VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713

valFORTH T.M.

SOFTWARE SYSTEM
for ATARI*

valFORTH 1.1

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

© Copyright Valpar International 1982

valFORTH™
SOFTWARE SYSTEM

Stephen Maguire
Evan Rosen

(Atari interfaces based on work by Patrick Mullarky)

Software and Documentation
©Copyright 1982
Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

Table of Contents

STROLLING THROUGH valFORTH 1.1

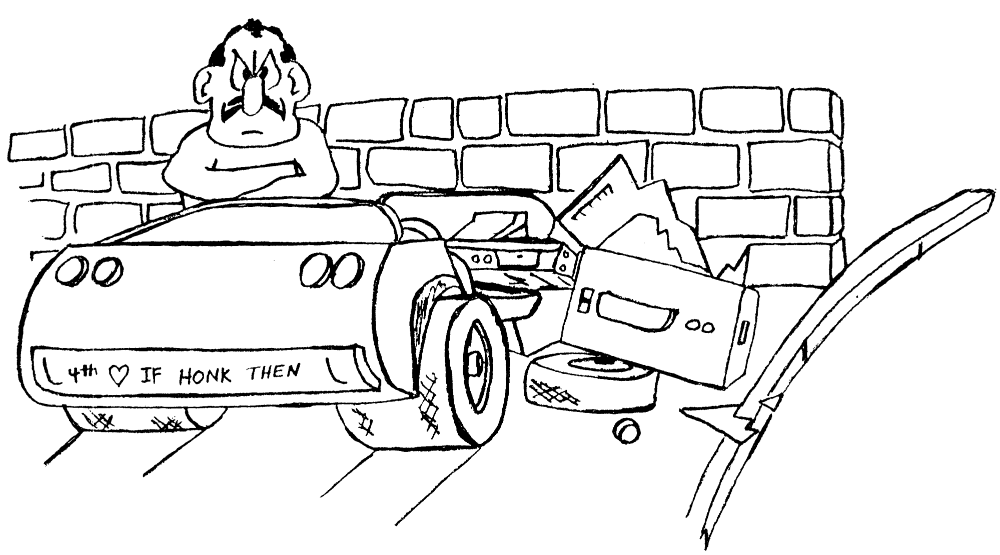
Welcome. For this excursion you'll need an ATARI 800 (or 400) with at least 24K, a disk drive, monitor, a printer, and valFORTH 1.1. You could even do without the printer. Please get everything up and running, and boot valFORTH.

(To boot the disk, turn the drive(s) on and the computer off. Insert the disk in drive 1 and turn the computer on. The disk should now be booting, and the monitor speaker should be going beep-beep-beep-beep as valFORTH loads.)

ERRORS, RECOVERIES, CRASHES

Before we get started, let's mention the inevitable: Most of the time when you make an error you'll receive one of the fairly lucid fig-Forth error messages. If you just get a number, this will probably refer to the Atari error message list which you can find in the documentation that came with your computer. Since the Atari is a rather complex beast, you may sometimes get into a tangle that looks worse than it is. Keep your head. If you have party-color trash on the screen, for instance, and yet you can still hear the "peek-peek-peek" of the key when you hit return, you may have merely blown the display list without hurting your system. Try Shift-Clear followed by 0 GR. . Very often you're home again. If this doesn't work, try a warm start: Hold down a CONSOLE button, say START, and while you've got it down, press SYSTEM RESET and hold both for a moment until the "valFORTH" title comes up. (If you were to push the SYSTEM RESET button alone, you'd get a cold start, which takes you back to just the protected dictionary.) A warm start gets you back to the "ok" prompt without forgetting your dictionary additions. If warm start doesn't work, your system is being kept alive only by those wires connected to it; it no longer has a life of its own. The standard procedure now is to push SYSTEM RESET alone a few times (cold start) in a superstitious manner, and then reboot the system.

Look carefully at the code that blew the system last time. If you're really having trouble debugging, sprinkle a bunch of WAIT's and/or .S's (Stack Printouts) through the code, and go through again. The best thing about those first few long debugging sessions in any computer language is that they teach you the value of writing code carefully.



FORMATTING AND COPYING DISKS

You may have noticed that your system came up in a green screen. In a little while you'll be able to change it to anything you like. We'll get to that in a moment, but right now type 170 LIST (and then hit RETURN.) Behold the table of contents. Our first priority should be to make a working disk by copying the original.

Let's assume that you have a blank, unformatted disk on which to make your copy. Notice the line called FORMATTER on screen 170. At the right side of this line is probably 92 LOAD, though the number may be different in later releases. Type 92 LOAD (or whatever the number is) and wait until the machine comes back with "ok". Now you're going to type FORMAT, but for safety's sake why not remove the valFORTH disk and insert the blank disk? One never knows if newly purchased software will give you warnings before taking action. ("Warnings" or "Prompts" make a system more friendly.) Ok, now type FORMAT. For the drive number you probably want to hit "1", unless you've got more than one drive and don't want to format on the lowest. In answer to the next prompt, hit RETURN unless you've changed your mind. Now wait while the machine does the job. If you get back "Format OK" you're in business. (If "Format Error" comes back, suspect a bad blank disk or drive.) You might as well format another disk at this time on which to store your programs.

Now to make the copy. Return the valFORTH disk to the drive and do 170 LIST again. Find DISK COPIERS and do 72 LOAD, or whatever number is indicated. When the "ok" prompt comes back, two different disk copying routines are loaded: DISKCOPY1 for single drive systems and DISKCOPY2 for multiple drive systems. Type whichever of these words is appropriate and follow the instructions. ("source" means the disk you want to copy. "dest." is the blank "destination" disk.) There are 720 sectors that have to be copied. Since this can't be done in one pass, if you are using DISKCOPY1 you will have to swap the disks back and forth until you're done. (The computer will tell you when.) The less memory you have, the more passes; there is great benefit in having 48K. If you have more than one drive, it still takes several internal passes, but there is no swapping required. Either way, the process takes several minutes with standard Atari disk drives.

Nice going. Now store the original disk in some safe place. Don't write protect your copy yet. First we'll adjust the screen color to your taste. Just to see if you really have a good copy, boot it. This can be done by the usual on-off method, or by typing BOOT.

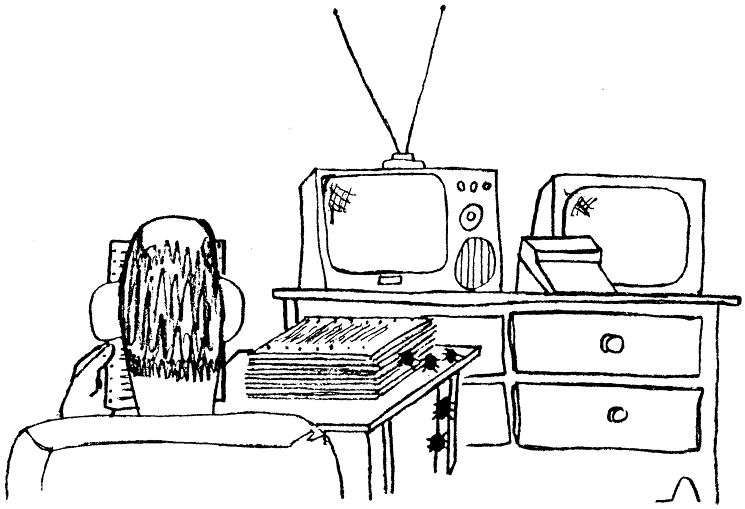
COLORS

Before playing with the colors, let's look at something else. Type VLIST, and watch the words go by. These are all of the commands that are currently in the "dictionary" in memory in your system. You can cause this listing, or any other, to pause by hitting CTRL and 1 at the same time. This is a handy feature of the Atari. The listing is restarted with the next CTRL 1. Additionally, in valFORTH most listings may be aborted by pressing any of the three yellow buttons START, SELECT, and OPTION. These three buttons together will be referred to as the CONSOLE.

Do VLIST again, and abort it with a CONSOLE press after a few lines. At the top of the list you should see the word TASK. Remember that for a moment. Do 170 LIST again. Look over the list and find COLOR COMMANDS, and LOAD as appropriate. Now do VLIST again, and stop the list when it takes up about half of the screen. Above TASK you now can see a number of new commands, or "words" as they are commonly called in Forth. These were added to the dictionary by the LOAD command. Here's what some of these words do:

Type BLUE 12 BOOTCOLOR, and you get a new display color. Try BLUE 2 BOOTCOLOR. If you try this action with the number as 4, the letters will disappear, and you'll have to type carefully to get them back. The number is the luminance or "lum" and is an even number in the range 0 to 14. The color-name is called the "hue." The word "color" will be used to refer to a particular combination of hue and lum; hence PINK 6 is a color, PINK is a hue. There are 16 hues available and you can read their names from the display, starting with LTORNG ("light orange") and ending with GREY. (The hues may not match their names on your monitor. Later on you'll be able to change the names to your liking, or eliminate them altogether to save memory, and just use numbers. For instance, PINK is equal to 4.)

Try out different colors using BOOTCOLOR, until you find one you can live with for a while. We usually use GREEN 10 or GREEN 12 in-house at Valpar. While you are doing this you'll probably make at least one mistake, and the machine will reply with an error message like "stack empty." Just hit return to get the "ok" back and start whatever you were doing again. Actually, you don't even have to get the "ok" back, but it's reassuring to see it there. When you've got a color you like, do VLIST again. Note the first word above TASK. It should be GREY. Carefully type FORGET GREY, and do VLIST again. Notice that GREY and all words above it are indeed forgotten. That's just what we want. Now type SAVE. You'll get a (Y/N) prompt back to give you a chance to change your mind, since SAVE involves a significant amount of writing to drive #1. For practice, check to see that you still have the copy in drive one, and if it is there, hit Y, and off we go. When "ok" comes back, remove the disk and apply a write protect tab to it. Boot this disk again to see that it will come up in your selected color.



DEBUGGING

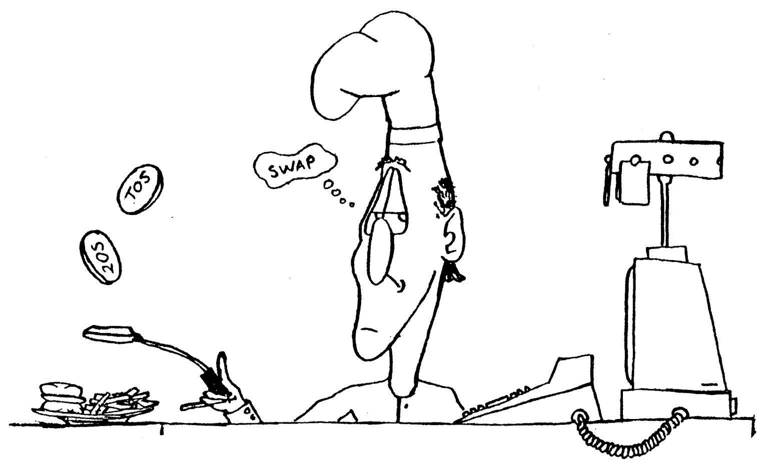
Look at 170 again. Load the DEBUGGING AIDS, and type ON STACK. You'll see that the stack is empty. Great, what's a stack? The best answer to this is to suggest that you read Leo Brodie's book, Starting FORTH. This amusing and thorough treatment of FORTH starts from the novice level and continues on to most of the advanced concepts in FORTH. Starting FORTH is available from many sources, including Valpar International. Included with your valFORTH package is a document called "Notes on Starting FORTH for the fig-FORTH User" which pinpoints differences between Brodie's dialect of FORTH, called 79 Standard, and the somewhat more common fig-FORTH on which valFORTH was based.
It is not feasible to present a course on FORTH in these pages, since FORTH is far more powerful than BASIC, which itself requires a fair amount of space to present. However, we'll try to be as considerate as possible to the FORTH-innocent.

The visible stack is a very good debugging and practice tool. Type a few integers, say 5 324 -19 0 and hit RETURN. The numbers are now visible on the stack. Top of stack, TOS, is at right. Print the top entry by typing " . " and then do DUP. Note that there are now two -19's on top. Do " * " to multiply them together. Now do DROP to discard the product, 361, currently at TOS. Ok, now do SWAP to exchange the 324 and 5 and then do " / " to divide 324 by 5. This should leave 64, since the answer is truncated. Now type 1000 * and notice that instead of getting 64000 you get -1536. This is of course two's complement on a two-byte number. Type U.S which will switch the visible stack to unsigned representation. Type .S to go back. The words .S and U.S may be used with the visible stack on or off to show the stack one time. Now type OFF STACK. Type in a few more numbers, say 1 2 3 4 5 6 7. ON STACK again, and observe that the entries are retained. Do OVER to bring a copy of the 6 over the 7. Now DROP it. Do ROT to rotate the third from top, 5, to the top. Now do <ROT to put it back. In addition to all of these normal routines, valFORTH supports PICK and ROLL both coded in 6502 for speed. Notice that the 5th on stack is a 3. 5 PICK will bring a copy of it to TOS. Do this and then DROP the 3. Do 5 ROLL to pull the 3 out of the stack and place it at TOS. DO SP! to clear the stack.

One point about number bases: Right now you're in DECIMAL. By typing HEX you go into hexadecimal, and typing DECIMAL or its abbreviation DCX you get back. And, as usual, virtually any base may be used by typing N BASE ! where N is the base you want. Thus, 2 BASE ! gives binary, etc. Some errors, particularly during loading, may leave you in an unexpected base, like base 0, for instance. If you find the machine acting normally except for numbers, this may have happened. A simple DCX will get you back to decimal. The word B? will print the current base in decimal. Put 30 on the stack and then do HEX. Now do B?. Do DCX to return to decimal.

While we're on the subject of numbers, do ON and note that it is just a CONSTANT equal to 1. Similarly, do OFF and see that it is zero. Try 0 STACK and then 1 STACK. The words ON and OFF are provided to enhance readability of code, but could be substituted by 1 and 0 if desired. The two representations are equally fast.

We mention to the newcomer to FORTH that the stack takes the place of dummy variables or dummy parameters in other languages. This reduces memory overhead in several ways but does exact a penalty of reduced readability of FORTH source code. Consistent and sensible source code formatting can significantly enhance readability. The source code on the present disk may be used as a reasonably good example of well-arranged code.



Now a few words about DECOMP. Clear the stack. Type in 3 and 4. Do OVER OVER followed by 2DUP and notice that these two phrases have the same effect. Clear the stack and then turn it off if you like, and do DECOMP 2DUP. What you see is a decompilation of 2DUP which indicates that it is indeed defined as OVER OVER. Decomp OVER. The word "primitive" in the decompilation of OVER indicates that OVER is defined in machine code.

Decomp LITERAL. The word (IMMEDIATE) after LITERAL in the decompilation indicates that LITERAL is immediate. Not all words can be decompiled by DECOMP, and sometimes trash will be printed with long pauses between lines. In this case, hold down any CONSOLE button (the three yellow ones, remember) until the "ok" comes back. This may take several seconds, but rarely much longer.

PRINTING

If you have a printer attached, we can generate some hardcopy. Look at screen 170 again. You can see the line labeled PRINTER UTILITIES. Don't load it, though. The printer utilities were loaded automatically when you loaded the debugging aids, and so are in the dictionary already. (There is no need to have them in twice, though it wouldn't hurt.) You have access to the words P:, S:, LISTS, PLISTS, PLIST, and a couple of others relating to output. Do VLIST and see if you can spot this group. As a matter of fact, do ON P: VLIST OFF P: all in one shot. ON P: is used to route output to the printer or not. OFF P: stops sending to the printer. Try ON P: OFF S: 170 LIST CR OFF P: ON S: and notice that this time text is not sent to the display screen, only to the printer. That's because of OFF S: .

Look at screen 170 again, either on display or in hardcopy, and note which screen the printer utilities start. Type this number in, but don't type load. Instead, after the number, type 10 PLISTS. This prints 10 screens starting from the first screen you just typed in. If you have a reasonably smart printer, it will automatically paginate, so that the screens are printed three to a page. If the printer acts peculiarly after printing each third screen, the pagination code in the word EJECT is probably not right for your printer. You'll be able to change this later on.

Now type 30 150 LISTS and after a few blank screens you'll see the entire disk go by, except for the boot code. You can pause any time by CTRL 1 or stop by holding a CONSOLE button.

Finally, do ON P: 30 179 INDEX OFF P: to print a disk index. The index is made up of the first line of each screen.

EDITING

Two editors have been included in this package. The fig (Forth Interest Group) Editor and the valFORTH 1.0 Editor. The latter, while a perfectly useable video-display editor in its own right, is actually a stripped-down version of the valFORTH 1.1 Editor, available with the Utilities/Editor package from Valpar International. The 1.0 Editor is provided to give the user some idea of what the very powerful 1.1 Editor is like, without actually providing it. (Among other things, the 1.1 Editor has a user-definable line buffer of up to 320 lines with a 5 line visible window at the bottom of the display. This window can be seen at the bottom of the 1.0 Editor, but is inactive.)

The fig Editor is a general-purpose FORTH line editor, and was the FORTH editing workhorse until good video-displays were developed.

The fig Editor User Manual is located just after this section. It is based on that by Bill Stoddart of FIG, United Kingdom, published in the fig-Forth installation manual 10/80, and is provided through the courtesy of the FORTH INTEREST GROUP, P.O. Box 1105, San Carlos, CA 94070. Serious Forth programmers should write FIG to request their catalog sheet of references and publications.

Let's look at the valFORTH editor 1.0. Refer to the directory again, screen 170, and load the valFORTH editor. (Don't load the fig Editor by mistake.) Before proceeding, make sure that the write-protect tab on your disk is secure. The word to enter the editor at the screen on top of stack is V. You can remember it by thinking of it as "view." Type 170 V. Screen 170 is now on the display again, but in the valFORTH 1.0 Editor rather than as a listing. This Editor is a subset of the valFORTH 1.1 Editor available in the Editor/Utilities package, which is MUCH more powerful and convenient, and is priced far lower than any comparable product of which we are aware. The Editor Command card provided shows all of the commands available with the 1.1 Editor. Commands available with the 1.0 Editor are marked with asterisks (*) on the card. Let's run through them:

The cursor can be moved as in the Atari "MEMO PAD" mode. That is, hold down the control key (CTRL) and move the cursor around the display with the four arrow keys. To enter text (replace mode only in 1.0), position the cursor and type it in. Delete characters with the backspace key as usual. The cursor will wrap to the next line at the end of a line, and to the top of the screen when it goes off the bottom. You can type at will on this screen since we won't save the changes to disk.

Do a Shift-Insert and notice that a blank line is inserted at the cursor line. The bottom line is lost, though it is recoverable in the 1.1 version. Now do Shift Delete to remove a line. (Delete is on the Backspace key). These are all of the Editing commands available in the 1.0 Editor. There are two methods of exiting the editor, CTRL S and CTRL Q. CTRL S marks the screen for saving to disk, and CTRL Q forgets the latest set of editing changes. As usual, changes are not saved immediately. This is accomplished with the word FLUSH or by bringing other screens into the buffers and pushing the edited ones out. Again, as usual, the EMPTY-BUFFERS command, or its valFORTH abbreviation, MTB, will clear all buffers, thus forgetting any changes that have not yet been written to disk.

Try CTRL Q to exit now. Reedit the screen by typing L. L does not require an argument on stack and will bring the last-editing screen into the editor. The words CLEAR and COPY have their normal meanings, as does WHERE, which has had the standard fig bug fixed. See the glossary for details. Note that since COPY in valFORTH does not FLUSH its changes, careful use allows transfers of single screens between disks by swapping disks after COPY and before FLUSH. This is particularly handy, for example, for transferring error message screens 176-179 between disks.

You can make this transfer by doing

```
176 176 COPY 177 177 COPY 178 178 COPY 179 179 COPY
```

and then swapping in the destination disk and typing FLUSH. You may want to define a word to do this automatically:

```
: ERRXFR                    ( -- )
  CR   ."  Insert source and press START" WAIT
  180 176
  DO I I COPY
  LOOP
  CR ." Insert dest. and press START" WAIT
  FLUSH ;
```

Because there are four 512 character screen buffers in memory in valFORTH, four 512 characters screens at a time is the maximum for this method. Bulk screen moves on a single disk or between disks are available with the Utilities/Editor Package.

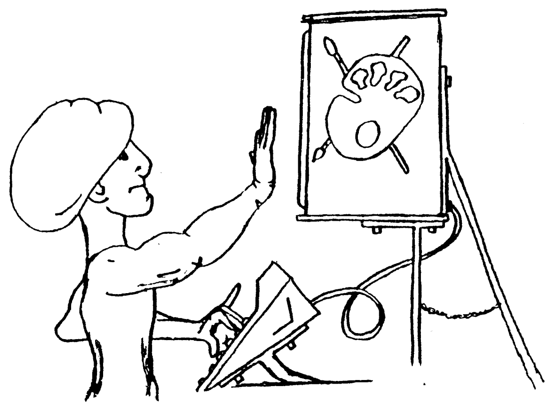
Note: The word "screen" in Forth refers to an area of the disk. When you do 170 LIST you are listing screen 170. In valFORTH there are 180 screens, numbered 0-179, on the disk in drive 1. In multiple-drive systems screen numbers continue across drives, so that screens 180-349 are on drive 2. 180 LIST will automatically read from drive 2. For technical reasons screen 0 should not be used for program code.

Whichever editor you use for the moment, you can write your programs to a blank disk and load them from there. Remember that in fig-FORTH (and so also in valFORTH), if you wish to continue loading from one screen to the next, all but the last screen should end in -->. You'll see this all through the valFORTH 1.1 code. You'll also see ==>. For present purposes you can use --> everywhere, and forget about ==>. ==> is actually a "smart" version of --> that does nothing if the system uses 1024 character screens instead of 512.

If you are a FORTHER, and wish to use 1024 byte screens, do FULLK. To return to 512 character screens, do HALFK. (A working disk may be SAVE'd in either condition.) Note that the valFORTH 1.0 Editor will not edit 1024 character screens, though the 1.1 version will, and includes special 1K notation. In the same vein, the word KLOAD that appears in the source code is a smart load. See the Glossary for details.

To terminate loading one simply omits the --> on the last screen. ;S may be used to end loading at any point. Also note that valFORTH --> and ==> are smart in the sense that if you wish to stop loading before the machine is ready to stop, simply hold down a CONSOLE button. When --> or ==> execute, they first check the CONSOLE. If a button is pressed, they stop loading instead of continuing with the next screen.

Before leaving editing practice, type MTB to empty the disk buffers and assure yourself that nothing will be flushed to disk accidentally as you read in new screens. Or else, do FLUSH if you really want to save your changes. (Remember to remove the write-protect tab if you do.)



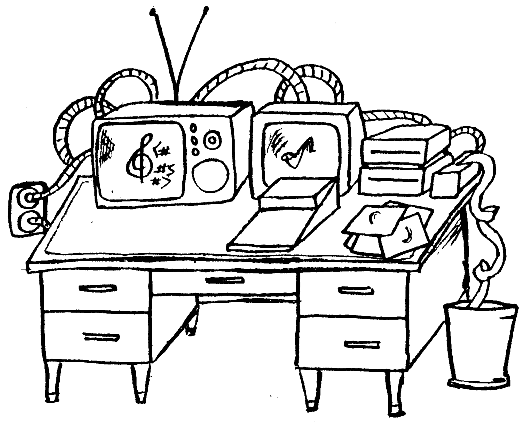
GRAPHICS

On to Graphics. Check screen 170 and load the Color Commands again, and then the Graphics Package. VLIST to see what you've got, and print the list if you like. You may notice that GR. is not among these freshly loaded words: It is in the kernel, that is, the booted code. Try the following sequence:

```
2 GR. (BASIC Graphics mode 2)
5 5 POS. (Move the graphics cursor)
G" TEST" (Send text to graphics area)
1  COLOR (Pick a new graphics color)
G" TEST" (More text)
: SMPL  4 0 DO I  COLOR G" TEST" LOOP ; (automate)
SMPL SMPL SMPL (Try it out)
: MANY BEGIN SMPL ?TERMINAL UNTIL ; (More automation)
1 GR. (Go somewhere else)
MANY (Press CONSOLE button to exit)
17 GR. MANY (Try it in full screen)
2 GR. 2 PINK 8 SE. MANY (Use SE. to change color 2)
4 GOLD 8 SE. (Use SE. to change background color).
0 GR. (Go back to normal text screen.)
```

You can also see a quick demonstration by loading the Graphics Demo Program listed on screen 170. If it's not listed on screen 170, do an INDEX in the area of the Graphics routine screens you loaded recently. When you find the Graphics examples screen, load it. Then do FBOX. Take a look at the code and then at the Glossary to get the idea.

As in Atari Basic, adding 16 to the graphics mode you want to enter gives non-split screen, and adding 32 suppresses erase-on-setup of the mode.



SOUNDS

As a final stop on this tour, load the SOUNDS words. The word SOUND acts similarly to the Basic command SOUND. In valFORTH it also has the abbreviation SO. and expects stack arguments like so:

channel(0-3) frequency#(0-255) distortion(0-14 evens) volume (0-15).
(We use "CatFish Don't Vote" as a mnemonic).

Try, for instance 0 200 12 8 SO. and then turn it off with 0 XSND which just shuts off the indicated voice, 0, or XSND4 which quiets everything. More about sound generation by the Atari may be found in the "sound" section.

Logical Line Input

One of the nice features of the Atari OS is that it lets you back the cursor over code that you've typed in already, even edit it with various inserts, deletes, and retypes, and then hit return to have it reinterpreted. This function is supported by valFORTH, and you can re-input up to two full lines of text, (and a wee bit more) at a time just by moving the cursor onto the "logical line" you wish to re-read. Try it.

THE GREAT SCREEN SIZE DEBATE

The "standard" Forth screen is composed of 1024 bytes. This is a nice round number, and on a good text display one can have room for that many characters plus a few more. However, beyond tradition, there is very little functional reason to have 1024 byte screens over several other power-of-2 sizes. In the case of Atari and Apple machines, 512 byte screens make video display editors much easier to work with, since one can get a whole screen in the display at once. valFORTH supports both 1024 and 512 byte screen modes, but in-house at Valpar we strongly prefer 512 byte screens and recommend that you adopt this as your personal standard. If at any time you wish to change to 1K to help compile software written on 1K screens, you can do so with one word, FULLK.

SAVING YOUR FAVORITE SYSTEM(S)

Well, you've seen many of the bells and whistles of valFORTH. When you are using the language for software development you will probably have a favorite set of capabilities that you always want aboard. Rather than loading them from scratch each time, why not SAVE them to a formatted disk? Just get everything you want into the dictionary. After it's all loaded, put a formatted disk into drive 1 and type SAVE. Answer the prompt by pressing "Y" unless you have changed your mind, and the computer will save a bootable copy of your system dictionary on the blank disk.

DISTRIBUTING YOUR PROGRAMS

If you have a program you wish to distribute, there are two ways in which to proceed:

(1) Make a PROTECTED auto-booting copy of your software by using the word AUTO as detailed in the "compiling Auto-Booting Software" section of this manual.

(2) Make a TARGET-COMPILED version of your software, using the valFORTH Target Compiler, scheduled for release approximately 9/82. Target Compilers allow production of much smaller final FORTH products by allowing elimination of unnecessary code, e.g., headers, compiler, buffers, etc.

In addition to the above procedures, Valpar International also requires that the message:

Created in whole or part using valFORTH products of
Valpar International, Tucson, AZ 85713, USA
Based on fig-FORTH, provided through the courtesy of
Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070

Hope you've enjoyed the tour. Bye now.

FIG EDITOR USER MANUAL

Based on the Manual
by Bill Stoddart
of FIG, United Kingdom

valFORTH organizes its mass storage into "screens" of 512 characters, with the option of 1024. If, for example, a diskette of 90K byte capacity is used entirely for storing text, it will appear to the user as 180 screens numbered 0 to 179. Screen 0 should not be used for program code. Each screen is organized as 16 lines with 32 characters per line.

Selecting a Screen and Input of Text

To start an editing session the user types EDITOR to invoke the appropriate vocabulary.

The screen to be edited is then selected, using either:

```
n LIST  ( list screen n and select it for editing ) OR
n CLEAR ( clear screen n and select for editing )
```

To input new text to screen n after LIST or CLEAR the P (put) command is used.

Example:

```
0 P THIS IS HOW
1 P TO INPUT TEXT
2 P TO LINES 0, 1, AND 2 OF THE SELECTED SCREEN.
```

Based on material provided through the courtesy of the FORTH INTEREST GROUP, P.O. Box 1105, San Carlos, CA 94070.

Line Editing

During this description of the editor, reference is made to PAD. This is a text buffer which may hold a line of text used by or saved with a line editing command, or a text string to be found or deleted by a string editing command.

PAD can be used to transfer a line from one screen to another, as well as to perform edit operations within a single screen.

Line Editor Commands

n H — Hold line n at PAD (used by system more often than by user).

N D — Delete line n but hold it in PAD. Line 15 becomes blank as lines n+1 to 15 move up 1 line.

n T — Type line n and save it in PAD.

n R — Replace line n with the text in PAD.

n I — Insert the text from PAD at line n, moving the old line n and following lines down. Line 15 is lost.

n E — Erase line n with blanks.

n S — Spread at line n. n and subsequent lines move down 1 line. Line n becomes blank. Line 15 is lost.

Cursor Control and String Editing

The screen of text being edited resides in a buffer area of storage. The editing cursor is a variable holding an offset into this buffer area. Commands are provided for the user to position the cursor, either directly or by searching for a string of buffer text, and to insert or delete text at the cursor position.

Commands to Position the Cursor

TOP — Position the cursor at the start of the screen.

N M — Move the cursor by a signed amount n and print the cursor line. The position of the cursor on its line is shown by a ● (solid circle).

String Editing Commands

F text — Search forward from the current cursor position until string "text" is found. The cursor is left at the end of the text string, and the cursor line is printed. If the string is not found an error message is given and the cursor is repositioned at the top of screen.

B — Used after F to back up the cursor by the length of the most recent text.

N — Find the next occurrence of the string found by an F command.

X text — Find and delete the string "text."

C text — Copy in text to the cursor line at the cursor position.

TILL text — Delete on the cursor line from the cursor till the end of the text string "text."

NOTE: — Typing C with <u>no text</u> will copy a null (represented by a heart) into the text at the cursor position. This will abruptly stop later compiling! To delete this error type TOP X 'return'.

Screen Editing Commands

n LIST — List screen n and select it for editing

n CLEAR — Clear screen n with blanks and select it for editing

n1 n2 COPY — Copy screen n1 to screen n2.

L — List the current screen. The cursor line is relisted after the screen listing, to show the cursor position.

FLUSH — Used at the end of an editing session to ensure that all entries and updates of text have been transferred to disc.

Editor Glossary

TEXT c ---
Accept following text to pad. c is text delimiter.

LINE n --- addr
Leave address of line n of current screen. This address will be in the disc buffer area.

WHERE n1 n2 ---
n2 is the block no., n1 is offset into block. If an error is found in the source when loading from disc, the recovery routine ERROR leaves these values on the stack to help the user locate the error. WHERE uses these to print the screen and line nos. and a picture of where the error occurred.

R# --- addr
A user variable which contains the offset of the editing cursor from the start of the screen.

#LOCATE --- n1 n2
From the cursor position determine the line-no n2 and the offset into the line n1.

#LEAD --- line-address offset-to-cursor

#LAG --- cursor-address count-after-cursor-till-EOL

-MOVE addr line-no ---
Move a line of text from addr to line of current screen.

H n ---
Hold numbered line at PAD.

E n ---
Erase line n with blanks.

S n ---
Spread. Lines n and following move down. n becomes blank.

D n ---
Delete line n, but hold in pad.

M n ---
Move cursor by a signed amount and print its line.

T n ---
Type line n and save in PAD.

L ---
List the current screen.

R n ---
Replace line n with the text in PAD.

n ---
Put the following text on line n.

I n ---
Spread at line n and insert text from PAD.

TOP ---
Position editing cursor at top of screen.

CLEAR n ---
Clear screen n, can be used to select screen n for editing.

FLUSH ---
Write all updated buffers to disc.

COPY n1 n2 ---
Copy screen n1 to screen n2.

-TEXT Addr 1 count Addr 2 -- boolean
True if strings exactly match.

MATCH cursor-addr bytes-left-till-EOL str-addr str-count
--- tf cursor-advance-till-end-of-matching-text
--- ff bytes-left-till-EOL
Match the string at str-addr with all strings on the cursor line forward from the cursor. The arguments left allow the cursor R# to be updated either to the end of the matching text or to the start of the next line.

1LINE --- f
Scan the cursor line for a match to PAD text. Return flag and update the cursor R# to the end of matching text, or to the start of the next line if no match is found.

FIND ---
Search for a match to the string at PAD, from the cursor position till the end of screen. If no match found issue an error message and reposition the cursor at the top of screen.

DELETE n ---

Delete n characters prior to the cursor.

N ---
Find next occurrence of PAD text.

F ---
Input following text to PAD and search for match from cursor position till end of screen.

B ---
Backup cursor by text in PAD.

X ---
Delete next occurrence of following text.

TILL ---
Delete on cursor line from cursor to end of the following text.

C ---
Spread at cursor and copy the following text into the cursor line.

RELOCATING BUFFERS

The purpose of this section is to show you how to avoid incorporating buffer space into an auto-booting program, thereby saving more than 2K in memory requirement for the machine on which the program will eventually run.

Fig-FORTH (and so valFORTH) uses a virtual memory arrangement which allows disk areas to be accessed in a manner similar to that used to access semiconductor memory. We won't go into detail here; those wishing to find out more about this can contact FIG for documentation at:

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

or they can puzzle out the process by starting at the word BLOCK. For our present purposes, however, we simply note that the virtual memory scheme requires that some continuous area of memory be allotted as buffer space for disk operation. valFORTH as delivered has buffer space for four 512 byte "screens" at a time. Each screen is composed of four blocks of 132 bytes each: 128 bytes of actual data, corresponding to a sector, and four bytes of identification and delimiting data. This produces a total of 4 x 4 x 132 = 2112 bytes that are needed for programming and compilation* but are generally not required when software is actually run. In order to get the full use of your computer, particularly for the purposes of producing auto-booting software like games, you'll need to know how memory is mapped and what changes you can make in the mapping. During the following discussion refer to the memory map provided with your documentation.

You will note from the memory map that the buffers are placed just above the kernel (boot-up) valFORTH dictionary. The dictionary pointer is set just past the buffers, so new word definitions will be compiled in above the end of the buffers. Why such an odd location? Read on...

* Those used to seeing the buffers at the top of memory will quickly realize that this is impractical on the Atari, since that area is used for display lists. Although it is possible to an extent to fool the operating system into thinking that it has less memory than it actually has, and thus "reserve" an area at the top of memory, this is a troublesome proposition.

* Another approach is to put the buffers just below the kernel dictionary, which has been done in at least one FORTH-for-Atari release. While this is safe, it sacrifices 2K bytes during run time unless rather clever programming techniques are used on each program to put code into the dormant buffer area.

* Clearly, the buffers should be put somewhere above the dictionary but below the display-list area, and a simple means to relocate them should be supported. This is precisely what you have in valFORTH.

---

* In a pinch, you can <u>compile</u> using only 264 bytes of buffer memory.

When you have a program that will compile and run, preferably without errors, and you'd like to create a smaller auto-booting version, follow this procedure:

* Boot the valFORTH disk.

* Decide on the area to which to relocate the buffers: If the program can be loaded without leaving the 0 Graphics mode or doing anything else to high memory while loading, then the result printed by the sequence

```
0 GR. DCX 741 @ 2113 - U
```

(See note below.)

will be a safe place to put the buffers; 741 @ is the Atari OS pointer to just below the current display list. (If you will be using Transients, a capability of the Utilities/Editor package, their default location is

```
DCX 741 @ 4000 -
```

so you would be better off to put the buffers at, say,

```
DCX 741 @ 6113 -
```

to avoid conflict).

* Find the buffer relocation utility listed in the table of contents starting on screen 170 of the valFORTH disk, and load it. This is a self-prompting utility that directs you to relocate the buffers and then forget the utility. Follow the directions. You'll receive a verification message after the buffers have been moved.

* Type

```
' TASK DP !
```

to move the dictionary pointer below the old buffer area. (Advanced programmers: This is not a typo. The cfa of TASK points to NEXT.)

* Now load your program as usual. You should probably create an auto-booting program at this point, rather than doing anything else, since if you run the program now it may write into your relocated buffers and conceivably even attempt a write to your disk. So, create an auto-booting version as directed in the Auto-booting section above. Remember that if the program is for distribution, you MUST protect your software and ours by using the AUTO command.

*****CAUTION*****

The buffers start out just above the kernel dictionary, as indicated, and for normal programming they should be LEFT THERE: Several routines on the valFORTH disk and other disks in this product line use the area between pad and the bottom of the display list as a scratch area for extensive disk transfers. DISKCOPY1 and DISKCOPY2 on the valFORTH disk are examples.

Note: The buffers should generally be relocated to an <u>even</u> address because of an Atari OS bug. See also Note 1 at end of valFORTH 1.1 Glossary.

COMPILING AUTO-BOOTING SOFTWARE

Your purchase of valFORTH and its associated packages also grants you a single-user license for the software. You may not copy valFORTH or its associated Valpar International products for any purpose other than for your own use as back-up copies. However, a word called AUTO has been provided to allow you to create a copy of your software that is suitable for distribution. The word AUTO does several things.

* AUTO provides extensive protection both for your software and the valFORTH and auxiliary programs on which it is based. Your product may still be copied by normal methods, but the programming concepts on which it is based will be very difficult to analyze. The valFORTH and auxiliary programs will be rendered useless except to run your program. Since AUTO scrambles all headers in the code before saving to disk or cassette, even direct examination of the code on the medium is not very revealing. This provides essentially all the protection of headerless code.

* AUTO will create a disk that autoboots to the FORTH word of your choice. This usually will be the last word defined in your program. In addition, a disk created using AUTO will not have exit points: That is, even if your program terminates, or makes an error because of an undiscovered bug, it will not exit to valFORTH and the "ok" prompt. Instead, it will automatically attempt to start again at the original auto-boot word, and will do so unless an error has disabled the system.

* AUTO allows repetitive saving of your protected software to disk and cassette in one sitting, with extensive prompting. This provides a short-run production environment. (Remember that if you want to save to cassette, the cassette recorder should be attached to the system at boot time; if it is attached after booting, the computer may not know that the recorder is there and may fail when trying to AUTO to cassette).

To run AUTO and create your bootable software:

(1) Load valFORTH.
(2) Relocate buffers to save 2K+, if desired (see below).
(3) Load your program.
(4) DISPOSE transients, if you use them. (The Transient utilities come with the Utilities/Editor package, and allow use of "disposable assemblers" and the like).
(5) Find the Auto-Boot Utility section on the valFORTH disk by referring to the directory starting on screen 170, and load as indicated.
(6) Type AUTO cccc where "cccc" is the word which you wish to execute on auto-booting the software. You will now be prompted through the rest of the procedure. On exiting from AUTO you will fall through to the auto-booting program that you have just protected.

DISTRIBUTING YOUR PROGRAMS

If you have a program you wish to distribute, there are two ways in which to proceed:

(1) Make a PROTECTED auto-booting copy of your software by using the word AUTO as detailed in the "Compiling Auto-Booting Software" section of this manual.

(2) Make a TARGET-COMPILED version of your software, using the valFORTH Target Compiler, scheduled for release approximately 9/82. Target Compilers allow production of much smaller final FORTH products by allowing elimination of unnecessary code, e.g., headers, compiler, buffers, etc.

In addition to the above procedures, Valpar International also requires that the message:

Created in whole or part using valFORTH products of
Valpar International, Tucson, AZ 85713, USA
Based on fig-FORTH, provided through the courtesy of
Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070

be included either on the outside of the media (diskette, cassette, or other) as distributed, or in the documentation provided with the product. Please note that failure to include this message with products that include valFORTH code may be regarded as a copyright violation.

GRAPHICS, COLORS, AND SOUNDS

Graphics

The Graphics package follows the Atari BASIC graphics set as closely as possible, and is identical in most respects. As in BASIC, the most complex parts of Graphics are DRAWTO (abbreviated "DR.") and FIL, and even these are not too obscure. Find the Graphics Demo by looking at the directory starting on screen 170, and load it. Try the word FBOX. Now look at the code that produced this effect, if you like. The general explanation is as follows:

Display positions are denoted by two coordinates, a horizontal and a vertical. The 0,0 point is in the upper left hand corner, and the vertical coordinate increases as you go down the display, while the horizontal coordinate increases as you go to the right. This is all familiar from BASIC.

In graphics modes, a single point at position X Y can be plotted by X Y PLOT. The color of the point will be that in the color register declared by the last COLOR command. A line, again of the color in the register declared by the last color command, may then be drawn to point X1 Y1 by X1 Y1 DR. . The word FIL may be used to fill in an area as described in the Atari manual, and as illustrated in the FBOX example. The color register for the fill is the one whose number is on the stack when FIL is executed. Essentially, to set up FIL you draw in boundaries and pick two points you wish to FIL between. The first of these points is set up either by a DR. or PLOT command, or by valFORTH's POSIT command. POSIT has the advantage of not requiring that you put anything into the place where you are positioning yourself. The second point for the FIL command is then set up by using POS. . The fill is then performed by putting a number on stack (the color register for the fill) and then doing FIL.

If you are in a text mode, a single character, c , can be sent to the display by ASCII c CPUT. Text strings can be sent to the display with G" cccc " and in addition will have the color in the register specified by the last COLOR command before the string is output. This is a significant enhancement to BASIC.

Graphics and Color Glossary:

SETCOLOR n1 n2 n3 --
Color register n1 (0...3 and 4 for background) is set to hue n2 (0 to 15) and luminance n3 (0-14, evens).

SE. n1 n2 n3 --
Alias for SETCOLOR.

GR. n --
Identical to GR. in BASIC. Adding 16 will suppress split display. Adding 32 will suppress display preclear. In addition, this GR. will not disturb player/missiles.

POS. x y --
Same as BASIC POSITION or POS. Positions the invisible cursor if in a split display mode, and the text cursor if in 0 GR. .

POSIT x y --
Positions and updates the cursor, similar to PLOT, but without changing display data.

PLOT x y --
Same as BASIC PLOT. PLOTs point of color in register specified by last COLOR command, at point x y.

DRAWTO x y --
Same as BASIC DRAWTO. Draws line from last PLOT'ted, DRAWTO'ed or POSIT'ed point to x y, using color in register specified by last COLOR command.

DR. x y --
Alias for DRAWTO.

FIL b --
Fills area between last PLOT'ed, DRAWTO'ed or POSIT'ed point to last position set by POS., using the color in register b.

G" --
Used in the form G" ccccc". Sends text cccc to text area in non-0 Graphics mode, starting at current cursor position, in color of register specified by last COLOR command prior to cccc being output. G" may be used within a colon definition, similar to .".

GTYPE addr count --
Starting at addr, output count characters to text area in non-0 Graphics mode, starting at current cursor position, in color of register specified by last COLOR command.

LOC. x y -- b
Positions the cursor at x y and fetches the data from display at that position. Like BASIC LOCATE and LOC. . Note that since the word LOCATE has a different meaning in valFORTH (it is part of the advanced editor in the Utilities/Editor package), the name is not used in this package. (Advanced users: We could put Graphics in its own vocabulary, but this would add some inconvenience.)

```
(G")       --
     Run-time code compiled in by G".

POS@      -- x y
     Leaves the x and y coordinates of the cursor on the stack.

CPUT    b --
     Outputs the data b to the current cursor position.

CGET      -- b
     Fetches the data b from the current cursor position.

>SCD    c1 -- c2
     Converts c1 from ATASCII to its display screen code, c2.
     Example:  ASCII A >SCD 88 @ C!
     will put an "A" into the upper left corner of the display.

SCD>    c1 -- c2
     Converts c1 from display screen code to ATASCII c2.
     See >SCD.

>BSCD  addr1 addr2 count --
     Moves count bytes from addr1 to addr2, translating from ATASCII
     to display screen code on the way.

BSCD>  addr1 addr2 count --
     Moves count bytes from addr1 to addr2, translating from display
     screen code to ATASCII on the way.

COLOR  b --
     Saves the value b in the variable COLDAT.

COLDAT  -- addr
     Variable that holds data from last COLOR command.

GREY    --  0
GOLD    --  1
ORNG    --  2
RDORNG  --  3
PINK    --  4
LVNDR   --  5
BLPRPL  --  6              (CONSTANTs)
PRPLBL  --  7
BLUE    --  8
LTBLUE  --  9
TURQ    -- 10
GRNBL   -- 11
GREEN   -- 12
YLWGRN  -- 13
ORNGRN  -- 14
LTORNG  -- 15

BOOTCOLOR   hue lum --
     Sets up hue for playfield 2 (text background) and lum for playfield 1
     (letter intensity) in 0 Graphics mode.  Lum of playfield 2 is set at 4.
     After using BOOTCOLOR, doing SAVE will create a system disk with the
     selected color.
```

Sounds

The actual production of sound by the Atari machines is rather complex and the reader is referred to the many recent (first half 1982) articles on this subject in various magazines. Here we will restrict comments to the function of the Atari audio control register. This is an eight bit register which valFORTH shadows by the variable AUDCTL. The bits have the following functions:

bit 7: Change 17 bit polycounter to 9 bit polycounter.
       Affects distortions 0 and 8.
bit 6: Clock channel 0 with 1.79 Mhz instead of 64 Khz.
bit 5: Clock channel 2 with 1.79 Mhz instead of 64 Khz.
bit 4: Clock channel 1 with channel 0 instead of 64 Khz.
bit 3: Clock channel 3 with channel 2 instead of 64 Khz.
bit 2: Use channel 2 as crude high-pass on channel 0.
bit 1: Use channel 3 as crude high-pass on channel 1.
bit 0: Change normal 64 Khz to 15 Khz.

The value n may be sent to the audio control register by doing n FILTER!.

SOUND   chan freq dist vol --
    Sets up the sound channel "chan" as indicated.
    Channel: 0-3.
    Frequency: 0-255, 0 is highest pitch.
    Distortion: 0-14, evens only.
    Volume: 0-15.
    Suggested mnemonic: CatFish Don't Vote

SO.   chan freq dist vol --
    Alias of SOUND.

FILTER!   n --
    Stores n in the audio control register and into the valFORTH shadow register, AUDCTL. Use AUDCTL when doing bit manipulation, then do FILTER!. (FILTER! does a number of housekeeping chores, so use it instead of a direct store into the hardware register.)

AUDCTL   -- addr
    A variable containing the last value sent to the audio control register by FILTER!. Used for bit manipulation since the audio control register is write-only.

XSND   n --
    Silences channel n.

XSND4   --
    Silences all channels.

S: flag --
If flag is true, enables handler that sends text to text screen. If false, disables the handler. (See PFLAG in main glossary.) ON S: etc.

P : flag --
If flag is true, enables handler that sends text to printer. If false, disables the handler. (See PFLAG in main glossary.) OFF P: etc.

BEEP --
Makes a raucous noise from the keyboard. Is put in this package for lack of a better place.

ASCII c, -- n (executing)
c, -- (compiling)
Converts next character in input stream to ATASCII code. If executing, leaves on stack. If compiling, compiles as literal.

EJECT --
Causes a form feed on smart printers if the printer handler has been enabled by ON P:. May need adjustment for dumb or nonstandard printers.

LISTS start count --
From start, lists count screens. May be aborted by CONSOLE button at the end of a screen.

PLIST scr --
Lists screen scr to the printer, then restores former printer handler status.

PLISTS start cnt --
From start, lists cnt screens to printer three to a page, then restores former printer handler status. May be aborted by CONSOLE button at the end of a screen.

FORMAT --
With prompts, will format a disk in drive of your choice.

(FMT) n1 -- n2
Formats disk in drive n1. Leaves 1 for good format, otherwise error number. Note: Because of what appears to be an OS peculiarity, this operation must not be the first disk access after a boot.

DISKCOPY1 --
With prompts, copies a source to a destination disk on single drive, with swapping. Smart routine uses all memory from PAD to bottom of Display List, producing minimum number of swaps.

DISKCOPY2 --
With prompts, copies disk in drive 1 to disk in drive 2 using memory like DISKCOPY1.

DECOMP cccc

Does a decompilation of the word cccc if it can be found in the active vocabularies.

Although DECOMP is very smart, like most FORTH decompilers it will become confused by certain constructs, and will begin to print trash, with pauses in between while it looks for more trash to print. When this happens, simply hold down a CONSOLE button until DECOMP exits. This sometimes takes as much as 10 seconds, depending on luck.

CDUMP addr n --

A character dump from addr for at least n characters. (Will always do a multiple of 16.)

#DUMP addr n --

A numerical dump in the current base for at least n characters. (Will always do a multiple of 8.)

(FREE) -- n

Leaves number of bytes between bottom of display list and PAD. This is essentially the amount of free dictionary space, if additional memory is not being used for player/missiles, extra character sets, and so on.

FREE --

Does (FREE) and then prints the stack and "bytes".

H. n --

Prints n in HEX, leaves BASE unchanged.

STACK flag --

If flag is true, turns on visible stack. If flag is false, turns off visible stack.

.S ... -- ...

Does a signed, nondestructive stack printout, TOS at right. Also sets visible stack to do signed printout.

U.S ... -- ...

Does unsigned, nondestructive stack printout, TOS at right. Also sets visible stack to do unsigned printout.

B? --

Prints the current base, in decimal. Leaves BASE undisturbed.

CFALIT cccc, -- cfa (executing)
cccc, -- (compiling)

Gets the cfa (code field address) of cccc. If executing, leaves it on the stack; if compiling, compiles it as a literal. Not precisely a debugging tool, but finds use in DECOMP.

FLOATING POINT WORDS

The floating-point package uses the Atari floating point routines in the operating system ROM in the same way that Atari Basic does. The routines are rather slow, and there are no trigonometric functions internal to the Atari. (SIN, COS, TAN, ATN, and ATN2 have been programmed and are available in the Advanced Graphics/Floating Point Package.) LOG and EXP are included in the operating system ROM and are supported in the present package, in base 10 and base e. Note that in the directory on screen 170 it is indicated that the ASSEMBLER must be loaded before loading the floating-point package.

Floating point words have a six byte representation in the Atari OS, and since the stack has a 60 byte maximum, a maximum of 10 floating point numbers can be on the stack at a time. In practice, this maximum often becomes 9 since some fp routines use the stack as a scratch area.

Operations involving floating-point numbers generally leave floating-point results. Exceptions are the words FIX, which takes a positive floating pointer number less than 32767.5 and leaves a rounded integer; and the floating-point comparison operators, F=, F<, etc., which leave flags. To get a floating-point number on the stack, use the word FLOATING or its alias, FP, followed by a number in Fortran "E" format. For example,

```
        FP  12345
        FP  12345.6
        FP  -12345.8
        FP  +5432E-16
    and FP  -8E18
```

will all leave floating-point numbers on the stack. Floating-point variables and constants are also supported.

It has been our experience that mistakes are common when first using this package. One must remember to use F* and not *, F+ and not +, and so on, when doing fp operations. Remember also that integers and fp numbers can't be mixed by operations: Either convert the fp number by FIX, or the integer by FLOAT, and then use the appropriate operation.

Create new words as usual. For instance, to define a floating-point square root function, write

```
: FSQRT                   ( fp -- fp )
  LOG  FP  2  F/  EXP ;
```

Overflow and underflow, and illegal operations such as dividing by 0, taking logarithms of negative numbers, or FIXing a negative number cause undefined and rather unpredictable results, though they do not harm the system. (Additional words in the Utilities/Editor Package cause all but one of these operations to give correct or useable results; logarithms of negatives cannot be approximated with Real numbers.)

The maximum and minimum numbers are generous, about 1E97 and 1E-97, and it is sometimes possible to exceed these limits during computation. Atari's internal representation of floating point numbers is awkward. Refer to the Atari OS manual, available from Atari, for details if needed.

FLOATING-POINT GLOSSARY

In the following, "fp" is used to indicate a floating-point number (six bytes) on the stack. The terms "top-of-stack," "2nd-on-stack" etc., have been used with the obvious meanings even though, because fp numbers are six bytes, their physical positions on the stack will not match the usual ones.

FCONSTANT   cccc, fp --
              cccc: --fp

The character string is assigned the constant value fp. When cccc is executed, fp will be put on the stack.

Example: FP 3.1415926 FCONSTANT PI

FVARIABLE   cccc, fp --
              cccc: addr --

The character string cccc is assigned the initial value fp. When cccc is executed, the addr (two bytes) of the value of cccc will be put on the stack.

Example: FP 0 FVARIABLE X
FP 18.4   X   F!

FDUP   fp1 -- fp1 fp1

Copies the fp number at top-of-stack.

FDROP   fp --

Discards the fp number at top-of-stack.

FOVER   fp2 fp1 -- fp2 fp1 fp2

Copies the fp number at 2nd-on-stack to top-of-stack.

FLOATING   cccc,   -- fp

Attempts to convert the following string, cccc, to a fp number. Stops on reaching first unconvertible character and skips the rest of the string. If no characters convertible, leaves unpredictable fp number on stack.

FP   cccc,   --fp

Alias for FLOATING.

F@   addr -- fp

Fetches the fp number whose address is at top-of-stack.

F!   fp addr --

Stores fp into addr. Remember that the operation will take six bytes in memory.

F.   fp --

Type out the fp number at top-of-stack. Ignores the current value in BASE and uses base 10.

F?   addr --

Fetches a fp number from addr and types it out.

F+ fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their fp sum, fp3.

F- fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their difference, fp3=fp2-fp1.

F* fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their product, fp3.

F/ fp2 fp1 -- fp3
Replaces the two top-of-stack fp items, fp2 and fp1, with their quotient, fp3=fp2/fp1.

FLOAT n -- fp
Replaces number at top-of-stack with its fp equivalent.

FIX fp (non-neg, less than 32767.5) -- n
Replaces fp number at top-of-stack, constrained as indicated, with its integer equivalent.

LOG fp1 -- fp2
Replaces fp1 with its base e logarithm, fp2. Not defined for fp1 negative.

LOG10 fp1 - fp2
Replaces fp1 with its base 10 decimal logarithm, fp2. Not defined for fp1 negative.

EXP fp1 -- fp2
Replaces fp1 with fp2, which equals e to the power fp1.

EXP10 fp1--fp2
Replaces fp1 with fp2, which equals 10 to the power fp1.

F0= fp -- flag
If fp is equal to floating-point 0, a true flag is left. Otherwise, a false flag is left.

F= fp2 fp1 -- flag
If fp2 is equal to fp1, a true flag is left. Otherwise, a false flag is left.

F> fp2 fp1 -- flag
If fp2 is greater than fp1, a true flag is left. Otherwise, a false flag is left.

F< fp2 fp1 -- flag
If fp2 is less than fp1, a true flag is left. Otherwise, a false flag is left.

FLITERAL fp --
If compiling, then compile the fp stack value as a fp literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

```
: xxx [ calculate ] FLITERAL ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is resumed and FLITERAL compiles the value on stack.

FLIT -- fp
Within a colon definition, FLIT is automatically compiled before each fp number encountered as input text. Later execution by the system of FLIT as it is encountered in the dictionary cause the context of the next 6 dictionary addresses to be pushed to the stack as a fp number. FLIT is also compiled in explicitly by FLITERAL.

ASCF addr -- fp
An ASCII-to-floating-point conversion routine. Uses Atari OS routine. The routine reads string starting at addr and attempts to create a floating point number. If string is not a valid ASCII floating-point representation, leaves undefined result on stack. Used by FLOATING.

FS fp --
System routine. Sends fp argument on stack to Atari register FR0. Experts only.

>F -- fp
System routine. Fetches fp argument from Atari register FR0. Experts only.

<F fp1 fp2 --
System routine. Sends fp1 and fp2 to Atari registers FR1 and FR0 respectively. Experts only.

F.TY --
System routine. Types out last fp number converted by FASC.

CIX addr --
System variable. One byte offset pointer in buffer pointed to by INBUF. Experts only.

INBUF addr --
System variable. Used by ASCF to know where ASCII string to be converted is located.

FR1 -- n
System constant. Atari internal register address.

FR0 --n
System constant. Atari internal register address.

FPOLY addr count --
A system routine for advanced users doing polynomial evaluation.
The polynomial P(Z) = SUM(i=0 to n) (A(i)*Z**i) is computed by the following standard method:
P(Z) = (...(A(n)*Z + A(n-1))*Z + ... + A(1))*Z + A(0)
The address addr points to the coefficients A(i) stored sequentially in memory, with the highest order coefficient first. The count is the number of coefficients in the list. The independent variable Z, in floating-point, should be sent to FR0 using FS. FPOLY is then executed. The result put on the stack using >F. Note that FPOLY is intended to be used in a Forth word. Trigonometric functions and general polynomial expansions, for example, may be defined more simply with the help of this routine.

FLG10 --
System routine used by LOG10.

FLG --
System routine used by LOG.

FEX --
System routine used by EXP.

FEX10 --
System routine used by EXP10.

FDIV --
System routine used by F/.

FMUL --
System routine used by F*.

FSUB --
System routine used by F-.

FADD --
System routine used by F+.

FPI --
System routine used by FIX.

IFP --
System routine used by FLOAT.

FASC --
System routine, Does floating-point-to-ASCII conversion on the fp number in FR0 and leaves string at address pointed to by INBUF. Last byte of string has most significant bit set. Used by F.TY.

AFP --
System routine used by ASCF.

(intentionally left blank)

OPERATING SYSTEM

This package implements the computer's Operating System I/O routines. The 850 (RS-232C) driver package may be loaded into the dictionary by using the word RS232, which will then support references to devices "R1" through "R4."

The code for this section was originally written by Patrick Mullarky, and published through the Atari Program Exchange. It is used here by permission of the author.

OS GLOSSARY

OPEN addr n0 n1 n2 -- n3
: This word opens the device whose name is at addr. The device is opened on channel n0 with AUX1 and AUX2 as n1 and n2 respectively. The device status byte is returned as n3. The name of a device may be produced in various ways: For a single character name, say "S" for the screen handler,

```
ASCII S PAD C!
```

will leave the ASCII value of S at PAD. Then

```
PAD 3 8 0 OPEN
```

will open the screen handler on channel 3 with AUX1 = 8 (write only) and AUX2 = 0. If you have the UTILITIES/EDITOR Package, longer names may be setup simply by using the word " .

CLOSE n --
: Closes channel n.

PUT b1 n -- b2
: Outputs byte b1 on channel n, returns status byte b2.

GET n -- b1 b2
: Gets byte b1 from channel n, returns status byte b2.

GETREC addr n1 n2 -- n3
: Inputs record from channel n2 up to length n1. Returns status byte n3.

PUTREC addr n1 n2 -- n3
: Outputs n1 characters starting at addr through channel n2. Returns status byte n3.

STATUS n -- b
: Returns status byte b from channel n.

DEVSTAT n -- b1 b2 b3
: From channel n1 gets device status bytes b1 and b2, and normal status byte b3.

SPECIAL b1 b2 b3 b4 b6 b6 b7 b8 -- b9
: Implements the Operating System "Special" command. AUX1 through AUX6 are b1 through b6 respectively, command byte is b7, channel number is b8. Returns status byte b9.

RS232 --
: Loads the Atari 850 drivers into the dictionary (approx 1.8K) through a three-step bootstrap process. Executing this command more than once without turning the 850 off and on again will crash the system.

valFORTH
T.M.

Advanced 6502 Macro Assembler

Version 2.0
April 1982

Although the FORTH language is many times faster than BASIC or PASCAL, there are still times when speed is so critical that one must turn to assembly language programming as a matter of necessity. Not wanting to give up the advantages of the FORTH language, FORTH programmers typically use an assembler designed specifically for the FORTH system. valFORTH incorporates a very powerful FORTH style 6502 assembler for these special programming jobs.

© Copyright Valpar International 1982

Overview

Most programming applications can be undertaken completely in high level FORTH. There are times, due to speed constraints, when assembly language must be used. Typically, "number crunching" and high speed graphic routines must be machine coded. valFORTH provides a powerful 6502 FORTH assembler for these special occasions.

FORTH assemblers differ from standard assemblers by making the best use of the stack and the FORTH system as a whole. The FORTH assembler is smaller than a standard assembler. In the case of the valFORTH assembler, this is particularly true.

The valFORTH assembler offers the programmer the following improvements over a standard assembler:

1) IF...THEN...ELSE structures which use positive logic rather than negative logic.

2) BEGIN...UNTIL structures for post-testing indefinite loops.

3) WHILE...REPEAT structures for pre-testing indefinite loops.

4) BEGIN...AGAIN structures for unconditional looping.

5) Full access to the FORTH operating system and its capabilities such as changing bases.

6) Complex assembly time calculations.

7) Mixed high level FORTH with assembly code to take full advantage of each.

8) Full macro capability.

The following is a complete description of the valFORTH assembler. This description assumes a working knowledge of 6502 assembly language programming and related terms.

The purpose of the FORTH assembler is to allow machine language programming without the need to abandon the FORTH system. Words coded in assembly language must follow the standard FORTH dictionary format and must adhere to certain guidelines regarding their coding.

Assembly language programmers typically have two methods of storing programs into RAM. The machine code can be poked directly into memory, or an assembler can be used to accomplish this. The former method is brutal, but it has the advantage that precious memory is not taken up by the assembler. The drawback, of course, is loss of readability and ease of modification. FORTH allows both of these methods to be employed.

The words "," and "C," can be used to poke any machine language program into the dictionary. This is used only when memory restrictions prohibit the use of an assembler or if it is assumed that no assembler is available.

In high level FORTH, words are compiled into the dictionary using the following form:

```
: name     high-level-FORTH... ;
```

When compiling a machine coded word, this becomes:

```
CODE name     machine-code... C;
```

In this example, the word "CODE" creates a header for the next word in much the same way ":" creates a header. The difference lies in the fact that ":" informs the system that the following definition is high level FORTH, while "CODE" indicates that the definition is a machine or assembly language definition. In the same manner, ";" terminates a high level FORTH definition while "C;" terminates a code definition.

To clarify this, a code definition will be programmed that will clear the top line of the current video display on an Atari 800 microcomputer. Note that video memory is pointed to by the address stored in locations 88 and 89 (decimal). The 6502 code is shown in listing 1.

```
CLR   TYA           ; Y comes in with 0; 0 means a blank
      LDY #39       ; 40 characters/line (0 thru 39)
LOOP  STA (88), Y   ; Fill from end to beginning
      DEY           ; Done?
      BPL LOOP      ; Keep going if not
      JMP NEXT      ; Re-enter the FORTH operating system
```

Listing 1

The CODE definition equivalent to listing 1 would be:

```
HEX                                  (put in hex mode)
CODE CLR                             (define code word)
      98 C,                          (poke in code)
      A0 C, 27 C,
      91 C, 58 C,     <--+
      88 C,              |
      10 C, FB C,     ---+
C; DECIMAL                           (end assembly)
```

First, the FORTH system is put into the hexadecimal mode so that opcode values need not be converted to decimal. Next, the word CODE puts the system into an assembly mode and enters the new word CLR into the dictionary as a machine language word. The opcodes are then byte compiled ("C,") into the dictionary. Note that for the final jump to re-enter FORTH, the predefined word NEXT was word compiled (",") into the dictionary. The word C; terminates the assembly process. The system is then restored to the decimal mode.

This method can always be used, but it is very tedious. Each opcode must be looked up, and all relative branches calculated. Besides introducing a great source for error, if a single opcode is added or deleted, it is possible that many jumps must be re-calculated. For this reason, using the assembler is the prescribed method for entering machine language routines.

Unlike the standard assembler which has four fields (the label field, the operation field, the operand field, and the comment field), the FORTH assembler has only three fields. In a FORTH assembler, there is no explicit label field, but there is an implied label field through the use of the assembler constructs IF, and BEGIN, described later. In addition, the remaining three fields in the FORTH assembler are in reversed order (as is standard for the FORTH language). In other words, the operand precedes the operation, and remarks can be embedded anywhere.

In compiling an assembly word, the FORTH assembler ultimately uses either "," or "C," and for this reason assembly mnemonics traditionally end with a comma. valFORTH equivalents are shown in chart 1.

Standard Assembler		valFORTH Assembler
LDX COUNT		COUNT LDX,
JMP COUNT+1		COUNT 1+ JMP,
LDA #3		# 3 LDA,
ADC N		N ADC,
STY TOP,X		TOP ,X STY,
INC BOT,Y		BOT ,Y INC,
STA (TOP,X)		TOP X) STA,
AND (BOT),Y		BOT)Y AND,
JMP (POINT)		POINT)JMP,
DEC N+4		N 4 + DEC,
DEX		DEX,
ROL A		.A ROL,
	or	ROL.A,

```
Note:      # 9 LDA, =    9  #  LDA,
        TOP ,X ROL, =   ,X TOP ROL, etc.
```

Chart 1

Converting the program given in listing 1 to FORTH assembly mnemonics we have:

```
DECIMAL
CODE CLR
     TYA,                 (    TYA              )
     # 39 LDY,            (    LDY #39          )
     BEGIN,               ( LOOP                )
        88 )Y STA,        (    STA (88),Y       )
        DEY,              (    DEY              )
     MI UNTIL,            (    BPL LOOP         )
     NEXT JMP,            (    JMP NEXT         )
C;
```

In the above example, a BEGIN ... UNTIL, clause (described in the next section) is used. By using this structure, no labels are necessary and positive logic is used rather than negative logic (i.e., "repeat until minus" instead of "if NOT minus, then repeat"). Note that the FORTH assembler compiles exactly the same machine code as the standard assembler, it simply makes the assembly coding easier.

Control Structures

Allowing labels within assembly language programming would make the FORTH assembler needlessly long and slow. To get around the problem of test branching, the ValFORTH assembler has a very powerful set of control structures similar to those found in high level FORTH.

The IF,...ENDIF, and IF,...ELSE,...ENDIF, clauses

The IF, construct which handles conditional downward branches has the following two forms:

```
...code...                      ...code...
flag IF,                        flag IF,
    ...true code...                 ...true code...
ENDIF,                          ELSE,
...code...                        ...false code...
                                ENDIF,
                                ...code...
```

where "flag" is one of the 6502 statuses: NE , EQ , CC , CS , VC , VS , MI , or PL. The following are a few examples of how these are used.

Note: When the FORTH inner interpreter passes control to an assembly language routine, the Y register always contains a zero value and the X register must be preserved as it is used by the FORTH system to maintain the parameter stack. See the section on parameter passing for more information.

```
;  Code routine for 1+
ONEPL INC 0,X        ; increment low byte of 16 bit value
      BNE THERE      ; carry out of low?
      INC 1,X        ; increment high byte if so
THERE JMP NEXT       ; re-enter FORTH system
```

Now in ValFORTH assembly language:

```
CODE ONEPL           (define word)
     0 ,X INC,       (increment low byte)
     EQ IF,          (if result was zero,)
        1 ,X INC,    (then bump the high byte)
     ENDIF,
     NEXT JMP,       (exit to FORTH)
C;
```

Note: In the following example, CONIN is assumed to be predefined.

```
; Input routine
INPUT JSR CONIN     ; Go get character, comes back in A
      CMP #$0D      ; Is it a carriage return?
      BNE INP1      ; If not, do something else
      ...code1...   ; execute code for carriage return
      JMP INP2      ; do not execute "normal" code
INP1  ...code2...   ; execute code for normal keys
INP2  ...code3...   ; execute code more common code
      JMP NEXT      ; re-enter FORTH system
```

The equivalent valFORTH version would be:

```
HEX
CODE INPUT
     CONIN JSR              (Get character         )
     # 0D CMP,              (carriage return?      )
     EQ IF,                 (If so, then           )
        ...code1...         (execute c/r code      )
     ELSE,                  (otherwise             )
        ...code2...         (execute normal code   )
     ENDIF,
     ...code3...
     NEXT JMP,              (re-enter FORTH system )
C; DECIMAL
```

The BEGIN,...UNTIL, clause

Another useful structure is the BEGIN,...UNTIL, construct which allows for post-testing indefinite looping. The BEGIN,...UNTIL, construct has the following form:

```
...code1...
BEGIN,                      code2 is repeatedly
   ...code2...              executed until "flag"
flag UNTIL,                 is true.
...code3...
```

The following 6502 routine waits until a carriage return has been typed.

```
; WAIT until c/r
WAIT  JSR CONIN     ; Go get a character, comes back in A
      CMP #$0D      ; Is it a carriage return?
      BNE WAIT      ; Ask again if not
      JMP NEXT      ; Return to FORTH
```

Using the BEGIN, clause, this becomes

```
NEXT
CODE  WAIT                  (Code name WAIT   )
      BEGIN,                (Begin waiting    )
         CONIN JSR,         (Get a character  )
         # 0D CMP,          (Carriage return?)
      EQ UNTIL,             (loop up until so)
      NEXT JMP,
C; DECIMAL
```

The BEGIN,...WHILE,...REPEAT, clause

In the valFORTH assembler, there is another valuable control structure. It is the BEGIN,...WHILE,...REPEAT, structure. The WHILE, clause allows pre-testing indefinite loops to be easily programmed. It takes the form:

```
...code1...
BEGIN,
   ...code2...
flag WHILE,
   ...code3...
REPEAT,
...code4...
```

Code2 and code3 are repeatedly executed until "flag" become false, at which time program control proceeds to code4.

A common example of the WHILE, clause is getting a line of input text terminated by a carriage return.

```
;  Get line of text  (note: Y=0 on entry always)
GETLN JSR CONIN          ; Get one character
      CMP #$0D           ; C/R terminates input
      BEQ GETL1          ; If not a C/R then
      STA BFFR,Y         ; store the character
      INY                ; Bump buffer pointer
      JMP GETLN          ; Go back for more
GETL1 JMP NEXT           ; Exit to FORTH
```

Using the WHILE, clause in valFORTH, we have:

```
HEX
CODE GETLN
     BEGIN,
        CONIN JSR,       (Get a character          )
        # 0D CMP,        (Carriage return?         )
     NE WHILE,           (If not,                  )
        BFFR ,Y STA,     (then store the character )
        INY,             (and bump the pointer     )
     REPEAT,             (Repeat all of the above  )
     NEXT JMP,
C; DECIMAL
```

The BEGIN,...AGAIN, clause

The final control structure is the BEGIN,...AGAIN, structure. This structure allows the use of unconditional looping in assembly language routines. Although its use is rare, it can reduce code size considerably. It takes the following form:

```
...code1...
BEGIN,                          Repeatedly execute code2
   ...code2...                  and code4 until "flag"
   flag IF,                     becomes true, in which
      ...code3...               case, program execution
      re-entry-point JMP,       continues with code3 and
   ENDIF,                       a system re-entry made.
   ...code4...
AGAIN, C;
```

The best example of the AGAIN, clause is in the coding of the CMOVE routine:

```
;  Byte at a time front end memory move
CMOVE LDA #3             ; Get top three stack items
      JSR SETUP          ; Move them to N scratch area
CMOV1 CPY N              ; Time to decrement COUNT high?
      BNE CMOV2          ; Nope
      DEC N+1            ; Yes, so do it
      BPL CMOV2          ; Bypass exit if not done
      JMP NEXT           ; Exit to FORTH system
CMOV2 LDA (N+4),Y        ; Get byte to move
      STA (N+2),Y        ; Move it!
      INY                ; Bump byte pointer
      BNE CMOV1          ; Keep going until ready to
      INC N+5            ;   bump high bytes of both
      INC N+3            ;   "to" and "from" addresses
      JMP CMOV1          ; Do it all again
```

Using the AGAIN, clause, this becomes:

```
CODE CMOVE
     # 3 LDA,             (Prepare for memory move)
     SETUP JSR,
     BEGIN,
        BEGIN,            (Start the process)
           N CPY,         (done?)
           EQ IF,
              N 1+ DEC,   (Maybe, keep checking)
              MI IF,
                 NEXT JMP,  (Re-enter FORTH system)
              ENDIF,
           ENDIF,
           N 4 + )Y LDA,  (Get byte to copy)
           N 2+ )Y STA,   (Store in new location)
           INY,           (Bump pointer)
        EQ UNTIL,
        N 5 + INC,        (Bump addresses)
        N 3 + INC,
     AGAIN,               (Do it all again)
C; DECIMAL
```

Parameter Passing

One of the most useful features of the FORTH language is its ability to use a parameter stack for passing values from one word to another. For assembly language routines to really be useful in the FORTH system, there must be some facility for these routines to access this stack. Likewise, there should be some way in which to access the return stack as well. This section details exactly how to make the best use of both stacks.

Since the FORTH system maintains dual stacks and the 6502 supports only one, it is necessary to simulate one of the stacks. For ease of stack manipulation, the parameter is simulated; the return stack uses the hardware stack of the microprocessor.

The simulated stack uses the 0-page,X addressing mode of the 6502. For example, the following statements show how the parameter stack is organized.

```
LDA 0,X      Low byte of item on top of stack
INC 1,X      High byte of top item
ADC 2,X      Low byte of item second on stack
EOR 3,X      High byte of 2OS
RNL 4,X      Low byte of item third on stack
AND 5,X
...          etc.
```

In high level FORTH, the word DROP drops (or pops) the top value from the stack. The code definition for DROP is:

```
CODE DROP     INX, INX, NEXT JMP, C;
```

In the same way, values can be "pushed" to the stack. Note that the X register must be preserved between FORTH words or the parameter stack is lost! Thus if the X register is needed in a code definition, it must be saved upon entry to the routine and restored before returning to the FORTH system. The special location XSAVE is reserved for this: (The word XSAVE has been defined as a FORTH constant.)

```
STX XSAVE        Save the X register
LDX XSAVE        Restore the X register
```

In all the examples given so far, the code definitions have re-entered the FORTH system through the normal re-entry point called NEXT. The following is a complete description of all possible re-entry points: (In all of the following code examples, standard 6502 assembler format has been used for ease of comprehension. All valFORTH assembler equivalents can be found in appendix A.)

The NEXT re-entry point

The NEXT routine transfers control to the next FORTH word to be executed. All FORTH words eventually come through the NEXT routine. Likewise, all other re-entry points come through NEXT once they have completed their special tasks. The next routine is typically used by words

The NEXT re-entry point (cont'd)

such as 1- which do not modify the number of arguments on the stack. The word NEXT is defined as a FORTH constant. NXT, is an abbreviation for NEXT JMP, .

Example:

```
;  1- routine
ONEM LDA 0,X       ; Borrow from low byte?
     BNE ONE1      ; If not, ignore correction
     DEC 1,X       ; Decrement high byte
ONE1 DEC 0,X       ; Now do the low
     JMP NEXT      ; Re-enter FORTH
```

Listing 2

The PUSH re-entry point:

The PUSH routine pushes a 16 bit value to the parameter stack whose low byte is found on the 6502 return stack and whose high byte is found in the accummulator. The X register is automatically decremented twice for the two bytes. This routine is typically used for words such as OVER or DUP which leave one more argument than they expect. The word PUSH has been defined as a FORTH constant. PSH, is an abbreviation for PUSH JMP, .

Example:

```
;  DUP routine
DUP  LDA 0,X       ; Get low byte of TOS
     PHA           ; Push it
     LDA 1,X       ; Put high byte in A
     JMP PUSH      ; Put it on the P-stack
```

Listing 3

The PUT re-entry point:

The PUT routine replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is found on the 6502 stack and whose high byte is in the accumulator. This is used by words such as ROT or SWAP which do not change the number of values on the stack. The word PUT has been defined as a FORTH constant. PUT, is an abbreviation for PUT JMP, .

Example:

```
;  SWAP routine
SWAP LDA 2,X       ; Low byte of 2nd value
     PHA           ; Save it
     LDA 0,X       ; Put low byte of TOS
     STA 2,X       ; into low byte of 2OS
     LDA 3,X       ; Hold high byte of 2OS
     LDY 1,X       ; Put high byte of TOS
     STY 3,X       ; into high byte of 2OS
     JMP PUT       ; Replace TOS no
```

Listing 4

The PUSHOA re-entry point

The PUSHOA re-entry point pushes the 8 bit unsigned value in the accumulator as a 16 bit value with the upper 8 bits zeroed. This word is very commonly used by words which leave a boolean flag on the parameter stack such as ?TERMINAL. The word PUSHOA has been defined as a FORTH constant. PSHA, is an abbreviation for PUSHOA JMP, .

```
Example:  ;  ?TERMINAL routine
          QTERM LDA $D01F      ; Read Atari CONSOLE keys
                EOR #7         ; Anything pressed?
                BEQ QT1        ; If not, go push false
                INY            ; Else push a true
          QT1   TYA            ; Put Y (0 or 1) in A
                JMP PUSHOA     ; Go push the result
```

Listing 5

The PUTOA re-entry point:

The PUTOA routine replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is in the accumulator and whose high byte is set to zero. This is used by words such as C@ which simply replace their arguments on the stack. The word PUTOA is defined as a FORTH constant. PUTA, is an abbreviation for PUTOA JMP, .

```
Example:   ;  Byte fetch
           CFCH LDA (0,X)      ; Load byte indirectly
                JMP PUTOA      ; Replace the address
                               ; with the contents
```

Listing 6

The BINARY re-entry point

The BINARY re-entry point drops the value on top of the parameter stack and then performs the PUT operation described above. This word is commonly used by words such as XOR which use one more argument than they leave. The word BINARY has been defined as a FORTH constant.

```
Example:  ;  Exclusive or TOS with 2OS
          XOR   LDA 0,X        ; Get low byte of top value
                EOR 2,X        ; XOR it with low of 2OS
                PHA            ; Save it
                LDA 1,X        ; Now do same for high bytes
                EOR 3,X        ; Result in A
                JMP BINARY     ; Go DROP , PUT
```

Listing 7

POP and POPTWO re-entry points

The POP and POPTWO re-entry points are used when values must be dropped from the parameter stack. POP performs a DROP, while POPTWO performs a 2DROP. Most words which can use BINARY can use POP. The words POP and POPTWO have been defined as FORTH constants. POP, is an abbreviation for POP JMP, and POP2, is an abbreviation for POPTWO JMP, .

Examples:

```
;  Another XOR routine
XOR  LDA 0,X        ; Get low byte
     EOR 2,X        ; XOR with other low byte
     STA 2,X        ; Put directly on stack
     LDA 1,X        ; Do the same for high bytes
     EOR 3,X
     STA 3,X
     JMP POP        ; Remove unneeded TOS item
```

Listing 8

```
;  C! routine
CSTR LDA 2,X        ; Get byte to store
     STA (0,X)      ; Store it!
     JMP POPTWO     ; Drop byte and address
```

Listing 9

The SETUP routine

A very useful routine in the FORTH system is the code routine SETUP. On the 6502, 0-page addressing is typically faster than absolute addressing. Also, some instructions, such as indirect-indexed addressing, can use only 0-page addresses. The SETUP routine allows the assembly language programmer to transfer up to four stack values to a scratch pad in the 0-page for these operations. The predefined name for this area is N. The calling sequence for the SETUP routine is:

```
LDA #num      ; Move "num" values to N, ("num" = 1-4)
JSR SETUP     ; then drop "num" values from the stack
```

The SETUP routine moves one to four values to the N scratch area and drops all values moved from the parameter stack. These values are stored in the following order:

```
LDA N       ; Low byte of value that was TOS
EOR N+1     ; High byte            ( N 1+ EOR, )
ADC N+2     ; Low byte of value that was 2OS
STY N+3     ; High byte            ( N 3 + STY, )
INC N+4     ; Low byte of 3OS      ( N 4 + INC, )
...
DEC N+7     ; High byte of value that was 4OS
```

Words such as CMOVE and FILL which use indirect-indexed addressing typically use the SETUP routine (see the BEGIN,...AGAIN, example). The word SETUP has been defined as a FORTH constant.

Return stack manipulation

The FORTH return stack is implemented as the normal 6502 hardware stack. To push and pop values, the 6502 stack instructions PHA and PLA can be used. Sometimes it is also necessary to manipulate the data on the return stack (such as for DO looping). Using the normal stack operations to do this can be tedious. Using indexed addressing, the return stack can be manipulated in the same manner as the parameter stack.

Examples:
```
;   >R routine
TOR  LDA 1,X        ; Pick up high byte
     PHA            ; Push it to R
     LDA 0,X        ; Now do the low byte
     PHA            ; It's done!
     JMP POP        ; Now, "lose" TOS
```

Listing 10

```
;   3rd loop index (I , I' , J , ... K )
K    STX XSAVE      ; Save P-stack pointer
     TSX            ; Get R-stack pointer
     LDA $109,X     ; 101-102,...,109-10A, (L-H)
     PHA            ; Push low byte of 3rd item
     LDA $10A,X     ; A now has high byte
     LDX XSAVE      ; Restore P-stack pointer
     JMP PUSH       ; Push the index
```

Listing 11

Machine Language Subroutines in valFORTH

When coding in assembly language, it is often useful to be able to make subroutine calls for often used operations. Using CODE makes it possible to do this, but it is not recommended. The following subroutine uses CODE.

```
CODE S1+                    ( Subroutine 1+      )
     0 ,X INC,              (    INC 0,X         )
     EQ IF,                 (    BNE *+4         )
        1 ,X INC,           (    INC 1,X         )
     ENDIF,                 (                    )
     RTS,                   (    RTS             )
C;
```

This subroutine could now be used in assembly language routines in the following way:

```
CODE 1+                     (Another 1+ routine  )
     ' S1+ JSR,             (   JSR S1+          )
     NEXT  JMP,             (   JMP NEXT         )
C;
```

This works fine, but there is one slight problem. If the user types S1+ as a command (i.e., it is not called, but executed) the FORTH system will "crash" when the RTS statement is encountered. This is because FORTH does not call its words, but jumps to them. For this reason, CODE is not used. A word which acts like CODE but protects the system is needed.

In the code for 1+ above, it was necessary to ' (tick) the subroutine to find its address. It would be desirable if we could simply type its name and have it return its address (just as NEXT and PUSH do). This is possible. The word SUBROUTINE below allows this (note that this word is not automatically loaded with the assembler, it must be typed in by the user).

```
: SUBROUTINE                  (new word SUBROUTINE     )
  0 VARIABLE                  (is like a VARIABLE      )
  -2 ALLOT                    (discard the value of 0  )
  [COMPILE] ASSEMBLER         (Put into assembly mode  )
  ?EXEC  !CSP ;               (Set/check for errors    )
```

The word SUBROUTINE can be used in the same way CODE is except that SUBROUTINEs end with an RTS instruction while CODE routines must end with a jump to a re-entry point. When the word defined using SUBROUTINE is executed, the entry point to the routine is left on the stack similar to the way in which a word defined using VARIABLE leaves an address. The following is an example of subroutine usage.

```
SUBROUTINE 2'SCOMP            (Two's complement  )
   SEC,                       (routine           )
   0  # LDA,
   0 ,X SBC,                  (i.e., TOS => - TOS)
   0 ,X STA,
   0  # LDA,
   1 ,X SBC,
   1 ,X STA,
   RTS,
C;
```

It can now be used as such:

```
CODE ABS                      (Take abs. value of TOS )
     1 ,X LDA,                (Is TOS < 0 ?           )
     MI IF,
        2'SCOMP JSR,          (If so, TOS => -TOS     )
     ENDIF,
     NEXT JMP,                (Exit to FORTH system   )
```

When the new word 2'SCOMP is executed directly, it leaves its address on the stack. When it is called by a subroutine, it performs a two's complement on the top stack value. This dual type of execution allows safe access to assembly language subroutines.

Macro Assemblies in valFORTH

FORTH assemblers use a reversed form of notation so that all the benefits of the standard FORTH system are available. In other words, anything that can be done in FORTH can be done during assembly time in a code definition. This is because all of the assembler opcodes are actually FORTH words which take arguments from the parameter stack. Thus

```
NEXT JMP,
```

actually puts the address of the NEXT routine (NEXT is a FORTH constant) onto the parameter stack. The word JMP, then compiles the address into the dictionary. Here is a simplified definition (it does not test for indirect jumping) for the JMP, opcode:

```
HEX
: JMP,                    (address --- )
  4C C,                   (compile in JMP opcode )
  ,                       (compile in the address )
; DECIMAL
```

All assembly words are designed in this fashion. Thus the necessity for operands to precede opcodes becomes clear. This allows the use of complex assembly time calculations that no ordinary assembler would ever support (e.g. no standard 6502 assembler would allow the use of the SIN function for generating a data table).

Most assemblers do allow the use of the basic operations: + , - , * , / , and &. These are easily used in the valFORTH system:

```
LDA #COUNT&$FF               COUNT FF AND # LDA,
LDY #NAME/$100               COUNT 100 /  # LDY,
EOR N+6                        N     6 +    EOR,
LDX #"A"+$80                 ASCII A 80 + # LDX,
etc.
```

The looping structures IF, and BEGIN, each leave two values on the stack during assembly time. The first is a branch address, the second is an identification code. When ENDIF, is executed, it checks the identification code to verify that structures have not been illegally interleaved (i.e., BEGIN, ... ENDIF,). If everything checks out, ENDIF, then calculates the branch offset required by the IF, clause, otherwise an error is reported. The BEGIN, clause functions in the same manner. Thus, the words IF, and BEGIN, are predefined macro instructions in the valFORTH assembler.

The fact that a FORTH assembler is nothing but a collection of words means that the assembler, like the FORTH language itself, is extensible. In other words, macro assemblies can easily be performed by defining new assembler directives. Take the following code extract which outputs a text string:

```
...code...                  ; ...
JSR CRLF                    ; Skip to next line
JSR PRTXT                   ; Call print routine
.BYTE 11,'valFORTH 1.'      ; String to output
LDA REL                     ; Get release number
JSR PRTNM                   ; Print the number
JSR CRLF                    ; Issue c/r
...code...                  ; ...
```

This code prints out the string "valFORTH 1.x" where "x" is the release number. Note that the routine PRTXT does not exist, it is simply used here for example purposes. The PRTXT routine "pops" the return address which points to the output string, picks up the length byte and adds it to the return address. The return address, which now points to the LDA instruction is "pushed" back onto the stack. The PRTXT routine still has a pointer to the string which it then prints out. Finally, it dones an RTS and returns control to the calling program. The release number is then printed out.

Assuming that the PRTXT routine is used quite often, it would be desirable to make it an assembler macro. A word which automatically assembles in the subroutine call to PRTXT and then assembles in a user specified string would be quite handy. In valFORTH, this is easily accomplished:

```
ASSEMBLER DEFINITIONS         (This is an assembler word)
HEX                           (Put system in base 16)
: PRINT"                      (Command form:  PRINT" text")
  20 C,  PRTXT ,              (compile in JSR PRTXT)
  22 WORD                     (Now the string upto ")
  HERE C@ 1+ ALLOT            (Bump dictionary pointer)
; DECIMAL                     (all done,)
IMMEDIATE                     (make word execute even at)
FORTH DEFINITIONS             (compile time.)
```

This word could now be used in ValFORTH assemblies in the following manner:

```
...code...
CRLF JSR,                   (Skip to next line)
PRINT" ValFORTH 1."         (Print out string)
REL LDA,                    (Get release number)
PRTNM JSR,                  (Go print it)
CRLF JSR,                   (Skip to next line)
...code...
```

Using the newly defined PRINT" macro, strings no longer need to be counted, and since there is less text to enter, typing errors are reduced. Other useful macros which could be designed are words which allow conditional assembly or automatically set up IOCB blocks for Atari operating system calls. Experienced assembly language programmers typically have a set of often used routines defined as macro instructions for quicker program development.

Compatability With Other Popular Assemblers

There are several other versions of FORTH out which have 6502 assemblers. The two major versions are the Forth Interest Group's written by William Ragsdale, and the version put out by the Atari Program Exchange written by Patrick Mullarky. The valFORTH assembler is a superset of both of these fine assemblers and is fully compatible with both versions.

Although not stated previously in the documentation, there are several ways in which to implement the IF, , WHILE, and UNTIL, structures. The valFORTH assembler was designed with transportability in mind. Although the recommended method is the valFORTH version, each of the following may be used.

valFORTH	Fig version	APX version
EQ IF,	0= IF,	IFEQ,
NE IF,	0= NOT IF,	IFNE,
CS IF,	CS IF,	IFCS,
CC IF,	CS NOT IF,	IFCC,
VS IF,	-------	IFVS,
VC IF,	-------	IFVC,
MI IF,	0< IF,	IFMI,
PL IF,	0< NOT IF,	IFPL,
EQ WHILE,	-------	-----
NE WHILE,	-------	-----
CC WHILE,	-------	-----
CC WHILE,	-------	-----
VS WHILE,	-------	-----
VC WHILE,	-------	-----
MI WHILE,	-------	-----
PL WHILE,	-------	-----
EQ UNTIL,	0= UNTIL,	0= UNTIL,
NE UNTIL,	0= NOT UNTIL,	0= NOT UNTIL,
CS UNTIL,	CS UNTIL,	-----
CC UNTIL,	CS NOT UNTIL,	-----
VS UNTIL,	-------	-----
VC UNTIL,	-------	-----
MI UNTIL,	0< UNTIL,	-----
PL UNTIL,	0< NOT UNTIL,	-----

Chart 2

In all versions, the word END, is synonymous with the word UNTIL,. Likewise, THEN, is synonymous with ENDIF,.

In the valFORTH and Fig assemblers, compiler security is performed to give added protection to the user against assembly errors. To accomplish this, the word C; or its synonym END-CODE is used to terminate the assembly word and perform the check. To remain compatible with APX FORTH, C; is not required in this release of valFORTH. However, it is strongly recommended that C; be used. Although C; and END-CODE are identical, C; is used in-house at Valpar for brevity. (Note that in later releases of valFORTH, C; will become mandatory).

There are several ways in which the indirect jump in the 6502 architecture is implemented in FORTH assemblers. The valFORTH assembler supports three common versions. Thus,

```
                JMP (VECTOR)
can be:
                VECTOR    )JMP,
                VECTOR  )  JMP,
          or    VECTOR   JMP(),
```

It is recommended that the first version be used.

It must be remembered that valFORTH's additional constructs may not be recognized by assemblers available from other vendors. If assembly listings are to be published for general 6502 FORTH users, it is suggested that valFORTH's advanced features not be used so that novice programmers can still make use of valuable pieces of code.

Appendix A — valFORTH Code Equivalents

This appendix gives the valFORTH assembly code for all 6502 code listings which are marked. Although listing 1 has already been translated to valFORTH assembly code, it is reproduced here for completeness.

Listing 1

```
DECIMAL
CODE CLR
     TYA,                      (Move a blank [0] into A)
     # 39 LDY,                 (Move count into Y)
     BEGIN,                    (Start looping)
        88 )Y STA,             (Move in a blank)
        DEY,                   (decrement pointer)
     MI UNTIL,                 (Go until count < 0)
     NEXT JMP,                 (Do a normal re-entry)
C;
```

Listing 2

```
CODE 1-                        (Decrement 16 bit value)
     0 ,X LDA,                 (Get the low byte)
     NE IF,                    (If a borrow will occur,)
        1 ,X DEC,              (then borrow from high...)
     ENDIF,
     0 ,X DEC,                 (Decrement low)
     NEXT JMP,                 (Re-enter FORTH)
C;
```

Listing 3

```
CODE DUP                       (Duplicate TOS)
     0 ,X LDA,                 (Get low byte)
     PHA,                      (Set up for PUSH)
     1 ,X LDA,                 (Put high in Accumulator)
     PUSH JMP,                 (Push 16 bit value)
C;
```

Listing 4

```
CODE SWAP                      (Exchange top stack items)
     2 ,X LDA,                 (Get low byte of 2OS)
     PHA,                      (Save it)
     0 ,X LDA,                 (Put low byte of TOS)
     2 ,X STA,                 (  into low byte of 2OS
     3 ,X LDA,                 (Save high byte of 2OS)
     1 ,X LDY,                 (Put high byte of TOS)
     3 ,X STY,                 (  into high byte of 2OS)
     PUT JMP,                  (Put old 2OS into TOS)
C;
```

Listing 5

```
     HEX
     CODE ?TERMINAL                         (Any console key pressed?)
          D01F LDA,                         (Load status byte)
          # 7 EOR,                          (Any low bits reset?
          NE IF,                            (If so,)
             INY,                           (then leave a true value)
          ENDIF,
          TYA,                              (Put true or false into A)
          PUSHOA JMP,                       (Push to parameter stack)
     C; DECIMAL
```

Listing 6

```
     CODE C@                                (Byte fetch routine)
          0 X) LDA,                         (Load from address on TOS)
          PUTOA JMP,                        (Push byte value)
     C;
```

Listing 7

```
     CODE XOR                               (One example of XOR)
          0 ,X LDA,                         (Get low byte of TOS)
          2 ,X EOR,                         (Exclusive or it with 2OS)
          PHA,                              (Push low result)
          1 ,X LDA,                         (Get high byte of TOS)
          3 ,X EOR,                         (XOR it with high of 2OS)
          BINARY JMP,                       (Drop TOS and replace 2OS)
     C;
```

Listing 8

```
     CODE XOR                               (Another exclusive or)
          0 ,X LDA,                         (Get low byte of TOS)
          2 ,X EOR,                         (XOR with low of 2OS)
          2 ,X STA,                         (Put in low of 2OS)
          1 ,X LDA,                         (Get high byte of TOS)
          3 ,X EOR,                         (XOR with high of 2OS)
          3 ,X STA,                         (Put in high of 2OS)
          POP JMP,                          (Drop TOS)
     C;
```

Listing 9

```
     CODE C!                                (Byte store routine)
          2 ,X LDA,                         (Pick up byte to store)
          0 )X STA,                         (Indirectly store it)
          POPTWO JMP,                       (Drop address and byte)
     C;
```

Listing 10

```
CODE >R                         (Transfer TOS to R-stack)
     1 ,X LDA,                  (Pick up high of TOS)
     PHA,                       (Put on R-stack)
     0 ,X LDA,                  (Pick up low of TOS)
     PHA,                       (Put on R-stack)
     POP JMP,                   (Lose top stack item)
C;
```

Listing 11

```
HEX
CODE K                          (3rd inner DO loop index)
     XSAVE STX,                 (Save P-stack pointer)
     TSX,                       (Pick up R-stack pointer)
     109 ,X LDA,                (Pick up low byte of value)
     PHA,                       (Save it)
     10A ,X LDA                 (Put high byte of value in A)
     XSAVE LDX,                 (Restore P-stack pointer)
     PUSH JMP,                  (Push 16 bit index value)
C; DECIMAL
```

Appendix B — Quick Reference Chart

valFORTH 6502 Assembly Words

ASSEMBLER (---)

Calls up the assembler vocabulary for subsequent assembly language programming.

CODE cccc (---)

Enters the new word "cccc" into the dictionary as machine language word and calls up the assembler vocabulary for subsequent assembly language programming. CODE also sets the system up for security checking.

C; (---)

Terminates an assembly language definition by performing a security check and setting the CONTEXT vocabulary to the same as the CURRENT vocabulary.

END-CODE (---)

A commonly used synonym for the word C; above. The word C; is recommended over END-CODE.

SUBROUTINE cccc (---)

Enters the new word "cccc" into the dictionary as machine language subroutine and calls up the assembler vocabulary for subsequent assembly language programming. SUBROUTINE also sets the system up for security checking.

;CODE (---)

When the assembler is loaded, puts the system into the assembler vocabulary for subsequent assembly language programming. See main glossary for further explanation.

Control Structures

IF, (flag --- addr 2)

Begins a machine language control structure based on the 6502 status flag on top of the stack. Leaves an address and a security check value for the ELSE, or ENDIF, clauses below. "flag" can be EQ , NE , CC , CS , VC , VS , MI , or PL . Command forms:

```
...flag..IF,..if-true..ENDIF,...all...
...flag..IF,..if-true..ELSE,..if-false..ENDIF,..all...
```

ELSE, (addr 2 --- addr 3)

Used in an IF, clause to allow for execution of code only if IF, clause is false. If the IF, clause is true, this code is bypassed. See IF, above for command form.

ENDIF, (addr 2/3 ---)

Used to terminate an IF, control structure clause. Additionally, ENDIF, resolves all forward references. See IF, above for command form.

BEGIN, (--- addr 1)

Begins machine language control structures of the following forms:

...BEGIN,...AGAIN,...
...BEGIN,...flag..UNTIL,...
...BEGIN,...flag..WHILE,..while-true..REPEAT,...

where "flag" is one of the 6502 statuses: EQ , NE , CC , CS , VC , VS , MI , and PL . See the very similar BEGIN in the main glossary for additional information.

UNTIL, (addr 1 flag ---)

Used to terminate a post-testing BEGIN, clause thus allowing for conditional looping of a program segment while "flag" is false. See BEGIN, above for more information.

WHILE, (addr 1 flag --- addr 4)

Used to begin a pre-testing BEGIN, clause thus allowing for conditional looping of a program segment while"flag" is true. See BEGIN, above for command format.

REPEAT, (addr 4 ---)

Used to terminate a pre-testing BEGIN,..WHILE, clause. Additionally, REPEAT, resolves all forward addresses of the current WHILE, clause. See BEGIN, above.

AGAIN, (addr 1 ---)

Used to terminate an unconditional BEGIN, clause. Execution cannot exit this loop unless a JMP, instruction is used. See BEGIN, clause for more information.

Parameter Passing

NEXT (--- addr)

Transfers control to the next FORTH word to be executed. The parameter stack is left unchanged.

PUSH (--- addr)

Pushes a 16 bit value to the parameter stack whose low byte is found on the 6502 return stack and whose high byte is found in the accumulator.

PUSHOA (--- addr)

Pushes a 16 bit value to the parameter stack whose low byte is found in the accumulator and whose high byte is zero.

PUT (--- addr)

Replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is found on the 6502 stack and whose high byte is in the accumulator.

PUTOA (--- addr)

Replaces the value currently on top of the parameter stack with the 16 bit value whose low byte is in the accumulator and whose high byte is set to zero.

BINARY (--- addr)

Drops the top value of the parameter stack and then performs a PUT operation described above.

POP and POPTWO (--- addr)

POP drops one value from the parameter stack. POPTWO drops two values from the parameter stack.

SETUP (--- addr)

Moves one to four values to the N scratch area in the zero page and drops all values moved from the parameter stack.

N (--- addr)

Points to a nine-byte scratch area in the zero page beginning at N-1 and going to N+7. Typically used by words which use indirect-indexed addressing where addresses must be stored in the zero page. See SETUP above.

Opcodes (---)

ADC,	AND,	ASL,	BIT,	BRK,	CLC,	CLD,	CLI,
CLV,	CMP,	CPX,	CPY,	DEC,	DEX,	DEY,	EOR,
INC,	INX,	INY,	JSR,	JMP,	LDA,	LDX,	LDY,
LSR,	NOP,	ORA,	PHA,	PHP,	PLA,	PLP,	ROL,
ROR,	RTI,	RTS,	SBC,	SEC,	SED,	SEI,	STA,
STX,	TAX,	TAY,	TSX,	TXA,	TXS,	TYA,	

Aliases

NXT,	=	NEXT JMP,
PSH,	=	PUSH JMP,
PUT,	=	PUT JMP,
PSHA,	=	PUSHOA JMP,
PUTA,	=	PUTOA JMP,
POP,	=	POP JMP,
POP2,	=	POPTWO JMP,
XL,	=	XSAVE LDX,
XS,	=	XSAVE STX,
THEN,	=	ENDIF,
END,	=	UNTIL,

VII. valFORTH 1.1 SUPPLIED SOURCE LISTING

```
Screen:  30
( Auto command                    )

BASE @ HEX

: ZAP                 ( addr # -- )
  -DUP
  IF 0+S
   DO D20A C@ 7F AND I C!
   LOOP
  ELSE DROP
  ENDIF ;

: -WAIT                      ( -- )
   BEGIN ?TERMINAL NOT UNTIL ;

DCX                              ==>
```

```
Screen:  31
( Auto command                    )

: BEHEAD                     ( -- )
  0 >R CR ." Now protecting..."
  CR VOC-LINK @
  BEGIN
   DUP 2- @
   BEGIN
    DUP 1+ OVER R> 1+ >R
    R 15 MOD NOT IF ." ." ENDIF
    R 495 MOD NOT IF CR ENDIF
    C@ 63 AND WIDTH @ MIN 1-
    ZAP PFA LFA @ DUP NOT
   UNTIL
   DROP @ DUP NOT
  UNTIL R> 2DROP ;               -->
```

```
Screen:  32
( Auto command                    )

: QUEST                      ( -- )
  CR
  ." Save on disk:           "
  ." press OPTION" CR
  ." Save on cassette:       "
  ." press SELECT" CR
  ." Exit:                   "
  ." press START" CR CR ;

: DODISK                     ( -- )
  (SAVE) ' SAVE 32 + @EX
  741 @ 128 - 1 1 R/W ;

                                 ==>
```

```
Screen:  33
( Auto command                    )

: QUEST2                   ( -- n )
  ." Format and save:      "
  ." press OPTION" CR
  ." Just save:            "
  ." press SELECT" CR CR
  WAIT ?TERMINAL -WAIT
  ." Prepare disk -- "
  ." press START"
  WAIT -WAIT ;
: CSV                        ( -- )
  ." Prepare cassette "
  ." (play/record) -- " CR
  ." press START"  CR
  WAIT CSAVE -WAIT ;             -->
```

```
Screen:  34
( Auto command                    )

: DSV
  QUEST2
  4 =
  IF
   1 (FMT) 1 <>
   IF
    CR ." Format error"
   ELSE
    DODISK
   ENDIF
  ELSE
   DODISK
  ENDIF CR ;
                                 ==>
```

```
Screen:  35
( Auto command                    )

: DECIS                      ( -- )
  BEGIN
   QUEST WAIT ?TERMINAL -WAIT
   DUP 1 =
   IF DROP 1
   ELSE
    2 =
    IF
     CSV
    ELSE
     DSV
    ENDIF 0
   ENDIF
  UNTIL ;                        -->
```

Screen: 36

```
( Auto command                          )

: AUTO                           ( -- )
  [COMPILE] ' CR
  ." Auto? Y/N " KEY 89 = CR
  IF
   CFA ' ABORT 6 + !
   ' COLD CFA ' ABORT 8 + !
   -1 26 +ORIGIN !
   ' ZAP NFA DP !
   BEHEAD DECIS ABORT
  ELSE
   DROP ." Auto aborted..." CR
  ENDIF ;                   BASE !
```

Screen: 37

Screen: 38

```
( Text output:  S:  P:                 )

BASE @ 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 ! ;


                                    ==>
```

Screen: 39

```
( Text output:  BEEP  ASCII           )

: BEEP                           ( -- )
  0C0 0
  DO
    08 0D01F C! 6 0 DO LOOP
    00 0D01F C! 6 0 DO LOOP
  LOOP ;

: ASCII                ( ccc, -- (b))
  BL WORD
  HERE 1+ C@
  STATE @
  IF
    COMPILE CLIT C,
  ENDIF ;  IMMEDIATE              -->
```

Screen: 40

```
( Text output:  EJECT  LISTS          )
DCX

: EJECT                          ( -- )
  12 EMIT ;

: LISTS                      ( s # -- )
  0 <ROT O+S
  DO
    CR I LIST
    1+ DUP 3 MOD 0=
    IF EJECT ENDIF
    ?EXIT
  LOOP
  DROP ;
                                    ==>
```

Screen: 41

```
( Text output:  PLISTS  PLIST         )

: PLISTS                     ( s # -- )
  PFLAG @ <ROT
  ON P:
  LISTS
  CR PFLAG ! ;

: PLIST                        ( s -- )
  1 PLISTS ;






                              BASE !
```

Screen: 42

```
( Debug:  B?  [FREE]  FREE      )
BASE @ DCX
'( S: )( 19 KLOAD )
HEX

: B?                          ( -- )
  BASE @ DUP           ( Display )
  DECIMAL .            ( current )
  BASE ! ;             ( radix )

: (FREE)                     ( -- n )
  2E5 @ PAD - ;

: FREE                        ( -- )
  (FREE) U. ." bytes" CR ;
                                 ==>
```

Screen: 43

```
( Debug:  H.  CFALIT            )

: CFALIT           ( ccc, -- (b))
  STATE @
  [COMPILE] [
  [COMPILE] ' CFA
  SWAP IF [COMPILE] ] ENDIF
  [COMPILE] LITERAL ;
  IMMEDIATE

'( H. --> )(  )
: H.                       ( b -- )
  BASE @ HEX           ( Display )
  SWAP 0               ( # in hex )
  <# # # #> TYPE
  BASE ! ;                       -->
```

Screen: 44

```
( Debug:  #DUMP                 )

( memory dump  )

: #DUMP                  ( a # -- )
  O+S
  DO
    CR I 5 U.R I
    DUP 8 + SWAP
    DO
      I C@ 4 .R
    LOOP
    ?EXIT
  8 /LOOP
  CR ;
                                 ==>
```

Screen: 45

```
( Debug:  CDUMP                 )

( Character dump routine        )

: CDUMP                  ( a # -- )
  PFLAG @ <ROT OFF P:
  OVER + SWAP
  DO
    CR I 5 U.R
    SPACE I DUP 10 + SWAP
    DO
      I C@ SPEMIT
    LOOP
    ?EXIT
  10 /LOOP
  CR PFLAG ! ;                   -->
```

Screen: 46

```
( Stack print:  DEPTH           )

: DEPTH                    ( n -- )
  S0 @ SP@ - 2/ 1- ;

CFALIT . VARIABLE X.S

: XDOTS                      ( -- )
  DEPTH
  IF
    SP@ 2- S0 @ 2-
    DO I @ X.S @ EXECUTE
    -2 +LOOP
  ELSE
    ." Stack empty "
  ENDIF ;                        ==>
```

Screen: 47

```
( Stack print:  .S  U.S  STACK )

: .S    CFALIT  . X.S ! XDOTS ;
: U.S   CFALIT U. X.S ! XDOTS ;

: STKPRT
  CR ." ( " XDOTS ." ) " ;

: STACK                    ( f -- )
  IF
   CFALIT STKPRT
  ELSE
   CFALIT NOOP
  ENDIF
  [ ' PROMPT 11 + ]
  LITERAL ! ;                    -->
```

```
Screen:  48
( FORTH colon decompiler         )


0 VARIABLE .WORD

: PWORD
  2+ NFA ID. ;

: 1BYTE
  PWORD .WORD @ C@ .
  1 .WORD +! ;

: 1WORD
  PWORD .WORD @ @ .
  2 .WORD +! ;
                                 ==>
```

```
Screen:  49
( FORTH colon decompiler         )
: NP                      ( n -- n )
  DUP CFALIT ;S = OVER
  CFALIT (;CODE) = OR
  IF PWORD CR CR PROMPT QUIT
  ENDIF ?TERMINAL
  IF DROP PROMPT QUIT ENDIF ;

: BRNCH
  PWORD ." to " .WORD @ .WORD @
  @ + U. 2 .WORD +! ;

: NXT1
  .WORD @ U. 2 SPACES
  .WORD @ @ 2 .WORD +! ;
                                 -->
```

```
Screen:  50
( FORTH colon decompiler         )

: STG
  PWORD 22 EMIT .WORD @
  DUP COUNT TYPE 22 EMIT
  C@ .WORD @ + 1+ .WORD ! ;

: CKIT
  DUP  CFALIT 0BRANCH =
  OVER CFALIT BRANCH  = OR
  OVER CFALIT (LOOP)  = OR
  OVER CFALIT (+LOOP) = OR
  OVER CFALIT (/LOOP) = OR

  IF BRNCH
  ELSE DUP CFALIT LIT =
    IF 1WORD                     ==>
```

```
Screen:  51
( FORTH colon decompiler         )

    ELSE
      DUP CFALIT CLIT =

      IF 1BYTE
      ELSE
        DUP CFALIT COMPILE =
        IF PWORD CR NXT1 PWORD
        ELSE
          DUP 4 - @ A922 =
          IF STG
          ELSE PWORD ENDIF
        ENDIF
      ENDIF
    ENDIF
  ENDIF ;                        -->
```

```
Screen:  52
( FORTH colon decompiler         )

: ?DOCOL
  DUP 2- @
  [ ' : 12 + ] LITERAL -
  IF ." Primitive pfa dump:"
    2- @ 18 #DUMP
    PROMPT QUIT
  ENDIF ;







                                 ==>
```

```
Screen:  53
( FORTH colon decompiler         )

: DCMPR                   ( PFA -- )
  DUP NFA CR CR DUP ID.
  C@ 40 AND
  IF ." (IMMEDIATE)"
  ENDIF
  CR CR ?DOCOL .WORD !
  BEGIN NXT1 NP CKIT CR AGAIN ;

: DECOMP
  [COMPILE] ' DCMPR ;



                        BASE ! ;S
```

Screen: 54

Screen: 55

Screen: 56

```
( fig editor:  TEXT  LINE      )
BASE @  HEX
( This editor is based on the   )
( example editor supplied in    )
( the "fig-FORTH Installation   )
( Manual."                      )
: TEXT
  HERE C/L 1+ BLANKS WORD
  HERE PAD C/L 1+ CMOVE ;
: LINE
  DUP FFF0 AND 17 ?ERROR
  SCR @ (LINE) DROP ;          ==>
```

Screen: 57

```
( fig editor:  MARK            )
: MARK
  10 0
  DO
    I LINE UPDATE
    DROP
  LOOP ;
                               -->
```

Screen: 58

```
( fig editor:  WHERE           )
VOCABULARY EDITOR IMMEDIATE
( Note: the fig bug is fixed )
(       in WHERE below.      )
: WHERE                 ( n n -- )
  2DUP DUP B/SCR / DUP SCR !
  ." Scr # " DECIMAL . SWAP
  C/L /MOD C/L * ROT BLOCK +
  CR C/L -TRAILING TYPE
  CR HERE C@ - 2- 0 MAX SPACES
  1 2FE C! 1C EMIT 0 2FE C!
  [COMPILE] EDITOR QUIT ;
                               ==>
```

Screen: 59

```
( fig editor:  #L's  -MOVE     )
EDITOR DEFINITIONS
: #LOCATE               ( -- n n )
  R# @ C/L /MOD ;
: #LEAD                 ( -- n n )
  #LOCATE LINE SWAP ;
: #LAG                  ( -- n n )
  #LEAD DUP >R + C/L R> - ;
: -MOVE                 ( n n -- )
  LINE C/L CMOVE UPDATE ;
                               -->
```

Screen: 60

```
( fig editor:  H  E  S          )
: H                     ( n -- )
  LINE PAD 1+ C/L
  DUP PAD C! CMOVE ;
: E                     ( n -- )
  LINE C/L BLANKS UPDATE ;
: S                     ( n -- )
  DUP 1- 0E
  DO
    I LINE
    I 1+ -MOVE
  -1 +LOOP
  E ;                          ==>
```

Screen: 61

```
( fig editor:  D  M             )
: D                     ( n -- )
  DUP H 0F DUP ROT
  DO
    I 1+ LINE
    I -MOVE
  LOOP
  E ;
: M                     ( n -- )
  R# +! CR SPACE
  #LEAD TYPE 14 EMIT #LAG
  TYPE #LOCATE . DROP ;
                               -->
```

Screen: 62

```
( fig editor:  T  L  R  P       )
: T                     ( n -- )
  DUP C/L * R# !
  DUP H 0 M ;
: L                       ( -- )
  SCR @ LIST 0 M ;
: R                     ( n -- )
  PAD 1+ SWAP -MOVE ;
: P                     ( n -- )
  1 TEXT R ;
                               ==>
```

Screen: 63

```
( fig editor:  I  TOP           )
: I                     ( n -- )
  DUP S R ;
: TOP                     ( -- )
  0 R# ! ;
                               -->
```

Screen: 64

```
( fig editor:  CLEAR  COPY      )
: CLEAR                 ( n -- )
  SCR ! 10 0
  DO
    FORTH I EDITOR E
  LOOP ;
: COPY                ( n n -- )
  B/SCR * OFFSET @ + SWAP
  B/SCR * B/SCR OVER + SWAP
  DO
    DUP FORTH I BLOCK
    2- ! 1+ UPDATE
  LOOP
  DROP ;                       ==>
```

Screen: 65

```
( fig editor:  1LINE  FIND      )
: 1LINE                 ( -- f )
  #LAG PAD COUNT
  MATCH R# +! ;
: FIND                    ( -- )
  BEGIN
    3FF R# @ <
    IF
      TOP PAD HERE C/L
      1+ CMOVE 0 ERROR
    ENDIF
    1LINE
  UNTIL ;
                               -->
```

Screen: 66

```
( fig editor:  DELETE                )

: DELETE                    ( n -- )
  >R #LAG + FORTH R -
  #LAG R MINUS R# +! #LEAD
  + SWAP CMOVE R> BLANKS
  UPDATE ;








                                  ==>
```

Screen: 67

```
( fig editor:  N  F  B  X           )

: N                          ( -- )
  FIND 0 M ;

: F                          ( -- )
  1 TEXT N ;

: B                          ( -- )
  PAD C@ MINUS M ;

: X                          ( -- )
  1 TEXT FIND
  PAD C@ DELETE
  0 M ;
                                  -->
```

Screen: 68

```
( fig editor:  TILL  C              )

: TILL                       ( -- )
  #LEAD + 1 TEXT
  1LINE 0= 0 ?ERROR
  #LEAD + SWAP -
  DELETE 0 M ;

: C
  1 TEXT PAD COUNT #LAG ROT
  OVER MIN >R FORTH R R# +!
  R - >R DUP HERE R CMOVE
  HERE #LEAD + R> CMOVE R>
  CMOVE UPDATE 0 M ;

                                  ==>
```

Screen: 69

```
( End of fig-FORTH editor           )

FORTH DEFINITIONS DCX













                       BASE !
```

Screen: 70

Screen: 71

Screen: 72

```
( Disk copy routines                 )
BASE @ DCX

0 VARIABLE SEC/PAS
0 VARIABLE SECNT

: AXLN                    ( system )
  4 PICK 0
  DO 3 PICK I 128 * +
       3 PICK I + 3 PICK R/W
  LOOP 2DROP 2DROP ;

: DCSTP
  741 @ PAD DUP 1 AND - -
  0 128 U/ SWAP DROP
  SEC/PAS ! 0 SECNT ! ;          ==>
```

Screen: 73

```
( Disk copy routines                 )

: DISKCOPY1                   ( -- )
  DCSTP
  BEGIN
   CR CR ." Insert source and pu
sh START" WAIT
   720 SECNT @ - SEC/PAS @ MIN
   DUP >R PAD DUP 1 AND - SECNT
   @ 2DUP 5 PICK <ROT 1 AXLN
   CR CR ." Insert dest.  and pu
sh START" WAIT 0 AXLN CR
   R> SECNT +! SECNT @ DUP .
   ." sectors copied" 720 =
  UNTIL EMPTY-BUFFERS
  CR ." Done" CR ;               -->
```

Screen: 74

```
( Disk copy routines                 )

: DISKCOPY2                   ( -- )
  DCSTP
  CR ." Insert source in drive 1
" CR ." Insert dest.  in drive 2
" CR ." Press START to copy"
  WAIT
  BEGIN
   720 SECNT @ - SEC/PAS @ MIN
   DUP >R PAD DUP 1 AND - SECNT
   @ 2DUP 5 PICK <ROT
   1 AXLN 720 + 0 AXLN
   R> SECNT +! SECNT @ 720 =
  UNTIL EMPTY-BUFFERS
  CR ." Done" CR ;    BASE !
```

Screen: 75

Screen: 76

```
( 6502 Assembler in FORTH      )
(
( Originally written by
( Patrick Mullarky.
(
( Modified extensively 2/82
( by Stephen Maguire,
( Valpar International
(
(
( This assembler conforms to the
( fig "INSTALLATION GUIDE" and
( to APX versions of FORTH.
(
(   )
                                ==>
```

Screen: 77

```
( 6502 Assembler in FORTH      )
(
( Now supports:
(
( IF,...ELSE,...ENDIF,
( BEGIN,...WHILE,...REPEAT,
( BEGIN,...AGAIN,
( BEGIN,...any flag UNTIL,
( C; & END-CODE
( ;CODE
(
( Also supports:
(
( compiler security
( definition checking           )
                                -->
```

Screen: 78

```
( 6502 Assembler in FORTH      )
'( TRANSIENT TRANSIENT )(   )
BASE @ HEX
ASSEMBLER DEFINITIONS

: SB
  <BUILDS C, DOES> @ C, ;

 000 SB BRK,     018 SB CLC,
 0D8 SB CLD,     058 SB CLI,
 0B8 SB CLV,     0CA SB DEX,
 088 SB DEY,     0E8 SB INX,
 0C8 SB INY,     0EA SB NOP,
 048 SB PHA,     008 SB PHP,
 068 SB PLA,     028 SB PLP,
 040 SB RTI,     060 SB RTS,   ==>
```

Screen: 79

```
( 6502 Assembler in FORTH      )
 038 SB SEC,     0F8 SB SED,
 078 SB SEI,     0AA SB TAX,
 0BA SB TSX,     08A SB TXA,
 09A SB TXS,     098 SB TYA,
 0A8 SB TAY,

0 VARIABLE )J   : ) 1 )J ! ;

: 3BY
  <BUILDS C, DOES> C@ DUP 4C =
  IF )J @ IF DROP 6C ENDIF
  ENDIF C, , 0 )J ! ;

 04C 3BY JMP,  06C 3BY JMP(),
 020 3BY JSR,  06C 3BY )JMP,  -->
```

Screen: 80

```
( 6502 Assembler in FORTH      )

: 256<  DUP 100 ( HEX ) U< ;

70 CONSTANT VC  ( over clear )
50 CONSTANT VS  ( over set )
B0 CONSTANT CC  ( carry clear )
90 CONSTANT CS  ( carry set )
D0 CONSTANT EQ  ( zero )
F0 CONSTANT NE  ( non-zero )
30 CONSTANT PL  ( positive )
   1 ?PAIRS 4C C, , ;  IMMEDIATE

: IF,
  C, 0 C, HERE 2 ;     IMMEDIATE
                               ==>
```

Screen: 81

```
( 6502 Assembler in FORTH      )

: ENDIF,
  DUP 2 = IF
    DROP DUP HERE SWAP -
    DUP  7F > 5 ?ERROR
    DUP -80 < 5 ?ERROR
    SWAP 1- C!
  ELSE
    3 ?PAIRS HERE SWAP !
  ENDIF ;                IMMEDIATE

: ELSE,
  DUP 2 ?PAIRS 4C C, HERE 0 ,
  <ROT [COMPILE] ENDIF, 3 ;
                               -->
```

Screen: 82

```
( 6502 Assembler in FORTH      )

: THEN,
  [COMPILE] ENDIF, ;  IMMEDIATE

: BEGIN,
  HERE 1 ;            IMMEDIATE

: UNTIL,
  SWAP 1 ?PAIRS C,
  HERE 1+ - DUP -80
  < 5 ?ERROR C, ;     IMMEDIATE

: END,
  [COMPILE] UNTIL, ;  IMMEDIATE
                               ==>
```

Screen: 83

```
( 6502 Assembler in FORTH      )

: WHILE,
  SWAP 1 ?PAIRS [COMPILE] IF,
  DROP 4 ;            IMMEDIATE

: REPEAT,
  4 ?PAIRS SWAP 4C C, , 2
  [COMPILE] ENDIF, ;  IMMEDIATE

10 CONSTANT MI  ( negative )

: END-CODE
  [COMPILE] C; ;      IMMEDIATE
                               -->
```

Screen: 84

```
( 6502 Assembler in FORTH          )
0D VARIABLE MODE    ( ABS mode )
00 VARIABLE ACC     ( A-reg? )

: BIT,
  256< IF 24 C, C,
  ELSE 2C C, , ENDIF ;

: CKMODE
  MODE @ =
  IF          ( MODE = MODE - 8 )
    256<      ( if addr < 256    )
    IF
      -08 MODE +!
    ENDIF
  ENDIF ;                      ==>
```

Screen: 85

```
( 6502 Assembler in FORTH          )

: M0
  <BUILDS
     C,
  DOES>
     SWAP 0D CKMODE
     1D CKMODE SWAP
     C@ MODE @ OR C,
     256< IF C, ELSE , ENDIF
     0D MODE ! ;    ( ABS mode )

  00 M0 ORA,       20 M0 AND,
  40 M0 EOR,       60 M0 ADC,
  80 M0 STA,       A0 M0 LDA,
  C0 M0 CMP,       E0 M0 SBC,   -->
```

Screen: 86

```
( 6502 Assembler in FORTH          )
: !ADDR   C, 256< IF C, ELSE ,
  ENDIF 0D MODE ! ;

: ZPAGE
  OVER 100 U< IF F7 AND ENDIF ;

: M1
  <BUILDS C, DOES> C@ ACC @
  IF FB AND C, ELSE MODE @ 1D =
  IF 10 ELSE 0 ENDIF OR ZPAGE
  !ADDR ENDIF 0 ACC ! ;

 00E M1 ASL,     02E M1 ROL,
 04E M1 LSR,     06E M1 ROR,
 0CE M1 DEC,     0EE M1 INC,   ==>
```

Screen: 87

```
( 6502 Assembler in FORTH          )

: OPCODE
  C@ ZPAGE MODE @ 1D =
  MODE @ 19 = OR
  IF 10 OR ENDIF ;

: M2
  <BUILDS C,
  DOES> OPCODE MODE @ 9 =
  IF 4 - ENDIF !ADDR ;

: M3
  <BUILDS C,
  DOES> OPCODE !ADDR ;
                               -->
```

Screen: 88

```
( 6502 Assembler in FORTH          )
  0AC M2 LDY,     0AE M2 LDX,
  0CC M2 CPY,     0EC M2 CPX,
  08C M3 STY,     08E M3 STX,

: X)   01 MODE ! ;   ( [addr,X] )
: #    09 MODE ! ;   ( immediate )
: )Y   11 MODE ! ;   ( [addr],Y  )
: ,X   1D MODE ! ;   ( addr,X )
: ,Y   19 MODE ! ;   ( addr,Y )
: .A   01 ACC  ! ;   ( a - reg )

0A SB ASL.A,    2A SB ROL.A,
4A SB LSR.A,    6A SB ROR.A,

                               ==>
```

Screen: 89

```
( 6502 Assembler in FORTH          )

: IFVC,  VC [COMPILE] IF, ;
: IFVS,  VS [COMPILE] IF, ;
: IFCC,  CC [COMPILE] IF, ;
: IFCS,  CS [COMPILE] IF, ;
: IFEQ,  EQ [COMPILE] IF, ;
: IFNE,  NE [COMPILE] IF, ;
: IFPL,  PL [COMPILE] IF, ;
: IFMI,  MI [COMPILE] IF, ;

: 0= EQ ; : 0<  MI ; : >= EQ ;
: NOT 20 XOR ;   : RP) 101 ,X ;
: BOT 0 ,X ; : SEC 2 ,X ;

                               -->
```

Screen: 90

```
( End of 6502 assembler        )
HEX
: XS,     XSAVE STX,  ;
: XL,     XSAVE LDX,  ;
: NXT,    NEXT JMP,  ;
: POP,    POP JMP,  ;
: POP2,   POPTWO JMP,  ;
: PSH,    PUSH JMP,  ;
: PSHA,   PUSH0A JMP,  ;
: PUT,    PUT JMP,  ;
: PUTA,   PUT0A JMP,  ;
FORTH DEFINITIONS
'( PERMANENT PERMANENT )(   )
                       BASE !
```

Screen: 91

Screen: 92

```
( FORMAT                          )
BASE @ HEX
: FORMAT
  CR CR ." Enter Drive#: " KEY
  DUP EMIT 30 - 1 MAX 4 MIN CR
 ." Hit RETURN to format drive "
  DUP . CR
 ." Hit any other key to abort "
  KEY 9B =
  IF (FMT) 1 = CR CR ." Format "
   IF ." OK" ELSE ." ERROR"
   ENDIF
  ELSE CR ." Format aborted..."
    DROP
  ENDIF CR CR ;        BASE !
```

Screen: 93

Screen: 94

```
( Buffer relocation               )
BASE @ DCX
: RELOCBUFS            ( addr -- )
  DUP 1 AND
  IF CR ." Odd buffer address."
     CR ." Try again." DROP QUIT
  ENDIF
  DUP          ' FIRST !
  DUP 2112 + ' LIMIT !
  DUP            PREV  !
  DUP            USE   !
  MTB CR 156 EMIT 156 EMIT
  ." Buffers relocated to "
  U.  ." and emptied" CR ;
                                ==>
```

Screen: 95

```
( Buffer relocation               )
CR CR ." The buffers take 2112 b
ytes decimal." CR
CR ." To relocate buffers, put t
he new addr on stack and do:" CR
 CR 7 SPACES ." RELOCBUFS FORGET
 RELOCBUFS" CR CR   BASE !
```

Screen: 96

Screen: 97

Screen: 98

Screen: 99

Screen: 100

```
( Colors:  hue CONSTANTs          )

BASE @ DCX

  0 CONSTANT GREY
  1 CONSTANT GOLD
  2 CONSTANT ORNG
  3 CONSTANT RDORNG
  4 CONSTANT PINK
  5 CONSTANT LVNDR
  6 CONSTANT BLPRPL
  7 CONSTANT PRPLBL
  8 CONSTANT BLUE
  9 CONSTANT LTBLUE
 10 CONSTANT TURQ
 11 CONSTANT GRNBL                      ==>
```

Screen: 101

```
( Colors:  hue CONSTANTs          )

 12 CONSTANT GREEN
 13 CONSTANT YLWGRN
 14 CONSTANT ORNGRN
 15 CONSTANT LTORNG










                              BASE !  -->
```

```
Screen: 102
( Colors:  SETCOLOR  BOOTCOLOR )
BASE @ DCX

: SETCOLOR        ( # hue lum -- )
  SWAP 16 * OR SWAP
  708 ( COLPF0 ) + C! ;

: SE.  SETCOLOR ;

: BOOTCOLOR         ( hue lum -- )
  SWAP 16 * DUP 4 + DUP
  [ ' COLD 35 + ] LITERAL C!
  710 C! OR DUP
  [ ' COLD 40 + ] LITERAL C!
  709 C! ;
                          BASE !
```

```
Screen: 103
```

```
Screen: 104
( Graphics:  CGET                 )

BASE @ DCX '( )SCD )( 68 KLOAD )

HEX
CODE CGET                ( -- b )
  B5 C, 00 C, 48 C,
  86 C, XSAVE C,
  A2 C, 30 C, A9 C, 07 C,
  9D C, 342 , 98 C,
  9D C, 348 , 9D C, 349 ,
  68 C, 20 C, CIO ,
  A6 C, XSAVE C,
  4C C, PUSH0A ,
C;
                                 ==>
```

```
Screen: 105
( Graphics:  COLOR  POS.  LOC. )

0 VARIABLE CLRBYT

: COLOR                    ( b -- )
  CLRBYT ! ;

: POS.                   ( h v -- )
  54 C! 55 ! ;

: POSITION POS. ;        ( h v -- )

: LOC.                 ( x y -- b )
  POS. CGET ;

                                 -->
```

```
Screen: 106
( Graphics:  CPUT                 )

HEX
CODE CPUT                  ( b -- )
  B5 C, 00 C, 48 C,
  86 C, XSAVE C,
  A2 C, 30 C, A9 C, 0B C,
  9D C, 342 , 98 C,
  9D C, 348 , 9D C, 349 ,
  68 C, 20 C, CIO ,
  A6 C, XSAVE C,
  4C C, POP ,
C;

                                 ==>
```

```
Screen: 107
( Graphics:  POS@  POSIT  PLOT )

: POS@                   ( -- h v )
  55 @ 54 C@ ;

: POSIT                  ( h v -- )
  POS. 54 C@ 5A C!
       55  @ 5B  ! ;

: PLOT                 ( b h v -- )
  POS. CLRBYT C@ CPUT ;

                                 -->
```

Screen: 108

```
( Graphics:  GTYPE             )

: GTYPE            ( cnt adr -- )
  0 MAX -DUP
  IF O+S
   DO I C@ >SCD CLRBYT C@
      40 * OR SCD> CPUT
   LOOP
  ENDIF ;






                               ==>
```

Screen: 109

```
( Graphics:  [G"]   G"          )

: (G")                   ( -- )
  R COUNT DUP 1+ R> + >R
  GTYPE ;

: G"                     ( -- )
  22 STATE @
  IF
    COMPILE (G")
    WORD HERE C@ 1+ ALLOT
  ELSE
    WORD HERE COUNT GTYPE
  ENDIF  ; IMMEDIATE

                               -->
```

Screen: 110

```
( Graphics:  GCOM  DR.  FIL     )

CODE GCOM                ( n -- )
  86 C, D1 C, B5 C, 00 C,
  A2 C, 30 C, 9D C, 342 ,
  20 C, CIO , A6 C, D1 C,
  4C C, POP ,

: DR.                  ( x y -- )
  CLRBYT C@ 2FB C!
  POS. 11 GCOM ;

: DRAWTO DR. ;

: FIL              ( fildat -- )
  2FD C!  12 GCOM ;   BASE !
```

Screen: 111

Screen: 112

```
( Graphics Demo                    )
BASE @ DCX

: BOX
  1 COLOR 20 10 POSIT
  50 10 DR.  50 28 DR.
  20 28 DR.  20 10 DR.  ;

: FBOX
  5 GR. BOX
  20 28 POS. 2 FIL ;


( LOAD THIS SCREEN AND EXECUTE )
( FBOX.  WHEN YOU'RE DONE, DO  )
( FORGET BOX )          BASE !
```

Screen: 113

Screen: 114

```
( 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: 115

```
( Sound:  XSND  XSND4           )

DCX

: XSND                ( voice# -- )
  2* 53761 +
  0 SWAP C! ;

: XSND4                     ( -- )
  53760 8 0 FILL
  0 FILTER! ;




                       BASE !
```

Screen: 116

Screen: 117

Screen: 118

Screen: 119

Screen: 120

```
( Floating:  FDROP FDUP FSWAP   )
BASE @  HEX
CODE FDROP                        ( fp --)
  INX, INX, POPTWO JMP,
C;
CODE FDUP              ( fp -- fp fp )
  # 6 LDY,
  BEGIN,
   DEX,  6 ,X LDA,   0 ,X STA,
   DEY, 0=
  UNTIL, NEXT JMP,
C;
                                    ==>
```

Screen: 121

```
( Floating:  FSWAP  FOVER       )
CODE FSWAP ( fp1 fp2 -- fp2 fp1)
  XSAVE STX,   # 6 LDY,
  BEGIN,
   0 ,X LDA,   PHA,   6 ,X LDA,
   0 ,X STA,   PLA,   6 ,X STA,
   INX, DEY, 0=
  UNTIL, XSAVE LDX, NEXT JMP, C;
CODE FOVER ( fp fp -- fp fp fp )
  # 6 LDY,
  BEGIN,
   DEX,   0C ,X LDA,   0 ,X STA,
   DEY, 0=
  UNTIL, NEXT JMP, C;          -->
```

Screen: 122

```
( Floating:  conversion words   )
CODE AFP
  XS, D800 JSR, XL, NXT,
C;
CODE FASC
  XS, D8E6 JSR, XL, NXT,
C;
                                    ==>
```

Screen: 123

```
( Floating:  FADD FSUB FMUL ...)
CODE IFP  XS, D9AA JSR, XL, NXT,
CODE FPI  XS, D9D2 JSR, XL, NXT,
CODE FADD XS, DA66 JSR, XL, NXT,
CODE FSUB XS, DA60 JSR, XL, NXT,
CODE FMUL XS, DADB JSR, XL, NXT,
CODE FDIV XS, DB28 JSR, XL, NXT,
CODE FLG  XS, DECD JSR, XL, NXT,
                                  -->
```

Screen: 124

```
( Floating:  FLG10  FEX  FPOLY )
CODE FLG10
  XS, DED1 JSR, XL, NXT,  C;
CODE FEX
  XS, DDC0 JSR, XL, NXT,  C;
CODE FEX10
  XS, DDCC JSR, XL, NXT,  C;
CODE FPOLY
  XS,  0 ,X LDA,   PHA,
  3 ,X LDA,   XSAVE LDY,
  2 ,Y LDX,   TAY,   PLA,
  DD40 JSR,   XL, POP2,  C;   ==>
```

Screen: 125

```
( Floating:  system constants   )
D4 CONSTANT FR0
E0 CONSTANT FR1
F3 CONSTANT INBUF
F2 CONSTANT CIX
                                  -->
```

```
Screen: 126
( Floating:  F@  F!  F.TY       )
: F@                     ( a -- fp )
  >R R @ R 2+
  @ R> 4 + @ ;
: F!                     ( fp a -- )
  >R R 4 + !
  R 2+ ! R> ! ;
: F.TY                      ( a -- )
  BEGIN
    INBUF @ C@ DUP
    7F AND EMIT
    1 INBUF +! 80 >
  UNTIL ;                        ==>
```

```
Screen: 127
( Floating:  F.  F?  <F  >F     )
: F.                       ( fp -- )
  FR0 F@ FSWAP FR0 F! FASC
  F.TY SPACE FR0 F! ;
: F?                        ( a -- )
  F@ F. ;
: <F                    ( fp fp -- )
  FR1 F! FR0 F! ;
: >F                       ( -- fp )
  FR0 F@ ;
                                 -->
```

```
Screen: 128
( Floating:  FS  floating +-*/ )
: FS                       ( fp -- )
  FR0 F! ;
: F+                 ( fp fp -- fp )
  <F FADD >F  ;
: F-                 ( fp fp -- fp )
  <F FSUB >F  ;
: F*                 ( fp fp -- fp )
  <F FMUL >F  ;
: F/                 ( fp fp -- fp )
  <F FDIV >F  ;                  ==>
```

```
Screen: 129
( Floating: FLOAT FIX FLOG FEXP)
: FLOAT                  ( n -- fp )
  FR0 ! IFP >F  ;
: FIX                    ( fp -- n )
  FS FPI FR0 @ ;
: LOG                   ( fp -- fp )
  FS FLG >F  ;
: LOG10                 ( fp -- fp )
  FS FLG10 >F  ;
: EXP                   ( fp -- fp )
  FS FEX >F  ;                   -->
```

```
Screen: 130
( Floating: FEXP10 ASCF FLIT...)
: EXP10                 ( fp -- fp )
  FS FEX10 >F  ;
: ASCF                   ( a -- fp )
  0 CIX ! INBUF ! AFP >F ;
: FLIT  ( in dict. only: -- fp )
  R> DUP 6 + >R F@ ;
: FLITERAL            ( fp -- [fp])
  STATE @
  IF
   COMPILE FLIT HERE F! 6 ALLOT
  ENDIF ;   IMMEDIATE            ==>
```

```
Screen: 131
( Floating:  FLOATING  FP       )
: FLOATING          ( nnnn, -- fp )
  BL WORD HERE 1+ ASCF
  [COMPILE] FLITERAL ; IMMEDIATE
( Float the following literal )
( Ex:  FLOATING 1.2345 )
( or   FLOATING  -1.67E-13 )
: FP                ( nnnn, -- fp )
  [COMPILE] FLOATING ;
  IMMEDIATE
                                 -->
```

Screen: 132

```
( Floating: FVARIABLE FCONSTANT)

: FVARIABLE        ( xxxx, fp -- )
                   ( xxxx: -- a   )
  <BUILDS
    HERE F! 6 ALLOT
  DOES> ;

: FCONSTANT        ( xxxx, fp -- )
                   ( xxxx: -- fp )
  <BUILDS
    HERE F! 6 ALLOT
  DOES> F@ ;



                                ==>
```

Screen: 133

```
( Floating:  F0=  F=  F<  F>    )

: F0=                  ( fp -- f )
  OR OR 0= ;

: F=                ( fp fp -- f )
  F- F0= ;

: F<                ( fp fp -- f )
  F- DROP DROP 80 AND 0> ;

: F>                ( fp fp -- f )
  FSWAP F< ;


                          BASE !
```

Screen: 134

Screen: 135

Screen: 136

```
( 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: 137

```
( 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: 138

```
( 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: 139

```
```

Screen: 140

```
( ValFORTH Video editor   V1.0 )

BASE @ DCX '( >SCD )( 68 KLOAD )

VOCABULARY EDITOR IMMEDIATE
EDITOR DEFINITIONS

0 VARIABLE XLOC    ( X coord.    )
0 VARIABLE YLOC    ( Y coord.    )
0 VARIABLE LSTCHR  ( last key    )
0 VARIABLE ?ESC    ( coded char?)
0 VARIABLE TBLK    ( top block   )
0 VARIABLE UPSTAT 2 ALLOT ( map)

 15 CONSTANT 15  32 CONSTANT 32
128 CONSTANT 128             ==>
```

Screen: 141

```
( ValFORTH Video editor   V1.0 )

: LMOVE 32 CMOVE ;

: BOL 88 @ YLOC @ 1+ 32 * + ;

: SBL  88 @ 544 + ;

: CURLOC                  ( --- )
  BOL XLOC @ + ;     ( SCR ADDR )

: CSHOW                   ( --- )
  CURLOC DUP       ( GET SCR ADDR )
  C@ 128 OR        ( INVERSE CHAR )
  SWAP C! ;        ( STORE ON SCR )
                              -->
```

Screen: 142

```
( ValFORTH Video editor   V1.0 )

: CBLANK                  ( --- )
  CURLOC DUP C@ 127
  AND SWAP C! ;

: 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: 143

```
( ValFORTH Video editor   V1.0 )

: 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 + C! ; -->
```

Screen: 144
```
( ValFORTH Video editor   V1.0 )
: LNINS                          ( -- )
  CBLANK
  4 YLOC @ 4 /
  DO 1 I UPSTAT + C! LOOP
  YLOC @ 15 <
  IF
    BOL DUP 32 +
    15 YLOC @ - 32 *
    <CMOVE
  ENDIF
  BOL 32 ERASE
  CSHOW EDMRK ;
                                   ==>
```

Screen: 145
```
( ValFORTH Video editor   V1.0 )
: LNDEL                          ( -- )
  CBLANK
  4 YLOC @ 4 /
  DO 1 I UPSTAT + C! 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: 146
```
( ValFORTH Video editor   V1.0 )
: RUB                            ( -- )
  XLOC @ 0= NOT    ( ON L-EDGE? )
  IF LFCUR 0 CURLOC C!
     CSHOW EDMRK
  ENDIF ;
: PTCHR                          ( -- )
  EDMRK
  LSTCHR @ 127 AND
  DUP LSTCHR !
  >SCD CURLOC C!
  RTCUR XLOC @ 0=
  IF DNCUR ENDIF
  0 ?ESC ! CSHOW ;                 ==>
```

Screen: 147
```
( ValFORTH Video editor   V1.0 )
: SCRSV                          ( -- )
  88 @ 32 + PAD 512 BSCD>
  4 0
  DO
    I UPSTAT + C@
    0 I UPSTAT + C!
    IF
      PAD 128 I * +
      TBLK @ I + BLOCK
      128 CMOVE UPDATE
    ENDIF
  LOOP
  0 XLOC ! 0 YLOC ! ;
                                   -->
```

Screen: 148
```
( ValFORTH Video editor   V1.0 )
: SCRGT                          ( -- )
  4 0
  DO
    TBLK @
    I + BLOCK
    PAD 128 I * +
    128 CMOVE
  LOOP
  PAD 88 @ 32 +
  512 >BSCD ;
                                   ==>
```

Screen: 149
```
( ValFORTH Video editor   V1.0 )
: 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: 150

```
( ValFORTH Video editor   V1.Ø )

: SPLCHR  1 ?ESC ! ;         ( -- )

: EXIT                       ( -- )
  CBLANK 19 LSTCHR !
  Ø XLOC ! Ø YLOC ! ;

: EDTABT                     ( -- )
  UPSTAT 4 Ø FILL
  EXIT ;

                                ==>
```

Screen: 151

```
( ValFORTH Video editor   V1.Ø )

: CONTROL                  ( n -- )
  DUP  19 = IF DROP EXIT   ELSE
  DUP  17 = IF DROP EDTABT ELSE
  DUP  28 = IF DROP UPCUR  ELSE
  DUP  29 = IF DROP DNCUR  ELSE
  DUP  3Ø = IF DROP LFCUR  ELSE
  DUP  31 = IF DROP RTCUR  ELSE
  DUP 126 = IF DROP RUB    ELSE
  DUP 157 = IF DROP LNINS  ELSE
  DUP 156 = IF DROP LNDEL  ELSE
       27 = IF DROP SPLCHR ELSE

                                -->
```

Screen: 152

```
( ValFORTH Video editor   V1.Ø )

  PTCHR   ( IF NOTHING SPECIAL )
  ENDIF ENDIF ENDIF ENDIF
  ENDIF ENDIF ENDIF ENDIF
  ENDIF ENDIF ;

                                ==>
```

Screen: 153

```
( ValFORTH Video editor   V1.Ø )

: (V)                     ( TBLK -- )
  DECIMAL
  DUP BLOCK DROP TBLK !
  UPSTAT 4 Ø FILL
  1 PFLAG ! Ø GR.
  1 752 C! CLS
  1 559 C@ 252
  AND OR 559 C!
  112 56Ø @ 6 + C!
  112 56Ø @ 23 + C!
  ." Screen #" 11 SPACES
  ." ValFORTH"
  Ø NWSCR
                                -->
```

Screen: 154

```
( ValFORTH Video editor   V1.Ø )


( Main loop of editor )

  BEGIN
    KEY DUP LSTCHR !
    ?ESC @
    IF
      PTCHR Ø LSTCHR !
    ELSE
      CONTROL
    ENDIF
    LSTCHR @ 19 =
  UNTIL
                                ==>
```

Screen: 155

```
( ValFORTH Video editor   V1.Ø )

  CBLANK SCRSV Ø 767 C!
  2 56Ø @ 6 + C!
  2 56Ø @ 23 + C!
  2 559 C@ 252
  AND OR 559 C!
  Ø 752 C! CLS CR
  ." Last edit on screen # "
  SCR @ . CR CR ;

FORTH DEFINITIONS

: V                          ( s -- )
  1 MAX B/SCR *
  EDITOR (V) ;                  -->
```

Screen: 156

```
( ValFORTH Video editor   V1.0 )

: 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: 157

```
( ValFORTH Video editor   V1.0 )

: 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 ;                      -->
```

Screen: 158

```
( ValFORTH Video editor   V1.0 )


( Note: the fig bug is fixed )
(       in WHERE below.      )

HEX
: WHERE                ( [ n n ] -- )
  2DUP DUP B/SCR / DUP SCR !
  ." Scr # " DECIMAL . SWAP
  C/L /MOD C/L * ROT BLOCK +
  CR C/L -TRAILING TYPE
  CR HERE C@ - 2- 0 MAX SPACES
  1 2FE C! 1C EMIT 0 2FE C!
  [COMPILE] EDITOR QUIT ;
                         BASE !
```

Screen: 159

Screen: 160

Screen: 161

Screen: 162

```
( DOS:  input/output routines  )

BASE @ HEX

340 VARIABLE IOCB
  0 VARIABLE IO.X
  0 VARIABLE IO.CH

: IOCC
  10 * 70 MIN DUP IO.X C!
  340 + IOCB !  ;

: (IO)
  <BUILDS ,
  DOES> @ IOCB @ + ;
                              ==>
```

Screen: 163

```
( DOS:  system words            )

2 (IO) ICCOM  3 (IO) ICSTA
4 (IO) ICBAL  8 (IO) ICBLL
A (IO) ICAX1  B (IO) ICAX2
C (IO) ICAX3  D (IO) ICAX4
E (IO) ICAX5  F (IO) ICAX6


CODE XCIO
  XSAVE STX, IO.X LDX,
  IO.CH LDA, E456 JSR,
  XSAVE LDX, IO.CH STA,
  TYA, PUSH0A JMP,
C;
                              -->
```

Screen: 164

```
( DOS:  OPEN CLOSE PUTC GETC    )

: OPEN     ( adr n1 n2 n3 -- n4 )
  IOCC ICAX2 C! ICAX1 C!
  ICBAL ! 03 ICCOM C! XCIO ;

: CLOSE                 ( n1 -- )
  IOCC 0C ICCOM C! XCIO DROP ;

: PUT               ( c n1 -- n2 )
  IOCC IO.CH C! 0B
  ICCOM C! XCIO ;

: GET               ( n1 -- c n2 )
  IOCC 7 ICCOM C! XCIO
  IO.CH C@ SWAP ;            ==>
```

Screen: 165

```
( DOS:  GET/PUTREC STATUS DEV  )

: GETREC       ( adr n1 n2 -- n3 )
  IOCC 5 ICCOM C! ICBLL !
  ICBAL ! XCIO ;

: PUTREC       ( adr n1 n2 -- n3 )
  IOCC 9 ICCOM C! ICBLL !
  ICBAL ! XCIO ;

: STATUS                ( n1 -- n2 )
  IOCC ICSTA C@ ;

: DEVSTAT       ( n1 -- n2 n3 n4 )
  IOCC 0D ICCOM C! XCIO
  >R 2EA @ 2EC @ R> ;          -->
```

Screen: 166

```
( DOS:  SPECIAL                  )

: SPECIAL
( n1 n2 n3 n4 n5 n6 n7 n8 -- n9)
  IOCC ICCOM C! ICAX6 C!
  ICAX5 C! ICAX4 C! ICAX3 C!
  ICAX2 C! ICAX1 C! XCIO ;









                        BASE !
```

Screen: 167

Screen: 168

```
( Atari 850 download                    )

BASE @ HEX

CODE DO-SIO
   XSAVE STX, 0 # LDA,
   E459 JSR,
   XSAVE LDX, NEXT JMP,

: SET-DCB
    50 300 C!      1 301 C!
    3F 302 C!     40 303 C!
   500 304  !      5 306 C!
     0 307 C!      C 308 C!
     0 309  !      0 30B C! ;
                                      ==>
```

Screen: 169

```
( Atari 850 download                    )

CODE RELOCATE
   XSAVE STX, 506 JSR,
   HERE 8 + JSR,  XSAVE LDX,
   NEXT JMP,   0C )JMP,


: RS232                          ( -- )
   HERE 2E7 ! SET-DCB DO-SIO
   500 300 0C CMOVE DO-SIO
   RELOCATE 2E7 @ HERE - ALLOT
   HERE FENCE ! ;


                          BASE !
```

Screen: 170

```
CONTENTS OF THIS DISK:

PRINTER UTILITIES:          38 LOAD
DEBUGGING AIDS:             42 LOAD
VALFORTH EDITOR 1.0:       140 LOAD
ASSEMBLER:                  76 LOAD
COLOR COMMANDS:            100 LOAD
GRAPHICS:                  104 LOAD
GRAPHICS DEMO:             112 LOAD
SOUNDS:                    114 LOAD
FLOATING POINT:            120 LOAD
 (FP REQUIRES ASSEMBLER FIRST)
SCREEN CODE CONVERS.:      136 LOAD
FORMATTER:                  92 LOAD
DISK COPIERS:               72 LOAD
   (continued on next screen)
```

Screen: 171

```
CONTENTS OF THIS DISK, cont:

fig EDITOR:                 56 LOAD
BUFFER RELOCATION:          94 LOAD
AUTO-BOOT UTILITY:          30 LOAD
OPERATING SYS. WORDS:      162 LOAD
850 DOWNLOAD (RS-232):     168 LOAD
 (OPSYS AND 850 NEED ASSEMBLER)
```

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

valFORTH
T.M.

SOFTWARE SYSTEM
for ATARI*

GENERAL UTILITIES and VIDEO EDITOR

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

© Copyright Valpar International 1982

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.

© Copyright Valpar International 1982

valFORTH UTILITIES/EDITOR USER'S MANUAL

Table of Contents

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 psuedo 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
```

Current:

```
( Example screen )  ( line 0 )

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

■
: +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 (scr# ---)
Enter the edit mode and view the specified screen.

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 (ON/OFF ---)
When ON, allows all words compiled until the next OFF to be locatable using the LOCATE command above.

#BUFS (#lines ---)
Sets the length (in lines) of the storage buffer. The default is five.

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
```

<u>The SEL Structure</u>

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 >> .

Note also that COND structures may take more than one stack argument, or none at all.

(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 2112 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 @ +
   Ø 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 -- )
  Ø DO
   9300 Ø
   DO
   LOOP
  LOOP ;

: MSEC                    ( n -- )
  Ø DO
   6 Ø
   DO
   LOOP NOOP
  LOOP ;
                                ==>
```

```
Screen: 141
( Utils:  H->L  L->H  H/L      )

HEX

CODE H->L              ( n -- n )
     B5 C, Ø1 C, 95 C, ØØ C,
     94 C, Ø1 C, 4C C, NEXT , C;

CODE L->H              ( n -- n )
     B5 C, ØØ C, 95 C, Ø1 C,
     94 C, ØØ C, 4C C, NEXT , C;

CODE H/L             ( n -- n n )
     B5 C, ØØ C, 94 C, ØØ C,
     4C C, PUSHØA , C;
DCX                            -->
```

```
Screen: 142
( Utils:  BIT  ?BIT  TBIT      )
HEX
CODE BIT               ( b -- n )
 B4 C, ØØ C, C8 C, A9 C, ØØ C,
 95 C, ØØ C, 95 C, Ø1 C, 38 C,
 36 C, ØØ C, 36 C, Ø1 C, 18 C,
 88 C, DØ C, F8 C, 4C C, NEXT ,
C;
: ?BIT BIT AND Ø# ; ( 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   Ø , 1 , -1 , Ø ,

CODE STICK           ( n -- h v )

  B4 C, ØØ C, B9 C, 78 C, Ø2 C,
  48 C, CA C, CA C, 29 C, Ø3 C,
  ØA C, A8 C, B9 C,     , 95 C,
  Ø2 C, C8 C, B9 C,     , 95 C,
  Ø3 C, 68 C, 4A C, 4A C, 29 C,
  Ø3 C, ØA C, A8 C, B9 C,     ,
  95 C, ØØ C, C8 C, B9 C,     ,
  95 C, Ø1 C, 4C C, ' SWAP ,

CURRENT @ CONTEXT !            -->
```

Screen: 144

```
( Utils:  STRIG  PADDLE           )
HEX
CODE PADDLE                ( n -- n )
  B4 C, ØØ C,  B9 C, 27Ø ,
  4C C, PUTØA , C;
CODE STRIG                 ( n -- f )
  B4 C, ØØ C, B9 C, 284 ,
  49 C, Ø1 C, 4C C, PUTØA , C;
CODE PTRIG                 ( n -- f )
  B4 C, ØØ C, B9 C, 27C ,
  49 C, Ø1 C, 4C C, PUTØA , C;
                                ==>
```

Screen: 145

```
( Utils:  ATRACT  NXTATR          )
DCX
: ATTRACT                  ( f -- )
  IF 255 ELSE Ø ENDIF 77 C! ;
: NXTATR
  255 2Ø C! ;                ( -- )
( Changes user clock )
: HLDATR
  Ø 2Ø C! ;                  ( -- )
( Changes user clock )
                                -->
```

Screen: 146

```
( Utils:  16TIME                  )
HEX
CODE 16TIME
  CA C, CA C,
  A5 C, 13 C,  95 C, Ø1 C,
  A5 C, 14 C,  95 C, ØØ C,
  DØ C, Ø4 C,
  A5 C, 13 C,  95 C, Ø1 C,
  4C C, NEXT , C;
                                ==>
```

Screen: 147

```
( Utils:  8RND  16RND  CHOOSE  )
HEX
CODE 8RND                    ( -- b )
  AD C, D2ØA ,
  4C C, PUSHØA ,
C;
CODE 16RND                   ( -- n )
  AD C, D2ØA , 48 C, 68 C, 48 C,
  68 C, 48 C, AD  C, D2ØA ,
  4C C, PUSH ,  C;
 : CHOOSE                   ( n -- n )
  16RND U* SWAP DROP ;
                                -->
```

Screen: 148

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

April, 1982

Notes on Starting FORTH for the fig-Forth User

A very popular book on the FORTH language called Starting FORTH has recently been published. The author, Leo Brodie, gives an excellent description of the FORTH language as implemented at FORTH, Inc. fig-FORTH differs from that implementation in some areas, and this document explains those differences. All comments that apply to fig-FORTH also apply to valForth.

BLANK = BLANKS (page 285)

Brodie describes the word BLANK. In fig-FORTH, this word is BLANKS.

EMPTY-BUFFERS vs. EMPTY-BUFFERS (page 283)

Brodie's word EMPTY-BUFFERS does not necessarily change the buffers. In fig-FORTH, EMPTY-BUFFERS zero fills the buffers.

CONTEXT vs. CONTEXT (page 247)

These two words are not synonymous in the two versions. fig-FORTH uses a system of VOC-LINKS with CONTEXT, while FORTH, Inc. does not.

EXIT = ;S (page 246)

The word EXIT, as Brodie describes it, is identical in function to ;S in fig-FORTH.

'S = SP@ (page 247)

The word 'S in FORTH, Inc.'s is SP@ in fig-FORTH.

EMPTY (page 84)

Not yet implemented in fig-FORTH.

WIPE vs. CLEAR (page 84)

CLEAR requires a screen number while WIPE clears the last screen edited.

ABORT" (page 103)

Not implemented in fig-FORTH.

PROVIDED THROUGH THE COURTESY OF VALPAR INTERNATIONAL CORPORATION; 3801 East 34th Street, Suite 105; Tucson, Arizona. Further distribution of this public domain document must include this notice.

?DUP = -DUP (page 103)

The word ?DUP in FORTH, Inc.'s is -DUP in fig-FORTH.

?STACK vs. ?STACK (page 103)

?STACK as described by Brodie as incorrect for fig-FORTH. ?STACK in fig-FORTH automatically aborts if there is a stack error.

NEGATE = MINUS, DNEGATE = DMINUS (pages 123, 178)

The words NEGATE and DNEGATE in FORTH, Inc.'s are MINUS and DMINUS respectively in fig-FORTH.

+LOOP vs. +LOOP (page 143)

The word +LOOP, as Brodie describes it, works differently for negative stepping than the +LOOP in fig-FORTH. fig-FORTH always ends if the index equals the limit, even for negative stepping.

PAGE = CLS (page 143)

Brodie's PAGE is called CLS in valForth. It has no equivalent in fig-FORTH.

U/MOD = U/ (page 177)

Brodie's U/MOD is U/ in fig-FORTH.

CREATE vs. CREATE (page 209)

Brodie's CREATE works differently from CREATE in fig-FORTH. A word using CREATE in fig-FORTH must unSMUDGE the header before the word can be used. The ";" unsmudges headers automatically. In addition, Brodie's CREATE and fig-FORTH CREATE move different default values in the CFA of the created header (see below).

CREATE = <BUILDS (page 209)

In Brodie's chapter 11 on extending the compiler, he uses the series CREATE... DOES>. In fig-FORTH, this should be <BUILDS...DOES>.

NUMBER vs. NUMBER (page 285)

Brodie's NUMBER only converts numbers to double length if the double word set is loaded. fig-FORTH always converts numbers to double length.

PROVIDED THROUGH THE COURTESY OF VALPAR INTERNATIONAL CORPORATION; 3801 East 34th Street, Suite 105; Tucson, Arizona. Further distribution of this public domain document must include this notice.

>IN = IN, H = DP (page 247)

The variable >IN and H in Brodie's FORTH are IN and DP respectively in fig-FORTH.

VARIABLE vs. VARIABLE (page 209)

The word VARIABLE, as Brodie describes it, accepts no value from the stack. fig-FORTH, on the other hand, does expect an initialization value from the stack.

PROVIDED THROUGH THE COURTESY OF VALPAR INTERNATIONAL CORPORATION; 3801 East 34th Street, Suite 105; Tucson, Arizona. Further distribution of this public domain document must include this notice.

INDEX 5/3/82

KEY TO INDEX ABBREVIATIONS

ABBR	MEANING	TAB	PACKAGE
$$	STRINGS	$-ARY-CASE-DBL	UTILITIES/EDITOR
ARY	ARRAYS	$-ARY-CASE-DBL	UTILITIES/EDITOR
CSE	CASE	$-ARY-CASE-DBL	UTILITIES/EDITOR
DBL	DBL# EXTENSIONS	$-ARY-CASE-DBL	UTILITIES/EDITOR
ASM	ASSEMBLER	ASSEMBLER	valFORTH 1.1
DBG	DEBUGGER	1.1 EXTENSIONS	valFORTH 1.1
FP	FLOATING POINT	1.1 EXTENSIONS	valFORTH 1.1
GCS	GRAF-COL-SOUND	1.1 EXTENSIONS	valFORTH 1.1
TOPD	TXT OUT, DISK PREP	1.1 EXTENSIONS	valFORTH 1.1
FE	fig EDITOR	fig EDITOR	valFORTH 1.1
VG1	valFORTH GLOSS	1.1 GLOSSARY	valFORTH 1.1
HRT	HI-RES TEXT	HRT-MSC-TRNS	UTILITIES/EDITOR
MSC	MISC. UTILITIES	HRT-MSC-TRNS	UTILITIES/EDITOR
TRNS	TRANSIENTS	HRT-MSC-TRNS	UTILITIES/EDITOR
VED1	valFORTH Ed. 1.1	valFORTH Ed. 1.1	UTILITIES/EDITOR

VALPAR
INTERNATIONAL
3801 E. 34TH STREET
TUCSON, ARIZONA 85713

valFORTH T.M.

SOFTWARE SYSTEM
for ATARI*

PLAYER-MISSILE GRAPHICS

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

Software and Documentation
©Copyright 1982
Valpar International

valFORTH

T.M.

SOFTWARE SYSTEM

Stephen Maguire
Evan Rosen

(Atari interfaces based on work by Patrick Mullarky)

Software and Documentation
© Copyright 1982
Valpar International

Purchasers of this software and documentation package are authorized only to make backup or archival copies of the software, and only for personal use. Copying the accompanying documentation is prohibited.

Copies of software for distribution may be made only as specified in the accompanying documentation.

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.

Software and Documentation
©Copyright 1982
Valpar International

PLAYER/MISSILE GRAPHICS PACKAGE

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  BLDPLY       ( 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!".

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 ---)

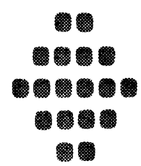
The (PMINIT) command (or PMINIT below) must be used to initialize the player missile routines before any other player missile command may be used. (PMINIT) expects both the address of player/missile memory and a 1 or a 2 indicating whether single or double resolution is desired.

NOTE: The difference between single and double resolution is shown graphically below:

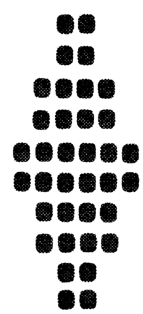
Player as defined in memory:

```
00011000
00111100
01111110
00111100
00011000
```

single res on screen:



double res on screen:



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# --)

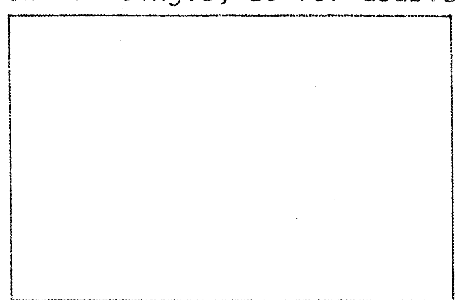
The PLYSEL command is used to select image "#" out of a table of images of the same length and assigns that image to the specified player. PLYSEL is typically used to animate players. An example usage of this can be found in Player/Missile Example #2 found in the directory of the disk.

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:



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:

15	14		4	3	2	1	0
0	0	...	0	1	r	t	b

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 (ml# --- 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 AND        ( Leaves 1 if collision with pf#0, else 0 )
2 AND        (   "    1     "        "   pf#1,   "  0 )
4 AND        (   "    1     "        "   pf#2,   "  0 )
8 AND        (   "    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

(---)

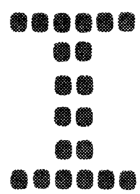
The HITCLR command clears all collision registers. In other words, it sets the collision monitor to a state which indicates that no collisions have occurred.

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-in dependent 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, ØØ C, B9 C, Ø MSLDAT ,
  2D C, MSLSZ , HERE
  88 C, 3Ø C, 7 C, 16 C, Ø2 C,
  16 C, Ø2 C, 4C C, , 15 C,
  Ø2 C, 8D C, MSLSZ , 8D C,
  DØØC , 4C C, POPTWO ,
C;





                                  ==>
```

Screen: 49

```
( PlyMsl:  PLYLOC MSLLOC MCPLY )

CODE PLYLOC          ( pl# -- h v )
 94 C, Ø1 C, B4 C, Ø C,
 B9 C, Ø PLYHRZ , 95 C, Ø C,
 B9 C, Ø PLYVRT , 4C C, PUSHØA ,

CODE MSLLOC          ( ml# -- h v )
 94 C, Ø1 C, B4 C, Ø C,
 B9 C, Ø MSLHRZ , 95 C, Ø C,
 B9 C, Ø MSLVRT , 4C C, PUSHØA ,

: MCPLY                      ( f -- )
  26F C@ SWAP
  IF 2Ø OR ELSE DF AND ENDIF
  26F C! ;                        -->
```

Screen: 50

```
( PlyMsl:  ?COL HITCLR ?MXPF...)

CODE ?COL                    ( -- f )
 CA C, CA C, 98 C, AØ C, ØF C,
 19 C, DØØØ , 88 C, 1Ø C, FA C,
 C8 C, 94 C, Ø1 C, 95 C, ØØ C,
 4C C, ' Ø# ( CFA @ ) ,
C;

CODE ?MXPF               ( ml# -- n )
  B4 C, ØØ C,  B9 C, DØØØ ,
  4C C, PUTØA ,  C;

CODE ?PXPF               ( pl# -- n )
  B4 C, ØØ C,  B9 C, DØØ4 ,
  4C C, PUTØA ,  C;               ==>
```

Screen: 51

```
( PlyMsl:  ?MXPL ?PXPL PLYBND  )

CODE ?MXPL               ( ml# -- n )
  B4 C, ØØ C,  B9 C, DØØ8 ,
  4C C, PUTØA ,  C;

CODE ?PXPL               ( pl# -- n )
  B4 C, ØØ C,  B9 C, DØØC ,
  4C C, PUTØA ,  C;

CODE HITCLR                   ( -- )
  8C C, DØ1E , 4C C, NEXT , C;

CODE ?BND                ( xl# -- n )
  AD C, BNDCOL ,
  4C C, PUSHØA ,  C;              -->
```

Screen: 52

```
( PlyMsl:  MSLBND  ?BND            )

CODE ?PLYSTT             ( pl# -- n )
  B4 C, ØØ C, B9 C, Ø PLYSTT ,
  4C C, PUTØA ,  C;


CODE ?MSLSTT             ( ml# -- n
  B4 C, ØØ C, B9 C, Ø MSLSTT ,
  4C C, PUTØA ,  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 + 1Ø O+S
  DO I C! 4 /LOOP ;

: PMCOL          ( pl# col lum -- )
  SWAP 1Ø * +
  SWAP DUP 4 =
  IF
    DROP 2C7 C!
  ELSE
    2CØ + 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
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 73
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 74
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 75
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 76
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 77
Ø
1
2
3
4
5
6
7
8
9
1Ø
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
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 85
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 86
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 87
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 88
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

Screen: 89
Ø
1
2
3
4
5
6
7
8
9
1Ø
11
12
13
14
15

```
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
```