Using
fig-FORTH
on the Atari 800

Stephen A. Cohen
May 2, 1983

This manual teaches:

How to use FORTH with enough proficiency to be able to write most basic programs. This manual will not make you an expert in FORTH, but following the directions in this manual will give you enough capability on your disk to do integer or Floating Point math, numeric input and output, and random disk accessing, which is enough to complete most programs. Your FORTH disk will also allow you to do graphics and sound on the Atari 800 by using the Atari BASIC graphics and sound commands, as explained on pages 13 and 14 of the APX manual which comes with the FORTH disk. This manual does not deal with sound and graphics; however, almost everything else about fig-FORTH except the assembler is explained in detail in this manual.

General notes regarding this manual:

1. <CR> stands for Carriage Return. Where <CR> appears, press the key marked **RETURN** on the keyboard.
2. This manual took much time and effort to produce. Please treat it with care. It is the author's sincere hope that this manual will be helpfull to you and to others. It is not easy to make copies of such a lengthy manual as this, so please take care of it so that others may read it after you.
3. If you would like to become an expert in FORTH, which is an extremely powerful and fast language once it has been mastered fully, you will have to read some of the following books and spend some time on the computer experimenting with FORTH in addition to reading this manual:

   Starting FORTH by Leo Brodie
   Systems Guide to fig-Forth by C. H. Ting
   The APX manual on Extended fig-FORTH by Patrick L. Mullarky

   It is of use to know something about assembly languages if you would like to become an expert in FORTH or any other language.

CONTENTS

Making yourself a FORTH disk	1-1
Editing in Forth	2-1
screen editing	2-1
disk editing	2-3
fig-FORTH vs FORTH-79	3-1
The Printer	3-1
A self-teaching lesson (lesson 1)	A-1
A self-teaching lesson (lesson 2)	B-1
Glossary of most FORTH commands	C-1
Index to Glossary of most FORTH commands	D-1
Numeric input from the keyboard	E-1
Complete VLIST of FORTH VOCABULARY	F-1
Sample Random Disk Access Program	G-1

Though he has tried to avoid any errors in this manual, the author does not guarantee that he is perfect; errors may appear. This manual is believed to be accurate but its accuracy is not guaranteed. If any reader should find errors or have suggestions for improving the manual, the author would appreciate being contacted at school or (preferably) at home.

HOME:
Stephen Cohen
6961 E. Walsh Pl.
Denver, CO 80224
(303) 377-6869

SCHOOL that Author attends:
George Washington High School
655 S. Monaco Pkwy.
Denver, CO 80224
(303) 399-7770 (computer lab phone)

HOW TO MAKE YOURSELF A FORTH DISK

1. TURNING THE COMPUTER ON:
   A. Make sure there are **NO DISKS** in any disk drives. A disk can easily be erased by being in the drive when the drive is turned on.
   B. Turn on the main power supply.
   C. Turn on the disk drive. If there are 2 drives on the computer then the one on top is drive 1 and that is the one you need. The POWER ON light will go on, and so will the BUSY light. The drive should make a little noise for a few seconds, then the BUSY light will go out.
      WHENEVER you turn on an ATARI you MUST wait for this light to go out before you put a disk in the drive.
   D. Turn on the monitor and turn the volume up to some audible level.
   E. Make sure that there are no cartridges in the cartridge slots of the computer. Above the keyboard of the computer the words PULL OPEN appear. Open the computer. (By the way, opening the computer turns it off if it was already turned on.) There are 2 cartridge slots inside. If there are any cartridges in the slots then pull them out. Then shut the computer by pushing the swinging door back down until it clicks shut.
   F. If the computer has a printer, turn on the interface and the printer. If the computer has another drive, you may turn this on, too.
   G. Now you are ready to insert any disk into the disk drive and turn on the computer, thus BOOTING the disk.
      The on/off switch for the computer is on the right side as you face the computer near the back on the side of the computer.

2. Formatting your disk for Atari DOS:
   A. Boot an Atari DOS Disk.
   B. The DOS MENU should appear on the screen. Select choice **I** (FORMAT DISK).
      The computer will ask --> WHICH DRIVE TO FORMAT ?
      Answer --> **1**
      The computer will say --> TYPE "Y" TO FORMAT DISK 1
      BEFORE YOU TYPE ANYTHING remove the DOS disk from the drive and insert the disk that you want formatted.

      WARNING: Formatting a disk erases the disk.

      Type --> **Y**
      The formatting will be complete when the computer says:
      SELECT ITEM OR RETURN FOR MENU

3. SAVEing FORTH onto your disk:

A. Boot a fig-FORTH Master disk.
The computer should greet you with --> fig-FORTH 1.1

B. Type the following (spacing is important) :
18 LOAD 36 LOAD 5Ø LOAD 56 LOAD 6Ø LOAD <CR>

C. The computer should LOAD the above-numbered screens, which takes it a couple of minutes.
During this the computer should print:
R Isn't unique
I Isn't unique
N Isn't unique
Ø= Isn't unique
Isn't unique

D. When the computer is done LOADing screens 18,36,5Ø,56 and 6Ø it will print **ok.**
ok is the FORTH prompt and is FORTH's way of telling you that it has done whatever you wanted and is waiting for you to type something.

E. Type the following, and remember that the computer should print "ok" after everything you type:

EDITOR DEFINITIONS <CR>
: CLEAR DUP 15 > IF CLEAR ELSE ." NO " DROP THEN ; <CR>
(note: the ." must be typed without a space between the period and the quote. A space should follow the quote, though.)
The computer will print --> CLEAR Isn't unique ok
If anything else appears on the screen, you have done something wrong. In this case, go to the top of this page and start over with step 3.

If everything seems to be working correctly, type:
FORTH DEFINITIONS <CR>

F. Do not turn the computer off, but remove the FORTH Master disk from the drive and insert your disk (which you have formatted for DOS.)

G. Type --> **SAVE** <CR>
The computer should now save FORTH onto your disk. This takes about a minute, after which the computer prints **ok.**

H. Remove your disk from the drive and insert the Master disk again. Type:
14 LIST MARK 15 LIST MARK <CR>
The computer should read and list screens 14 and 15 from the Master disk. These screens contain the error messages which you are now going to save onto your disk.

I. Remove the Master disk and reinsert your disk. Type:
FLUSH <CR>
Screens 14 and 15 (the error messages) will now be copied onto your disk. Check to see if this worked by typing:
14 LIST 15 LIST <CR>
The screens should list exactly as they did before. If they do not, then ask someone for help.

Now you have a working FORTH disk. Whenever you want to use FORTH, all you will have to do is boot your disk.

EDITING

There are 2 main types of editing. They are:
1. Screen Editing
2. Disk Editing

1. The Atari screen editor (for editing a line while typing it.)

This editing is done by using the Control key with the up, down, forward, and back arrow keys, and by using the Control key with the INSERT, DELETE, and CLEAR keys.

Holding down the CTRL key and pressing one of the arrow keys (on which the arrow is in inverse print) will move the cursor anywhere on the screen, depending on which arrow you press. Going off the top of the screen will bring you to the bottom, and vice versa. Going off one of the sides will bring you back to the opposite side. Holding a key down will cause it to repeat.

To delete characters to the right of the cursor, press CTRL-DELETE. To erase characters to the left of the cursor, just press DELETE (or BACKSPACE) without holding down the CTRL key. BACKSPACE will replace a character to the left of the cursor with a blank; CTRL-DELETE will delete a character to the right of the cursor, thus shortening the line the cursor is on. To delete the entire line that the cursor is on, press SHIFT-DELETE.

To clear the screen, type CTRL-CLEAR or SHIFT-CLEAR. Whenever you are typing a line of text and you have messed it up so much that it would be easier to do it over instead of editing it, type CTRL-CLEAR or SHIFT-CLEAR. The screen will be cleared and you can start over.

To insert spaces to the right of the cursor, type CTRL-INSERT. Then whatever you type will be typed over these spaces. EXAMPLE:

Screen says: EXAPLE.
You move the cursor so that it is over the P.
You type: CTRL-INSERT
Screen says: EXA PLE (cursor is over space)
You type: M
Screen now says: EXAMPLE

To insert an entire line of spaces on the line that the cursor is on, press SHIFT-INSERT. This moves the rest of the lines on the monitor screen down and creates a blank line where the cursor is.

When you press **RETURN** the entire line of text that the cursor is over is entered into the computer's input buffer. This means that even if you are in the middle of a line, the whole line will be accepted as input. The computer does not usually interpret anything you type until you press **RETURN.** Pressing RETURN does not clear to the end of the line.

WARNING: FORTH can interpret only 80 characters at a time. Having more than 80 characters on a line of input will cause the computer to do weird things.

When you are editing a line that has an error message on it, delete the error message before pressing RETURN. Otherwise, the computer will think that you typed in the error message. Example:

You type a line and accidentally miss one letter in the word **OVER** so that it says --> OVR

The computer types --> OVR ? at the end of the line that you have just typed.

Instead of retyping the whole line, you move the cursor up on the screen to the word 'OVR' and insert an 'E' in the proper place using the method described on the previous page. Then you move the cursor to the beginning of the error message that the computer printed and, using CTRL-DELETE you delete this message from the screen. Then you type RETURN. The computer will now accept the line as though you had correctly re-typed it.

There are a couple of special keys on the Atari that you should know about:

The key that has an Atari symbol on it, just to the left of the right SHIFT key, changes from inverse to normal print when you press it. Pressing it again will change back to normal print.

The **CAPS lowr** key, just above the right SHIFT key, will shift to lower case if you press it and to upper case if you press it while holding down the SHIFT key.

Stay in upper case normal print when using FORTH, except in special cases where you might want to print something in lower case. All FORTH commands are in UPPER CASE and in normal, not inverse, print. Using inverse print can cause the computer to lock up.

2. The FORTH EDITOR (for editing programs or data on the disk)

The disk is divided into 90 screens. Screens 0-15 are used by FORTH and hold the language itself and the error messages. You can type programs, data, messages, or anything onto any of the other screens, numbers 16-89. In FORTH, when you want to load a program from the disk, you will type the number of the screen that the program is on, not the name of the program.

To list a screen type: (screen#) **LIST**
Note: wherever spaces appear in this manual, it is necessary that you type them. For instance, here there must be a space between the screen number and the word **LIST**.
LISTing a screen makes that screen the current screen. To edit a screen you must first make that screen the current screen by LISTing it.

Example: Type **14 LIST**
The computer will list screen number 14, which has on it some of the error messages used by FORTH.
You will notice that these messages are on separate lines, and that each line has a number (0-15). A screen consists of 16 lines, numbered 0-15. Each line holds 64 characters, but trailing blanks are not typed out when the screen is listed.

Now if you type **16 LIST** you will see that screen 16 lists as a bunch of little hearts. Actually, there is nothing on the disk on screen 16 (assuming that you have just formatted a new disk). A heart is the Atari character for 'nothing'.

Suppose we want to type a program that will print
I ADORE FORTH

Suppose we want this program to be on screen 16 of the disk. First, we must enter the editor. This is done by typing --> **EDITOR**
The computer will print --> **ok**
Now we must make 16 the current screen. Type --> **16 LIST**
This makes 16 the current screen and lists screen 16.
Next we type --> **16 CLEAR** (Type the word 'CLEAR', not the CLEAR key.)
This clears screen 16 of the hearts and fills screen 16 with blanks.
(If you forget to type the screen number, or try to CLEAR a screen below number 16, the computer should print NO and the screen will not be cleared.)
To list the current screen we can now type --> **L**
L lists the current screen. Note: it is a good idea to clear the monitor by typing CTRL-CLEAR (the clear key, not the word CLEAR) before listing a screen on the monitor. Typing **L** or **LIST** to list a screen can cause problems when

there is text on the screen.

COMMENTING AND TITLING YOUR SCREEN

Now type --> **Ø P (MY FİRST SCREEN)**
Typing a number (Ø-15) and then **P** lets you put a line of text on the line whose number you typed. If you type **L** now you will see that

(MY FIRST SCREEN)

now appears on line Ø. You should always put the title of the screen on line Ø in a comment. In FORTH a comment is put inside parentheses, with at least one space after the first parentheses. The title of screen 16, then, is now

MY FIRST SCREEN

Type --> **1 P : FEELINGS ." I ADORE FORTH" CR ;**
If you now list the screen you will see that the definition of the word FEELINGS is on line 1.
Type --> **FLUSH**
This will save the work you have done onto the disk.
Next type --> **16 LOAD**
This loads the definition of FEELINGS from screen 16.
If you now type --> **FEELINGS**
the computer will print --> **I ADORE FORTH**

Note: though the computer should print 'ok' whenever you use the **P** command, 'ok' should not appear on the line when you list the screen. If 'ok' appears on the line, get rid of it. Otherwise, LOADing the screen will cause characters to disappear from the monitor screen.

For now, if you have any trouble or make any errors and this program does not work, go back to where you first typed **16 LIST** and redo everything from there down.

You have now been introduced to the editor and to a few of the editor commands. Following is a list of all the commands.

COMMAND:	EXPLANATION
(n,m stand for numbers. <string> stands for any character string.)	
EDITOR	Enters the Editor vocabulary. You must type **EDITOR** in order to be able to use many of the commands listed below. Type **EDITOR** before you do any editing.
n LIST	Lists screen n and makes screen n the current screen.

n CLEAR	Clears screen n, filling it with blanks. Also makes screen n the current screen. Type the word 'CLEAR', not the CLEAR key on the keyboard.
L	Lists the current screen. Also lists, at the bottom, the line that the Character Pointer (CP) is on. The CP is symbolized by something that looks like a 'T'. The number of the line that the CP is on will be listed last.
n T	Types line n and puts the CP at the beginning of line n.
n P <string>	Puts <string> on line n. Erases whatever was on line n. <string> may be up to 64 characters long. Any characters beyond that will be cut off. Note: Though the limit is 64, you should type a maximum of 63 characters so that the 64th will be a space. Note: Though, as was explained earlier, FORTH allows you to type up to 80 characters without giving an error, 64 is all that will fit on one line of disk space.
n E	Erases line n. (Fills the line with blanks.)
n D	Deletes line n and moves up all the following lines, and creates a new blank line 15. The contents of line n will be saved in a buffer so that the line may be reinserted later by the I command.
n I	Inserts the buffer from the previous D command into a new line created immediately preceding line n and then moves all the following lines down one line. The last line (line 15) will be lost.
n S	Spreads the current screen at line n, creating a new blank line immediately preceding line n and moving all the following lines down. The last line (line 15) will be lost.
F <string>	Finds <string> in the current screen, starting from where the CP is. The CP will be left at the end of the string.
B	Backs up the CP to the beginning of the string just found using F.
n M	Moves the CP n characters forward (or backward if n is negative).

X <string> Finds <string> and extracts it, thus shortening up the line. This is the main find-and-delete command. The search for <string> will start at the current position of the CP.

C <string> Inserts <string> in the current line at the CP. Warning: You cannot insert blanks using this command, except when they are inside a string, as in "57 *".

FLUSH Saves your work onto the disk. Actually, it only saves the changes you have made onto the disk, so that if you haven't yet edited the screen it will not be written to the disk.

WARNING:

DO NOT Type SAVE (this reformats your disk)
ALWAYS use **FLUSH** to save your work.

If you accidentaly use SAVE instead of FLUSH, it is suggested that you reformat your disk for FORTH starting with step 3 under MAKING YOURSELF A FORTH DISK in this manual. If you start from step 3, you will have lost the screen you just edited but you will not have lost the rest of the screens 18-89 on your disk.

EMPTY-BUFFERS Empties the disk buffers. This is what you type if you want FORTH to forget about everything you have just edited. The screen you were working on will be left as it was before you started editing.

n m COPY copies screen n onto screen m. Destroys any old information on screen m.

MARK Marks the entire current screen as having been modified. FLUSH will then write the entire screen onto the disk. Use MARK to copy a screen onto another disk in this way:

With a disk in the drive, type --> **n LIST MARK**
Then replace the disk with another one in the drive. Type --> **FLUSH**
Screen n will now be copied onto the new disk.

n m INDEX Lists, on the monitor, the first line of all the screens n through m. N must be smaller than m. This is why it is a good idea to put the title of each screen on line Ø of the screen: so that by INDEXing your disk you can quickly see what is on each screen. **INDEX** is used as a type of directory, or catalog, of the disk.
Note: using INDEX often writes any changes you have made onto your disk, so that if you want to

abort what you have done you should type **EMPTY-BUFFERS** before you type **n m INDEX.**

A HELPFUL HINT FOR EASIER EDITING

The author of this manual has found it much easier to re-enter a line using the "P" command when it needs changing than to edit it using the F, X, C, and other commands. A way not to have to retype the entire line is to do the following:

1) List the screen
2) Move the cursor up to the line that needs changing, using the up arrow with the CTRL key as described in the screen editing section of this manual.
3) Insert the letter "P" after the line number, which the computer printed when it listed the screen. Make sure to insert a space before and after the P.
4) By using the screen editor, make whatever changes you need to make to the line.
5) Press RETURN.
6) Hit the CLEAR key while holding down the CTRL key to clear the monitor.
7) Type **L** to list the screen and see if you have correctly changed it.

This is much faster and easier than editing the line any other way. In effect, you are retyping the entire line but taking advantage of the screen editor to save you work.

EXAMPLE:
If line 12 lists as
```
12 : JUNK   1Ø 2Ø RT +;/;
```
and you want to change it to
```
12 : JUNK   1Ø 2Ø ROT + ;
```
you first move the cursor to over the colon with the up and right arrow keys and the CTRL key as described in the section on the screen editor in this manual, then type CTRL-INSERT twice so that the screen says:
```
12   : JUNK   1Ø 2Ø RT +;/;
```
The cursor will now be over the second space after the 12.
Next you type **P** so that the screen says:
```
12 P : JUNK   1Ø 2Ø RT +;/;
```
Now move the cursor so it is over the "T" by typing the right-arrow with the CTRL key. Then press CTRL-INSERT:
```
12 P : JUNK   1Ø 2Ø R T +;/;
```
Now type **O** so the screen says:
```
12 P : JUNK   1Ø 2Ø ROT +;/;
```
Now move the cursor so it is over the first semicolon and type the space bar to replace the semicolon with a space:
```
12 P : JUNK   1Ø 2Ø ROT + /;
```
The cursor will be over the slash. Type CTRL-DELETE to get rid of the slash. The screen says:
```
12 P : JUNK   1Ø 2Ø ROT + ;
```

Press RETURN.
The computer should put "ok" at the end of the line.
Now, before you type anything else, you must press CTRL-CLEAR to clear the monitor. Otherwise, you may have problems.
Now, if you list the screen using **L** line 5 should list as:
5 : JUNK 1Ø 2Ø ROT + ;

Thus, by pressing a few keys, you can edit any line!
The only other commands you really need to use are:
CLEAR, LIST, L, D, S, FLUSH, and INDEX. (as well as P)

Of course, you can use the rest of the commands if you want to, but, as I said, I have found it easier to combine the **P** command with the screen editing capabilities of the ATARI as this example showed.

LEAVING THE EDITOR

You exit the editor whenever you type any of the following:
(This may not be a complete list)

FORTH
Any colon definition

Sometimes you will accidentally exit the editor. You will know this has happened when you type an editor command (such as **L**) and the computer prints a question mark. Reenter the editor by typing **EDITOR.** Your work will not have been lost, so don't worry!

To purposefully exit the editor, type --> **FORTH**

LOADING PROGRAMS FROM THE DISK

Once you have FLUSHed your program onto the disk, you must type (screen#) **LOAD** before you can run the program.

n LOAD interprets (compiles) screen number n from the disk.

Typing a program on a screen does not allow you to run it until you type (screen#) **LOAD**
LOAD interprets, or compiles, the screen so that you can run your program using any words you define on the screen.
You will usually want to type **FLUSH** after you edit a screen and before you load the screen so that the program will be saved on the disk.

WARNING: Make sure you know the difference between LOAD and LIST

Whenever you want to EDIT a screen, you must first **LIST** it.
Whenever you want to run a screen, you must first **LOAD** it.

Editing a screen that you have just LOADed instead of LISTed can cause your disk to be ruined.

DIFFERENCES BETWEEN fig-FORTH AND FORTH-79 (in Starting FORTH)

(This is for those who have read Starting FORTH which is about FORTH-79 and is a little different from fig-FORTH.)

(To those who have not read Starting FORTH: it is a good book!)

In fig-FORTH, to define a variable, type
n VARIABLE <name>
n will be the starting value of the variable.
(In the book, you just type VARIABLE <name>)

In fig-FORTH, each disk block is 128 bytes and each screen is made up of 8 blocks. Each block is 2 lines of the screen. This is important to know when reading from or writing to the disk from a program.
(In the book, a screen is the same as a block and both are 1024 bytes)

The word CREATE is not used in fig-FORTH as it is in FORTH-79. You should not normally use this word in fig-FORTH.

****

THE PRINTER

To print on the printer, type --> **PON**
Everything that is printed on the screen will now print also on the printer (assuming that the printer is turned on!) until you type **POFF**.
Note: 'ok', the FORTH prompt, will not print on the printer.

To quit printing on the printer, type --> **POFF**

To list a screen number n on the printer type --> **PON** n **LIST POFF**

PFLAG is the printer flag. It is a system variable that is set to 1 by **PON** and to 0 by **POFF**. If you type --> **1 PFLAG !**
this is the same as **PON**.
Typing **PFLAG @** will put the value of **PFLAG** on top of the stack, so that you can see whether the printer is on or not.

To use the printer, both it and the interface must be turned on.

To advance the paper in the printer, first turn the switch on the left of the printer to **LOCAL**. Then push the second switch from the left to **FORWARD** and hold it there until the paper has advanced far enough. When you are done, switch back to **ON LINE** from **LOCAL**. The printer will not accept input from the computer when the switch is on **LOCAL**.

You can also "rewind" paper back through the printer by following the above procedure but holding the switch on **REVERSE** instead of **FORWARD**.

A SELF-TEACHING PROGRAM TO START YOU IN FORTH

Boot your FORTH disk.

Note: if you are doing this as an assignment for a class type **PON** to connect the printer to the computer so that you will have a printed record of everything that you do. (Make sure that you are using a computer that has a printer attached and that has paper in the printer.) For information on the use of the printer read the section on the printer in this manual.

Type --> **1Ø 4 + .** (including the period).
The computer should print --> 14 ok
FORTH can be used in immediate mode; a set of commands need not be saved onto a disk to be executed. What you have just done is calculated 1Ø+4 in immediate mode. Words in the input stream (the stream of characters that you type or, later, that you will load from the disk) are interpreted in a left-to-right order, so that the number 1Ø was interpreted and put on top of the stack, followed by the number 4, then the word **+** (which adds the top of the stack to the second item on the stack) was executed, then the word **.** (which prints the number on top of the stack and removes the top number from the stack) was executed. The stack was left empty. If there had been anything on top of the stack before you typed "1Ø 4 + ." it would be still on top of the stack.

Type --> **: FOUR-MORE 4 + . ;**
Now type --> **8 FOUR-MORE**
The computer should print the number 12, followed by **ok** which is the FORTH prompt.
You have just typed a simple program: it adds 4 to the number on top of the stack and prints the sum. In FORTH, you define words (commands) that "chunk" a whole series of commands together so that the series of commands can be carried out merely by typing the word you have defined. In this case, you defined a sequence that will add 4 to the top of the stack as **FOUR-MORE**
The hyphen in FOUR-MORE is important; a word must not have spaces in it because looking for spaces is how the FORTH interpreter separates one word (command) from another. Note that you must always have at least one space between commands for this reason.

Type --> **: *2+4 2 * FOUR-MORE ;** (make ***2+4** one word)
Now type --> **1Ø *2+4**
The computer will print --> 24 (followed by ok)
This demonstrates 2 points:

1) Any character (except a space) can be used in the name of a word. The "*" and the "2" in "*2+4" are only characters in the word "*2+4". They have no other significance. However, the "2" and the "*" in the definition itself are significant, they stand for "put a 2 on top of the stack" and "multiply the top two numbers on the stack". The first word following the colon (:) can have any arbitrary characters in it, though, as you

are defining this word. ("*2+4" in the example above)
2) Previous definitions can be used in the definition of new words. You are using FOUR-MORE (which you defined above) as part of the definition of *2+4.

An explanation of what just happened to the stack follows.

The stack started empty. Then the number 10 was put on top of the stack, so the stack looked like this:

```
| 10 |
|    |
```

Then the word "*2+4" was executed. Remember that you defined "*2+4" as ": *2+4 2 * FOUR-MORE ;". First a 2 was put on the stack:

```
|  2 |
| 10 |
|    |
```

Then the word "*" was executed, so the stack looked like this:

```
| 20 |
|    |
```

Then the word "FOUR-MORE" was executed. Remember that you defined "FOUR-MORE" as ": FOUR-MORE 4 + . ;". First, a 4 was put on the stack:

```
|  4 |
| 20 |
|    |
```

Then the word "+" was executed:

```
| 24 |
|    |
```

Then the word "." was executed. This printed the number on top of the stack (24) and left the stack empty:

```
|    |
```

Now that you fully understand the workings of the stack (you do, don't you?) we shall continue...

Type --> **: FOUR-MORE 13 + . ;**
Of course this is an unusual definition for a word called FOUR-MORE, but it is to demonstrate a point.
The computer will respond --> FOUR-MORE Isn't unique
This is because you have now defined FOUR-MORE more than once.
Type --> **9 FOUR-MORE**
The computer will print 22. The latest definition of a word is

always used.
Type --> **5 *2+4**
The computer prints 14 instead of 23. (remember that you defined *2+4 to double a number and then execute FOUR-MORE.) The reason that 4 was added instead of 13 is that the current definition of FOUR-MORE at the time that you defined *2+4 was the definition that added 4. Note that by later changing the definition of FOUR-MORE you did not also change *2+4, even though it uses FOUR-MORE as part of its definition. A definition of a word is not changed by changing the definitions of the words that it uses after the word has been defined.

You should be able to see that if a word has a complicated, long definition that relies on previous definitions, it will be very hard to change the program; you will most likely have to redefine the words that are called upon and then retype the definition of the final word. Disk editing makes this easier...

Type --> **17 LIST**
Screen 17 should list as a bunch of hearts. This means the screen is totally blank; it has not been used before. Note: if you are doing this with the printer, you will note that the hearts are printed on the monitor only, not also on the printer. If screen 17 has something on it, then use a different screen by typing --> (screen#) **LIST** Then subsitute (screen#) for 17 from here on in this self-teacher. Note: valid screen numbers are 16-89.

Type --> **EDITOR**
This gets you into the EDITOR vocablulary, which means that you will be able to edit the disk. Whenever you want to start a session of editing the disk you must first type EDITOR.
Type --> **17 CLEAR** (Not the CLEAR key, which will clear the monitor screen, but the word "CLEAR")
This clears the hearts from screen 17 and fills screen 17 with blanks.
Type --> **L**
This will list the screen again. You can use **L** instead of **LIST** only when you are in the EDITOR vocabulary and you have already referenced the screen number using LIST or CLEAR.
The screen should now be blank.
Type --> **5 P : FOUR-MORE 4 + . ;**
This Puts the line of text following the space after the P on line 5 of the screen.
List the screen using **L**
You will notice that the definition of FOUR-MORE appears on line 5.
Now type --> **8 P : *2+4 2 * FOUR-MORE ;**
When you list the screen you will see the definition of *2+4 on line 8.
Type --> **FLUSH**
This saves the work you have done onto the disk.

Type --> **17 LOAD**
This reads screen 17 into the computer exactly as if you had

typed everything on the screen into the computer. If you have been following along from the beginning of this section, the computer will print --> FOUR-MORE Isn't unique
*2+4 Isn't unique
because those two words are not being defined for the first time.
Type --> **2Ø *2+4**
The computer will print 44. Now, how do we quickly change the definition of FOUR-MORE and *2+4 at the same time, to multiply by 2 and then add 13 instead of 4?
Type --> **EDITOR** (You must type "EDITOR" to reenter the editor, because by loading the new definitions of FOUR-MORE and *2+4 you left the editor. See the EDITING manual under LEAVING THE EDITOR if you would like a further explanation of why you have to type "EDITOR".)
Type --> **17 LIST**
We want to change the definition of FOUR-MORE, which is on line 5.
One way to do this is to completely retype line 5.
Type --> **5 P : FOUR-MORE 13 + . ;**
Now list the screen by typing **L**
Line 5 should contain the new definition of FOUR-MORE.

FLUSH and then LOAD screen 17 again. Remember that whenever you want to LOAD a screen, you must always type the number of the screen first: for instance here you type **17 LOAD**
If you just type "LOAD" without a number, the computer may lock up.
Type --> **5 *2+4**
The computer prints 23 because we have now not only redefined FOUR-MORE, we have also redefined *2+4 to use the new definition of FOUR-MORE. (Remember that a screen that is LOADed from the disk is treated by FORTH exactly as though it had been typed in by hand.)

You can see that the EDITOR makes it easy to change a program (a program is a set of definitions of words) by allowing you to redefine only one word and yet, at the same time, redefine all the words that call on that word.

Now let us see how we can better use the editor.

Get into the EDITOR and list screen 17.
If you do not remember how to do this, type --> **EDITOR** and then type **17 LIST.**
Let's give screen 17 a title.
Type --> **Ø P (MY FAVORITE SCREEN — *2+4 DEFINITION)**
Anything that appears within parentheses is a comment in FORTH; FORTH will ignore it. It is there only for the benefit of people (including yourself!) who will want to know what a screen does. Note that in a comment, there must be a space after the opening parentheses and that the comment continues until a closing parentheses is encountered by FORTH.

Now list screen 17. The comment should appear on line Ø.
What if we want to change a line? There must be an easier way

than retyping it, as we did before.
Read the section on the EDITOR in this manual and then change line Ø to say (MY SCREEN -- *2+4 WORD DEFINITION)

If you have too many problems, you can abort by typing:
EMPTY-BUFFERS
This makes the computer forget any editing you have done since you last FLUSHed your work onto the disk. You can then get back into the EDITOR, type 17 LIST, and start all over.

When you have successfully changed line Ø without having to retype the entire line by hand, you will have learned a valuable lesson: how to use the EDITOR.

Now change line 5 so that the word FOUR-MORE again adds 4 instead of 13 to the top of the stack and prints the result.
List screen 17 now to see that you changed it correctly.

Make sure to save your work onto the disk by typing **FLUSH.**

Now, why did you really want a title for the screen on line Ø in the first place?

Type --> **14 2Ø INDEX**
Line Ø of screens 14 through 2Ø will be printed. Titles for your screens should always appear on line Ø so that you can easily see what is on your disk by typing INDEX.

If you now LOAD screen 17, the title will not hamper the computer because it is a comment (it appears in parentheses).

You can now test your program out by typing --> **3 *2+4**
The computer should print 1Ø, which is the correct answer.

If you have been doing all this on the printer, now type --> **POFF**
This stops everything from going to the printer as well as the screen.
Note: if the printer was already not in use, then typing POFF does nothing.

A SELF-TEACHING LESSON ON REAL NUMBERS IN FORTH AND SOME BASIC LOOPS

Boot your FORTH disk.

If you are doing this lesson for credit for a class, type **PON** (you should be working on a computer that has a printer connected) so that you will have a printed record of the fact that you have done this record. (See section 3 in this manual about the printer if you do not understand why you should type PON.)

Type **5 4 / .**
The computer prints 1 as the answer because FORTH deals in integers unless specifically instructed otherwise.
Type **5 FLOAT 4 FLOAT F/ F.**
The computer prints 1.25
The word **FLOAT** will make the number on top of the stack into a real number. A real number (floating point number) uses 6 bytes, or 3 stack positions. When you typed **5** that put the integer 5 on top of the stack as a single-length integer. This used one position on the stack, or 2 bytes. When you typed **FLOAT** the integer 5 was dropped from the top of the stack and FORTH put a six-byte representation of the real number 5.Ø on top of the stack. The stack looked like this:

```
¦ high part of 5.Ø      ¦
¦ medium part of 5.Ø    ¦
¦ low part of 5.Ø       ¦
¦                       ¦
```

When you typed **4 FLOAT** the number 4.Ø was similarly put on top of the stack in 3 stack positions. Now the stack looked like this:

```
¦ high part of 4.Ø      ¦
¦ medium part of 4.Ø    ¦
¦ low part of 4.Ø       ¦
¦ high part of 5.Ø      ¦
¦ medium part of 5.Ø    ¦
¦ low part of 5.Ø       ¦
¦                       ¦
```

The word **F/** treats the top 6 positions of the stack as 2 real numbers and calculates the second real number on the stack divided by the top real number on the stack, removing the 2 real numbers from the stack and putting the quotient (a real number) on the stack. The stack now looked like this:

```
¦ high part of 1.25     ¦
¦ medium part of 1.25   ¦
¦ low part of 1.25      ¦
¦                       ¦
```

The word **F.** prints the floating point number on top of the stack and drops it from the stack. The stack was left empty.

Note: **F.** will print the number in Atari BASIC format. If the number is very large or very small, it will be printed in E notation. This is scientific notation. For instance, if you calculate 90000 times 90000, the result will be printed as 8.1E+9 which means 8.1 X 109.

One word that switches from integer to real representation on the stack is **FLOAT**, as we have seen. FLOAT is not the only way to get a real number onto the stack, though.

Type **FP 3 FP 1.5 F* F.**
The computer prints 4.5
FP is a word that converts the character string immediately following (and delimited by a blank) to a real number and puts that number on top of the stack.
Type **FLOATING 1.5 FP 2.1 F+ F.**
The computer prints 3.6
FLOATING and **FP** are synonymous; they do exactly the same thing.

Be sure that you know the difference between FLOAT and FP or FLOATING:
FLOAT makes the number on top of the stack real; there must already be an integer on top of the stack when you execute FLOAT.
FP and FLOATING interpret the following character string and convert it to a real number and put this number on the stack.

Now that you know how to put a real number on the stack, how do you convert it back to an integer?

Type **5 FLOAT 2 FLOAT F/**
Since you did not ask the computer to print the result, the real number 2.5 is now on top of the stack in the top 3 positions of the stack.
Type **FIX .**
The computer first converts the top three positions on the stack (which are one real number) into one integer (after rounding) and then prints the result, 3. (2.5 rounds up to 3.)

Get into the EDITOR and title a blank screen **(LESSON 2)**
On lines 1-4 put the following program:

```
 FP 0 FVARIABLE NUM
: REAL-DIV.   FLOAT NUM F! FLOAT NUM F@ F/ F. ;
: PROMPT-ME   ." TYPE 2 INTEGERS WITH A SPACE AFTER EACH" CR
   ." THEN TYPE REAL-DIV." CR ;
```

List the screen to make sure it is right, then save it on your disk (using FLUSH as always) and LOAD the screen.

Type **PROMPT-ME**
The computer should print:

```
TYPE 2 INTEGERS WITH A SPACE AFTER EACH
THEN TYPE REAL-DIV.
ok
```

Type **6 4 REAL-DIV.**
The computer should print 1.5

This program demonstrates a few points:

1) To declare a real variable, put a real number on the stack, then type **FVARIABLE** <name>
   The variable will be assigned a starting value of the real number on the stack
2) After FLOATing the top of the stack we must temporarily get it out of the way by storing it in a variable (there are other ways to get a real number out of the way, but that is one of the easiest) before we can FLOAT the second. Then we fetch the first real number from the variable and put it back on top of the second one on the stack
3) To print text we use the word ." followed by the text we wish to print. A quote mark marks the end of the text that is to be printed. There must be a space after the ." and the first space following ." will not be printed. The fact that the last character printed is a period is only a coincidence; ." is not a command at the end of the sentence; it is merely the end of the text and the quote that marks the end of the text.
4) CR is a command to print a carriage return and go to the next line.
5) If a definition is too long to fit on one line on the screen, you can continue it on the next line. Actually, a definition can be quite long, using up many lines on the screen. No one line can have more than 64 characters, however, as explained in the section on the EDITOR.

VALID COMMANDS USING REAL NUMBERS

You might have gotten the impression that any command can be made to work for real numbers by putting an F in front of it as in F+, F., F*, and FVARIABLE. This is not the case; there are only a few specific commands for real numbers. They are all described in the glossary of FORTH commands, appendix C in this manual.

This is a list of Floating Point commands in fig-FORTH:
(This list may not be complete)

FCONSTANT	FVARIABLE	FDUP	FDROP	FSWAP
FOVER	FLOATING	FP	F0	F!
F.	F+	F-	F*	F/
FLOAT	FIX	FLOG	FLOG10	FEXP
FEXP10	F0=	F=	F<	

There are no commands that will add, subtract, multiply or divide an integer and a real number. You must first change the integer to a real number and perform a floating-point operation (F+, F-, F*, F/, etc) to get a real result or change the real number to an integer and perform an integer operation to get an integer result. Integers and real numbers are not interchangeable; you must always be aware that a real number is not represented on the stack in the same way as an integer and that it uses 3 stack positions.

Some operations with real numbers leave a flag on the stack, though, and a flag is only one stack position. For instance, F= will leave a 1 (true) or a Ø (false) flag on the stack. You cannot SWAP or FSWAP this flag and any real number on the stack because the flag is one stack position and the real number is 3 stack positions. (For an explanation of F= or any other command, see the glossary of FORTH commands.)

DOING A LOOP

On the same screen, now put the following definition starting on line 7:

: TEN-TENS 11 1 DO 1Ø . CR LOOP ;

FLUSH the screen and LOAD it. (Ignore any **Not unique** messages that appear on the monitor.) Type **TEN-TENS**
The computer will print 1Ø ten times.
The command **DO** takes the top two numbers off of the stack (a 1 and an 11 in this example) and stores them as the index and limit to a loop (1 is the index; 11 is the limit.) Execution continues until the word **LOOP** is encountered. LOOP increments the index by one and then compares the index to the limit. If the index is less than the limit, everything from the first word after **DO** is repeated. If the index is equal to or greater than the limit, the loop is not repeated.
Note that the index is incremented by the word LOOP, not by DO, and that the index is not compared to the limit by DO but by LOOP. Because of these facts, a loop will always be executed at least once (even if the limit is less than the starting index) and that the index will go from the lower bound (the starting index) to one less than the upper bound. That is why 1Ø was printed ten times and not eleven; the index went from 1 to 1Ø, not from 1 to 11.

How do we find out what the index to the loop is?
On line 8 of your screen put
: DIGITS 1Ø Ø DO I . CR LOOP ;
FLUSH and LOAD the screen.
Execute the word DIGITS by typing **DIGITS**
The computer should print the digits Ø through 9.
The word **I** puts the index of the loop on the stack. Note again that the loop went from Ø to 9, not from Ø to 1Ø. The loop repeats while the index is less than the limit (has not yet reached the limit).

What if we want to execute a loop that increments by some value other than 1? We use **+LOOP** instead of **LOOP** in the following way:

On line 9 of your screen put
: ODDS 11 1 DO I . CR 2 +LOOP ;

When you execute the word ODDS (after FLUSHing and LOADing the screen) the odd digits will be printed.

DO ... +LOOP acts like DO ... LOOP except that the index is incremented by whatever number is on top of the stack when +LOOP is executed instead of always being incremented by 1.

On line 10 put
: BACKWARDS 10 20 DO I . CR -1 +LOOP ;
When you execute the word BACKWARDS the numbers 20 downto 11 should be printed. Remember that the loop is not executed when the limit has been reached. This means that when the increment is negative, the loop repeats while the index is greater than the limit.

How about nested loops?

On line 11 of your screen put
: NESTED 6 1 DO I 1 DO I . LOOP CR LOOP ;

Note that **I** always puts the index of the innermost loop that it is in on the stack and predict what NESTED should do.

Test your prediction? Were you right? This is what should have been printed by the computer:

```
1
1
1 2
1 2 3
1 2 3 4
```

Can you see why? The index for the outer loop acted as the limit for the inner loop! CR was not executed until the inner loop was done executing each time, so the inner loop printed its indices all on the same line.
When the limit was 1 and the starting index for the inner loop was 1, the inner loop executed itself once because a loop will always execute at least once no matter what the limit.
For each of the succesive times through the outer loop, the inner loop executed one time less than the index for the outer loop because a loop will repeat only while its index has not yet reached the limit.

TWO THINGS TO REMEMBER ABOUT LOOPS:

I is the only word that will allow you to see the index for a loop. It always puts the index of the innermost loop it is in on the top of the stack.

A loop must be contained entirely in a definition. You cannot define a word such as
: WRONG 10 0 DO ;
The words DO and LOOP (or +LOOP) must be in the same definition. Each DO must have exactly one LOOP or +LOOP associated with it in a definition.

In addition to DO ... LOOP and DO ... +LOOP there are the following control structures which you can read about in the GLOSSARY OF FORTH COMMANDS in this manual:

BEGIN ... UNTIL
BEGIN ... WHILE ... REPEAT
IF ... THEN
IF ... ELSE ... THEN

If you have been printing on the printer, type **POFF** to stop printing to the printer. Take the paper out of the printer and show it to your teacher to show that you have completed this lesson. If you have not been printing on the printer, typing **POFF** will not do anything.

You may now take your disk out of the drive and then turn the computer off.

GLOSSARY OF FORTH COMMANDS

A Quick note on the data stack (just 'stack' for short):
Arithmetic, comparisons, and most other operations in FORTH are done on the stack. The stack holds numbers in a First-In-Last-Out way. This means that if you put a 1 on the stack, then a 2, and then get the top item from the stack, you will get the 2 and the 1 will be left as the top item on the stack.

There are 2 ways to represent the stack:

or (6,10,4,1)

both represent a situation in which 1 is the top item on the stack, 4 is the second item, 10 is the third, and 6 is the fourth. Taking the top item off the stack would make 4 the new top item, 10 the new second item, etc.

Abbreviations used in this glossary:

n, n1, n2, etc. stand for integers. An integer uses 2 bytes, or 1 position on the stack.
An integer must be between -32768 and 32767. In some unusual cases, an integer is unsigned (must be positive) and must be between 0 and 65535, but for most purposes use the -32768 to 32767 boundaries.

c, c1, c2, etc. stand for an integer between 0 and 255, inclusive. An integer such as this can be stored in one byte of memory (but uses a full position when put onto the stack.) Integers between 0 and 255 can be treated as string characters because each character has an ASCII value between 0 and 255. Example: 65 is a capital A.

d, d1, d2, etc. stand for double-length integers, or for a combination of 2 integers. A double-length integer uses 4 bytes, or 2 positions on the stack.
Double length integers must be in the range of approximately plus or minus 2 billion. (2 to the 31st power).

fp, fp1, fp2, etc. stand for floating-point (real) numbers. These use 6 bytes, or 3 positions on the stack.

<string> stands for any character string as read in from the input stream at the time the command is executed.

addr, addr1, etc. stand for addresses in memory. An address is a single-length number, so **addr** is the same as **n**. The abbreviation **addr** is used to show that the number will refer to a memory location in the computer. Each byte of memory in the computer has its own address.

f, f1, etc. stand for boolean flags. A flag is a single-length number. Ø is false; anything other than Ø is true. Usually, 1 is used for true, though any other non-zero value would also work.

Whenever it is said that a string has a byte count in the first byte, that means that the first byte of the string holds a number between 1 and 255 that is the length of the string. The actual string starts in the next byte of memory. Example: if the string "COMP" is at address 5427 with a byte count, that means memory location 5247 holds the number 4 (the length), location 5248 has the ASCII value of **C**, 5249,525Ø,and 5251 hold the values for **O**, **M**, and **P** respectively. If the string is not said to have a byte count in the first byte, then the string actually starts at the given address.

In the STACK EFFECTS column, whatever the command expects on top of the stack will be on the left, and what will be left on the stack will be on the right. For instance, the stack effects of **MOD** are: (n1,n2--n3). This means the word **MOD** needs two arguments on the stack (n2 is on top of the stack; n1 is the second item from the top) and will replace the two arguments with one number. The explanations of what the arguments should be and what the result is will appear in the EXPLANATION column of each command.

****

COMMAND	STACK EFFECTS	EXPLANATION
STACK MANIPULATION COMMANDS:		
DUP	(n--n,n)	Duplicates an item on the stack
DROP	(n--)	Drops the top item from the stack.
SWAP	(n1,n2--n2,n1)	Swaps the first and second items on the stack.
OVER	(n1,n2--n1,n2,n1)	Copies the second item on the stack to the top of the stack.
ROT	(n1,n2,n3--n2,n3,n1)	Rotates the third item to the top of the stack.

Note: to rotate the other direction, type **ROT ROT**. That will do this:
(n1,n2,n3--n3,n1,n2)

-DUP (n--n) or (n--n,n)
Duplicates the top item on the stack if it is not Ø. Example: **5 -DUP** leaves two 5s on top of the stack, but **Ø -DUP** leaves only one Ø on top of the stack.

2DUP (d--d,d)
Duplicates a double-length integer on the stack, or does this: (n1,n2--n1,n2,n1,n2)

2SWAP (d1,d2--d2,d1)
SWAPs double-length integers.

2OVER (d1,d2--d1,d2,d1)
OVERs double-length integers.

FDUP (fp--fp,fp)
Duplicates a real number on the stack. Will also do this:
(n1,n2,n3--n1,n2,n3,n1,n2,n3)

FDROP (fp1--)
DROP for real numbers.

FSWAP (fp1,fp2--fp2,fp1)
SWAPs real numbers.

FOVER (fp1,fp2--fp1,fp2,fp1)
OVER for real numbers.

FROT (fp1,fp2,fp3--fp2,fp3,fp1)
ROT for real numbers.

>R (n--)
Move item from data stack (regular stack) to return stack (a special stack used by the FORTH interpreter to keep track of the program that is running.) Use this word only if you know exactly what you are doing. Otherwise, you could lock up the computer.

R (--n)
Copy top item from return stack to data stack.

R> (--n)
Møve top item from return stack to data stack.
Note: if, in a definition, you do **>R** and then **R>**, you will not crash the system. Both must appear in the same definition.

In this way, you can use the return stack as temporary storage for a number. Example: to define a word to duplicate the second item on the stack (n1,n2--n1,n1,n2) one way would be --> **: DUP-2ND >R DUP R> ;**

ARITHMETIC OPERATION COMMANDS:

+ (n1,n2--n3)
Adds n1+n2. Example: **1 4 +** would leave a 5 on the stack. The 4 and the 1 are lost.

- (n1,n2--n3)
Subtracts n1-n2. Example: **5 3 -** will leave a 2 on the stack.

* (n1,n2--n3)
Multiplies n1*n2.

/ (n1,n2--n3)
Divides (integer division) n1/n2. The quotient will be an integer, and will always round down. Examples: **10 2 /** leaves a 5 on the stack. **11 2 /** leaves a 5 on the stack, also. **5 9 /** leaves a 0 on the stack.

MOD (n1,n2--n3)
Leaves n1 mod n2 on the stack. **5 2 MOD** leaves a 1 on the stack. **9 5 MOD** leaves a 4 on the stack. Note: MOD is the same as the remainder of an integer division.

/MOD (n1,n2--n3,n4)
n3 is the mod, n4 is the quotient. Calculates the integer quotient and the remainder. **11 3 /MOD** leaves a 3 on top of the stack (the quotient) and a 2 as the second item on the stack (the mod).

*/ (n1,n2,n3--n4)
Calculates n1*n2/n3.

*/MOD (n1,n2,n3--n4,n5)
Multiplies n1*n2, then divides by n3. n4 is the mod (remainder) and n5 is the quotient.

MAX (n1,n2--n1) or (n1,n2--n2)
Leaves the maximum of the two top numbers on the stack.

MIN (n1,n2--n1) or (n1,n2--n2)
Minimum.

ABS (n1--n2)
Absolute value.

MINUS (n-- -n)
Change the sign of the number on top of the stack.

D+ (d1,d2--d3)
Adds double-length integers.

DABS (d1--d2)
Absolute value for double-length integers.

DMINUS (d-- -d)
Change the sign of a double-length integer.

F+ (fp1,fp2--fp3)
Adds two real numbers.

F- (fp1,fp2--fp3)
Subtracts fp1-fp2.

F* (fp1,fp2--fp3)
Multiplies fp1*fp2.

F/ (fp1,fp2--fp3)
Divides (real number division) fp1/fp2. If fp1 is 5 and fp2 is 2 then fp3 will be 2.5

FLOG (fp1--fp2)
Calculates the natural logarithm (to the base **e**) of fp1.

FEXP (fp1--fp2)
Calculates the antilog to the base **e** of fp1.

FLOG10 (fp1--fp2)
Replaces fp1 on top of the stack with its common logarithm (base 10).

FEXP10 (fp1--fp2)
Calculates the antilog (base 10) of fp1. This is the same as calculating 10 raised to the fp1th power. If fp1 were a 3, then fp2 would be 1000.

NUMERIC CONVERSION COMMANDS:

FLOAT (n--fp)

Converts an integer to a floating point number. To get the real quotient of 3/4 you would have to type: **3 FLOAT 4 FLOAT F/** This would leave .75 on top of the stack.

FIX (fp--n)

Converts a real number to an integer. The integer is rounded. FIXing 8.3 would leave 8 on top of the stack, but FIXing 8.6 would leave 9 on the stack.

FLOATING <string> (--fp)

Reads from the input stream and converts it to a real number. Example: if you type **FLOATING 6.42** the number 6.42 will be left on top of the stack. The string must be a valid FORTRAN-style string such as 1.23, -5.36E9, etc. If the string is invalid, fp will be an unpredictable value.

FP <string> (--fp)

Synonymous with FLOATING (above). FP does the same thing as FLOATING.

NUMBER (addr--d)

Converts the character string stored at addr (with a byte count in the first byte) to a double-length integer.

Note: if there is a decimal point in the string, then the location of the decimal point, relative to the end of the string, will be stored in the user variable DPL. Example: converting the string "123.4" would yield the double-length integer 1234 and DPL would be set to equal 1. If there is no decimal point in the string, DPL will be Ø.

Note: a double-length integer is stored on the stack with the high 2 bytes first. This means that to convert a double-length integer to a single-length integer, (assuming that the double-lenght integer is in the range of a single-length integer) you just have to **DROP** once. To do this: (d--n) just type **DROP**. To convert a single-length integer to a double length (n--d) just put a Ø on top of the stack.

COMPARISON

`<` (n1,n2--f)
: f will be 1 (true) if n1<n2. Otherwise, f will be Ø. Example: typing **3 8 <** leaves a 1 on top of the stack.

`>` (n1,n2--f)
: True if n1>n2.

`=` (n1,n2--f)
: True if n1=n2.

`Ø<` (n--f)
: True if n is negative.

`Ø=` (n--f)
: True if n=Ø.
Note: Ø= also will change the value of a true flag to false and of a false flag to true. Example: if the top of the stack is Ø (false) then Ø= leaves a 1 (true). If the top of the stack is anything other than Ø (true) then Ø= leaves a Ø (false). In this way, Ø= is a logical NOT operator.

`U<` (n1,n2--f)
: True if unsigned n1 is less than unsigned n2. Any integer can be treated as though it has a sign (in which case it ranges from -32768 to +32767) or as though it is unsigned (Ø to 65535). If the number is unsigned, then the sign bit is used as the highest-order bit.

`FØ=` (fp--f)
: True if the top of the stack is a real number Ø.

`F=` (fp1,fp2--f)
: True if fp1=fp2.

`F<` (fp1,fp2--f)
: True if fp1<fp2.

Note: - which is described on page C-4 can be used as a test for unequality because it will leave a non-zero value on top of the stack if two numbers are not equal. Note that this value may not be a 1 but it will be a non-zero value and thus be a true flag. To make any non-zero value a 1 use **Ø= Ø=**. This can be understood better if you read the note on the following page.

LOGICAL OPERATORS

AND (n1,n2--n3)

Logical AND. Both arguments must be true for the result to be true.
1 and 1 is 1.
1 and Ø is Ø.
Ø and 1 is Ø.
Ø and Ø is Ø.
Note: The arguments are ANDed bit by bit. For instance, if you calculate 25 and 11, the following happens:

```
25 is binary          11ØØ1
11 is binary          Ø1Ø11
Result is binary      Ø1ØØ1   which is 9.
```

Thus, typing **25 11 AND** leaves 9 on top of the stack.
The most common use of AND is with flags, so remember that if you are ANDing two flags that are 1 (true) or Ø (false) then both must be true for the result to be true.

OR (n1,n2--n3)

Logical OR. Either argument must be true for the result to be true. Note: the arguments are ORed bit by bit.
1 or 1 is 1.
1 or Ø is 1.
Ø or 1 is 1.
Ø or Ø is Ø.

XOR (n1,n2--n3)

Logical exlusive OR. One of the arguments must be true for the result to be true. Note: the arguments are XORed bit by bit.
1 xor 1 is Ø.
1 xor Ø is 1.
Ø xor 1 is 1.
Ø xor Ø is Ø.

Note: **Ø=** which is described on the previous page can be used as a logical NOT operator because it will change any non-zero value (true) to Ø (false) and Ø (flase) to 1 (true).

MEMORY ACCESS -- VARIABLES, ETC.

CONSTANT <string> (n--)

<string> (--n)

Creates a constant called <string> (<string> must be one word) with value n. Then whenever the word <string> is executed, n will be put on the stack. Example: if you type **12 CONSTANT MONTHS/YR** then whenever you type **MONTHS/YR** the number 12 will be put on the stack. Note: Never use the word CONSTANT inside a colon definition. You set all constants before you write colon definitions.

VARIABLE <string> (n--)

<string> (--addr)

Creates a variable called <string> (<string> must be a valid name) and gives the variable a starting value of n. When the word <string> is executed, the address of the variable in memory will be put on the stack. Example: if you type **9 VARIABLE LIVES** then the value of LIVES is nine. If you want to see what the value of LIVES is, you type **LIVES @**. Typing **LIVES** put the address of the variable on the stack, and then the word **@** fetched the value stored at that address and put it on top of the stack. If you want to change the value of LIVES to 8, for instance, you type **8 LIVES !**. This puts an 8 on top of the stack, then puts the address of LIVES on top of the 8, then executes the command ! which stores the second value on the stack in the address on top of the stack.

! (n, addr--)

Stores the 2-byte number (single-length integers use 2 bytes) in the address. This word is used most often for storing values in variables.

@ (addr--n)

Fetches the 2-byte value stored in the address on top of the stack and puts this single-length integer on top of the stack.

+! (n, addr--)

Adds n to the contents of the address. This is used most often for incrementing variables (or decrementing them if n is negative).
Example: if there is a variable called COOKIES which is set to 9, then typing **12 COOKIES +!** would change the value of COOKIES to 21.

C! (c, addr--)

Stores c into addr. This is used when you want to store the number c into only one byte, instead of 2 as you would with !. This is most useful for storing characters, which use only one byte of memory.

C@ (addr--c)

Fetches the value of the byte whose address is addr. Useful for fetching characters from memory.

ALLOT (n--)

Adds n bytes to the parameter field of the most recently defined word. In plain English (for those who don't speak FORTH fluently) this means that you can set aside memory in the computer to be used for storing an array of numbers.
Example: If you type
Ø VARIABLE SCORES 12 ALLOT
you have an array called SCORES which will hold 14 characters (variables get 2 bytes automatically, and allotting 12 more gives SCORES 14 bytes) or 7 single-length integers (integers use 2 bytes). Then, to access any position of the array, you merely type **SCORES** which puts the address of the beginning of the array on top of the stack, then add an offset. Example: to get the address of the 3rd position in the SCORES array, (which holds, let us say, integers) you type --> **SCORES 4 +**. SCORES put the address of scores on the stack. 4 + added the offset (for the 1st position in the array you would add nothing, for the 2nd you add 2 (remember that an integer uses 2 bytes), and for the 3rd you add 4. On top of the stack you now have the address of the 3rd position of the SCORES array.
Note: to easily calculate the offset into an integer array, you can define a word such as **: OFF 1 - 2 * + ;**
Then to put the address of the 3rd position

of the SCORES array on top of the stack, type **SCORES 3 OFF**
Can you see how this is the same as typing **SCORES 4 +** ?
WARNING: If you try to use more positions in an array than you allotted to the array, the computer can lock up. Do not, for example, try to access the 18th position of an array that you only alloted 17 positions to.

FCONSTANT <string> (fp--)
<string> (--fp)
Creates a real-number constant. Example: **FLOATING 3.1415 FCONSTANT PI** creates a constant called PI. When you type **PI** the real number 3.1415 will be put on the stack.

FVARIABLE <string> (fp--)
<string> (--addr)
Creates a real-number variable called <string> with a starting value of fp. Executing the word <string> will then put the address of the variable in memory on top of the stack. Example: if you type
FLOATING 100 FVARIABLE AREA
you will have a variable named AREA that is set to the value of 100. To change the value of AREA to 99.32 you type
FLOATING 99.32 AREA F!
(F! is explained below).
To put the value of AREA on top of the stack as a real (floating-point) number you type **AREA F@** (F@ is explained below)

F@ (addr--fp)
Put the real number stored starting at addr on top of the stack. This command is mostly used with floating point variables. Note that a real number is stored in 6 bytes.

F! (fp,addr--)
Store the real number fp in the 6 bytes starting at addr. This command is mostly used with floating point variables.

CMOVE (addr1, addr2, n --)

Move (or copy) n bytes in memory starting at addr1 to addr2. Example: if a string is stored at memory location 100 and is 6 bytes long, and you want to copy this string to start at memory location 500, you would type **100 500 6 CMOVE**
Now there is a copy of bytes 100 through 105 at bytes 500 through 505. Note that whatever was in bytes 500 through 505 is now lost.

FILL (addr, n, c--)

Fill n bytes in memory, starting at addr, with the value c.

ERASE (addr, n--)

Fill n bytes in memory with zeroes, starting at addr. Example: **1000 10 ERASE** will fill bytes 1000 through 1009 with zeroes.
Note: this command is useful for clearing arrays. If you have allotted 100 bytes to the array SCORES, then to clear the array you would type **SCORES 100 ERASE**

BLANKS (addr, n --)

Fills n bytes in memory, starting at addr, with the number 32, which is the ASCII code for a blank, or space. This is very useful for clearing strings in memory. If you have a string starting at address 5000 and with a length of 20 characters, then you can clear this string to blanks by typing **5000 20 BLANKS**

NUMBER BASES

DECIMAL (--)

Set decimal base (base 10).

HEX (--)

Set hexadecimal base (base 16).

OCTAL (--)

Set octal base (base 8).

BASE (--addr)

A system variable which holds the value of the current base. To change the base, store the value you want in BASE. To put the value of the current base on top of the stack, type **BASE @**

For instance, the definition of HEX is
: HEX DECIMAL 16 BASE ! ;

TERMINAL OUTPUT

. (n--)

Note: this is a period, or dot. It prints the number on top of the stack. Note that the number is then dropped from the stack. To print the number on top of the stack without losing it, you could type **DUP .**
Note: the number will be printed starting wherever the cursor is and there will be a trailing space printed after the number. Example: if you type **5 .** then the screen will look like this: **5 . 5 ok**
Note: the number on top of the stack is printed using the current base. This can be useful in the following way:
To see what the number 19 is in binary, you could type: **DECIMAL 19 2 BASE ! .**
The computer will print: 10011
Now type **DECIMAL** to get back into decimal base.

.R (n1,n2--)

Prints n1 right-justified in a field n2 characters wide. Example: if you type **37 5 .R** the computer will print 3 spaces and then the number 37, which is 2 digits. This way, the computer printed a total of 5 characters.

? (addr--)

Prints the single-length integer stored starting at addr. (Remember that an integer is 2 bytes.) Typing **500 ?** is the same as typing **500 @ .**

D. (d--)

Prints a double-length integer.

D.R (d,n--)

Prints a double-length integer, right-justified in a field of width n.

F. (fp--)

Prints the floating-point (real) number that is on top of the stack.

F? (addr--)
Prints the floating-point (real) number that is stored starting at addr.

CR (--)
Does a carriage return.

SPACE (--)
Prints a space.

SPACES (n--)
Prints n spaces.

." <string>" (--)
Prints a string on screen, delimited by a quote. The first space after the ." is ignored; it separates the command ." from the string to be typed. Note that there must be no space between the period and the first quote.
Example: **." I LIKE FORTH"** prints
I LIKE FORTH
Note that the computer would print "ok" after it though: I LIKE FORTHok
so you might want to type instead:
." I LIKE FORTH " with a space before the last quote so that the computer will print:
I LIKE FORTH ok
Another trick would be to type:
." I LIKE FORTH" CR so that the computer will print:
I LIKE FORTH
ok

EMIT (c --)
Prints the character whose ASCII value is on top of the stack. Example: **65 EMIT** prints a capital "A". **66 EMIT** prints a capital "B".
Note: it is useful to know the ASCII values for every character or to have a chart of these values.

TYPE (addr, n --)
Types a string of length n that is stored starting at the address in memory.
Example: if the string "STUDENTS" were stored in memory starting at location 1000,
1000 8 TYPE would print "STUDENTS".
1000 4 TYPE would print "STUD".
1003 5 TYPE would print "DENTS".

COUNT (addr1--addr2,n)

Puts the byte count of the string stored at addr1 and the address of the actual beginning of the string on the stack. This is used mostly to prepare the stack for the word **TYPE** (above.)

-TRAILING (addr,n1--addr,n2)

Adjusts the character count on the stack of a string stored at addr to ignore trailing blanks. This word is most often used immediately before TYPE so that trailing blanks are not printed after the string.

TERMINAL INPUT

KEY (--c)

Wait for someone to press a key on the keyboard, then put the ASCII value of the key on the stack. The character that is typed is not printed on the monitor. Example: if you type **KEY** then nothing will happen until you press a key. Then, for example, say you press the "A" key. The computer will now print "ok" but the "A" that you typed will not be shown on the screen. The number 65 (the ASCII value of "A") will be on top of the stack.
Note: a simple way to have the computer print the character you pressed is:
KEY DUP EMIT
This waits for you to press a key and prints it out while still keeping its value on the stack. Can you see why this works?

EXPECT (addr, n --)

Read n characters (or until carriage return) from the keyboard into the computer storing the input string at address addr in memory. An ASCII NUL Character (a zero) is put at the end of the string in memory. Example: **5000 20 EXPECT** will wait for you to type a string up to 20 characters in length, then move that string to start at memory location 5000. The next byte after the end of the string, no matter what length the string is (less than or equal to 20 characters), will be a zero. For instance, if you type **ATARIS ARE NEAT** that string will be stored in memory starting at 50000 (byte 50000 will hold the ASCII value of "A", etc.) and byte 50015 will hold a zero (the length of the string is 15, so

bytes 50000-50014 hold the string itself.) Because of this, you will be able to find the end of the string by searching the memory for a zero.

WORD (c--)

Read one word from the input stream, using the given character as a delimiter. The character is usually a blank (ASCII 32) as this way you can separate words from each other. The word is read to the address HERE (for an explanation of HERE see **HERE** in this glossary) with a preceding byte count. This means that HERE will contain the length byte and the actual string will begin at HERE + 1.

QUERY (--)

Waits for up to 80 characters to be typed at the keyboard (followed by <CR>) and moves the input string to an address called TIB (mnemonic for Terminal Input Buffer) and sets the system variable IN to 0. For explanations of IN and TIB see below Special System Words.

note: all of these are sequences of words. The entire sequence must appear in one definition. The stack effects of each word in the sequence are listed after the sequence. Where three dots appear (...) they stand for any group of commands that may be inserted in the sequence.

DO ... LOOP

DO (n1,n2--)

LOOP (--)

Do everything between the DO and the LOOP n1-n2 times. The computer keeps a counter for the loop. The counter goes from n2 to n1 but the loop does not repeat once the counter reaches n1. The counter is incremented by 1 each time through the loop. Example: **1Ø 1 DO CR LOOP** will print 9 carriage returns (1Ø-1 is 9). The counter goes from 1 to 9 and then when the counter is incremented to 1Ø (the word LOOP actually increments the counter) the loop does not repeat.

The way the computer actually keeps track of a loop is that the word DO puts limit and the counter on the Return Stack. The word LOOP increments the counter then compares the counter and the limit and, if the counter has not yet reached the limit, loops back to the first word after DO. If the counter has reached the limit, the computer drops the counter and limit from the Return Stack and continues with the first word after LOOP.

I (--n)

Puts the counter (index) of the loop on the stack. Use this word inside loops, as in **1Ø Ø DO I . CR LOOP** which will print the digits Ø through 9.

LEAVE (--)

Causes the index to equal the limit so that the loop will be exited when the word LOOP is executed.

DO ... +LOOP

DO (n1,n2--)

+LOOP (n--)

Just like DO ... LOOP except that the increment does not have to be 1. Use in

this way:
10 1 DO I . CR 2 +LOOP will print the odd integers 1 to 9. The increment can be negative:
-10 0 DO I . CR -1 +LOOP will print the negative numbers 0 to -9. (The loop does not repeat once the limit has been reached, so -10 is not printed.)
Note that the words **I** and **LEAVE** may be used in this loop exactly as in a DO ... LOOP loop.

BEGIN ... UNTIL

BEGIN	(--)
UNTIL	(f--)

Do everything between the BEGIN and UNTIL until there is a true flag (a non-zero value) on top of the stack when the word UNTIL is executed. The word until takes the top item off the top of the stack and, if it is a 0, loops back to the first word after BEGIN. The word BEGIN does nothing except mark the beginning of the loop.
Example: if you define TEST as follows
: TEST ." PRESS A TO CONTINUE " BEGIN KEY ASCII A = UNTIL ." THANK YOU" CR ;
the word TEST will wait print a message, wait for you to press the A key on the keyboard, and print another message. If you press any key other than A the computer will simply wait for you to type another key, until you type A.
Note that you can create an infinite loop by putting a 0 on the stack before saying **UNTIL**, as in **BEGIN CR 0 UNTIL** which will print carriage returns forever. FORTH provides another way to do this. The word **AGAIN** which does not appear other than right here in this glossary acts just like **0 UNTIL.** You could write a word to print 'FOREVER' over and over again as follows:
: INFINITY BEGIN ." FOREVER" CR AGAIN ;
If you execute INFINITY there is no escape except pressing the SYSTEM RESET key or turning the computer off.

BEGIN ... WHILE ... REPEAT

BEGIN	(--)
WHILE	(f--)
REPEAT	(--)

In this loop, a flag on top of the stack is tested by the word **WHILE.** If the flag is false (0) execution skips to the first

command after **REPEAT.** Otherwise, execution continues up to **REPEAT**, then loops back to the first word after **BEGIN.**
Between BEGIN and WHILE there should be a test that leaves a flag on the stack. The word WHILE takes the flag off of the stack and, if the flag is true, everything between WHILE and REPEAT is executed. Then the program loops back to the first word after BEGIN to repeat the loop. The words BEGIN and REPEAT do nothing except mark the beginning and end of the loop.
Example: here is a program that will print the numbers from 1 to 10. (You could use a DO loop, as shown on the previous page, but this is an example of a WHILE loop)

```
0 VARIABLE X
: TEST2  1 X ! BEGIN X @ 11 < WHILE X @ .
   CR X @ 1 + X ! REPEAT ;
```

First, we declared our variable X. Then we defined the word TEST2 to store 1 into X and, while X was less than 11, print X and then add 1 to its value. The word TEST2 should, therefore, print the numbers 1 through 10 when executed.

IF ... THEN or IF ... ENDIF (THEN and ENDIF are synonyms)

```
IF      (f -- )
THEN    ( -- )
```

If the flag is true then everything between the **IF** and the **THEN** will be executed. Otherwise, the program will jump to the first word after THEN. The word **IF** expects a flag on the stack and will remove this flag from the stack. The word **THEN** simply marks the end of the conditional. Everything after THEN will be executed whether or not the flag is true.
Example: A word that will print 'NOT EQUAL' if the top two items on the stack are equal and 'EQUAL' if they are equal is defined below:

```
: WHICH  - IF ." NOT " THEN ." EQUAL " ;
```

Note that this word takes advantage of the fact that if two numbers are not equal their difference is non-zero and any non-zero number can act as a true flag.
If, after defining WHICH, you typed:
4 5 WHICH the computer would print:

NOT EQUAL
If you typed:
23 23 WHICH the computer would print:
EQUAL
This works because the word NOT is only printed if the numbers are not equal but the word EQUAL is printed either way.

IF ... ELSE ... THEN or IF ... ELSE ... ENDIF

IF (f --)
ELSE (--)
THEN (--)

This is like an IF ... THEN loop except that if the flag is true, everything between IF and ELSE is executed and if it is false everything between ELSE and THEN is executed. Either way, the program continues with the first word after THEN.
The word IF takes a flag off of the stack. The words ELSE and THEN do nothing to the stack; they only mark the positions that the program will jump to.
Example: the following definition will compare the top two numbers on the stack and print 'YES' if they are equal and 'NO' if they are not equal:
: WHAT = IF ." YES" ELSE ." NO" THEN CR ;
Whether or not the two numbers on top of the stack were equal, the program printed a Carriage Return (<CR>) after printing the proper message.
If you type **8 8 WHAT** the computer will print YES.
If you type **8 -4 WHAT** the computer will print NO.

SPECIAL SYSTEM WORDS

HERE (-- addr)

Puts the address of the top of your dictionary on the stack. This is a special address because the word **WORD** moves text to that address and because your dictionary ends right below that address. For instance, if HERE puts 3Ø984 on the stack, you should not use any address below 3Ø984 unless you know exactly what you are doing and it is for a special purpose.
Every time that you define a new word, it gets added to your dictionary and the value

returned by HERE will increase. The value returned by HERE will not change unless you change the length of your dictionary by defining new words or getting rid of old words.

PAD (-- addr)

Puts the address of the pad on top of the stack. The pad is an area of memory that can be used by the programmer for whatever use he desires and is of length at least 80 bytes. The pad is often used as an output buffer which means that text is moved to the pad and then whatever text is at the pad is typed out. If you need to manipulate text, the pad is a good place to do this. The pad starts about 68 bytes beyond HERE.

?TERMINAL (--f)

The flag will be true if a terminal break request is present. This means that if a key on the keyboard has been pressed, a true flag will be put on the stack. Otherwise, a flase flag will be put on the stack.

SP@ (-- addr)

Addr will be the address of the top of the stack. Example: If there is a 7 on top of the stack and you type
SP@ @ .
The number 7 will be printed and 7 will still be on top of the stack.

ABORT (-- empty stack)

Empties the stack, aborts whatever it was doing, enters the FORTH vocabulary, and enters the main FORTH interpreter mode. ABORT is usually executed by FORTH after every error, and you can use it in your programs if a situation arises that would cause you to want to abort your program.

COMMENTS

((--)

Begin a comment. The comment must end with a closed parentheses. A comment is ignored by FORTH and is in a program only for the reader of the program. *Example:*
(FORTH IS WONDERFUL)

is a comment. Note that a comment must have at least one space after the opening parentheses but does not need a space before the closing parentheses.

DISK ACCESSING WORDS

LIST (n--)

List screen number n and make screen number n the current screen for editing. See the editing section of this manual for a full description of this word.

LOAD (n--)

Load screen number n from the disk as though everything on the screen had been typed into the computer from the keyboard. See the editing section in this manual for more on this word.

BLOCK (n--addr)

Block number n will be read from the disk into a buffer that the computer finds avbailable. The address of the beginning of this buffer will be put on top of the stack. A block is 128 bytes long and is the basic unit of a FORTH disk. There are 8 blocks to a Screen.
Example: if you want to print out the 5th through 10th characters on screen 20, you execute
20 8 * BLOCK 5 + 6 TYPE
See if you can figure out how this works.
Note: every 2 lines on a screen are 1 block, so to print the 3rd line of screen 30 you execute
30 8 * 1 + BLOCK 64 TYPE
Note: To print out the 2nd line of a block (a block is 2 lines) just add 64 to the starting address. To print line 1 of screen 25 (remember that lines are numbered 0-15) execute
25 8 * BLOCK 64 + 64 TYPE

Note: if all the buffers are full, the word **BLOCK** will pick a buffer and, if it has been updated (see below under **UPDATE**) it will write the contents of this buffer back onto the disk and then use this buffer for the block now being loaded. If a buffer has not been updated, FORTH will overwrite the buffer with the block now being loaded.

FLUSH (--)

Writes any disk buffers that have been updated (see **UPDATE** below) onto the disk. This command is used to write any block onto the disk. Some explanation of FLUSH is given in the editing section of this manual. FLUSH will see if each disk buffer has been updated, and if it has it will write the buffer to its appropriate block on the disk and then mark the buffer as being empty. If the buffer has not been updated, FLUSH simply marks it as being empty, because if the contents of the block have not been changed, FORTH needs not bother to write the block back onto the disk.

UPDATE (--)

Marks the last disk buffer that was accessed as updated, or having been changed, so that a subsequent **FLUSH** command will write the block stored in that buffer to the disk.
Note: The EDITOR command, **MARK**, (see the section on editing in this manual) UPDATEs all 8 blocks in the most recently LISTed or CLEARed screen.

EMPTY-BUFFERS (--)

Empties all disk buffers without writing them to the disk (erases all the disk buffers.) This command is used to forget any changes that have been made to the disk since it was last written to.
Note: Remember that a block is written to a disk without the use of the FLUSH command if the buffer the block is in is needed for some other purpose. For this reason, you should use the EMPTY-BUFFERS command as soon as you realize that you do not wish to save your work. If you wait, the work may be saved to the disk inadvertantly.
See the editing section of this manual for another explanation of this command.

DR1 (--)

Sets the system variable **OFFSET** to 720 so that drive 2 will be the current disk drive. See **OFFSET** in this manual.
Note: the screens on drive 2 are numbered 90 through 179 (remember that on drive 1 they are 0 through 89). Thus, two ways to list screen 20 of drive 2 are:
110 LIST or

DR1 20 LIST
As you can see, DR1 sets an offset of 720 blocks (remember there are 8 blocks per screen). This offset stays in effect until the word **DRO** is executed or the variable **OFFSET** is changed.

DRO (--)

Sets the system variable **OFFSET** to 0 so that drive 1 will be accessed.

R/W (addr,n,f--)

Read or Write a disk block. This command is very seldom used by the programmer because he usually can do everything he needs with the the commands listed above.
If f is 1, block n will be read from the disk into the 128 bytes starting at addr.
If f is 0, the 128 bytes starting at addr will be written to block n.
This command gives the programmer complete control over the contents of the disk and should be used with caution.
Note: for most purposes, the other commands listed above will suffice for disk accessing. R/W is used by the BLOCK and FLUSH commands.
There may be no error message printed if the R/W command is unsuccessful.

-DISK (addr,n1,n2,f--n3)

This is another seldom-used disk-read/write command. It reads (if the flag is 1) or writes (if the flag is 0) a sector (128 bytes) to or from the disk. Addr is the starting memory location to be written to the disk or that the disk should be read into. N1 is the sector number (0 to 719) and n2 is the drive number (1 to 4). N3 will be 0 if everything worked, or it will be the DOS error number if there was an error.

DEFINING WORDS

: <string> (--)

Begin a colon definition of the first word (delimited by a blank) following the colon. The definition must be in high-level FORTH, and the definition will be compiled into the parameter field address of the word being defined.

Example:

: SQUARE DUP * ;

will define a new word, or command, called SQUARE that will execute first the command **DUP** and then the command *****.

; (--)

End compilation of a colon definition.

VARIABLE

This defining word is explained in this manual in the MEMORY ACCESS section of this glossary.

CONSTANT

This defining word is explained in this manual in the MEMORY ACCESS section of this glossary.

<BUILDS ... DOES> ...

<BUILDS (--)

DOES> (-- addr)

This is a structure used to create new defining words. <BUILDS and DOES> go in a colon definition of a new defining word. The section after the word <BUILDS will be executed by the defining word in creating a new command. The section after DOES> will be executed by the word defined by the defining word. The address of the parameter field of the word defined by the defining word will be put on the stack before execution of the code after DOES> begins.

Example:

The way to create an array normally would be (for instance, an array called THESE of 10 integers):

0 VARIABLE THESE 18 ALLOT

(because THESE will need 20 bytes, 2 of which it is automatically given by the word VARIABLE. See ALLOT in this manual for an explanation.)

Then, the way to get the address of one of these integers (for use with @ or !) would be (for instance, to fetch the 3rd position of the array):

```
THESE 4 + @
```

(because the first position is THESE+0, the second is THESE+2, etc. Again, see ALLOT if you do not understand arrays.)

A much easier way would be to define a defining word called INT-ARRAY that would define THESE as an array that could be accessed easily.

```
: INT-ARRAY   <BUILDS 0 VARIABLE 1 - 2 *
        ALLOT DOES> SWAP 1 - 2* + ;
```

Now when we type

```
10 INT-ARRAY THESE
```

we will create (define) an array called THESE that has room for 10 integers. We have just done the same thing as we did before (see above) when we created the array.

To get the 3rd position of THESE we merely now have to type

```
3 THESE @
```

To store the number 73 in the 5th position of THESE, we merely type

```
73 5 THESE !
```

Now we shall see how this works. The <BUILDS definition of INT-ARRAY creates an integer array with the same number of positions as the number that is on the stack when INT-ARRAY is executed. The word VARIABLE creates a variable named whatever the next word in the input stream is, and the next word that is found in the input stream when INT-ARRAY is executed will be the name of the array we are defining, so the array will be first defined as a variable. Then some calculations are done to the number on top of the stack so that the right number of bytes are allotted to the array.

The DOES> definition of INT-ARRAY defines what the word defined by INT-ARRAY (in this case THESE) will do. The first thing that it will do is put its own address (parameter field address) on top of the stack. All words defined by DOES> put their own address on top of the stack before doing anything else. Then it will SWAP this address with the index to the array, which is expected to be on the stack when THESE is executed. Then some calculations will be done so that the address of the nth position of THESE (n is the index) is left on the stack.

The word INT-ARRAY can now be used to define any number of integer arrays, for instance 100 INT-ARRAY SCORES would define the array SCORES.
The author hopes that this example of defining a defining word called INT-ARRAY explains the use of <BUILDS ... DOES>.

CODE <string> (--)

Begin an assembly-language definition of the first word following the word CODE.
When this word is executed, the assembly-language routine in the definition will be executed. The definition will be ended with a JMP instruction to some other routine.

;CODE (--)

;CODE is used in this way:
: <string> ... ;CODE <assembly-language>
<string> is the name of a defining word being defined (see <BUILDS ... DOES> to find out what a defining word is). The difference between this and <BUILDS ... DOES> is that you need no word such as <BUILDS to start you off and the DOES> part is defined in assembly language instead of in high-level FORTH. Everything following the word ;CODE should be an assembly-language routine that will be executed by any word defined by the defining word (<string>).

IMMEDIATE (--)

Make the most recently defined word immediate so that it will be executed instead of compiled in a definition.
EXAMPLE:
: HELLO ." HELLO THERE" ; IMMEDIATE
defines HELLO to print HELLO THERE instead of being compiled in a future definition.
For instance, if you now type the following:
: DOTHIS 3 HELLO 4 + . ;
the computer will print HELLO THERE while compiling DOTHIS. Executing DOTHIS will print the number 7 and not print HELLO THERE.

[COMPILE] (--)

Compile the next immediate word, instead of executing it. If you defined HELLO as in the example above, then typed:
: DOTHAT 3 COMPILE HELLO 4 + . ;
the computer would treat HELLO as though it were not an immediate word. It would

simply compile DOTHAT, and when DOTHAT was executed the computer would print
HELLO THERE 7

COMPILE (--)

When the word now being defined is run, compile the next word into the dictionary. Example:
You type
```
: TYPE-IT   ." GOOD " ;
: DO-IT   5 . COMPILE TYPE-IT ; IMMEDIATE
: TRY-IT   3 DO-IT 7 + . ;
```
The computer immediately prints 5 because DO-IT is an immediate word that prints 5. However, if you now type TRY-IT the computer prints
GOOD 10
The computer prints **GOOD** because the word DO-IT immediately compiled TYPE-IT, which prints "GOOD", into the definition of TRY-IT. The number 10 was printed because TRY-IT also prints the sum of 3 and 7.
Try to understand how this example worked, and if you do, then you understand COMPILE (as well as IMMEDIATE).

, (n--) (comma)
Compiles the number n into the dictionary at HERE and adds 2 to the address of the top of the dictionary.
Example:
```
1 VARIABLE ODDS 3 , 5 , 7 , 9 ,
```
creates an array called ODDS that contains the odd integers 1 through 9.

' <string> (--addr) (single-quote)
Puts the address of the parameter field of the word <string> on top of the stack.
Note: <string> is the next word in the input stream when ' is executed.
Example:
```
' SOMECOMMAND
```
would put the parameter field address of SOMECOMMAND on top of the stack.

[...] (--)
Everything between the [and the] will be executed immediately, not compiled. The command [stops compilation by setting STATE to equal 0. (See STATE under SYSTEM VARIABLES in this glossary.) The command] sets STATE to equal 192, thus reentering compiler mode. Example:
If you type
```
: TEST   7 [ ." COMPILING TEST " ] 8 + . ;
```

the computer will immediately print "COMPILING TESTIT" and TESTIT will be compiled as : TESTIT 7 8 + . ; so that executing TESTIT will cause the computer to print 15.

VOCABULARIES AND VLISTING AND FORGETTING

FORTH (--)

Enter the FORTH vocabulary (make FORTH the context vocabulary). When you enter a command, if it is not in the FORTH vocabulary the computer will not recognize it. All of the commands in this glossary are in the FORTH vocabulary.

EDITOR (--)

Enter the EDITOR vocabulary (make EDITOR the context vocabulary). When you enter a command, the EDITOR vocabulary will be searched. If the command is found there, that command will be executed. If the command is not found in the EDITOR vocabulary, the FORTH vocabulary will be searched for the command. This is how all vocabularies other than the FORTH vocabulary work. The purpose of having a separate vocabulary is so that there can be more than one command with the same name, as long as they are in different vocabularies.
A list of commands in the EDITOR vocabulary appears in the Editing section of this manual.

ASSEMBLER (--)

Enter the ASSEMBLER vocabulary (make ASSEMBLER the context vocabulary). The APX manual EXTENDED fig-FORTH by Patrick L. Mullarky explains the ASSEMBLER that is on your disk.

VLIST (--)

List all the commands in the context vocabulary.

FORGET <string> (--)

Forget all definitions after and including <string>. For instance, if you type
: THIS ." THIS" ;
:THAT ." THAT" ;
and then type **FORGET THIS** the computer will no longer recognize THIS or THAT.
The FORGET command is used if you want to get rid of old versions of a word so that you will get no more "Isn't unique" messages or to get rid of unneeded words to save memory.
NOTE: You cannot FORGET the self-booting FORTH definitions because they are protected by FENCE (see FENCE under SYSTEM VARIABLES in this glossary).

DEFINITIONS (--)

This makes the context vocabulary also the current vocabulary. This means that any definitions you now type will be in the context vocabulary.
EXAMPLE: to define a new EDITOR command you type **EDITOR DEFINITIONS** then type the definition of the new command. When you are done, it is a good idea to type
FORTH DEFINITIONS

VOCABULARY <string> (--)

Create a new vocabulary called <string>.
EXAMPLE: The EDITOR vocabulary was created by the creator of the fig-FORTH MASTER disk typing **VOCABULARY EDITOR.**
Note: FORTH allows for a maximum of 4 vocabularies, so you can only define a maximum of 1 vocabulary of your own (in addition to the FORTH, EDITOR, and ASSEMBLER vocabularies.)

NUMERIC OUTPUT FORMATTING COMMANDS

Note:

To get fancier output of numbers than the commands .,U., etc. (above, under terminal output) allow, you can format the output of numbers.
The format for the number is entered between the <# and #> commands (explained below) so all the commands listed below must appear within the phrase <# ... #>

<# (--)

Begins the number formatting process. There must be an unsigned double-length number on top of the stack, or an unsigned single-length integer as the second item and a 0 as the top of the stack. If, in the output, you are going to want to print the sign of the number, there must be a signed number as the third item on the stack.

#> (d--addr,n)

Completes the number formatting process and leaves the length of the string and the address of the string on top of the stack, so that a subsequent **TYPE** command will type out the text string.

SIGN (n,d--d)

Inserts the sign of n into the output string.

(d1--d2)

Converts one digit of the double-length number on top of the stack, using the current base, into the otput character string.

#S (d--0,0)

Converts the double-length number into the output text string.

HOLD (c--)

Inserts the character whose ASCII value is c into the output string.

Note: The stack effects of an entire number formatting sequence are as follows:
<# ... #> (d--addr,n) or (n1,0--addr,n2)
<# ... SIGN #> (n1,|d|--addr,n2) or (n1,|n1|,0--addr,n2)
where |n| means the absolute value of n.

Note: numbers are converted "backwards"; the first digit converted is the character printed last.

EXAMPLE:
The following definition of **TIME.** will print out the time if the number of seconds is on top of the stack.

```
58 CONSTANT ASCII-OF-COLON
: BASE-SIX   6 BASE ! ;
: 2-DIGITS   DECIMAL # BASE-SIX # ASCII-OF-COLON HOLD ;
: TIME.   <# 2-DIGITS 2-DIGITS DECIMAL #S #> TYPE CR ;
```

If you were to put the double-length number 4610 on top of the stack (or the single length number 4610 followed by 0) and then execute the word **TIME.** the computer would print 1:16:50.

The second number after each colon is converted in base 10, the first digit after each colon is converted in base 6, and the hours are converted in base 10.

ANOTHER EXAMPLE:
To define a word **NET** that would print the number -3294 as $32.94-
we would use the following definition:

```
36 CONSTANT ASCII-$
46 CONSTANT ASCII-POINT
: PREPARE   DUP ABS 0 ;     ( n--n,!n!,0)
: NET   PREPARE <# SIGN # # ASCII-POINT HOLD
   #S ASCII-$ HOLD #> TYPE ;
```

Now, typing -3294 NET would cause the computer to print $32.94-
If we wanted the computer to print -$32.94 instead, we would have defined NET as

```
: NET   PREPARE <# # # ASCII-POINT HOLD
   #S ASCII-$ HOLD SIGN #> TYPE ;
```

CONSTANTS USED BY THE FORTH SYSTEM

BL (--n)

Puts the number 32 (ASCII blank character) on top of the stack.

C/L (--n)

Puts the number 64 (characters per line on a disk screen) on top of the stack.

0 (--n)

Puts the number 0 on top of the stack. The most-often used numbers are defined as constants because a constant is executed faster than a number that has to be interpreted every time that it is used.
Note: if your program needs to use a number more than 3 times, it is to your advantage to declare that number as a constant, for example: 18 CONSTANT 18
Note: most numbers (those that have not been declared as constants) do the same thing that constants do--they put their own value on top of the stack. They execute more slowly than constants, however.

1 (--n)

Puts the number 1 on top of the stack.

2 (--n)

Puts the number 2 on top of the stack.

3 (--n)

Puts the number 3 on top of the stack.

B/SCR (--n)

Puts the number 8 (the number of disk blocks per disk screen) on top of the stack.

B/BUF (--n)

Puts the number 128 (the number of bytes per disk block or disk buffer) on top of the stack.

FIRST (--addr)

Puts the beginning address of the of the disk buffers (the address of the first disk buffer) on top of the stack. This number is: HEX 3BE0

LIMIT (--addr)

Puts the address of the first byte after the last disk buffer on top of the stack. This is the first byte in memory that is "free", or not used by the FORTH system.
Note: All memory between FIRST and LIMIT is used for disk buffers.

SYSTEM VARIABLES (USER VARIABLES)

Note:

All of these commands are variables.
They all do the following to the stack:
(--addr)
addr will be the address that must be fetched with @ or changed with ! to get or change the value of the variable.
In this section, the explanations of the commands will only be of what the variables hold.

SCR

The number of the current screen (the screen most recently referenced by **LIST** or **CLEAR**)

OFFSET

The offset (in number of blocks) for the disk drives. The command **BLOCK** adds the number stored in OFFSET to the top of the

stack before it gets the block with that number. Since block number 720 is block 0 of drive 2, storing 720 in OFFSET will cause drive 2 to be referenced by future BLOCK (and most other disk-referencing) commands.

BLK

The number of the block being interpreted (as by a LOAD command). BLK will be set to 0 if input is coming from the Terminal Input Buffer (the computer is being operated from the keyboard and not, at the moment, from the disk).

TIB

The address of the Terminal Input Buffer, into which all text from the keyboard is put before being interpreted.

IN

The offset (in number of bytes) into the input text buffer (whether the TIB or a disk buffer) to the text being currently interpreted. The command **WORD** changes the value of IN when it finds text in the input buffer.

WARNING

If WARNING holds a 1, there is a disk in drive 1 and error messages will be read from the disk. If WARNING holds a 0, there is not a disk with error messages and if there is an error the computer will simply print "ERROR # n", where n is the error number. If WARNING holds a -1, then the command **(ABORT)** will be executed whenever there is an error.

DP

The address of the first free byte of memory above the dictionary. The command **HERE** reads DP onto the top of the stack. The command **ALLOT** changes the value stored in DP.

R#

The location of the Character Pointer in the current screen. This variable is used by the EDITOR.

STATE

If STATE is 0, the system is in immediate mode (interpreting or executing). Otherwise, the system is in compiling state

(the input stream is being compiled into the dictionary.) While a new word is being defined, STATE is non-zero.

FENCE

No definitions can be forgotten using **FORGET** below this address. The contents of FENCE must be changed to FORGET a definition below this address. The word **SAVE** sets FENCE so that no self-booting definitions can be forgotten.

HLD

This is the address of the latest character of text that has been converted from a number during a formatted numeric output conversion.

FLD

This is the field width (number of characters) for formatted output of numbers.

CSP

The data stack pointer is stored in this variable during compilation of a new definition for error checking. If the data stack pointer is not the same after compilation, there will be an error.

DPL

The number of digits to the right of the decimal point in a double-length integer that has been input. If a number is typed with a decimal point in it, the number will be double-length, the decimal point will be taken out and DPL will be set. Example: If you type **12.3** the stack will have a double-length number (123) on top and DPL will be 1, because there is 1 digit to the right of the decimal place. If no decimal point is typed, DPL will hold -1.

OUT

This is an offset into the Text Output Buffer. The command **EMIT** increments OUT.

CURRENT

This holds the address of the link field address of the last word defined in the current vocabulary. In other words, it is a pointer to the vocabulary into which new definitions will be put. (See under VOCABULARIES in this glossary.)

CONTEXT

This holds the address of the link field of the last word in the vocabulary that is to be searched first in searching the dictionary for words. In other words, it is a pointer to the context vocabulary. (See under VOCABULARIES in this glossary.)

VOC-LINK

This holds the link field address of the most recently defined vocabulary. (It could hold the link field address of the word EDITOR, for example.)
When a new vocabulary is defined, it is linked to the the previously defined vocabulary by using the number in VOC-LINK, then it changes VOC-LINK to point to itself.

DEBUGGING COMMANDS

DUMP (addr,u--)

Dump (print out) the contents of u (u is an unsigned integer) memory locations starting at addr. If you first type **HEX** the dump will be in hexadecimal.

CDUMP (addr,u--)

Dump the characters determined by the ASCII values of u bytes starting at addr. A clever use of this command is to print out the contents of any disk sector in the following way: **n BLOCK 128 CDUMP**
(This example, and much of this section on debugging, was taken from the manual EXTENED fig-FORTH by Patrick L. Mullarky, copyrighted in 1981 by the author. That manual, which comes with the EXTENDED fig-FORTH master disk, explains the debugger commands fully.)

B? (--)

Types out the current radix, or base, in decimal without changing the base. If you were to simply type **BASE @ .** the result would always be 10, since no matter what base you are in that number, when expressed in its own base, is 10. B? will print out the number 16 if you are in base 16, though, and still leave you in base 16.

DECOMP <string> (--)

Decompiles the colon definition whose name is <string>. The computer will print each address of the parameter field address of the command whose name is <string>, followed by the name of the word that is called on in that address. For example, if you had previously defined the word SQUARE as **: SQUARE DUP * ;** then typing **DECOMP SQUARE** would cause the computer to print

addr1 DUP
addr2 *

Note: DECOMP will print out what was compiled into the dictionary, not necessarily what was typed when defining a word.

WARNING: The computer can lock up if you try to DECOMPILE a word that was not defined entirely in High-level FORTH by a colon definition (a word that was partially defined in assembly language).

Note: The computer will print **PRIMITIVE** if you try to DECOMPILE some things, such as variables, constants, or assembly language definitions.

FREE (--)

Types the number of free bytes of space left for the dictionary. This number will vary depending on what graphics mode you are in, since different graphics modes use different amounts of memory.

H. (n--)

Prints the top of the stack in hexadecimal (base 16).

S. (--)

Prints the contents of the stack in the current base, without changing the contents of the stack. The numbers will be printed as unsigned single-length integers, so -1 would print as 65535, -2 as 65534, -32768 as 32768, but 32767 would be printed as 32767.

INDEX TO GLOSSARY OF FORTH COMMANDS

(in order of appearance in glossary)

STACK MANIPULATION COMMANDS

DUP	C-2
DROP	
SWAP	
OVER	
ROT	
-DUP	C-3
2DUP	
2SWAP	
2OVER	
FDUP	
FDROP	
FSWAP	
FOVER	
FROT	
>R	
R	
R>	

ARITHMETIC OPERATION COMMANDS

+	C-4
-	
*	
/	
MOD	
/MOD	
*/	
*/MOD	
MAX	
MIN	C-5
ABS	
MINUS	
D+	
DABS	
DMINUS	
F+	
F-	
F*	
F/	
FLOG	
FEXP	
FLOG10	
FEXP10	

NUMERIC CONVERSION COMMANDS

FLOAT	C-6
FIX	
FLOATING	
FP	
NUMBER	

COMPARISON
	<	C-7
	>
	=
	0<
	0=
	U<
	F0=
	F=
	F<

LOGICAL OPERATORS
	AND	C-8
	OR
	XOR

MEMORY ACCESS -- VARIABLES, ETC.
	CONSTANT	C-9
	VARIABLE
	!
	@
	+!	C-10
	C!
	C@
	ALLOT
	FCONSTANT	C-11
	FVARIABLE
	F@
	F!
	CMOVE	C-12
	FILL
	ERASE
	BLANKS

NUMBER BASES
	DECIMAL
	HEX
	OCTAL
	BASE

TERMINAL OUTPUT
. C-13
.R
?
D.
D.R
F.
F? C-14
CR
SPACE
SPACES
."
EMIT
TYPE
COUNT C-15
-TRAILING

TERMINAL INPUT
KEY
EXPECT
WORD C-16
QUERY

CONTROL STRUCTURES (LOOPS AND CONDITIONALS)
DO ... LOOP C-17
I
LEAVE
DO ... +LOOP
BEGIN ... UNTIL C-18
BEGIN ... WHILE ... REPEAT
IF ... THEN C-19
IF ... ENDIF
IF ... ELSE ... THEN C-20
IF ... ELSE ... ENDIF

SPECIAL SYSTEM WORDS
HERE
PAD C-21
?TERMINAL
SP@
ABORT

COMMENTS
(

DISK ACCESSING WORDS
LIST C-22
LOAD
BLOCK
FLUSH C-23
UPDATE
EMPTY-BUFFERS
DR1
DR0 C-24
R/W

DISK ACCESSING WORDS (continued from previous page)
-DISK C-24

DEFINING WORDS
: C-25
;
VARIABLE (see C-9)
CONSTANT (see C-9)
<BUILDS ... DOES>
CODE C-27
;CODE
IMMEDIATE
[COMPILE]
COMPILE C-28
, (comma)
' (single-quote)
[...]

VOCABULARIES AND VLISTING AND FORGETTING
FORTH C-29
EDITOR
ASSEMBLER
VLIST
FORGET C-30
DEFINITIONS
VOCABULARY

NUMERIC OUTPUT FORMATTING COMMANDS
<# C-31
#>
SIGN
#
#S
HOLD

CONSTANTS USED BY THE FORTH SYSTEM
BL C-32
C/L
0
1
2 C-33
3
B/SCR
B/BUF
FIRST
LIMIT

SYSTEM VARIABLES (USER VARIABLES)
SCR
OFFSET
BLK C-34
TIB
IN
WARNING
DP

SYSTEM VARIABLES (continued from previous page)
R# C-34
STATE
FENCE C-35
HLD
FLD
CSP
DPL
OUT
CURRENT
CONTEXT C-36
VOC-LINK

DEBUGGING COMMANDS
DUMP
CDUMP
B?
DECOMP C-37
FREE
H.
S.

NUMERIC INPUT FROM THE KEYBOARD

Because numeric input from the keyboard is commonly used in programming but is relatively difficult for a beginning FORTH programmer, included on the following page of this manual is a solution to the problem of keyboard numeric input. While this is not the only way to solve the problem, and not necessarily the best way, it is a valid solution. Merely copy the program listed on the two screens on the following page onto your disk, and all you will need to to do whenever you need to use it is LOAD screen 24 (or whatever screen you copy it onto on your disk). Then, to wait for a single-length number to be typed in from the keyboard and then put that number on the stack, execute INPUT#. Likewise, execute INPUTD# for double-length numbers and INPUTF for floating point (real) numbers.

A word of explanation is in order for how these commands work. First, you need to know that --> is the "next screen" command. It causes interpretation of a screen to cease and interpretation of the next screen to begin. In this case, it is used to stop interpreting screen 24 and start interpreting screen 25.

The way the D< command works is this:
The stack starts like this: (lo1 hi1 lo2 hi2)
Then, after ROT 2DUP: (lo1 lo2 hi2 hi1 hi2 hi1)
Then the command calculates ((hi2=hi1 AND lo1<lo2) OR hi2>hi1)
which is the same as d1<d2.
The rest of screen 24 is self-explanatory.

INPUTD# does a carriage return, waits for input from the keyboard followed by a carriage return, and converts the first number delimited by a blank to a double-length integer on top of the stack.
D-N-CONVERT simply DROPs the high-order part of the double-length integer.
INPUT# is like INPUTD# except the number entered must be between -32768 and +32767. The high order part of the number entered is dropped.
INPUTF uses ASCF, a word that is not described in the Glossary. ASCF converts a string starting at the address on top of the stack into a real number, which it leaves on top of the stack. INPUTF does a carriage return, waits for the user to type a number from the keyboard, and converts the first string delimited by a blank (space) to a real number, which it leaves on top of the stack. The number may be typed in E notation (as 8.32E-42 which is 8.32 times 10 to the -42nd power) if the user wishes, or it may be typed "regularly", but it must not have any spaces in it or everytning following the first space will be ignored.

;S is a word that, among other things, causes interpretation of the screen being interpreted to cease.

A sample program that uses numeric input is on the page following the next page.

```
SCR # 24
  0 ( NUMBER INPUT AND DOUBLE-LENGTH ROUTINES )
  1 : D<    ROT 2DUP     > >R    = >R    U<    R> AND    R> OR ;
  2 : D>     2SWAP D< ;
  3 : D=    ROT     = >R    =    R> AND ;
  4
  5 : D-    DMINUS D+ ;
  6 --)
  7
  8
  9
 10
 11
 12
 13
 14
 15

SCR # 25
  0 ( NUMBER INPUT ROUTINES CONTINUED )
  1 : INPUTD#    CR  QUERY BL WORD HERE NUMBER CR ;
  2 : D-N-CONVERT    DROP ;
  3 : INPUT#    BEGIN INPUTD# 2DUP 2DUP 32767 0 D> ROT ROT
  4 -32768 -1 D< OR WHILE CR ." MUST BE BETWEEN" CR
  5 ." -32768 AND 32767 " CR ." ENTER ANOTHER NUMBER " 2DROP
  6 REPEAT D-N-CONVERT ;
  7 : INPUTF    CR QUERY BL WORD HERE 1+ ASCF ;
  8 ;S
  9
 10
 11
 12
 13
 14
 15
```

```
SCR # 26
  0 ( QUADS )
  1 : FVAR    0 FLOAT FVARIABLE ;
  2 FVAR A    FVAR B    FVAR C    FVAR D    : -B    0 FLOAT B F@ F- ;
  3 : 2A    2 FLOAT A F@ F* ;    24 LOAD ( INPUT ROUTINES )
  4 : INPUT-EM ." A=" INPUTF A F! ." B=" INPUTF B F!
  5 ." C=" INPUTF C F! ;
  6 : DSET    B F@ B F@ F* 4 FLOAT A F@ F* C F@ F* F-    D F! ;
  7 : SQRT    FLOG 2 FLOAT F/ FEXP ;
  8 : NOT-QUAD     ." NOT A QUAD" CR ." X= " C F@ -B F/ F. ;
  9 : 1-SOLUTION    ." X= " -B 2A F/ F. ;
 10 : 2-SOLUTIONS    ." X= " -B D F@ SQRT F+ 2A F/ F. ." OR " CR
 11 ." X= " -B D F@ SQRT F- 2A F/ F. ;
 12 : IMAGINARY    ." NO REAL ROOTS." ;
 13 : QUADS    INPUT-EM A F@ F0= IF NOT-QUAD ELSE DSET D F@ F0=
 14 IF 1-SOLUTION ELSE D F@ 0 FLOAT F<
 15 IF IMAGINARY ELSE 2-SOLUTIONS THEN THEN THEN CR ;
```

VLIST of FORTH VOCABULARY

```
F(    F=    F0=    FCONSTANT    FVARIABLE    FP    FLOATING    FLITERAL
FLIT    ASCF    FEXP10    FEXP    FLOG10    FLOG    FIX    FLOAT    F/
F*    F-    F+    FS    F)    (F    F?    F.    F.TY    F!    F@    CIX    INBUF
FLPTR    FR1    FR0    FPOLY    FEX10    FEX    FLG10    FLG    FDIV    FMUL
FSUB    FADD    FPI    IFP    FASC    AFP    XL,    XLD    FOVER    XS,
XSAV    FSWAP    FDUP    FDROP    FILTER!    SOUND    AUDBASE    AUDCTL
FIL    DRAW    GCOM    FILDAT    ATACHR    G"    (G")    GTYPE    PLOT
POS    COLCRS    ROWCRS    CPUT    XGR    &GR    GR.    SNAME    IOCX    MASK
CIO    SE.    SETCOLOR    DISKCOPY    %COPY    MS1    WRTO    RDIN    GKEY
DSETUP    +BLK    WRT    RD    GET    ADRS    BLK#    BUFHEAD    CODE    ASSEMBLER
WHERE    EDITOR    MARK    LINE    TEXT    DECOMP    NXT1    .SETUP    ?DOCOL
T?PR    DOCOL    CKIT    .LIT    STG    BRNCH    NP    1WORD    1BYTE    PWORD
PDOTQ    PPLOOP    PLOOP    SEMIS    BRAN    ZBRAN    .CLIT    .WORD    S.
DEPTH    CDUMP    CDMP    DUMP    LDMP    U.R    ?EXIT    FREE    B?    H.
ASCII    BEEP    POFF    PON    CSAVE    SAVE    (SAVE)    (FMT)    BOOT
VLIST    TRIAD    INDEX    MATCH    LIST    U?    ok    2OVER    2SWAP
2DROP    2DUP    D!    D@    C?    U.    ?    .    .R    D.    D.R    #S    #
SIGN    #)    (#    SPACES    WHILE    ELSE    IF    REPEAT    AGAIN    END
UNTIL    +LOOP    LOOP    DO    THEN    ENDIF    BEGIN    BACK    FORGET
'    R/W    -DISK    EXPECT    --)    LOAD    MESSAGE    .LINE    (LINE)
BLOCK    BUFFER    DR1    DR0    EMPTY-BUFFERS    FLUSH    UPDATE    +BUF
PREV    USE    M/MOD    */    */MOD    MOD    /    /MOD    *    M/    M*
MAX    MIN    DABS    ABS    D+-    +-    S->D    COLD    ABORT    QUIT
(    PROMPT    DEFINITIONS    FORTH    VOCABULARY    IMMEDIATE    INTERPRET
?STACK    DLITERAL    LITERAL    [COMPILE]    CREATE    ID.    ERROR
(ABORT)    -FIND    NUMBER    (NUMBER)    WORD    GFLAG    PFLAG    SETUP
UP    XSAVE    N    W    IP    BINARY    POPTWO    POP    PUSH0A    PUT
PUSH    NEXT    PAD    HOLD    BLANKS    ERASE    FILL    QUERY    ."
(.")    -TRAILING    TYPE    COUNT    DOES)    (BUILDS    ;CODE    (;CODE)
DECIMAL    HEX    SMUDGE    ]    [    COMPILE    ?LOADING    ?CSP    ?PAIRS
?EXEC    ?COMP    ?ERROR    !CSP    PFA    NFA    CFA    LFA    LATEST
TRAVERSE    -DUP    SPACE    ROT    )    (    U(    =    -    C,    ,    ALLOT
HERE    2+    1+    HLD    R#    CSP    FLD    DPL    BASE    STATE    CURRENT
CONTEXT    OFFSET    SCR    OUT    IN    BLK    VOC-LINK    DP    FENCE
WARNING    WIDTH    TIB    +ORIGIN    B/SCR    B/BUF    LIMIT    FIRST
C/L    BL    3    2    1    0    USER    VARIABLE    CONSTANT    ;    :    C!
!    C@    @    TOGGLE    +!    DUP    SWAP    DROP    OVER    DMINUS    MINUS
D+    +    0(    0=    R    R)    )R    LEAVE    ;S    RP!    SP!    SP@    XOR
OR    AND    U/    U*    CMOVE    CR    ?TERMINAL    KEY    EMIT    ENCLOSE
(FIND)    DIGIT    I    (DO)    (+LOOP)    (LOOP)    0BRANCH    BRANCH
EXECUTE    CLIT    LIT
```

```
SCR # 28
  0  ( RANDOM DISK ACCESSING PROGRAM -- RDA )
  1 : NOT   0= ;
  2 : VARIABLE 0 VARIABLE ;   : CREATE   VARIABLE -2 ALLOT ;
  3 CREATE FLAG 0 , 1 ,   CREATE ID 1 , 2 ,   CREATE NAME 3 , 18 ,
  4 CREATE AREA 21 , 2 ,   CREATE PHONE 23 , 7 ,
  5 32 CONSTANT RECLEN   B/BUF RECLEN / CONSTANT REC/BLK
  6 VARIABLE REC# VARIABLE 'REC  VARIABLE NXT#
  7 85 B/SCR * CONSTANT FILES   B/SCR REC/BLK * CONSTANT RECS
  8
  9 : READ   REC# @ REC/BLK /MOD FILES + BLOCK SWAP
 10    RECLEN * + 'REC ! ;
 11 : OFF @ 'REC @ + ;
 12 : INCREC   1 REC# +! ;   : FLAG-IT   FLAG OFF C! ;
 13 : PAST-LAST  ID OFF @ 0= FLAG OFF C@ 0= AND ;
 14 : EXISTS   ID OFF @ = ;
 15 : DELETED  FLAG OFF C@ 0= PAST-LAST NOT AND ;     -->

SCR # 29
  0 ( RDA CONTINUED )
  1 : D<  ROT > IF 2DROP 1 ELSE < THEN ;
  2 : D>  2SWAP D< ;
  3 : INPUTD#   CR QUERY BL WORD HERE NUMBER CR ;
  4 : D-N-CONVERT   DROP ;
  5 : INPUT#   BEGIN INPUTD# 2DUP 2DUP 9999 0 D> ROT ROT
  6  1 0 D< OR WHILE CR ." MUST BE A POSITIVE NUMBER BETWEEN" CR
  7 ." 1 AND 9999" CR ." ENTER ANOTHER NUMBER " 2DROP
  8 REPEAT D-N-CONVERT ;
  9
 10 : ID#   ." ENTER ID# OF STUDENT" INPUT# ;
 11 : NA-ENTER   NAME OFF NAME 2 + @ BLANKS BEGIN CR ." ENTER NAME"
 12 ."  OF STUDENT" CR QUERY 0 WORD HERE C@ 18 > WHILE
 13 CR ." MUST BE LESS THAN 19 CHARACTERS" CR REPEAT HERE COUNT
 14 NAME OFF SWAP CMOVE ;
 15 -->

SCR # 30
  0 ( RDA CONTINUED ... )
  1 : CONT   CR ." PRESS RETURN TO CONTINUE " BEGIN KEY
  2 155 = UNTIL CR ;
  3 : ID-PUT   ID OFF ! ;
  4 : ID-FIND  ID# -1 REC# ! BEGIN INCREC READ DELETED IF 0 REC# @
  5 NXT# ! ELSE DUP EXISTS PAST-LAST OR THEN UNTIL ;
  6 : CO-ENTER  CR ." ENTER AREA CODE " CR BEGIN INPUT# DUP DUP
  7 100 < SWAP 999 > OR WHILE ." MUST BE A VALID AREA CODE" CR
  8 ." ENTER ANOTHER NUMBER" CR DROP REPEAT AREA OFF ! ;
  9 : NA.   NAME OFF 18 -TRAILING TYPE ;   : CO.   AREA OFF @ . ;
 10 : .ID   ID OFF @ . ;   : PH. PHONE OFF 3 TYPE 45 EMIT PHONE OFF
 11  3 + 4 TYPE ;
 12 : SHOWREC   CR ." ID#       " .ID CR ." NAME      " NA. CR
 13 ." AREA CODE " CO. CR ." PHONE #   " PH. CR ;
 14 -->
 15
```

```
SCR # 31
  0 ( CONTINUED RDA PROGRAM )
  1 : INFORM  CR ." MUST BE 7 CHARACTERS, OR 8 WITH A DASH" CR
  2 ." ENTER ANOTHER PHONE NUMBER" CR ;
  3 : EXTRACT   1 + DUP 7 + SWAP DO I C@ I 1 - C! LOOP ;
  4 : PH-ENTER   CR ." ENTER PHONE NUMBER" CR BEGIN BEGIN
  5 QUERY BL WORD HERE C@ DUP 7 ( SWAP 8 ) OR WHILE INFORM REPEAT
  6 HERE 9 + HERE 1 + DO BEGIN I C@ 45 = WHILE I EXTRACT HERE C@
  7 1 - HERE C! REPEAT LOOP HERE C@ 7 = NOT WHILE INFORM REPEAT
  8 HERE 1 + PHONE OFF 7 CMOVE ;
  9 : LIST-EM   CR ." Console or Printer (C/P) ? " BEGIN KEY DUP DUP
 10 67 = SWAP 80 = OR NOT IF DROP 0 ELSE 1 THEN UNTIL 80 = IF PON
 11 THEN -1 REC# ! 0 BEGIN INCREC READ PAST-LAST NOT WHILE DELETED
 12 NOT IF SHOWREC 1+ DUP 4 = IF PFLAG @ 0= IF CONT THEN
 13 DROP 0 THEN THEN REPEAT
 14 PFLAG @ 0= SWAP 0 ) AND IF CONT THEN POFF ;
 15 : PAGE   125 EMIT ;   -->

SCR # 32
  0 ( RDA CONTINUED MORE )
  1 : DELETE ID-FIND DROP PAST-LAST IF CR ." STUDENT NOT FOUND" CR
  2 ELSE 0 FLAG-IT UPDATE CR NA. ." IS NOW DELETED" CR THEN CONT ;
  3
  4 : ADD ID-FIND DUP EXISTS
  5 IF CR ." STUDENT ALREADY EXISTS" CR DROP ELSE NXT# @ 0= NOT
  6 IF NXT# @ REC# ! THEN READ ID-PUT NA-ENTER CO-ENTER PH-ENTER
  7 1 FLAG-IT UPDATE THEN CONT ;
  8 -->
  9
 10
 11
 12
 13
 14
 15

SCR # 33
  0 ( RDA CONTINUED STILL MORE )
  1 : MORE   CR ." MORE CHANGES? (Y/N) " BEGIN KEY DUP DUP
  2 89 = SWAP 78 = OR NOT IF DROP 0 ELSE 1 THEN UNTIL
  3 89 = IF 1 ELSE UPDATE 0 THEN ;
  4
  5 : MENU2   BEGIN CR CR ." CHANGE MENU" CR CR ." 1  NAME" CR
  6 ." 2  AREA CODE" CR ." 3  PHONE #" CR ." 4  ABORT CHANGES" CR
  7 CR ." ENTER YOUR CHOICE #" CR KEY DUP EMIT 48 - DUP DUP
  8 1 ( SWAP 4 ) OR IF DROP CR ." INVALID CHOICE" CR 0 ELSE
  9 1 THEN UNTIL ;
 10
 11 : CHANGE   ID-FIND DROP PAST-LAST IF CR ." STUDENT NOT FOUND"
 12 CR CONT ELSE BEGIN SHOWREC MENU2 DUP 1 = IF NA-ENTER ELSE DUP
 13 2 = IF CO-ENTER ELSE DUP 3 = IF ID PH-ENTER THEN THEN THEN
 14 4 = IF 1 ELSE MORE NOT THEN UNTIL THEN ;
 15 -->
```

```
SCR # 34
  0 ( THE END OF RDA.  TO EXECUTE, TYPE:  RDA )
  1
  2 : GOODBYE   CR CR FLUSH ." THANK YOU" CR CR ;
  3
  4 : MENU   BEGIN CR CR ." MAIN MENU" CR CR ." 1  ADD" CR
  5 ." 2  DELETE" CR ." 3  CHANGE" CR ." 4  LIST-EM" CR
  6 ." 5  QUIT" CR CR ." ENTER YOUR CHOICE # " KEY DUP EMIT CR
  7 48 - DUP DUP 1 < SWAP 5 > OR IF ." INVALID CHOICE"
  8 DROP 0 ELSE 1 THEN UNTIL ;
  9
 10
 11 : RDA   BEGIN PAGE MENU DUP 1 = IF ADD ELSE DUP 2 = IF
 12 DELETE ELSE DUP 3 = IF CHANGE ELSE DUP 4 = IF LIST-EM
 13 THEN THEN THEN THEN 5 = UNTIL GOODBYE ;
 14 ;S
 15
```