ;

: PHBAK  DILLO          ( color -- )
  DUP TO PHUCLR DRBAK ;



                                 --)
```

```
Screen:  80
( Dillo:  CENTER[0]  PHIL[TO]  )

: CENTER                       ( -- )
  0 0 GOTO ;

: CENTER0                      ( -- )
  CENTER 0 TO DAZM ;

: PHIL DILLO                 ( n -- )
  DRWSTT C@ -1 DRWSTT C!
  SWAP DRAW DRWSTT C! ;

: PHILTO DILLO            ( x y -- )
  DRWSTT C@ -1 DRWSTT C!
  (ROT DRAWTO DRWSTT C! ;
                                 ==)
```

```
Screen:  81
( Dillo:  WABS  WNDREL  WNUM   )

DILLO DEFINITIONS

: WABS     ( l r t b -- L R T B )
  DILLO ROT SWAP XF/L 2SWAP
  XF/L 2SWAP ROT SWAP ;

: WNDREL  ( l r t b -- L R T B )
  DILLO ROT (XF/L 2SWAP
  (XF/L 2SWAP ROT SWAP ;

: WNUM              ( l r t b -- )
  DILLO TO WNDS TO WNDN
  TO WNDE TO WNDW ;
                                 --)
```

```
Screen:  82
( Dillo:  WEDG  WRL,BT-  BASNUM)

: WEDG              ( l r t b -- )
  DILLO TO WNDBOT TO WNDTOP
  TO WNDRGT TO WNDLFT ;

: WRL-                       ( -- n )
  WNDR WNDL - 2/ ;

: WBT-                       ( -- n )
  WNDB WNDT - 2/ ;

: BASNUM                       ( -- )
  WRL- DUP MINUS SWAP
  WBT- DUP MINUS WNUM ;
                                 ==)
```

```
Screen:  83
( Dillo:  SCLSTP  RELOC         )

: SCLSTP                       ( -- )
  WRL- TO HSCL WBT- TO VSCL ;

: RELOC                        ( -- )
  91 @ 90 C@ (XF/L GOTO ;








                                 --)
```

```
Screen:  84
( Dillo:  M,SCOOR  SET-SCALE   )

: MCOOR DILLO                  ( -- )
  SCLSTP 1 TO ?MCOOR
  OFF XFORM ;           MCOOR

: SCOOR DILLO                  ( -- )
  HUSR TO HSCL VUSR TO VSCL
  HUSR DUP MINUS SWAP
  VUSR DUP MINUS WNUM
  0 TO ?MCOOR ON XFORM ;

: SET-SCALE     ( hscl vscl -- )
  DILLO TO VUSR TO HUSR ;

                                 ==)
```

Screen: 85

```
( Dillo:COOR DSCALE WNDASP WCTR)
: COOR  DILLO                  ( -- )
  ?MCOOR
  IF MCOOR ELSE SCOOR ENDIF ;
: DSCALE                       ( -- )
  WRL- WBT- SET-SCALE ;
: WNDASP                       ( -- )
  CASP SWAP CASP SWAP ;
FORTH DEFINITIONS
: WCTR DILLO                   ( -- )
  WNDE WNDW - 2/ WNDW +  WNDN
  WNDS - 2/ WNDS + GOTO ;     --)
```

Screen: 86

```
( Dillo:  WCTR0  BASWND  WINDOW)
: WCTR0                        ( -- )
  WCTR 0 TURNTO ;
: BASWND DILLO                 ( -- )
  WNDL TO WNDLFT WNDR TO WNDRGT
  WNDT TO WNDTOP WNDB TO WNDBOT
  BASNUM CENTER0 ;
: WINDOW  ( lft rgt top bot -- )
  DILLO WNDASP MCOOR 2OVER 2OVER
  BASWND WABS WEDG WNUM WCTR0 ;
: RELWND  ( lft rgt top bot -- )
  DILLO WNDASP SCOOR WABS
  WEDG CENTER0 ;              ==)
```

Screen: 87

```
( Dillo:  RELWND  FRAME        )
: FRAME DILLO                  ( -- )
  [ DRWSTT 8 + ]L
  DUP @ OVER 0 SWAP ! SWAP
  WNDLFT WNDTOP POSIT
  WNDRGT WNDTOP DRAWLN
  WNDRGT WNDBOT DRAWLN
  WNDLFT WNDBOT DRAWLN
  WNDLFT WNDTOP DRAWLN ! ;
                               --)
```

Screen: 88

```
( Dillo:  DINIT  WIPE          )
: DINIT DILLO                  ( -- )
  DRWSTT 16 ERASE
  ON RPHIL
  MCOOR BASWND 0 PHBAK 1 PEN ;
  DINIT
: WIPE DILLO      ( org dest -- )
  [ DRWSTT 9 + ]L )R R @ 0 R !
  )R PHCLR DRCLR PHUCLR PEN
  WNDBOT 1+ WNDTOP
  DO WNDLFT I POSIT
     WNDRGT I DRAWLN
  LOOP PEN PHPEN R) R) ! ;
                               ==)
```

Screen: 89

```
( Dillo:  ASPSTP               )
DILLO DEFINITIONS
: ASPSTP             ( tbladr -- )
  DUP C@ SWAP 1+ C@ * 32 SWAP
  / 833 * TO COSFAC ;
                               --)
```

Screen: 90

```
( Dillo:  DEFBAS  EDGES        )
: DEFBAS             ( L R T B -- )
  TO WNDB TO WNDT
  TO WNDR TO WNDL
  WRL- TO HABS WBT- TO VABS
  SCLSTP ;
: EDGES      ( scn/pxl pxl/ln -- )
  1- ( WNDR ) )R
  703 C@ 4 = ( ?split )
  IF 160 ELSE 192 ENDIF
  SWAP / 1-  ( WNDB )
  0 R) 0 4 ROLL DEFBAS
  BASWND ;
                               ==)
```

Screen: 91

```
( Dillo:  PXLTBL                   )
LABEL PXLTBL
( scn/pxl  pxl/ln/10  pxl/byt   )
  8 C,       4 C,       1 C,   (  0 )
  8 C,       2 C,       1 C,   (  1 )
 16 C,       2 C,       1 C,   (  2 )
  8 C,       4 C,       4 C,   (  3 )
  4 C,       8 C,       8 C,   (  4 )
  4 C,       8 C,       4 C,   (  5 )
  2 C,      16 C,       8 C,   (  6 )
  2 C,      16 C,       4 C,   (  7 )
  1 C,      32 C,       8 C,   (  8 )
  1 C,       8 C,       2 C,   (  9 )
  1 C,       8 C,       2 C,   ( 10 )
  1 C,       8 C,       2 C,   ( 11 )
  1 C,      16 C,       4 C,      -->
```

Screen: 92

```
( Dillo:  7PLUS                    )
CODE 7PLUS HEX              ( -- )
  A9 C, 07 C, 85 C, 57 C, AD C,
  30 C, 02 C, 85 C, N  C, AD C,
  31 C, 02 C, 85 C, N 1+ C, HERE
  B1 C, N  C, 29 C, FC C,
  C9 C, 40 C, F0 C,
  14 C, B1 C, N  C,
  85 C, N 2+ C, 29 C,
  0F C, C9 C, 0F C, D0 C, 06 C,
  C6 C, N 2+ C, A5 C, N 2+ C,
  91 C, N  C, C8 C, 4C C, ,
  4C C, NEXT ,  C; DCX
                                 ==>
```

Screen: 93

```
( Dillo:  DIMSTP                   )
: DIMSTP      ( stripped-GR-# -- )
              ( scn/pxl pxl/ln    )
  3 * PXLTBL + DUP ASPSTP
  DUP C@ ( scn/pxl ) SWAP
  DUP 2+ C@ TO PX/BYT
  1+ C@ 10 * ( pxl/ln, normal )
  559 ( DMACTL ) C@
  3 AND DUP 2 <>
  IF 1 = IF 4 ELSE 6 ENDIF 5 */
  ELSE DROP
  ENDIF  ( pxl/ln, actual )
  DUP PX/BYT / TO BYT/LN
                                 -->
```

Screen: 94

```
( Dillo:  DIMSTP  UGR.             )

  PX/BYT DUP 2 =
  IF DROP 12
  ELSE 4 =
   IF 8 ELSE 0 ENDIF
  ENDIF
  MSKTBL + MTBL1 8 CMOVE
  8 0
  DO
    MTBL1 I + C@
    255 XOR MTBL2 I + C!
  LOOP ;

: UGR.       ( stripped-GR-# -- )
  DIMSTP 2DROP MCOOR ;           ==>
```

Screen: 95

```
( Dillo:  GR.                      )

FORTH DEFINITIONS

: GR. DILLO               ( b -- )
  ( set up display, 7+ ? )
  DUP 15 AND SWAP OVER
  12 =
  IF 2- 2- GR. 7PLUS
  ELSE GR.
  ENDIF
  ( set up drawln & window data)
  DIMSTP EDGES
  ( initialization )
  0 PHBAK 1 PEN MCOOR DSCALE
   OFF ASPECT ;  FORTH
```

Screen: 96

Screen: 97

Screen: 98

Screen: 99

Screen: 100

```
( turntwd:  TURNTWD               )

'( QATN2 )( 17 KLOAD )

: TURNTWD DILLO          ( x y -- )
  DY1 - SWAP DX1 - SWAP
  QATN2 DUP ABS 87 + SWAP +-
  ( ROUNDING )
  180 31416 */
  ( DEGREES )
  90 SWAP- TURNTO ;
```

Screen: 101

Screen: 102

```
( P-naming:  NAMEPT  THISPT       )

: NAMEPT            ( xxx, x y -- )
  <BUILDS , ,       ( xxx: -- x y )
  DOES> DUP 2+ @ SWAP @ ;

: THISPT DILLO    ( xxx, --       )
                  ( xxx: -- x y )
  DX1 DY1 NAMEPT ;

     ( or write 2QUAN )
```

Screen: 103

Screen: 104

```
( lines:  2PT-LN                    )
DILLO DEFINITIONS QUAN LNA
QUAN LNB FORTH DEFINITIONS
: 2PT-LN ( x1 y1 x2 y2 --a b c )
  DILLO 2DUP >R >R
  ROT - TO LNA          (  dy = a )
  - TO LNB              ( -dx = b )
  LNA R> M*             ( -ax2-by2)
  LNB R> M* D+ DMINUS   ( = c )
  2DUP 32767 M/ ABS
  SWAP DROP -DUP
  IF 1+ >R R M/ SWAP DROP
     LNA R / TO LNA
     LNB R> / TO LNA
  ELSE DROP
  ENDIF LNA LNB ROT ;          ==>
```

Screen: 105

```
( lines:  2LNX                      )

'( D/ )( 60 KLOAD )

FORTH DEFINITIONS

: 2LNX   ( a1 b1 c1 a2 b2 c2 )
  DILLO                   ( -- x y )
  6 PICK 3 PICK M*       ( a1*b2 )
  5 PICK 8 PICK M*     ( - a2*b1 )
  DMINUS D+ >R >R
  4 PICK 4 PICK M*       ( c1*a2 )
  3 PICK 9 PICK M*     ( - c2*a1 )
  DMINUS D+
  I I' D/ DROP >R           ( y )
                              -->
```

Screen: 106

```
( lines:  2LNX  MAKLN               )

   ( a1 b1 c1 a2 b2 c2 -- x y )

  5 PICK 2 PICK M*     ( b1*c2 )
  4 PICK 7 PICK M*   ( - b2*c1 )
  DMINUS D+
  I' J D/ DROP >R           ( x )
  2DROP 2DROP 2DROP
  R> R> R> R> 2DROP ;

: MAKLN DILLO          ( -- a b c )
  DX1 DY1
  OVER 100 *SIN +
  OVER 100 *COS + 2PT-LN ;
```

Screen: 107

Screen: 108

```
( L-naming:  THISLN                 )

'( 2PT-LN )( 52 KLOAD )

: THISLN          ( xxx, --         )
  DILLO           ( xxx: -- a b c )
  <BUILDS
  MAKLN , , ,
  DOES>
  DUP 4 + @ OVER 2+ @ ROT @ ;
    ( or write 3QUAN )




                                ==>
```

Screen: 109

```
( L-naming:  NAMELN                )

: NAMELN ( xxx, x1 y1 x2 y2 -- )
  <BUILDS      ( xxx: -- a b c )
  2PT-LN , , ,
  DOES>
  DUP 4 + @ OVER 2+ @ ROT @ ;

    ( or write 3QUAN )
```

Screen: 110

```
( W-naming:  THISWND               )

: THISWND DILLO        ( xxx, -- )
  <BUILDS              ( xxx: -- )
  ?XFM ,
  WNDW , WNDE , WNDN , WNDS ,
  WNDLFT , WNDRGT ,
  WNDTOP , WNDBOT ,
  DOES> >R 18 0
  DO J I + @ 2 +LOOP R> DROP
  WEDG WNUM XFORM WCTR0 ;
```

Screen: 111

Screen: 112

```
( Dillo option load block         )

100 LOAD     ( TURNTWD      )

102 LOAD     ( POINT NAMING )

104 LOAD     ( LINE-INTERSECT.   )

108 LOAD     ( LINE NAMING   )

110 LOAD     ( WINDOW NAMING )
```

Screen: 113

Screen: 114

```
( Quan:  TO  AT                    )

: TO
  -FIND 0= 0 ?ERROR DROP
  STATE @
  IF ,
  ELSE EXECUTE
  ENDIF ; IMMEDIATE

: AT
  -FIND 0= 0 ?ERROR DROP
  4 + [COMPILE] LITERAL ;
  IMMEDIATE
```

==>

Screen: 115

```
( Quan:  [2@6]  [2!4]            )
ASSEMBLER HEX
LABEL (2@6)
   A0 C, 06 C, B1 C, W  C, 48 C,
   C8 C, B1 C, W  C, 4C C, PUSH ,
LABEL (2!4)
   A0 C, 04 C, B5 C, 00 C, 91 C,
   W  C, C8 C, B5 C, 01 C, 91 C,
   W  C, 4C C, POP ,
                                  -->
```

Screen: 116

```
( Quan:  [2!2]  [2V4]            )
LABEL (2!2)
   A0 C, 02 C,
   4C C, ' (2!4) 2 + ,
LABEL (2V4)
   A0 C, 05 C, B1 C, W  C, 48 C,
   88 C, B1 C, W  C, 85 C, W  C,
   68 C, 85 C, W 1+ C,
   A0 C, 00 C, 4C C, W 1- ,
                                  ==>
```

Screen: 117

```
( Quan:  patch for CREATE        )
DCX
: (PTCH)                ( system )
   SWAP )R R = 251 R = 249 R) =
   OR OR ;
: PTCH                  ( system )
   IF [ ' (PTCH) CFA ] LITERAL
   ELSE [ ' = CFA ] LITERAL
   ENDIF
   [ ' CREATE 63 + ] LITERAL ! ;
                                  -->
```

Screen: 118

```
( Quan:  QUAN  VECT
: QUAN
  ON PTCH LABEL -2 ALLOT
  (2@6) , (2!4) ,
  [ ' VARIABLE 4 + ] LITERAL ,
  2 ALLOT OFF PTCH ;
: VECT
  ON PTCH LABEL -2 ALLOT
  (2V4) , (2!2) ,
  [ ' NOOP CFA ] LITERAL ,
  OFF PTCH ;
```

Screen: 119

Screen: 120

```
( Dbls:  DU/MOD                  )
DILLO DEFINITIONS
HEX
CODE DU/MOD   ( d1 d2 -- dr dq )
   A9 C, 04 C, 20 C, SETUP ,
   CA C, 94 C, 00 C,
   CA C, 94 C, 00 C,
   CA C, 94 C, 00 C,
   CA C, 94 C, 00 C,
   A0 C, 04 C, CA C, B9 C, C5 C,
   00 C, 95 C, 00 C, 88 C, D0 C,
   F7 C, A0 C, 20 C, 16 C, 02 C,
   36 C, 03 C, 36 C, 00 C, 36 C,
   01 C, 36 C, 06 C, 36 C, 07 C,
   36 C, 04 C, 36 C, 05 C, 38 C,
                                  ==>
```

Screen: 121

```
( Dbls:  DU/MOD                       )

   B5 C, 06 C, E5 C, C4 C, 48 C,
   B5 C, 07 C, E5 C, C5 C,
   48 C, B5 C, 04 C, E5 C, C2 C,
   48 C, B5 C, 05 C, E5 C, C3 C,
   30 C, 10 C, 95 C, 05 C, 68 C,
   95 C, 04 C, 68 C, 95 C, 07 C,
   68 C, 95 C, 06 C, F6 C, 02 C,
   4C C, HERE 05 + , 68 C, 68 C,
   68 C, 88 C, D0 C, C4 C, 4C C,
   NEXT , C;

DCX

                                   --)
```

Screen: 122

```
( Dbls:  D/MOD D/                     )

: D/MOD
   DUP 4 PICK DUP >R XOR >R
   DABS 2SWAP DABS 2SWAP
   DU/MOD R> D+-
   2SWAP R> D+- 2SWAP ;

: D/
   D/MOD 2SWAP 2DROP ;
```

Screen: 123

Screen: 124

Screen: 125

Screen: 126

```
( Quan:  TO  AT                       )

: TO
   -FIND 0= 0 ?ERROR DROP
   STATE @
   IF ,
   ELSE EXECUTE
   ENDIF ; IMMEDIATE

: AT
   -FIND 0= 0 ?ERROR DROP
   4 + [COMPILE] LITERAL ;
   IMMEDIATE

                                   ==>
```

```
Screen: 127
( Quan:  QUAN                          )

ASSEMBLER HEX

LABEL (2@6)
   A0 C, 06 C, B1 C, W  C, 48 C,
   C8 C, B1 C, W  C, 4C C, PUSH ,

LABEL (2!4)
   A0 C, 04 C, B5 C, 00 C, 91 C,
   W  C, C8 C, B5 C, 01 C, 91 C,
   W  C, 4C C, POP ,



                                     -->
```

```
Screen: 128
( Quan:  [2!2]  [2V4]                  )

LABEL (2!2)
   A0 C, 02 C,
   4C C, ' (2!4) 2 + ,

LABEL (2V4)
   A0 C, 05 C, B1 C, W  C, 48 C,
   88 C, B1 C, W  C, 85 C, W  C,
   68 C, 85 C, W 1+ C,
   A0 C, 00 C, 4C C, W 1- ,



                                     ==>
```

```
Screen: 129
( Quan:  QUAN  VECT                    )

: QUAN
   LABEL -2 ALLOT
   (2@6) , (2!4) ,
   [ ' VARIABLE 4 + ] LITERAL ,
   2 ALLOT ;

: VECT
   LABEL -2 ALLOT
   (2V4) , (2!2) ,
   [ ' NOOP CFA ] LITERAL , ;


DCX

```

```
Screen: 130
( Demos:  BOX                          )

: BOX                          ( n -- )
   4 0
   DO DUPDRAW 90 TURN
   LOOP DROP ;









                                     -->
```

```
Screen: 131
( Demos:  QUBE                         )

: QUBE                         ( n -- )
   >R DX1 DY1 DAZM R>
   DUPDRAW   90 TURN
   DUPDRAW   45 TURN
   DUPDRAW   45 TURN
   DUPDRAW   90 TURN
   DUPDRAW   45 TURN
   DUPDRAW  135 TURN
   DUPDRAW  -90 TURN
   DUPDRAW  180 TURN
   DUPGO
   -45 TURN DRAW
   TURNTO GOTO ;
                                     -->
```

```
Screen: 132
( Demos:  TCIRCLE  QCIRCLE             )

: TCIRCLE                 ( chord -- )
   18 0
   DO DUPDRAW 20 TURN
   LOOP DROP ;

: QCIRCLE                ( radius -- )
   DAZM DX1 DY1 4 ROLL
   -10 TURN
   3473 10000 */ DUP
   10000 3473 */ GO
   100 TURN TCIRCLE
   GOTO TURNTO ;


```

Screen: 133

Screen: 134

Screen: 135

Screen: 136

Screen: 137

Screen: 138

Screen: 139

Screen: 140

Screen: 141

Screen: 142

Screen: 143

Screen: 144

Screen: 145

Screen: 146

Screen: 147

Screen: 148

Screen: 149

Screen: 150

```
( Fp ext:  useful constants     )

FP 3.14159265     FCONSTANT PI
FP 1.57079633     FCONSTANT PI/2
FP 1.04719756     FCONSTANT PI/3
FP  .785398163    FCONSTANT PI/4
FP  .523598776    FCONSTANT PI/6
FP  .0174532925   FCONSTANT RD/DG
FP 57.2957796     FCONSTANT DG/RD
FP  2.71828183    FCONSTANT EXP1
FP 1              FCONSTANT FP1
FP 0              FCONSTANT FP0
FP 1E+97          FCONSTANT FTOP
FP 1E-97          FCONSTANT FBOT


                                ==>
```

Screen: 151

```
( Fp ext: FMINUS 2FDUP F+! FMAX)
: FMINUS              ( fp -- fp )
  SP@ 4 + DUP C@
  128 XOR SWAP C! ;
: 2FDUP ( fp fp -- fp fp fp fp )
  FOVER FOVER ;
: F+!                 ( fp a -- )
  DUP >R F@ F+ R> F! ;
: FMAX            ( fp fp -- fp )
  2FDUP F<
  IF FSWAP ENDIF FDROP ;
                              -->
```

Screen: 152

```
( Fp ext: FMIN F0< FABS 2FDP F,)
: FMIN            ( fp fp -- fp )
  2FDUP F>
  IF FSWAP ENDIF FDROP ;
: F0<                ( fp -- f )
  FP0 F< ;
: FABS            ( fp -- fp )
  FDUP F0< IF FMINUS ENDIF ;
: 2FDROP          ( fp fp -- )
  FDROP FDROP ;
: F,                 ( fp -- )
  SWAP ROT , , , ;            ==>
```

Screen: 153

```
( Fp ext:  F>R  FR>  F.S      )
: F>R                ( fp -- )
  SWAP ROT R> <ROT
  >R >R SWAP >R >R  ;
: FR>                ( -- fp )
  R> R> SWAP R> R>
  ROT >R <ROT SWAP ;
: F.S                   ( -- )
  CR S0 @ SP@ - 2- 6 / -DUP
  IF -1 SWAP 1-
   DO SP@ I 6 * + F@ SWAP ROT F.
   -1 +LOOP
  ELSE ." No FP on stack... "
  ENDIF ;                     -->
```

Screen: 154

```
( Fp ext:  [F@]  FPICK  FROLL  )
CODE (F@)     ( system: -- [fp] )
  HEX  CA C, CA C, CA C, CA C,
  CA C, CA C, 4C C, NEXT , DCX
: FPICK ( fp..fp n -- fp..fp fp)
  1 - 6 * SP@ 2+ SWAP O+S
  6 - 6 CMOVE (F@) ;
: FROLL   ( fp..fp n -- fp..fp )
  0 MAX -DUP
  IF DUP 6 * >R FPICK SP@
  DUP 6 + R> <CMOVE FDROP
  ENDIF ;
                              ==>
```

Screen: 155

```
( Fp ext:  -FIX  -FLOAT        )
: -FIX              ( fp -- n )
  FP -32768 FMAX FP 32767 FMIN
  FDUP F0<
  IF FMINUS 1 ELSE 0 ENDIF
  >R FIX R>
  IF MINUS ENDIF ;
: -FLOAT            ( n -- fp )
  DUP 0< DUP >R
  IF MINUS ENDIF
  FLOAT R>
  IF FMINUS ENDIF ;
                              -->
```

Screen: 156

```
( Fp ext:  FLWCHK  [system]    )
: FLWCHK       ( system )
  DUP 173 >    ( fp fp n n n -- )
  IF DROP 128 ( fp fp 1 / fp 0 )
   AND SWAP 128 AND
   XOR >R 2FDROP
   FTOP R>
   IF FMINUS
   ENDIF 0
  ELSE 79 <
   IF 2DROP 2FDROP FP0 0
   ELSE 2DROP 1
   ENDIF
  ENDIF ;
                              ==>
```

Screen: 157

```
( Fp ext:  2EXP@&  [system]       )
: 2EXP@&                  ( system )
      ( fp fp -- fp fp n n n n )
  SP@ 10 + DUP C@ SWAP 6 - C@
  2DUP 127 AND SWAP
  127 AND SWAP ;
                                 -->
```

Screen: 158

```
( Fp ext:  F*OV  F/OV  1/FP       )
: F*OV              ( fp fp -- fp )
  2EXP@& + FLWCHK IF F* ENDIF ;
: F/OV              ( fp fp -- fp )
  FDUP F0=
  IF FDROP FBOT
  ENDIF
  2EXP@& 128 - MINUS + FLWCHK
  IF F/
  ENDIF ;
: 1/FP                 ( fp -- fp )
  FP1 FSWAP F/OV ;
                                 ==>
```

Screen: 159

```
( Fp ext:  ->RD  ->DG             )
: FR                     ( -- fp1 )
  4 RPICK J I' ;
: ->RD              ( fp1 -- fp2 )
  RD/DG F* ;
: ->DG              ( fp1 -- fp2 )
  DG/RD F* ;
```

Screen: 160

```
( Trig:  sin/cos & atn coefs      )
'( 1/FP )( 15 KLOAD )
LABEL SCCFS
FP  .260190304E-5 F,
FP -.198074187E-3 F,
FP  .833302514E-2 F,
FP -.166666567    F,
LABEL TCFS
FP -.509095825E-1 F,
FP -.470832514    F,
FP  .141250074E+1 F,
                                 ==>
```

Screen: 161

```
( Trig:  [S/C]                    )
: (S/C)              ( fp f -- fp )
  >R FDUP PI F/ FIX DUP 1 AND
  IF R> 1 XOR >R
  ENDIF
  FLOAT PI F* F-
  FDUP FDUP F* FDUP F>R
  SCCFS 4 FR> FS FPOLY >F F*
  FOVER F* F+
  R>
  IF FMINUS
  ENDIF ;
                                 -->
```

Screen: 162

```
( Trig:  SIN  COS  TAN            )
: SIN                  ( fp -- fp )
  FDUP F0< >R
  FABS R> (S/C) ;
: COS                  ( fp -- fp )
  FABS PI/2 F+
  0 (S/C) ;
: TAN                  ( fp -- fp )
  FDUP SIN FSWAP COS F/OV ;
                                 ==>
```

Screen: 163

```
( Trig:  [ATN]                          )

: (ATN)                 ( fp -- fp )
  FDUP FP1 F>
  IF 1/FP 2 ELSE 0 ENDIF >R
  FDUP FP .267949192 F>
  IF FDUP FP 1.73205081 F* FP1
     F- FSWAP FP 1.73205081
     F+ F/ R> 1+ >R
  ENDIF
  FDUP FDUP F* FDUP FDUP F>R
  TCFS 2 FR> FS FPOLY >F F*
  FSWAP TCFS 12 + F@ F+ F/
  FOVER F* F+
  I 1 >
                                  -->
```

Screen: 164

```
( Trig:  [ATN]                          )

  IF FMINUS
  ENDIF R> -DUP
  IF
   1- -DUP
   IF
    1-
    IF PI/3
    ELSE PI/2
    ENDIF
   ELSE PI/6
   ENDIF F+
  ENDIF ;

                                  ==>
```

Screen: 165

```
( Trig:  ATN  ATN2                      )

: ATN                   ( fp -- fp )
  FDUP F0< >R FABS (ATN) R>
  IF FMINUS
  ENDIF ;

: ATN2          ( fpx fpy -- fp )
  FSWAP 2FDUP FSWAP
  F0< >R F0< >R
  F/OV FABS (ATN) R>
  IF PI FSWAP F-
  ENDIF R>
  IF FMINUS
  ENDIF ;
```

Screen: 166

Screen: 167

Screen: 168

Screen: 169

Screen: 170

```
CONTENTS OF THIS DISK:

ARMADILLO GRAPHICS:          40 LOAD
 + TURN TOWARD:             100 LOAD
 + POINT-NAMING:            102 LOAD
 + LINE-INTERSECT FCNS:     104 LOAD
 + LINE-NAMING:             108 LOAD
 + WINDOW-NAMING:           110 LOAD
LOAD ALL ABOVE +'S:         112 LOAD
DEMOS:                      130 LOAD

FLOATING PT. EXTENSION:     150 LOAD
 ( REQUIRES VAL4TH FLT. PT. )
QTRIG (INTEGER) FNCTNS:      30 LOAD

QUAN STRUCTURES:            114 LOAD
```

Screen: 171

Screen: 172

Screen: 173

Screen: 174

Screen: 175

Screen: 176

```
( Error messages                               )
Stack empty
Dictionary full
Wrong addressing mode
Is not unique
Value error
Disk address error
Stack full
```

Screen: 177

```
Disk Error!
Dictionary too big
```

Screen: 178

```
( Error messages
Use only in Definitions
Execution only
Conditionals not paired
Definition not finished
In protected dictionary
Use only when loading
Off current screen
```

Screen: 179

```
Declare VOCABULARY
```

VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713
802-790-7141

valFORTH T.M.

SOFTWARE SYSTEM
for ATARI*

Text Compression and Auto Text Formatting

*Atari is a trademark of Atari, Inc., a division of Warner Communications.

valFORTH T.M.
SOFTWARE SYSTEM

Text Compression and Auto Text Formatting

Evan Rosen

VALPAR INTERNATIONAL

Disclaimer of Warranty
on Computer Programs

All Valpar International computer programs are distributed on an "as is" basis without warranty of any kind. The total risk as to the quality and performance of such programs is with the purchaser. Should the programs prove defective following their purchase, the purchaser and not the manufacturer, distributor, or retailer assumes the entire cost of all necessary servicing or repair.

Valpar International shall have no liability or responsibility to a purchaser, customer, or any other person or entity with respect to any liability, loss, or damage caused directly or indirectly by computer programs sold by Valpar International. This disclaimer includes but is not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer programs.

Defective media (diskettes) will be replaced if diskette(s) is returned to Valpar International within 30 days of date of sale to user.

Defective media (diskettes) which is returned after the 30 day sale date will be replaced upon the receipt by Valpar of a $12.00 Replacement Fee.

TEXT COMPRESSION AND AUTO TEXT FORMATTING

Table of Contents

LI OVERVIEW AND STROLLING THROUGH TCAF

LII TCAF - GLOSSARY

LIII TCAF - SUPPLIED SOURCE LISTING

NOTICE
TEXT COMPRESSION AND AUTOMATIC TEXT FORMATTING
CODE TRANSPORTATION

The routines in this package have been coded and presented so that they may be readily transported to other fig-FORTH systems on machines other than the Atari 400/800. This is in response to numerous requests to this effect from various "adventure" game authors. We note, however, that the same restrictions apply to the software in this package, whether run on the Atari 400/800 machines or any other:

First, the code may ONLY be used in either an AUTO'd system as described in valFORTH 1.1 documentation, or in a target-compiled system.

Second, any software written with these routines, on any machine, must contain the acknowledgement of Valpar International as the source of the code, as described and detailed in valFORTH 1.1 documentation.

Other distribution may be construed to be a violation of applicable copyright laws.

Overview

This package attempts to fill at least two common needs of the programmer who does verbal/interactive programming.

First, a group of automatic text formatting routines is provided that allow two different approaches:

*A non-wrap line formatter to both the video display and the printer, including variable-margin capability and inverse video option, and

*A versatile window formatting system, with scrolling, color and inverse video options as appropriate, and window naming. Notes on the creation of window types with different "generic" parameters are also included.

In both modes described above, user options of left-, right-, center-, or fill-justification are supported, as is numerical output formatting.

Second, two different approaches to the problem loosely termed "text compression" are implemented:

The first is intended for use in programs where run-time retrieval of text stored on disk is allowable, and provides a set of general virtual-memory operators for the creation and retrieval of messages from disk. A simple encryption scheme is provided as a (working) example for the software developer who wishes his or her messages to be not easily readable from disk with, for instance, a Forth screen editor. In addition, alternate points in the virtual memory routines are indicated where deeper encryption routines might be employed. Routines are provided for virtual memory message programming on both one- and two-drive development systems.

The second set of text compression routines is intended for use in "in memory" applications, such as cassette-booted programs, that do not have access to disk for message retrieval. In this case the most compact code practical is desired, and a system built around some of the basic aspects of Forth's co[illegible] threaded-code structure is provided.

Finally, we note that the autoformatting and text compression utilities are designed to be used in most any of their possible different combinations.

STROLLING THROUGH TCAF
(Text Compression and Autoformatting)

The organization of this disk is slightly different from the others in this series. While a table of contents may still be found on screen 170 as usual, in this package the "load chain," starting on screen 166, will get far more use. In general when one wishes to load a TCAF development system with a specific set of capabilities, one makes slight adjustments to the load chain option screen and then simply loads the first screen in the chain. The chain does the rest.

To start off, first prepare two blank, formatted disks. Make your normal working copy of TCAF on the first disk and leave it un-write-protected. The second disk will be used a little later.

Autoformatting

In order to select options you will want to make changes to the load chain option screen. This may be found by locating the load chain in the directory on screen 170, and then scanning through the screens in the chain until you find the one marked "options" in its first line. (This is on or near screen 167.) Look at this screen, and see that most of the lines have a left parenthesis in the left column, followed by a LOAD command and a comment. By removing selected left-column left parentheses you can activate various options. Right now on your working copy use an editor to remove all of the left-column left parentheses except for the one on the line that says "text compression." (Text compression uses transient structures and will be discussed separately.) And, of course, don't remove the one in the comment at the very top of the screen. OK, now boot a bare valFORTH 1.1 system, and load in the debugger, and swap in the TCAF disk, do MTB as usual, and load the first screen in the load chain on this disk. (Probably 166.)

When the prompt comes back, type

    ON STACK

since you'll want to watch the stack. Then type

    TYPEOUT

(Failure to execute this initialization word may cause a crash as you try to use words like *TYPE later on.) This command activates one of the two formatting modes. This mode, called "type-out mode," since it uses the word TYPE as its actual output word, can send formatted type to either the display or the printer. The other formatting mode is activated by WINDOUT and is called "window-out mode." It will be discussed a bit later.

Now type

```
 " Here is a simple example of the formatter's function."
```

using lower case as shown, and notice that an address, actually PAD, is left on the stack. Now reactivate upper case (press Shift and Caps-Lowr) if you haven't already, and type

```
COUNT
```

The address was bumped by one, and the string count was extracted from the first byte in the string created by " and placed on top of stack. All normal. Now type

```
2DUP CR CR TYPE CR
```

and see that the typed output wraps around as usual. Now try

```
2DUP CR CR *TYPE *CR
```

The word "formatter's" is no longer split. Let's try it again but with different formatting. Type

```
CTRJST  ("center justification")
2DUP CR CR *TYPE *CR
```

How about

```
FILJST  ("fill justification")
2DUP CR CR *TYPE *CR
```

The text is now spread or "filled" to take up the whole space between the margins. The last mode is

```
RGTJST  ("right justification")
2DUP CR CR *TYPE *CR
```

which gives the expected result. Finally, type

```
LFTJST  ("left justification," the default mode)
2DUP CR CR *TYPE *CR
```

and we're back where we started.

Well, what precisely is happening? The 2DUP each time is there of course to reproduce the two stack arguments, adress and count, for use by *TYPE (or TYPE). The two CR's each time are merely to space the result down the page a bit, and make it start at the left margin. As we will see, these two CR's will not generally be necessary in normal programs. The TYPE we'll assume you already know about. If not, look it up in the 1.1 glossary. While you're looking at TYPE's definition you might refresh your memory about how to allow it to type inverse video characters also. A short discussion about this follows TYPE's definition, and we may need this feature later on. OK, what about *TYPE? *TYPE, like TYPE, takes a count and address on the stack, but instead of routing the text directly to an output device, *TYPE sends it instead to a holding buffer, located at BUF, where it accumulates. As each character is sent to the buffer it may be colored, inversed, or capitalized, depending on whether these options are loaded and appropriate. Since we loaded all three of these options we'll try them presently. When the buffer at BUF overflows with a non-blank character, *TYPE formats the line (if any format routines were loaded) and then sends it out via a vect called *XMTLN. Roughly, a vect is a word that can be "assigned" the meaning of a second word so that when the vect is executed it acts precisely like the word last assigned to it.) "XMTLN" in *XMTLN stands for "transmit line." The word *XMTLNP is currently assigned to *XMTLN and is located, in the first release, on or near screen 73. *XMTLNP, and so now *XMTLN, types out the buffer at BUF and increments a line counter if the printer is on and does a few CR's if a printed page is full. After the buffer is output, cleared, and the overhanging characters have been moved to the beginning of the buffer, *TYPE continues to consume the character string. In general there will always be something in the buffer unless it has been cleared. *CR pushes the last of the text out of the buffer to the output device, and then clears and initializes the buffer with BUFINIT. You probably won't have to do BUFINIT yourself unless you are experimenting with the internals of the program. To illustrate this point about *CR, type

```
2DUP  CR CR *TYPE 2DUP *TYPE *CR
```

This time, since we didn't do *CR after the first *TYPE, the next *TYPE tacked its text right on to what was left in the buffer.

Now type

```
SP!  (we'll make a new message)
: SHOW 2DUP CR CR *TYPE *CR ;
" here is another message for another purpose."
COUNT
CTRJST
CAP SHOW
SHOW
```

See what CAP does? Now try

```
ON CAPS
SHOW
OFF CAPS
SHOW
```

And what about inverse video? Since *TYPE uses TYPE and TYPE as it now stands will not print inverse video (it strips the high bit before sending a byte out) we'll need the modification discussed in the 1.1 Glossary, under TYPE. Here it is, type it in, c a r e f u l l y:

```
HEX  FF  '  TYPE  14 + C!  DECIMAL
```

and then type

```
VLIST
```

to see an interesting side note. The high bits of the last byte of (almost) all names are [illegible] search words. But, back to business. Type

```
SHOW
```

And you get [illegible] word in VLIST uses PAD for something, so your mes[illegible] just a reminder. Type

```
SP! [illegible]
" Here [illegible] other features."
COUNT
ON CAPS
ON INVID
SHOW
ON INVBK [illegible] "inverse background")
SHOW
OFF INVID
SHOW
2DUP CR [illegible]
etc.
```

Play with these [illegible] if you like. When you're done, do

```
SP!
OFF CAPS
OFF INVID
OFF INVBK
```

and we'll continue

Virtual Memory [illegible]

We used [illegible] has a serious [illegible] that it won't [illegible] characters since it only keeps a one-byte [illegible] "extended quote") in this package allows longer [illegible] X" will not work from the keyboard. Use [illegible] the two-byte length count and leaves it on [illegible] of X" on screen 120. Take a look at it [illegible]

```
120 [illegible]
```

and a [illegible] of the demo. Turn your stack on if you've turned [illegible]

```
XCOUNT
OFF CAPS [illegible]
FILJST
SHOW
```

If you looked at the screen you may have noticed the right-arrow characters near the end of the text. These cause a *CR to be executed at that point in the text. See *EMIT code for details. You can make your own control characters in a similar fashion. (To type a right arrow character in the valFORTH 1.1 editor, do ESC followed by CTRL-*). Observe also that no --> was required for X" to cross the screen boundary. X" will only stop on finding a final " and so may run right on through a disk looking for one if you forget to put it in.

Well, X" is ok, but not as handy as it might be for general programming. Look at screen 122 and then type

```
SP!
122 LOAD
```

and a demo message will come back again, indicating that a new word, MSGDEM1, now exists. This word will actually pull its message text off the disk. Let's do it. Do MTB just to make sure it's not cheating, and then type

```
CR CR MSGDEM1
```

Notice that we didn't use SHOW this time, just the message name. The messages end with a right-arrow. Now, the method that generates this message, namely using a new word, V" , followed by a string and then a terminating " and then the word M: followed by the message name, does achieve the desired result, but at the price of leaving the V", ", M:, and name on the disk along with the message. This method is provided only because for those working with a one-drive development system it is the easiest, and does not involve any disk swapping during compile time. However, for those with two drive systems, and those with only one drive but also a tolerance for swapping disks every time a message is compiled, the next and last structure in this series is provided. It allows fully compact, text-only messages to be compiled on the final product, and also allows encryption of the text. We will first do it the way the one-drivers need to.

Look at screen 124. The 8D ALTINIT command sets up an alternate set of disk pointers to start at screen 8D. This is where, in our example, the text of the various messages compiled by this method will be stored on the extra disk we formatted at the beginning. Notice that the message starts with X" again. Hence, we see that it will first be assembled at PAD before being sent elsewhere. Now look at screen 125. There's the terminating " , the defining word, MSG:, and the message name, and a short message with ." This final message is just there for convenience in this demo and is not needed in general. Type

```
124 LOAD
```

and when it tells you to put in the destination disk, swap in the extra blank disk you formatted, then press START as directed. At the next prompt, swap back and press START again. When using this method you must be very careful not to reverse your disks or you may wipe out part of your source disk.

Well, the message is now written to the second disk on screen 80, and the word MSGDEM2 knows where to find it. Let's take a quick look to see that it's really there. Type

MTB

to empty the buffers, and then swap disks again (so that the destination disk is in the drive) and do 80 LIST, and then 81 LIST. There's the message. The first two strange bytes on screen 8D are the count. Now do

MTB CR CR MSGDEM2

and watch the routines pull the message from the disk. While we're here, let's send this to the printer. But since your printer may have characteristics different from the printer this package is initialized for, we want to adjust a couple of things. The first item is a quan named PWID. This is the actual number of columns your printer has. The default value is 80. To change it to 96, for example, type

96 TO PWID

The second item is the quan PRTWID which is the width of the area you'd like to print to. The default again is 8D. To set it to 60, say, type

60 TO PRTWID

The third item is how far you'd like to indent. This is the quan PRTIND and its initial value is D. To set it to 10 type

10 TD PRTIND

Finally, we want to tell the formatter to send its output to the printer now, so type

PRT:

(The default setting was to the video display, and will be called back by VID:) Is your printer ready? Let's try it. Type

MSGDEM2

Since many printers will get confused if a character with the high bit set is sent to them you might want to be careful about this.

Incidentally, the same options are available with the video display. PRTWID becomes VIDWID and PRTIND becomes VIDIND. PRT: becomes VID:. There is no "VWID" since the formatter derives this from the positioning of the margins. (The left margin is kept by the OS in the byte at 82 decimal, and the right margin byte is at 83. Default are 2 and 39 respectively.)

Any new printer settings only become active when PRT: is executed, and likewise with video settings and VID:.

Try

```
20 TO VIDWID
4 TO VIDIND
VID:
CR CR MSGDEM2
38 TO VIDWID  (back to default)
0  TO VIDIND  (ditto)
VID:   (move in new values)
```

OK, now swap the source disk back in, that is, the TCAF working disk, but keep the destination disk handy. Let's load a few (six) more messages. Type

```
MTB (to empty the buffers)
126 LOAD
```

and follow the prompts.

As you can see, any large amount of this single-drive compilation could be quite tiresome. Do a short VLIST (abort with any of the three yellow console buttons) and look at the new messages. Swap in the destination disk, do MTB, and then try

```
CR M0
CR M1
etc.
```

Encryption and 2-Drive Systems

There are two more features to point out. They are encryption/decryption (e/d) and adjustments for two-drive systems. Concerning e/d, look at screen 105, or wherever you find the title EN, DECRYPT or similar. (Do an INDEX if you can't find the right screen easily.) Notice that there is a --> at the top of this screen which is causing it not to load. Remove this arrow with your editor. (Since you're going to reload the system in a minute anyway, it's ok if you over-write the system to get an editor in. Get one in somehow.) Now on the next two screens you should find the words ENCRYPT and DECRYPT in parentheses. (DECRYPT is in three times.) Remove the parens to allow these two words to load. Now, you folks with two drives, find the screen where MSG: is defined. (On or near screen 112). There are several sets of parens. Leave the ones that enclose $ENCRYPT and ... $DECRYPT ... alone. Shift only the ones that are around " DR! or " to be around " or DSTDSK " and shift the ones around " DR0 or " to be around " or SRCDSK ." Just to be on the safe side, here's a picture of how the screens should look after these changes.

FOR ONE OR TWO DRIVE SYSTEMS

```
Scr # 105
 ( Vrtxt:  EN,DECRYPT  example  )



 : ENCRYPT                ( c1 -- c2 )
    117 - DUP 0(
    IF 256 + ENDIF ;

 : DECRYPT                ( c2 -- c1 )
    117 + DUP 255 )
    IF 256 - ENDIF ;





                                    --)
```

```
Scr # 106
 ( Vrtxt:  V$TP  V$@  V$*EMT      )

 : V$TP           ( -- X$ [=PAD] )
   VRTC@   DECRYPT   NXTVRT VRTC@
     DECRYPT   NXTVRT 256 * + ;

 : V$@            ( -- X$ [=PAD] )
   PAD 2+ V$TP DUP PAD ! 0
   DO VRTC@ OVER C! 1+ NXTVRT
   LOOP DROP PAD ;

 : V$*EMT         ( -- X$ [=PAD] )
   V$TP 0
   DO VRTC@   DECRYPT
      *EMIT NXTVRT
   LOOP ;                           --)
```

```
Scr # 107
 ( Vrtxt:   V$!                    )

 : V$!                     ( X$ -- )
   DUP @ 2+ 0
   DO DUP C@   ENCRYPT
      VRTC! 1+ NXTVRT
   LOOP DROP ;









                                    --)
```

FOR TWO DRIVE SYSTEMS ONLY

```
Scr # 112
 ( Vrtxt:  ALT$!  MSG:              )

 : ALT$!                   ( X$ -- )
   VRTSAV ALTREC V$!
   ALTSAV VRTREC ;

 : MSG:                    ( X$ -- )
   ( $ENCRYPT )
   (BUILDS   DR1 ( or   DSTDSK )
    ALTBLK , ALTIN , ALT$!
    FLUSH    DR0 ( or   SRCDSK )
   DOES) VRTSAV
    DUP @ SWAP 2+ @ IN ! BLK !
    V$*EMT ( or )
    ( V$@ $DECRYPT XCOUNT *TYPE )
    VRTREC ;
```

What this last change does is substitute an automatic disk-shift for the swap prompts driven by DSTDSK and SRCDSK. These words are no longer needed and may be bypassed later on, if you're comfortable with the way things are working. OK, now, although some of you might be able to get these changes in by doing some FORGETting and reloading (if you didn't have to overwrite the system before), why not just reload the whole (modified) system this time, starting from valFORTH 1.1. Don't forget the debugger if you want it, and remember to initialize with TYPEOUT. Go ahead. We'll wait.

Now you can repeat the exercises from before, starting where you did the 124 LOAD. Two drive systems should have the "destination" disk, the one with the messages on it, in the second drive, and the source disk, TCAF, in the first drive. Two-drive systems should execute DR1 (which in FORTH means the second drive, which is number 2 on Atari systems) before saying message names so that the code will read the right drive. Say DR0 to go back to the source-code drive. (Released programs will of course not want to say DR1 since they will expect the "game" or application disk to be in the first drive, DR0, which is default. When you are making messages in this way, be sure to start them high enough on disk that your AUTO'd program, which will actually do the message retrieval, will fit under them.) If you look at screen 80 on the destination disk, what you'll see is that the text is now scrambled. The encryption routine used is a simple off-set scheme, and would be easy to crack for a serious hobbyist, though not for the casual user. If you are interested in a higher degree of security, you can encode a whole string at a time with more sophisticated routines. A pair of names, $ENCRYPT and $DECRYPT, have been reserved for these routines. We don't provide any examples, but a modern text on cryptography might be a good place to start looking. Anyway, if you use the names $ENCRYPT and $DECRYPT, and if the routines expect an extended string on stack (that is, one with a two-byte count at its front end) then they will (hopefully) snap right into the spot designated by the parens.

Windows

Windows are rectangular areas of the video display. They are not supported on the printer, but are supported in both "black and white" graphics 0 mode, and colored graphics 1 and 2 modes. Windows may be set up on-the-fly or they may be given names so that words can call them up readily. The implementation provided here may be used as an example and guide, since you may want your windows to act somewhat differently.

Since it is tricky to interact with a graphics 0 window from the keyboard (there is no simple way that we can find to date to create a split-screen option) we'll illustrate windows in graphics 1, and so also show how color works. Type

```
1 GR.
3 4 10 5 MAKECW
```

This makes a Color Window whose upper left hand corner is at the 3rd column over, 4th row down (counting the left and top edges as zero), and which is

10 characters wide and 5 high. We have messages M0 through M5 still available, so let's send them to the window. Type

```
WINDOUT  (counterpart of TYPEOUT)
OFF CAPS
0 COLOR M0
1 COLOR M1
2 COLOR M2
3 COLOR M3
WCLR
```

Note the extra coloration caused by the mix of upper and lower clase. This can be canceled by ON CAPS though it restricts the user to two color for the letters instead of four. A good practice would be to put all the source text for colored windows in upper case. Coloration switches in the middle of a message could be implemented by control characters similar to the right-arrow character and its meaning of *CR. This is done in *EMIT, you'll remember, and you might even use a case statement.

Numerical formatting is also supported, by the words *. and *.R which are direct counterparts of . and .R, except that they go through the formatter before outputting. Try

```
3456 *. *CR
7890 7 *.R *CR
```

These routines should be used both with WINDOUT and TYPEOUT. Using just . or .R will upset the formatting.

There is also

```
2 2 15 8 NAMECW  BINGO
```

which names a color window with the given parameters as BINGO. When BINGO is executed it will clear itself and position the imaginary cursor at its the upper lefthand corner. By studying the code, this and other performance characteristics may be altered.

Text Compression

Finally, there is "true" text compression (TC) itself. TC is intended primarily for applications where disk access to messages is not available, such as in cassette-booted systems. This utility uses Transient structures which you have probably come across before in the packages in this series. Hence, all the warnings about memory collisions must to some extent apply. The text compression utilities themselves are fairly straightforward to use, but what they do is rather complex. Briefly, TC allows the creation of bits of headerless code called "tc-texts" that, when executed, put a string onto the stack and then jump to the appropriate Forth words, for formatting. These tc-texts come in three types in this package, namely, tc-words, tc-suffixes and tc-prefixes.

The general procedure is to:

(1) Load the text compression routines.
(2) Define all needed tc-texts.
(3) Define all words that use tc-texts within themselves.
(4) Execute DISPOSE which will sever links to all of the tc-text creating and compiling structures, leaving only minimal, minimal, headerless structures.

Since (it turns out) we can load the text-compression routines right on top of the rest of the code we already have in, let's do that. Look at the screen in the load chain which you modified at the beginning of this excursion. It was probably 167. Text compression was not loaded at that time. Note the load screen for text compression (probably screen 60) and load it. Because this section uses transients you cannot use SAVE to create a bootable copy until you have executed DISPOSE to break the links to the transient area. In addition, FORGET will act a bit odd, and may cause crashes, so try to avoid using it until you have disposed. There are only three new words to learn in text compression, namely, W= , P= , and S=. These words define tc-words, tc-prefixes and tc-suffixes. For example,

```
W= DOG
W= TAIL
P= SUPER
S= 'S
S= !
```

defines five tc-texts. Type in the five definitions above and then type

```
DOG DOG DOG DOG DOG CR *CR
SUPER DOG CR *CR
SUPER DOG 'S TAIL ! CR *CR
```

The justification, capitalization, coloring, window output, and other options will also function with tc-texts.

Obviously, there is potential for numerous word-name conflicts between tc-texts and FORTH. The punctuation marks, for instance P= . P= , P= ! and so on all are desirable and all already exist in the FORTH vocabulary. Hence the three defining words for tc-texts automatically put the words they define into a separate vocabulary named ^. In addition, the name | (Shift-=) has been assigned as an alias for FORTH to shorten source code and ease typing. For instance, one might have a FORTH word like:

```
: ?TL                                ( flag -- )
  IF  ^  A_DOG_'S_TAIL_IS_HERE !  *CR  |
  ENDIF ;
```

By going into the ^ vocabulary the tc-text ! was interpreted properly, instead of as the FORTH !. Similarly, by going back into the | (FORTH) vocabulary, the word ; was interpreted as the FORTH ; rather than as some prefix that might have been in the ^ vocabulary.

Several more points are worth noting:

* If you are programming short phrases that do not generally run together, you can save some memory by defining a <BUILDS DOES> construct that always attaches the *CR to the end of the operation, thus saving two bytes per message, with the new <BUILD DOES> that loaded with this package.

* If you want to create new types of tc-texts, such as one to deal with problems like SHINE ING, just follow the examples of how the words W= P= S= are constructed. Smart prefixes that strip trailing vowels, for example, would not be difficult to code, but would not necessarily be worth the memory cost. However, in a very large application it might well be worth coding a large number of spelling rules.

* To create tc-texts that contain blanks, create a control character that is not printed, and use this as the blank. The character we suggest for this is the underline, whose ATASCII is 95. Note how the right-arrow, ATASCII 31, is picked off by *EMIT. Do the same for 95, only make it perform *SPACE instead of *CR as right-arrow does.

* Some tc-texts will get rather long, and will be cumbersome in source text. They can be provided with a no-cost alias. For example, say we had

```
W= A_DOG_WITH_A_BONE
```

We could then add

```
TRANSIENT
: D&B ^ [COMPILE] A_DOG_WITH_A_BONE_ | ; IMMEDIATE
PERMANENT
```

This alias would be removed by DISPOSE, as would, of course, the tc-text name A_DOG_WITH_A_BONE, leaving only the headerless tc-text itself.

* As set up, the Transient system is 4000 bytes below the display list. This may not be enough for some applications. The way to find out how much room you have left in the transient area is to type

```
TRANSIENT 741 @ HERE - U. PERMANENT
```

You might even define a word to do this. Call it TFREE. A trap in CREATE, and so also in : is designed to keep you from actually running into the display list by simply aborting the definition in progress when there are less than 128 bytes left.

* Finally, always remember to DISPOSE when you're done with the transients. If you forget and do SAVE you will not get a working system.

This system of compression is quite compact, costing only two bytes to produce output from a tc-text. The cost in memory of producing the tc-text of a word of n letters, (not even counting the trailing blank) is only n + 2.

TEXT COMPRESSION AND AUTO TEXT FORMATTING GLOSSARY

Basic Commands

*." (--)

Like ." , but sends string to the active formatting/outputting routines.

*TYPE (addr count --)

Like TYPE, but sends string of count characters starting at addr to the active formatting/outputting routines.

*CR (--)

Somewhat like CR in that it causes a carriage return. In addition, *CR first formats and flushes the buffer to the output device, and clears the buffer after doing so.

*EMIT (c --)

Like EMIT except sends the character c to the formatter, instead of directly to the output device.

*SPACE (--)

Sends a single character of value in the quan BKGND to the formatter, through *EMIT.

*SPACES (n --)

Sends n characters of value in the quan BKGND to the formatter, through *EMIT.

*BACKS (--)

Similar to action of delete key. Backs up the formatter buffer pointer, BPTR, one location and fills new location with BKGND value.

RGTJST (--)

Sets up formatter for right justification.

LFTJST (--)

Sets up formatter for left justification.

CTRJST (--)

Sets up formatter for center justification.

FILJST (--)

Sets up formatter for fill justification.

INVID (f --)

ON INVID means text will be output in inverse video; OFF INVID means normal video.

INVBK (f --)

ON INVBK means background of text will be output in inverse video. OFF INVID means normal video.

CAP (--)

Causes capitalization of the next byte processed by *EMIT or *TYPE.

CAPS (f --)

ON CAPS means subsequent formatted text will be capitalized if lower case. OFF CAPS means text will be printed as-is.

COLOR (b --)

New color register b will be used for color of subsequent text output to windows in Graphics modes 1 and 2.

TYPEOUT (--)

Initialization routine for the formatter. Either TYPEOUT or WINDOUT must be executed before the first attempt to output text from the formatter or the system may crash. TYPEOUT directs the formatter to use TYPE as its actual output routine, allowing output to the display screen or printer.

WINDOUT (--)

Initialization routine for the formatter. Either TYPEOUT or WINDOUT must be executed before the first attempt to output text from the formatter or the system may crash. WINDOUT directs the formatter to use window routines for output. A window must be created before attempting to use window output or the system may crash. See also NAMWND.

Quans, vects, and subcommands

FDIR (-- +-1)

A quan that holds the next direction to be used by the fill-justification routines when padding the text in the formatting buffer with blanks.

*JUST (--)

A vect used to point to the routine that performs whatever justification action is current. Altered by LFTJST, RGTJST, CTRJST, and FILJST.

BKGND (-- n)

Quan which holds the value of the background character to be used when clearing the formatting buffer. Generally either 32 (blank) or 160 (inverse blank.) See INVBK.

EOB (-- n)

Quan which points to the location in the formatter buffer corresponding to the last allowable position in the current output width. Set up by various routines including PRT:, VID:, and window-creating routines. Stands for "end of buffer."

BPTR (-- n)

Quan which points to the next available location in the formatter buffer. May be user-altered for special purposes, but should not be placed lower than BUF or higher than EOB. Stands for "buffer pointer."

WWID (-- n)

Quan which holds width of field to which text will be output. Used to set up EOB, which is actually used by the formatting routines. See EOB. Stands for "window width" though windows as defined elsewhere need not exist.

*XMTLN (--)

A vect that points to the routine to be used to move text from the formatter buffer to the output device. Set up at present either by TYPEOUT or WINDOUT. Stands for "transmit line."

BUF (--)

A label that points to the beginning of the formatter buffer area. This area need only be three bytes longer than the longest line to be formatted.

INVBK (ON or OFF --)

When ON, background character output by formatter in 0 graphics mode will be inverse video blank. When OFF, this character will be normal video blank. Sets up BKGND. See BKGND.

BUFCLR (--)

Fills the formatter buffer with BKGND.

BUFINIT (--)

Fill the formatter buffer with BKGND, sets up EOB using WWID and BUF, and points sets BPTR equal to BUF.

*TINT (c -- c)

A vect that either points to the coloring routines when a color window is active, or to NOOP when a 0 graphics window is active.

*CAP (c -- c)

Capitalization routine.

*INV (c -- c)

A vect that either points to the inversing routine when a 0 graphics window is active, or to NOOP when a color window is active.

<u>Text Compression</u>

W= xxx, (--)
xxx: (--)

Creates a tc-word-compiling word, named xxx, and a headerless tc-word which when executed sends the string xxx through the formatter followed by *SPACE. xxx when executed, compiles in the cfa of this tc-word. W= and xxx are both in transient area and so are disposed by DISPOSE.

P= xxx, (--)
xxx, (--)

Creates a tc-prefix-compiling word, named xxx, and a headerless tc-prefix which when executed sends the string xxx through the formatter. xxx when executed, compiles in the cfa of this tc-prefix. P= and xxx are both in the transient area and so are disposed by DISPOSE.

S= xxx, (--)
xxx, (--)

Creates a tc-suffix-compiling word, named xxx and a headerless tc-suffix which when executed sends the string xxx through the formatter preceded by *BACKS and followed by *SPACE. xxx, when executed, compiles in the cfa of this tc-suffix. S= and xxx are both in the transient area and so are disposed by DISPOSE.

Typed Output

PRTWID (-- n)

A quan containing the width of the area to be printed when printer output from the formatter has been selected by PRT:. PRT:, among other things, moves PRTWID to WWID.

PRTIND (-- n)

A quan containing the number of spaces the printer is to indent when outputting from the formatter. PRTIND is moved to PVIND by PRT:

PVIND (-- n)

A quan containing the number of spaces the output device is to indent when outputting from the formatter. Set up by PRT: from PRTIND or by VID: from VIDIND.

PWID (-- n)

A quan containing the number of columns the printer is actually able to print as it is currently configured, and independent of the formatting routines.

VIDIND (-- n)

A quan containing the number of spaces the output routines is to indent when outputting from the formatter. VIDIND is moved into PVIND by VID:.

VIDWID (-- n)

A quan containing the width of the area to be written when video output from the formatter has been selected by VID:. VID:, among other things, moves VIDWID to WWID.

PRT: (--)

Directs TYPEd output to the printer, and moves appropriate values into WWID and PVIND.

VID: (--)

Directs TYPEd output to the video display, and moves appropriate values into WWID and PVIND.

PRINIT (--)

Resets PCTR, the printed line counter.

*XMTLNP (--)

Routine sent to the vect *XMTLN by TYPEOUT. Routes output through TYPE.

Windows

WADR (--)

Address in memory corresponding to character position in upper lefthand corner of current window.

WHGT (--)

Height in lines of currently active window.

LPTR (--)

Counter that holds number of next line in window to which text is to be written. If LPTR points beyond the window then scrolling will occur at next output.

B/LN (-- n)

Bytes per line. Necessary datum for scrolling and clearing routines for windows.

WCLR (--)

Fills the current window with BKGND.

NAMWND (wadr wid hght b/ch byt/ln --)

One of many possible window-defining structures. Accepts window upper lefthand corner address, its width, height, byte-character, and the bytes/ln of the current graphics mode.

NAMEBW xxx, (column row wid hgt --)
xxx: (--)

Names a 0 graphics window for later activation.

MAKEBW (col row wid hgt --)

Establishes a 0 graphics window immediately but does not name it for later retrieval.

NAMECW xxx, (col row wid hgt --)
xxx: (--)

Names a 1 or 2 graphics window for later activation.

MAKECW (col row wid hgt --)

Establishes a 0 graphics window immediately but does not name it for later retrieval.

Virtual (Disk-based) Memory

(A pointer to a byte on disk is implemented by the two system variables, BLK and IN in the fig model. BLK contains the block number pointed to and IN contains the number of bytes into the block the byte in question is located.)

VRTC@ (-- b)

Fetches the byte pointed to on disk by the system variable BLK and IN. (BLK is the block number, and IN is the number of bytes into the block the desired byte is located.)

VRTC! (b --)

Stores the byte on stack to the location on disk pointed to by BLK and IN. See VRTC@.

VRTSAV (--)

Saves the values of system variables BLK and IN to quans OBLK and OIN respectively.

VRTREC (--)

Recalls the values of the system variables BLK and IN from the quans OBLK and OIN respectively.

NXTVRT (--)

Bumps the system variables BLK and IN as required to point to the next location in virtual memory.

RELVRT (offset --)

Takes an offset on stack and alters the system variables BLK and IN as necessary to point offset bytes from their initial virtual memory location.

V" (-- blk in)

Leaves the values of BLK adn IN on the stack at the time it is executed and then scans the virtual memory pointer formed by BLK and IN forward until the next " character is encountered.

XMTV (--)

Starting from the location in virtual memory pointed to by BLK and IN, outputs characters through *EMIT until a " character is encountered, which it does not output.

XCOUNT (adr -- adr+2 xcount)

Extracts a two-byte count from an extended string, and leaves the count on top of the address + 2.

M: xxx, (blk in --)
xxx: (--)

Generally used after V". Takes a virtual memory pointer from the stack, and creates a word xxx which when executed will push the virtual memory pointer to BLK and IN and then exectue XMTV, thus retrieving a message from disk. See Strolling... for an example.

V: xxx, (blk in --)
xxx: (--)

Creates a word xxx which when executed pushes the virtual memory pointer which was on stack at the time of its creation to BLK and IN.

V$TP (-- XCOUNT)

Extracts a two-byte string count from the disk location to which BLK and IN point, leaves it on stack, and bumps the virtual memory pointer made up of BLK and IN twice.

V$@ (-- X$=PAD)

Extracts the extended string in virtual memory pointed to by BLK and IN. The string is left at PAD.

V$*EMT (--)

Sends the extended string pointed to by BLK and IN through *EMIT.

V$! (X$ --)

Stores the extended string on stack to virtual memory starting at the location pointed to by BLK and IN.

X" (---X$=PAD)

Reads the following characters until the delimeter " as an extended string and stores the string at PAD. Operates from screens only. Crosses block and screen boundaries without additional code. Do not use --> to cross screens, as --> will just become part of the string.

ALTSAV (--)

Copies variables BLK and IN to quans ALTBLK and ALTIN respectively.

ALTREC (--)

Copies quans ALTBLK and ALTIN to variable BLK and IN respectively.

ALTINIT (scr --)

Sets up ALTBLK and ALTIN to point to screen scr. ALTBLK and ALTIN form an auxiliary virtual memory pointer that is used to keep track of how far messages have been compiled onto the destination disk.

ALT$! (X$ --)

Like V$! except stores string through alternate virtual memory pointers made up of ALTBLK and ALTIN.

LIII. TEXT COMPRESSION AND AUTO TEXT FORMATTING

SUPPLIED SOURCE LISTING

Screen: 1

Screen: 2

Screen: 3

Screen: 4

```
( Transients:  setup            )
'( QUAN )( 5 KLOAD )

BASE @ DCX

HERE

741 @ 4000 - DP !
( SUGGESTED PLACEMENT OF TAREA )


HERE CONSTANT TAREA
QUAN TP
QUAN TPFLAG  1 TO TPFLAG
QUAN OLDDP ( old HERE ) TO OLDDP
                               -->
```

Screen: 5

```
( Xsients: TRANSIENT PERMANENT )
( Expanded from code by Phillip)
( Wasson, in Forth Dimensions  )

: TRANSIENT                ( -- )
  TPFLAG NOT
  IF HERE TO OLDDP TP DP !
     1 TO TPFLAG
  ENDIF ;

: PERMANENT                ( -- )
  TPFLAG
  IF HERE TO TP OLDDP DP !
     0 TO TPFLAG
  ENDIF ;
                               -->
```

Screen: 6

```
( Transients:  DISPOSE          )
: DISPOSE   PERMANENT
  CR ." Disposing..." VOC-LINK
  BEGIN DUP    0 53279 C!
   BEGIN @ DUP TAREA U<
   UNTIL DUP ROT ! DUP 0=
  UNTIL DROP VOC-LINK @
  BEGIN DUP 4 -
   BEGIN DUP    0 53279 C!
    BEGIN PFA LFA @ DUP TAREA U<
    UNTIL
     DUP ROT PFA LFA ! DUP 0=
   UNTIL DROP @ DUP 0=
  UNTIL DROP [COMPILE] FORTH
  DEFINITIONS ." Done" CR ;
  PERMANENT               BASE !
```

Screen: 7

Screen: 8

Screen: 9

Screen: 10

```
( Quan:  ASSIGN                 )

'( CFALIT
: ASSIGN [COMPILE] CFALIT ;
IMMEDIATE --> )(  )

: ASSIGN               ( -- cfa )
  STATE @
  [COMPILE] [
  [COMPILE] ' CFA SWAP
  IF ]
  ENDIF [COMPILE] LITERAL ;
  IMMEDIATE
                             -->
```

Screen: 11

```
( Quan:  TO  AT                 )

: TO
  -FIND 0= 0 ?ERROR DROP
  STATE @
  IF ,
  ELSE EXECUTE
  ENDIF ; IMMEDIATE

: AT
  -FIND 0= 0 ?ERROR DROP
  2+ STATE @
  IF ,
  ELSE EXECUTE
  ENDIF ; IMMEDIATE
  ( corrected )              -->
```

Screen: 12

```
( Quan:  [2@6]  [2!4]           )

ASSEMBLER HEX

LABEL (2@6)
  A0 C, 06 C, B1 C, W  C, 48 C,
  C8 C, B1 C, W  C, 4C C, PUSH ,

LABEL (2!4)
  A0 C, 04 C, B5 C, 00 C, 91 C,
  W  C, C8 C, B5 C, 01 C, 91 C,
  W  C, 4C C, POP ,

                             -->
```

```
Screen:  13
( Quan:  [2V6]                    )

LABEL (2V6)
   A0 C, 07 C, B1 C, W  C, 48 C,
   88 C, B1 C, W  C, 85 C, W  C,
   68 C, 85 C, W 1+ C,
   A0 C, 00 C, 4C C, W 1- ,








                                 --)
```

```
Screen:  14
( Quan:  patch for CREATE        )

DCX

: (PTCH)                ( system )
   SWAP )R R = 251 R = 249 R) =
   OR OR ;

: PTCH                  ( system )
   IF [ ' (PTCH) CFA ] LITERAL
   ELSE [ ' = CFA ] LITERAL
   ENDIF
   [ ' CREATE 63 + ] LITERAL ! ;


                                 --)
```

```
Screen:  15
( Quan:  QUAN  VECT               )

: QUAN
   ON PTCH LABEL -2 ALLOT
   (2@6) , (2!4) ,
   [ ' VARIABLE 4 + ] LITERAL ,
   2 ALLOT OFF PTCH ;

: VECT
   ON PTCH LABEL -2 ALLOT
   (2V6) , (2!4) ,
   [ ' VARIABLE 4 + ] LITERAL ,
   [ ' NOOP CFA ] LITERAL ,
   OFF PTCH ;
```

```
Screen:  16
( Utils:  UMAX  UMIN  HIDCHR    )

: UMAX             ( u1 u2 -- u3 )
   2DUP U(
   IF SWAP ENDIF
   DROP ;

: UMIN             ( u1 u2 -- u3 )
   2DUP U)
   IF SWAP ENDIF
   DROP ;

'( HIDCHR )(
: HIDCHR
   -1 94 ! ; )
                                --)
```

```
Screen:  17
( Utils:  S:  P:                 )

'( S: ;S )(  )
HEX

: S:                       ( f -- )
   PFLAG @ SWAP
   IF 1 OR ELSE FE AND ENDIF
   PFLAG ! ;

: P:                       ( f -- )
   PFLAG @ SWAP
   IF 2 OR ELSE FD AND ENDIF
   PFLAG ! ;

DCX
```

```
Screen:  18
```

```
Screen:  19
```

```
Screen:  20
 ( Screen code conversion words )

 BASE @ HEX

 CODE >BSCD              ( a a n -- )
    A9 C, 03 C, 20 C, SETUP ,
    HERE  C4 C, C2 C, D0 C, 07 C,
    C6 C, C3 C, 10 C, 03 C, 4C C,
    NEXT ,       B1 C, C6 C, 48 C,
    29 C, 7F C, C9 C, 60 C, B0 C,
    0D C, C9 C, 20 C, B0 C, 06 C,
    18 C, 69 C, 40 C, 4C C, HERE
 2 ALLOT 38 C, E9 C, 20 C, HERE
 SWAP !  91 C, C4 C, 68 C, 29 C,

                                  -->
```

```
Screen:  21
 ( Screen code conversion words )

    80 C, 11 C, C4 C, 91 C, C4 C,
    C8 C, D0 C, D3 C, E6 C, C7 C,
    E6 C, C5 C, 4C C, ,       C;

 CODE BSCD>              ( a a n -- )
    A9 C, 03 C, 20 C, SETUP ,
    HERE  C4 C, C2 C, D0 C, 07 C,
    C6 C, C3 C, 10 C, 03 C, 4C C,
    NEXT ,       B1 C, C6 C, 48 C,
    29 C, 7F C, C9 C, 60 C, B0 C,
    0D C, C9 C, 40 C, B0 C, 06 C,
    18 C, 69 C, 20 C, 4C C, HERE
 2 ALLOT 38 C, E9 C, 40 C, HERE
                                  -->
```

```
Screen:  22
 ( Screen code conversion words )

 SWAP !  91 C, C4 C, 68 C, 29 C,
    80 C, 11 C, C4 C, 91 C, C4 C,
    C8 C, D0 C, D3 C, E6 C, C7 C,
    E6 C, C5 C, 4C C, ,


 : >SCD   SP@ DUP 1 >BSCD ;
 : SCD>   SP@ DUP 1 BSCD> ;






                          BASE !
```

```
Screen:  23
```

```
Screen:  24
```

Screen: 25

Screen: 26

Screen: 27

Screen: 28

```
( AF0:  quans  vects            )
QUAN BKGND      ( background chr )
BL TO BKGND
QUAN EOB          ( end of buffer )
QUAN BPTR       ( buffer pointer )
QUAN WWID      ( characters/line )
QUAN B/C       ( bytes/character )
QUAN LWD  ( 1st chr of last wd )
VECT *XMTLN      ( send fmted ln )
LABEL BUF 123 ALLOT    ( buffer )
( Need only be longest line +3 )
                             -->
```

Screen: 29

```
( AF0:  ?BL   INVBK             )
: ?BL                     ( -- f )
  C@ 31 AND 0= ;
: INVBK                   ( f -- )
  IF 160
  ELSE BL
  ENDIF TO BKGND ;
                             -->
```

Screen: 30

```
( AF0:  BUFCLR  BUFINIT         )
: BUFCLR                   ( -- )
  BUF WWID BKGND FILL ;
: BUFINIT                  ( -- )
  WWID BUF + 1- TO EOB
  BUFCLR BUF TO BPTR ;
 38 TO WWID
 BUFINIT
 ( Setup for 0 GR. display )
```

Screen: 31

Screen: 32

```
( Justify:  *JUST  LCRH  ^LCHR )

VECT *JUST
QUAN LCHR

: ^LCHR                       ( -- )
  EOB
  BEGIN DUP BUF U) OVER ?BL AND
  WHILE 1-
  REPEAT TO LCHR ;
```

Screen: 33

Screen: 34

```
( R,cjust:  ^RJ  ^CJ                 )

'( *JUST )( 16 KLOAD )
: (RCJ)                        ( b -- )
  )R BUF BUF EOB LCHR -
  R) / DUP )R +
  LCHR BUF - 1+ (CMOVE
  BUF R) BKGND FILL ;

: ^RJ                          ( -- )
  1 (RCJ) ;

: ^CJ                          ( -- )
  2 (RCJ) ;

                                   --)
```

Screen: 35

```
( R,cjust:  RGT,LFT,CTRJST          )

: RGTJST                       ( -- )
  ASSIGN ^RJ TO *JUST ;

: LFTJST                       ( -- )
  ASSIGN NOOP TO *JUST ;

: CTRJST                       ( -- )
  ASSIGN ^CJ TO *JUST ;
```

Screen: 36

Screen: 37

Screen: 38

```
( Fjust:  quans  <FPTR>          )

'( *JUST )( 16 KLOAD )

QUAN FDIR  1 TO FDIR
QUAN FPTR
QUAN ?FJ

: <FPTR>                     ( f -- )
  FPTR BUF U< NOT
  FPTR LCHR U> NOT AND ;




                                   -->
```

Screen: 39

```
( Fjust:  FPASS                    )

: FPASS                      ( -- )
  0 TO ?FJ
  BEGIN LCHR EOB U< <FPTR> AND
  WHILE FPTR ?BL
   IF 1 TO ?FJ
      FPTR FPTR 1+ EOB FPTR -
      <CMOVE 1 AT LCHR +!
    BEGIN FDIR AT FPTR +!
      FPTR ?BL NOT <FPTR> NOT OR
     UNTIL
   ENDIF FDIR AT FPTR +!
  REPEAT ;

                                   -->
```

Screen: 40

```
( Fjust:  ^FJ  FILJST              )

: ^FJ                        ( -- )
  1 TO ?FJ
  BEGIN LCHR EOB U< ?FJ AND
  WHILE FDIR 0>
   IF BUF ELSE LCHR ENDIF
   TO FPTR FPASS
  REPEAT FDIR MINUS TO FDIR ;

: FILJST                     ( -- )
  ASSIGN ^FJ TO *JUST ;
```

Screen: 41

Screen: 42

```
( AF1:  [LWD]  MOVWD   RETWD   )
'( BKGND )( ;S )

: <LWD>                      ( -- )
  BPTR
  BEGIN 1- DUP BUF U<
        OVER ?BL OR
  UNTIL 1+ TO LWD ;

: MOVWD                      ( -- )
  LWD HERE BPTR LWD - <CMOVE ;

: RETWD                      ( -- )
  HERE BUF BPTR LWD -
  DUP >R CMOVE
  R> BUF + TO BPTR ;         -->
```

Screen: 43

```
( AF1:  [SNDLN]  SENDLN         )

: (SNDLN)                   ( -- )
  '( *JUST ^LCHR *JUST )(  )
  *XMTLN BUFCLR ;

: SENDLN                    ( -- )
  (LWD) LWD BUF U)
  IF MOVWD LWD EOB LWD - 1+
     0 MAX BKGND FILL
  ENDIF (SNDLN)
  LWD BUF )
  IF RETWD
  ELSE BPTR 1- C@ BUF C!
       BUF 1+ TO BPTR
  ENDIF ;                       --)
```

Screen: 44

```
( AF1:  *CR                     )

: *CR                       ( -- )
  BPTR BUF =
  IF BUF WWID BKGND FILL
    '( *JUST
  ELSE ^LCHR )(  )
    '( ^FJ ASSIGN ^FJ )( 0 )
    '( *JUST
    AT *JUST @ ()
    IF *JUST
    ENDIF (  )
  ENDIF
  *XMTLN BUFINIT ;

                                --)
```

Screen: 45

```
( AF1:  *EMIT                   )

: *EMIT                   ( c -- )
  DUP 31 =
  IF DROP *CR
  ELSE '( *TINT *TINT )(  )
       '( *CAP  *CAP  )(  )
       '( *INV  *INV  )(  )
   BPTR C! 1 AT BPTR +!
   BPTR EOB 1+ U)
   IF BPTR 1- ?BL
    IF BPTR EOB 2+ MIN TO BPTR
    ELSE SENDLN
    ENDIF
   ENDIF
  ENDIF ;                       --)
```

Screen: 46

```
( AF1:  *TYPE                   )

: *TYPE         ( addr count -- )
  BEGIN DUP 0)
  WHILE
   OVER C@ 127 AND
  *EMIT 1- SWAP 1+ SWAP
  REPEAT 2DROP ;








                                --)
```

Screen: 47

```
( AF1:  *SPACE[S]  *BACKS       )

: *SPACE                    ( -- )
  BKGND *EMIT ;

: *SPACES                 ( n -- )
  0 MAX -DUP
  IF 0 DO *SPACE LOOP
  ENDIF ;

: *BACKS
  BPTR 1- BUF UMAX TO BPTR
  BL '( *INV *INV )(  )
  BPTR C! ;

                                --)
```

Screen: 48

```
( AF1:  [*."]  *."              )

: (*.")                     ( -- )
  R COUNT DUP 1+
  R) + )R *TYPE ;

: *."
  ASSIGN TYPE ASSIGN (.")
  [ ' ." 13 + ] LITERAL
  ASSIGN (*.") OVER !
  [ ' ." 35 + ] LITERAL
  ASSIGN *TYPE OVER !
  [COMPILE] ."
  (ROT ! ! ;         IMMEDIATE
```

Screen: 49

```
```

Screen: 50

```
( Coloring:  *TINT  etc.         )

'( >SCD )( 10 KLOAD )

VECT *TINT

'( CLRBYT )(
0 VARIABLE CLRBYT
: COLOR CLRBYT ! ; )

: ^TINT                     ( c -- c )
   >SCD CLRBYT @
   64 * OR SCD> ;

ASSIGN ^TINT TO *TINT
```

Screen: 51

```
```

Screen: 52

```
( Capitalization:  CAP[S]  etc.)

QUAN ?CAP
QUAN ?CAPLK


: CAP                          ( -- )
  1 TO ?CAP ;

: CAPS                       ( f -- )
  DUP TO ?CAPLK TO ?CAP ;

  OFF CAPS


                                 --)
```

Screen: 53

```
( Capitalization:  *CAP         )

: *CAP                  ( c -- c )
  ?CAP
  IF
   DUP 127 AND DUP
   122 <= SWAP
   97 >= AND
   IF 32 -
   ENDIF ?CAPLK TO ?CAP
  ENDIF ;
```

Screen: 54

```
( Inverse Video:  *INV  etc.    )

QUAN ?INV
VECT *INV

: INVID                    ( f -- )
  128 * TO ?INV ;

: ^INV                   ( c -- c )
  ?INV OR ;

  ASSIGN ^INV TO *INV

  OFF INVID
```

Screen: 55

Screen: 56

```
( Efficient (BUILDS...DOES)    )
( Partly after G. B. Lyons )
--> ( Pick up C, code nxt scr )
ASSEMBLER  HEX

LABEL (WIP)
  W )Y LDA,  CLC,  3 # ADC,
  IP STA,  INY,  W  )Y LDA,
  0 # ADC,  IP 1+ STA,
  DEY, RTS,

LABEL (DOES)
  IP 1+ LDA, PHA, IP LDA, PHA,
  (WIP) JSR, ' VARIABLE 4 +
  JMP,
            DCX                -->
```

Screen: 57

```
( Efficient (BUILDS...DOES)    )

ASSEMBLER HEX

LABEL ^WIP
  B1 C,  W C, 18 C, 69 C, 03 C,
  85 C, IP C, C8 C, B1 C,  W C,
  69 C, 00 C, 85 C, IP 1+ C,
  88 C, 60 C,

LABEL (DOES)
  A5 C, IP 1+ C, 48 C,
  A5 C, IP C, 48 C, 20 C,
  ^WIP , 4C C, ' VARIABLE 4 + ,

                               -->
```

Screen: 58

```
( Efficient (BUILDS...DOES)    )

: DOES)
  COMPILE (;CODE)
  4C C, (DOES) , ; IMMEDIATE

: (BUILDS
  CREATE SMUDGE ;

  DCX
```

Screen: 59

Screen: 60

```
( Txt comp:  TLABEL            )

'( TRANSIENT )(  2 KLOAD )
'( ^WIP      )( 28 KLOAD )

TRANSIENT

: TLABEL                   ( -- )
  HERE TRANSIENT
  CONSTANT PERMANENT
  [COMPILE] ASSEMBLER ;

: | [COMPILE] FORTH ; IMMEDIATE
VOCABULARY ^ IMMEDIATE

PERMANENT                    -->
```

```
Screen:  61
( Txt comp:  DMCP$                  )

HEX

TLABEL DCMP$
  A5 C, IP 1+ C, 48 C,
  A5 C, IP C, 48 C, 20 C, ^WIP ,
  CA C, CA C, 18 C, A5 C, W C,
  69 C, 02 C, 95 C, 00 C,
  A5 C, W 1+ C, 69 C, 00 C,
  95 C, 01 C, A0 C, 01 C,
  C8 C, B1 C, W C,
  10 C, FB C,
  88 C, 98 C,
  A0 C, 00 C, 4C C, PUSH0A ,
                                  --)
```

```
Screen:  62
( Txt comp:  [W=]  [P=]  [S=]  )

TLABEL (W=) ASSEMBLER
  4C C, DCMP$ ,
  ] *TYPE *SPACE ;S [

TLABEL (P=) ASSEMBLER
  4C C, DCMP$ ,
  ] *TYPE ;S [

TLABEL (S=) ASSEMBLER
  4C C, DCMP$ ,
  ] *BACKS *TYPE *SPACE ;S [

DCX
                                  --)
```

```
Screen:  63
( Txt comp:  TC=                    )

TRANSIENT

: TC=
  [COMPILE] ^ DEFINITIONS
  HERE )R TRANSIENT
  <BUILDS [COMPILE] IMMEDIATE
   LATEST C@ 31 AND )R
   LATEST 1+ I' R CMOVE
   R I' + DUP C@ 128 AND SWAP C!
   R> R> 2- , PERMANENT ALLOT
  DOES> @ STATE @
   IF , ELSE EXECUTE ENDIF ; --)
```

```
Screen:  64
( Txt comp:  W=   P=   S=          )

: W=                          ( -- )
  (W=) , CURRENT @
  TC= CURRENT ! ;

: P=                          ( -- )
  (P=) , CURRENT @
  TC= CURRENT ! ;

: S=                          ( -- )
  (S=) , CURRENT @
  TC= CURRENT ! ;

PERMANENT
```

```
Screen:  65
```

```
Screen:  66
```

Screen: 67

Screen: 68

```
( Numerics:  FMT#                        )
: FMT#                          ( f -- )
  IF
        ASSIGN *TYPE
        ASSIGN *SPACES
        ASSIGN *SPACE
  ELSE
        ASSIGN TYPE
        ASSIGN SPACES
        ASSIGN SPACE
  ENDIF
  [ ' D.   4 + ] LITERAL !
  [ ' D.R 22 + ] LITERAL !
  [ ' D.R 24 + ] LITERAL ! ;
                                    -->
```

Screen: 69

```
( Numerics:  *.   *.R                    )
: *.                            ( n -- )
  ON FMT# . OFF FMT# ;
: *.R                         ( n r -- )
  ON FMT# .R OFF FMT# ;
```

Screen: 70

```
( Typed out:  quans                    )
QUAN PRTWID    ( printer ch/ln )
80 TO PRTWID   ( init value    )
QUAN VIDWID    ( video ch/ln   )
38 TO VIDWID   ( init value    )
QUAN PCTR    ( printer line ctr )
0 TO PCTR    ( init value       )
QUAN VIDIND    ( video indent )
0 TO VIDIND    ( init value )
QUAN PRTIND    ( printer indent )
0 TO PRTIND    ( init value )
QUAN PVIND     ( indention )  -->
```

Screen: 71

```
( Typed out:  ?CR PWID ?P,VCR  )
VECT ?CR
QUAN PFLG     ( value for PFLAG )
QUAN PWID
80 TO PWID     ( adjust to suit )
: ?PCR                          ( -- )
  WWID PVIND + PWID (=
  IF CR ENDIF ;
: ?VCR                          ( -- )
  WWID PVIND +
  83 C@ 82 C@ - 1+ (
  IF CR ENDIF ;
                                   -->
```

Screen: 72

```
( Typed out:  PRT: PRINIT VID: )
: PRT:                          ( -- )
  PRTIND TO PVIND
  PRTWID TO WWID
  ASSIGN ?PCR TO ?CR
  2 TO PFLG BUFINIT ;
: PRINIT                        ( -- )
  0 TO PCTR ;
: VID:                          ( -- )
  VIDIND TO PVIND
  VIDWID TO WWID
  ASSIGN ?VCR TO ?CR
  1 TO PFLG BUFINIT ;  VID:  -->
```

Screen: 73

```
( Typed out:  *XMTLNP            )

: *XMTLNP                  ( -- )
  PFLAG @
  PFLG PFLAG !
  BUF WWID PVIND SPACES TYPE
  ?CR PFLG 2 =
  IF 1 AT PCTR +!
     PCTR 60 = ( lines/page )
   IF CR CR CR CR CR CR
       PRINIT
   ENDIF
  ENDIF
  PFLAG ! ;

                              -->
```

Screen: 74

```
( Typed out:  TYPEOUT            )

: TYPEOUT                  ( -- )
  ASSIGN *XMTLNP TO *XMTLN ;

( for buffer fmting, no windows)

  TYPEOUT








                              -->
```

Screen: 75

Screen: 76

Screen: 77

Screen: 78

```
Screen:  79

Screen:  80
 ( Windows:  quans  etc.          )
 '( )SCD )( 10 KLOAD )
 QUAN WADR ( window uplftcr adr )
 88 @ 2+ TO WADR  ( crnt. uplft )
 QUAN WHGT   ( # lines in window )
 24 TO WHGT    ( setup for 0 GR. )
 QUAN LPTR   ( wndw line pointer )
 0 TO LPTR     ( default to top  )
 QUAN B/LN          ( bytes/line )
 40 TO B/LN    ( setup for 0 GR. )
                                 --)

Screen:  81
 ( Windows:  [SCROLL]  SCROLL    )
 : (SCROLL)                ( -- )
   WWID B/C * )R   ( # to cmove )
   B/LN B/C * )R   ( # to advance)
   R WHGT 1- * WADR + WADR
   DO I J + I 4 RPICK CMOVE
   J /LOOP
   WADR WHGT 1- R) * + R)
   BKGND )SCD FILL ;
 : ?SCROLL                 ( -- )
   LPTR WHGT =
   IF (SCROLL) -1 AT LPTR +!
   ENDIF ;
                                 --)

Screen:  82
 ( Windows:  WCLR                 )
 : WCLR                   ( f -- )
   B/C WHGT * B/LN * WADR + WADR
   DO I WWID
    BKGND
    )SCD FILL
   B/LN /LOOP
   0 TO LPTR ;
                                 --)

Screen:  83
 ( Windows:  NAMWND WSTP RECWND )
 : NAMWND ( wadr wid hgt b/ch   )
   (BUILDS          ( byt/ln -- )
   , , , , , ;
 : WSTP ( wa wid hgt b/c b/l -- )
   TO B/LN TO B/C TO WHGT
   TO WWID TO WADR
   BUF WWID + 1- TO EOB
   WCLR BUFINIT ;
 : RECWND               ( system )
   )R -2 8
   DO J I + @
   -2 +LOOP R) DROP WSTP ;     --)

Screen:  84
 ( Windows:  *XMTLNW  WINDOUT    )
 : *XMTLNW                 ( -- )
   ?SCROLL
   BUF LPTR B/LN * WADR +
   WWID )BSCD 1 AT LPTR +! ;
 : WINDOUT                 ( -- )
   ASSIGN *XMTLNW TO *XMTLN ;
   WINDOUT
                                 --)
```

```
Screen:  85
```

```
Screen:  86
 ( B&W windows:  BWPRM  BWVCT    )
 '( WADR )( 40 KLOAD )

 : BWPRM  ( col row wid hght -- )
          ( wa wid hgt b/c b/l   )
   ROT 40 * 4 ROLL +
   88 @ + <ROT     ( set up wadr )
   1 40 ;          ( b/chr b/ln )

 : BWVCT                      ( -- )
   '( *TINT ASSIGN NOOP
   TO *TINT )(  )
   '( *INV  ASSIGN ^INV
   TO *INV  )(  ) ;

                                 -->
```

```
Screen:  87
 ( B&W windows:  NAMEBW  MAKEBW )

 : NAMEBW ( col row wid hght -- )
   BWPRM NAMWND
   DOES> RECWND BWVCT ;

 : MAKEBW ( col row wid hght -- )
   BWPRM WSTP BWVCT ;
```

```
Screen:  88
 ( Color windows:  CRPM  CVCT    )
 '( WADR )( 40 KLOAD )

 : CPRM    ( col row wid hght -- )
           ( wa wid hgt b/c b/l   )
   ROT 20 * 4 ROLL +
   88 @ + <ROT     ( set up wadr )
   1 20 ;          ( b/chr b/ln )

 : CVCT                       ( -- )
   '( *TINT ASSIGN ^TINT
   TO *TINT )(  )
   '( (INV  ASSIGN NOOP
   TO *INV  )(  ) ;

                                 -->
```

```
Screen:  89
 ( Color windows:  NAME,MAKECW  )

 : NAMECW ( col row wid hght -- )
   CPRM NAMWND
   DOES> RECWND CVCT ;

 : MAKECW ( col row wid hght -- )
   CPRM WSTP CVCT ;
```

```
Screen:  90
```

Screen: 91

Screen: 92

Screen: 93

Screen: 94

Screen: 95

Screen: 96

Screen: 97

Screen: 98

Screen: 99

Screen: 100

```
( Vrtxt:  VRTADJ  VRTCX          )

: VRTADJ                    ( -- )
  ?LOADING
  IN @ B/BUF >=
  IF 0 IN ! 1 BLK +!
  ENDIF ;

: VRTCX                ( -- adr )
  VRTADJ
  BLK @ BLOCK IN @ + ;




                                -->
```

Screen: 101

```
( Vrtxt:  VRTC@,!  VRTSAV,REC  )

: VRTC@                  ( -- b )
  VRTCX C@ ;

: VRTC!                  ( b -- )
  VRTCX C! UPDATE ;

QUAN OBLK QUAN OIN

: VRTSAV            ( -- blk in )
  BLK @ TO OBLK  IN @ TO OIN ;

: VRTREC            ( blk in -- )
  OIN IN ! OBLK BLK ! ;
                                -->
```

Screen: 102

```
( Vrtxt:  NXTVRT  RELVRT          )

: NXTVRT                    ( -- )
  1 IN +! VRTADJ ;

: RELVRT            ( offset -- )
  ?LOADING
  IN @ + B/BUF /MOD BLK +!
  DUP 0<
  IF B/BUF + -1 BLK +!
  ENDIF IN ! ;




                                -->
```

```
Screen: 103
( Vrtxt:  V"  XMTV  XCOUNT      )

: V"                     ( -- blk in )
  VRTADJ BLK @ IN @
  BEGIN VRTC@ 34 = NXTVRT
  UNTIL ;

: XMTV                          ( -- )
  BEGIN VRTC@ DUP 34 <>
  WHILE *EMIT NXTVRT
  REPEAT NXTVRT DROP ;

: XCOUNT     ( adr -- adr+2 cnt )
  DUP @ SWAP 2+ SWAP ;

                                  -->
```

```
Screen: 104
( Vrtxt:  M:  V:                   )

: M:                    ( blk in -- )
  <BUILDS , ,
  DOES> VRTSAV
  DUP @ IN !
  2+ @ BLK !
  XMTV *CR VRTREC ;

: V:                    ( blk in -- )
  <BUILDS , ,
  DOES>
  DUP @ IN !
  2+ @ BLK ! ;

                                  -->
```

```
Screen: 105
( Vrtxt:  EN,DECRYPT  example   )

-->

: ENCRYPT               ( c1 -- c2 )
   117 - DUP 0<
   IF 256 + ENDIF ;

: DECRYPT               ( c2 -- c1 )
   117 + DUP 255 >
   IF 256 - ENDIF ;

                                  -->
```

```
Screen: 106
( Vrtxt:  V$TP  V$@  V$*EMT     )

: V$TP                ( -- XCOUNT )
  VRTC@ ( DECRYPT ) NXTVRT VRTC@
  ( DECRYPT ) NXTVRT 256 * + ;

: V$@               ( -- X$ [=PAD] )
  PAD 2+ V$TP DUP PAD ! 0
  DO VRTC@ OVER C! 1+ NXTVRT
  LOOP DROP PAD ;

: V$*EMT                        ( -- )
  V$TP 0
  DO VRTC@ ( DECRYPT )
    *EMIT NXTVRT
  LOOP ;                          -->
```

```
Screen: 107
( Vrtxt:  V$!                      )

: V$!                       ( X$ -- )
  DUP @ 2+ 0
  DO DUP C@ ( ENCRYPT )
     VRTC! 1+ NXTVRT
  LOOP DROP ;

                                  -->
```

```
Screen: 108
( Vrtxt:  X"                       )

: X"               ( -- X$ [=PAD] )
  0 PAD ! PAD 2+
  BEGIN VRTC@ DUP 34 <>
  WHILE OVER C! 1+
        1 PAD +! NXTVRT
  REPEAT NXTVRT 2DROP PAD ;

                                  -->
```

Screen: 109

```
( Vrtxt:                         )
                                 -->
```

Screen: 110

```
( Vrtxt :  ALTSAV,REC           )

QUAN ALTBLK  QUAN ALTIN

: ALTSAV                   ( -- )
   BLK @ TO ALTBLK
   IN @ TO ALTIN ;

: ALTREC                   ( -- )
   ALTBLK BLK !
   ALTIN  IN  ! ;




                                 -->
```

Screen: 111

```
( Vrtxt:  ALTINIT SCRDSK DSTDSK)

: ALTINIT          ( screen -- )
   B/SCR * TO ALTBLK
   0 TO ALTIN ;


: SRCDSK                   ( -- )
   CR ." Insert source disk and p
ress START." WAIT CR ;

: DSTDSK                   ( -- )
   CR ." Insert dest. disk and pr
ess START." WAIT CR ;

                                 -->
```

Screen: 112

```
( Vrtxt:  ALT$!  MSG:            )

: ALT$!                  ( X$ -- )
   VRTSAV ALTREC V$!
   ALTSAV VRTREC ;

: MSG:                   ( X$ -- )
   ( $ENCRYPT )
   <BUILDS ( DR1 or ) DSTDSK
    ALTBLK , ALTIN , ALT$!
    FLUSH ( DR0 or ) SRCDSK
   DOES> VRTSAV
    DUP @ SWAP 2+ @ IN ! BLK !
    V$*EMT ( or )
    ( V$@ $DECRYPT XCOUNT *TYPE )
    VRTREC ;
```

Screen: 113

Screen: 114

Screen: 115

Screen: 116
```
( For demos:  UMOVE  $!          )

'( $! ;S )(  )

: UMOVE                ( a a n -- )
  <ROT OVER OVER U<
  IF
    ROT <CMOVE
  ELSE
    ROT CMOVE
  ENDIF ;

: $!
  OVER C@ 1+ UMOVE ;
                                -->
```

Screen: 117
```
( For demos:  ["]  "             )

: (")                     ( -- $ )
  R DUP C@ 1+ R> + >R ;

: "
  34 ( Ascii quote )
  STATE @
  IF                  ( cccc" -- )
    COMPILE (") WORD
    HERE C@ 1+ ALLOT
  ELSE
    WORD HERE       ( cccc" -- $ )
    PAD $! PAD
  ENDIF ;
  IMMEDIATE
```

Screen: 118

Screen: 119

Screen: 120
```
( X" ... " demo                  )

X" When you are going to take in
 hand any act, remind yourself w
hat kind of an act it is.  If yo
u are going to bathe, place befo
re yourself what happens in the
bath:  some splashing the water,
 others pushing against one anot
her, others abusing one another,
 and some stealing:  and thus wi
th more safety you will undertak
e the matter, if you say to your
self, I now intend to bathe, and
 to maintain my will in a manner
 comformable to nature.  And so
```

Screen: 121

```
you will do in every act:  for t
hus if any hindrance to bathing
shall happen, let this thought b
e ready:  it was not this only t
hat I intended, but I intended a
lso to maintain my will in a way
 conformable to nature; but I sh
al not maintain it so, if I am v
exed at what happens.→→Epictetus
, translated by George Long, 187
7.→"
CR
." The X-quote string is loaded"
CR
```

Screen: 122

```
( V" ... " M: message-name demo)
V" There is an inconvenience whi
ch attends all abstruse reasonin
g, that it may silence, without
convincing an antagonist, and re
quires the same intense study to
 make us sensible of its force,
that was at first requisite for
its invention.  When we leave ou
r closet, and engage in the comm
on affairs of life, its conclusi
ons seem to vanish, like the pha
ntoms of the night on the appear
ance of the morning; and 'tis di
fficult for us to retain even th
```

Screen: 123

```
at conviction, which we had atta
in'd with difficulty.  This is s
till more conspicuous in a long
chain of reasoning, where we mus
t preserve to the end the eviden
ce of the first propositions, an
d where we often lose sight of a
ll the most receiv'd maxims, eit
her of philosophy or common life
.  I am not, however, without  h
opes...→→David Hume, 1793.→"
   M: MSGDEM1
CR
." MSGDEM1 now exists."
CR
```

Screen: 124

```
( X" ... " MSG: msg-name demo   )
80 ALTINIT
                              X"
'Accessory No. 5 is a pocket com
pass and is used in connections
with putting.  Like suppose for
inst. you land on the green abou
t 10 ft. from the cup, why the n
ext thing is to find out what di
rection the hole is at and this
can't be done and done right wit
hout a compass.→→ At lease I hav
e seen a whole lot of golfers tr
y and putt without no compass, a
```

Screen: 125

```
nd their ball has went from 10 t
o 45 ft. degrees to the right or
 left of where the hole is actua
lly located.  This is because th
ey was just guessing where as wi
th a compass they's no guess wor
k about it.  If you miss a putt
with a compass to tell you just
where a hole is at, why it's bec
ause you can't putt so good.'→→R
ing Lardner on New Golf Accesori
es, 1924.→"
  MSG:  MSGDEM2
CR ." MSGDEM2 now exists." CR
```

Screen: 126

```
( More MSG:'s                   )
  X" The rat the cat I bought ca
ught escaped.→" MSG: M0
X" There are gold coins here!→"
MSG: M1
     X" Aww, gee, Beave!→"
     MSG: M2
X" You see, Watson, but you do n
ot observe.→" MSG: M3
X" Never look back; something ma
y be gaining on you.→"        -->
```

Screen: 127

```
( More MSG:'s                      )

MSG: M4

X" 'The precise date at which th
e reversion to cap and gown took
 place, as well as the fact that
 it affected so large a number o
f schools at about the same time
, seems to have been due in some
 measure to a wave of atavistic
sense of comformity and reputabi
lity that passed over the commun
ity at that period.'→→Thorstein
Veblen, 1899.→" MSG: M5

```

Screen: 128

Screen: 129

Screen: 130

Screen: 131

Screen: 132

Screen 133 through 161 are blank.

Screen: 162

Screen: 163

Screen: 164

Screen: 165

Screen: 166

```
( Load Chain                          )

    '( QUAN  )( 10 LOAD )
    16 LOAD ( utilities )

 ( Do not modify these lines )









                                    --)
```

Screen: 167

```
( Load Chain, options screen    )
28     LOAD ( af, do not modify )
(   34 LOAD ( rgt & ctr justify )
(   38 LOAD       ( fill justify )
(   50 LOAD            ( coloring )
(   52 LOAD     ( capitalization )
(   54 LOAD      ( inverse video )
42     LOAD ( af, do not modify )
(   60 LOAD  ( text compression )
(   68 LOAD ( fmt'd num. output )
(   70 LOAD       ( typed output )
(   86 LOAD ( B&W window output )
(   88 LOAD ( Color wndw output )
  ( select )= 1 of above 3 )
( 100 LOAD ( Virtual mem. text )
( 116 LOAD ( " for demos )
```

Screen: 168

Screen: 169

Screen: 170

```
CONTENTS OF THIS DISK:

LOAD-CHAIN                   166 LOAD

EFFICIENT (BUILDS DOES)
(ALSO LOADED BY TXTCMP)       56 LOAD
QUAN STRUCTURES               10 LOAD
TRANSIENT STRUCTURES           4 LOAD
```

Screen: 171

Screen: 172

Screen: 173

```
Screen: 174
```

```
Screen: 175
```

```
Screen: 176
( Error messages                    )

Stack empty

Dictionary full

Wrong addressing mode

Is not unique

Value error

Disk address error

Stack full
```

```
Screen: 177
Disk Error!

Dictionary too big
```

```
Screen: 178
( Error messages                    )

Use only in Definitions

Execution only

Conditionals not paired

Definition not finished

In protected dictionary

Use only when loading

Off current screen
```

```
Screen: 179
Declare VOCABULARY
```