Panasonic

HHC T.M.

SnapFORTH

VOLUME I: TUTORIAL

THE HIGH-LEVEL PROGRAMMING LANGUAGE NATIVE TO THE HHC™

RL-S6003S8

SnapFORTH

the high-level programming
language native to the HHC™

VOLUME I:
TUTORIAL

TABLE OF CONTENTS

CHAPTER 4: THE EDITOR

CHAPTER 5: WORDS & THE DICTIONARY

CHAPTER 6: FLOW OF CONTROL

CHAPTER 7: MORE ABOUT NUMBERS

CHAPTER 8: STRINGS

CHAPTER 9: USER INTERFACE

CHAPTER 10: ADVANCED TOPICS

CHAPTER 1: GETTING STARTED

Hello! This manual will introduce you to the SnapFORTH language. It assumes you have not worked very much (if at all) with SnapFORTH before. SnapFORTH is a powerful and sophisticated language, and many of the details involved in programming in it are covered in the ***SnapFORTH Reference Manual***; only a fraction of its capabilities are covered in this introductory manual. For instance, you will be using programs that exist only as dictionary files and that you can access only from the SnapFORTH capsule. In addition, you will not be learning about assembly language programming or about how to make your programs into capsules or "RUN SNAP PROGRAMS". These topics are covered in the ***SnapFORTH Reference Manual***.

To use this manual, you must have a Hand-Held Computer (HHC) and a SnapFORTH capsule. No peripherals will be necessary.Insert the capsule into any spare socket in the back of the HHC. Be sure that the arrow on the capsule matches the arrow on the socket.

Note: While you need a SnapFORTH capsule to write a program, you can save the program in a form that will run on an HHC without a SnapFORTH capsule. For more information on that, see the ***SnapFORTH Reference Manual***.

Now let's get going. Turn on the HHC and press CLEAR twice. The primary menu will appear in the window, one line at a time:

```
1=CALCULATOR
2=CLOCK/CONTROLLER
3=FILE SYSTEM
. . .
n=SnapFORTH
```

These messages will repeat in sequence as long as it takes you to make a choice. Press the number n to select the SnapFORTH capsule. Remember, you can return to the primary menu at any time by pressing CLEAR twice. You will first see

```
1=New File
NO FILES
```

appearing alternately in the display. Since you have no choice, press the "1" key. You will be asked to:

```
Enter Filename
```

For now, answer:

```
myown
```

Don't forget to press ENTER. We will explain what "myown" is for a little later, in chapter 4.

You should now see:

```
ok
```

Congratulations! You are now up and running and ready to program in SnapFORTH.

Now let's review. Press CLEAR twice to return to the primary menu and select SnapFORTH. This time you will see:

```
1=New File
2=myown
```

Select 2 (myown). The word "myown" will appear briefly in the display. When you see the now familiar

```
ok
```

you are now ready to read chapter two.

CHAPTER 2: STACKS & NUMBERS

All operations in SnapFORTH are conducted by means of words. Some of SnapFORTH's words may not look like words to you— * / or + for example—but the computer sees them as such. As far as SnapFORTH is concerned, any character or group of characters separated from other characters or groups of characters by a space on each side is considered a word. For example,

```
hello $10,000 %%>&& bye
```

all could be valid SnapFORTH words. Therefore, you must get into the habit now of spacing between each word so that SnapFORTH will recognize your commands.

Numbers are also words in SnapFORTH. Numbers may not contain any punctuation and are normally no larger than 32767 and no smaller than -32768. (Larger numbers will be introduced in the chapters on Advanced Numbers and Floating Point.) Negative numbers are preceded by a minus sign. The following expressions—

```
1 0 -1 32000 -64
```

are all valid SnapFORTH numbers. Numbers are normally acted on by other SnapFORTH words and are saved in a special place called the "stack."

2.1 THE STACK

The stack is a "holding file" for numbers that you intend to work with by means of words. Any number you type into the display is entered onto the stack awaiting future use. The order in which these numbers are stored, however, is of crucial importance; you are responsible for understanding how the stack works and how to manipulate it to your advantage.

One of the most familiar ways of imagining the SnapFORTH stack is to think of cafeteria trays on a spring-loaded rack. If you label three trays 1, 2, and 3 and add them in that order to the top of the stack, when you take them off again their order will now be reversed: the first one you remove will be #3, the second one #2, and the third one #1. You cannot get at tray #1 until you have taken off tray #3 and then tray #2.

```
#1→_#1_ (top)        Push #1 on stack.

#2→_#2_ (top)        Push #2 on stack.
     #1  (2nd item)
```

```
#3→_#3_ (top)        Push #3 on stack.
    #2  (2nd item)
    #1  (3rd item)

  ←_#2_ (top)        Pop (remove) top item.
    #1  (2nd item)
```

2.1.1 The Print Command

The word . ("dot") will take whatever number is on top of the stack off and display it. This number, you should note, is not on the stack any longer; if you need to use it for another purpose, you will need another operation to save it—more on this later. If you type several dots in a row, one for each number, SnapFORTH will print all the numbers on the stack (in the order we have described). Now try this out. But first—do you see the "ok" prompt? Good. Begin by typing a series of numbers: 1 2 3 4 . Be sure to add enough dots to account for all the numbers:

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

Did you make any mistakes? Probably not, but if you ever do in these beginning exercises, remember that you can always back up with the left arrow key and write over your mistake with the correct input. As you back up, the information will remain in the window; nothing changes until you type over a character or press the ENTER key—at which point everything to the right of the cursor will be lost. Likewise, if you need to move over to the right of the line, you can accomplish this one character at a time by using the right arrow. Again, nothing is erased as you space across it.

To get to the beginning of a line in a hurry, you can do either of two things: Use the up arrow, which will move you there all at once, or hold down the left arrow key while it repeats. To get to the end of a line in a hurry, press the down arrow or hold down the right arrow.

Now press the ENTER key to let SnapFORTH recognize the commands, and it will answer you:

```
4 3 2 1 ok
```

If you type more dots than there are numbers in the stack, SnapFORTH will warn you:

```
ERROR: STACK UNDERFLOW .
```

Notice the dot at the end of the error message? SnapFORTH will always try to display the word that caused the error—in this case, the extra dot.

Note: You may encounter other error messages as you experiment with SnapFORTH. A full description of these and their meanings is in the SnapFORTH Reference Manual.

The concept of a stack must be mastered as you use SnapFORTH, as you will often be asked to remember how your information is stored there. Furthermore, you will be learning ways to manipulate the order of your data on the stack. A convenient way to remember the concept is the abbreviation LIFO—Last In First Out.

2.2 ARITHMETIC OPERATIONS

2.2.1 Addition

One of the unusual things about SnapFORTH is the order in which the elements of your arithmetic problem—in computer terminology, the "arguments"—are entered. Your normal inclination might be to write an addition problem like this:

```
2 + 3
```

and with many languages and calculators this would be quite correct. In SnapFORTH, however, the same operation would look like this:

```
2 3 +
```

Why? Mainly because it is simpler for SnapFORTH to handle. Also, as you will discover later, this system will eliminate the need for parentheses in more complicated algebraic operations.

But back to our simple addition problem. When you type

```
2 3 +
```

and ENTER, SnapFORTH stores 2 and 3 on the stack (in the LIFO order just described), adds them, and replaces them with the answer. What is now on the stack? Only the answer, 5. When you include the dot, the contents of the stack will now appear:

```
2 3 + .5 ok
```

Adding a long column of numbers is almost as easy, but you must remember to include sufficient + operators. SnapFORTH can only add two numbers at a time. It will take the top two numbers you put on the stack, add them, and replace them with the answer; then it will take the next number on the stack and add that, leaving just the answer, and so on. What you need to do is put a '+' after the first two numbers and after every other subsequent number:

```
9 17 + 31 + 47 + 998 + .1102 ok
```

Or the same problem can be written like this:

```
9 17 31 47 998 + + + + .1102 ok
```

Caution: As you remember, at this point you cannot work with any number greater than 32767 or less than -32768. If you try to input such a number or perform a computation that would result in such a number, you will get an answer—but it will be wrong.

2.2.2 1+ and 2+

For certain frequently performed operations, SnapFORTH saves you computer time and space by consolidating the elements of the operation. If you wanted to add 1 to 14, for example, you could use the command 1+ without having to type a 1 separately and a + sign as well. Try this both ways:

```
14 1 + .15 ok
14 1+ .15 ok
```

Surprise! 2+ works exactly like 1+, except with 2 instead of 1.

```
14 2 + .16 ok
14 2+ .16 ok
```

2.2.3 Subtraction

This operation follows the same basic logic as addition. You enter the two arguments (the numbers in your subtraction problem), the command for subtraction (-), and the dot. SnapFORTH destroys the original two numbers and replaces them with the answer.

If you were trying to subtract two from three, for example, the usual order "three minus two" would now become "three two minus:"

```
3 2 - .1 ok
```

Try a variety of subtraction problems to get comfortable with this operation.

2.2.4 1- and 2-

Subtracting 1 or 2 from a number is such a common operation that SnapFORTH provides separate words for them:

```
5 1- .4 ok
5 2- .3 ok
```

2.2.5 Multiplication

The only new thing to learn about multiplication is the command for it, which is an asterisk (*). (Using an X or a dot would be too confusing, as those symbols have other functions.) The reverse notation that you learned for addition and subtraction still applies. Knowing this, how would you multiply 15 by 5?

```
15 5 * .75 ok
```

2.2.6 Division and Remainders

Division is slightly more complicated because there are remainders to deal with. At this point, SnapFORTH can only deal with whole numbers; it cannot give you fractions or digits to the right of the decimal point to express remainders. But you can arrange to see the remainder as follows.

After you type two numbers, the command / ("divide") causes the first number to be divided by the second. If you tried

```
22 3 / ok
```

Your answer would be

```
.7 ok
```

If you wanted to know the remainder, you could type the same numbers plus a new operator "mod":

```
22 3 mod ok
```

and the answer would be the remainder:

```
.1 ok
```

A lot of trouble, right? But SnapFORTH has graciously created a new word that will do both at the same time: /mod ("divide-mod"). It is important, however, to remember to give SnapFORTH two print commands, one for the quotient (which will be on top of the stack and printed first) and the other for the remainder (which is next on the stack and printed second). Now try out this same problem with /mod:

```
22 3 /mod . .7 1 ok
```

2.2.7 2* and 2/

These operations are used so often that they have been especially written for extra speed. they multiply (2*) and divide (2/) by 2:

```
5 2* .10 ok
5 2/ .2 ok
```

Note: since 2/ works by shifting the number, it will not work for negative numbers.

2.2.8 Combination Operations

SnapFORTH can also handle several different kinds of arithmetic operations in quick succession.

For instance, suppose you wanted to add 15, 37, and 106, multiply the result by 5, divide by 7, and see the quotient as well as the remainder. You could reduce this to a very concise set of instructions, as follows:

```
15 37 + 106 + 5 * 7 /mod
   . . 112 6 ok
```

2.2.9 negate

The command "negate" will reverse the sign of whatever number is on the top of the stack. Try typing a -2 on the stack and changing it with a "negate":

```
-2 negate .2 ok
```

The same process will work with a +2 (written simply as 2):

```
2 negate .-2 ok
```

2.2.10 abs

"Abs" will give you the absolute value of the top number on the stack. Try the same two numbers with this new command:

```
-2 abs .2 ok
2 abs .2 ok
```

2.2.11 max

With "max," SnapFORTH will tell you which of the top two numbers on the stack is the greater by removing the smaller number. What SnapFORTH prints, then, is the larger number. Try a pair of numbers like 17 and 53:

```
17 53 max .53 ok
-17 -53 max .-17 ok
```

This will also work with combinations of signed and unsigned numbers:

```
78 -22 max .78 ok
```

2.2.12 min

As you might have guessed, "min" will give you the smaller of the top two numbers on the stack. The larger number is deleted from the stack, and SnapFORTH prints what is left—the smaller number. Try the same sets of numbers:

```
17 53 min .17 ok
78 -22 min .-22 ok
```

Why would anyone need a "max" or "min" command? Surely we can all tell just from looking whether one number is greater than another. But there are useful operations that can be accomplished by combinations of these two commands. For instance, suppose you would like to exclude numbers from your program that are less than -10 or greater than 100. If you put -8 on the stack and follow it with -10 and "max," SnapFORTH will leave -8 and delete -10 because -8 is the greater number.

```
-8 -10 max .-8 ok
```

When you try a number such as -23, your result

```
-23 -10 max .-10 ok
```

will exclude the -23, which is less than -10.

Half your problem is solved. The other half can be solved in symmetrical fashion by comparing numbers to 100 using "min." Test this on 56 and 897:

```
56 100 min .56 ok
897 100 min .100 ok
```

Lots of work, right? But SnapFORTH will allow you to perform both commands one after the other without retyping the original number. You type in the number in question and compare it to -10; in the same line you compare the result to 100, and SnapFORTH will print the result. Look closely at the following example:

```
43 -10 max 100 min .43 ok
```

Read this line the way SnapFORTH reads it. First, 43 is entered onto the stack, and -10 follows. The two numbers are compared by means of 'max,' and the greater, 43, is left on the stack while the lesser, -10, is deleted. The number 100 is now added to the stack, and it is compared to 43 by means of "min." The resulting smaller number, 43, is printed.

2.2.13 cr

The word " cr " ("carriage return") waits a reasonable amount of time to let you read a line and then moves you on to the next line in the display. You can control the speed of the display by pressing the STP/SPD key followed by a number from 0 to 9. The lower the number, the slower the display, except 0 (which gives the fastest display).

You can use " cr " to break up a long display into readable pieces:

```
cr 12345 . 22232 . cr 47 . 3 . 36 .
```

The result will look like this:

```
12345 22232
47 3 36 ok
```

2.3 STACK MANIPULATION COMMANDS

SnapFORTH is methodical, thorough, and entirely consistent. It will always enter your numbers in LIFO order. There may be times when you don't want them in that position, times when you want to get at something further down in the stack without going through all the numbers on top first. SnapFORTH provides for these occasions by offering a small but powerful set of stack manipulation commands.

2.3.1 dup

This is an especially useful command for cases when you want to use an argument several times or for several purposes. The word "dup" simply makes a copy of (duplicates) the first (top) number on the stack. If you typed in

```
5 7 9 ok
```

when you printed three dots, you would get

```
. . .9 7 5 ok
```

If you typed in

```
5 7 9 dup ok
```

when you printed four dots this time (to account for the duplication), you would get

```
. . . .9 9 7 5 ok
```

2.3.2 swap

This command, as its name implies, reverses the order of the top two numbers on the stack. If you typed in the same series of numbers as before plus a "swap" command, your

```
5 7 9 swap ok
```

would become

```
. . .7 9 5 ok
```

instead of the usual

```
. . .9 7 5 ok
```

2.3.3 over

The command "over" makes a copy of the second number from the top of the stack and puts the copy on top, pushing the former top number down one place. Your same series of numbers

```
5 7 9 ok
```

which usually yields

```
. . .9 7 5 ok
```

when printed with the addition of "over," thus:

```
5 7 9 over ok
```

will appear like this:

```
. . . .7 9 7 5 ok
```

(Remember to allow for the duplication of the second number and add another dot.)

2.3.4 rot

This command stands for "rotate"; it removes the third number on the stack and puts it on top of the stack, moving the other two numbers down to make room for it. In other words, the third item becomes the top item, the top becomes the second, and the second becomes the third. Your three numbers

```
5 7 9 ok
```

which usually come off the stack like this

```
. . .9 7 5 ok
```

will with the addition of "rot"

```
5 7 9 rot ok
```

look like this

```
. . .5 9 7 ok
```

Remember, this command only changes the order of numbers on the stack. None are copied or removed.

2.3.5 drop

The word "drop" does exactly what it says; it takes the top number off the stack and deletes it. Your familiar series

```
 5 7 9 ok
```

would usually yield

```
 . . .9 7 5 ok
```

when you printed it. With the use of "drop," the top number on the stack will disappear.

```
 5 7 9 drop . .7 5 ok
```

The word "swapdrop" combines the actions of a "swap" followed by a "drop":

```
 2 3 4 swapdrop . .4 2 ok
```

2.3.6 pick and roll

The words "pick" and "roll" are generalized stack manipulation words. To duplicate the nth-item of the stack and place it on top of the stack, use n pick:

```
 5 6 7 8 3 pick .6 ok
 4 pick .5 ok
```

The command 1 pick is the same as dup; 2 pick is the same as over; 0 pick (or less) is meaningless.

The command "roll" is the generalized form of "rot." Typing n roll moves the nth number (not counting itself) to the top of the stack, moving the remaining values into the vacated position:

```
 5 6 7 8 3 roll . . . .6 8 7 5 ok
 5 6 7 8 4 roll . . . .5 8 7 6 ok
```

The command 3 roll is the same as rot; 2 roll is the same as swap; 1 roll or 0 roll is meaningless.

2.4 SnapFORTH NOTATION

A standard way to indicate changes in the stack is to use arrows with the "before" status on the left and the "after" status on the right (before —> after). Numbers on the stack appear in order of entry, so the number furthest to the right will be the top number of the stack. If a blank appears before the arrow, that means that nothing is required for the word to execute; if a blank appears after the arrow, then the word leaves nothing

on the stack. The symbol "n" stands for number. Examples appear in the following tables.

For a complete listing of all words included in SnapFORTH, refer to the ***SnapFORTH Reference Manual***, where they are arranged in essentially alphabetical order.

TABLE 2.1 ARITHMETIC OPERATORS

WORD	STACK	ACTION
.	(n1 —>)	Prints number on top of stack.
+	(n1 n2 —> n-sum)	Adds two numbers.
1+	(n1 —> n1-plus1)	Adds 1 to number.
2+	(n1 —> n1-plus2)	Adds 2 to number.
-	(n1 n2 —> n-diff)	Subtracts n2 from n1.
1-	(n1 —> n1-minus1)	Subtracts 1 from number.
2-	(n1 —> n1-minus2)	Subtracts 2 from number.
*	(n1 n2 —> n-prod)	Multiplies n1 and n2.
/	(n1 n2 —> n-quot)	Divides n1 by n2.
mod	(n1 n2 —> n-rem)	Determines remainder of n1 divided by n2.
/mod	(n1 n2 —> n-rem n-quot)	Divides, leaving remainder and quotient.
2*	(n1 —> n1-times2)	Multiplies number by 2.
2/	(n1 —> n1-divby2)	Divides number by 2.
negate	(n —> n)	Changes the sign of n.
abs	(n —> n)	Returns the absolute value of n.

max	(n1 n2 —> n-max)	Returns the greater of n1 and n2.
min	(n1 n2 —> n-min)	Returns the lesser of n1 and n2.
cr	(—>)	Moves the display to the next line.

TABLE 2.2 STACK OPERATORS

WORD	STACK	ACTION
dup	(n1 —> n1 n1)	Duplicates the top number on the stack.
swap	(n1 n2 —> n2 n1)	Swaps the top two numbers on the stack.
over	(n1 n2 —> n1 n2 n1)	Pushes a copy of the second stack item onto the stack.
rot	(n1 n2 n3 —> n2 n3 n1)	Removes the third item in the stack and pushes it on top.
drop	(n —>)	Removes the top stack item.
pick	(n —> n-picked)	Duplicates the nth stack item.
roll	(n —>)	Moves the nth stack item to the top of the stack, moving the remaining items to the vacated position.
swapdrop	(n1 n2 —> n1)	Swaps the two top numbers on the stack and then drops the top number.

CHAPTER 3: CONSTANTS, VARIABLES, AND VECTORS

In your work with SnapFORTH, you will occasionally need to set up constants and variables that have specific values. As their names imply, constants have values that never (well, hardly ever) change, and variables have values that can be easily changed.

3.1 CONSTANTS

As you work with SnapFORTH, you will find it very handy to be able to set up constants. These essentially give names to numbers that you use often and make it easy to find, manipulate, and (if necessary) change those numbers. Setting up a constant is very simple. You specify a value, type the word "constant," and then give it a name. Here is an example:

```
 5 constant days/week ok
```

Whenever you want to use the value of a constant, you need only refer to its name, "days/week":

```
 days/week .5 ok
```

3.2 VARIABLES

Variables have values which can be changed. These values must therefore be stored in random-access memory (RAM). You can request a temporary storage area (TSA) in which to contain these variables which has a name of its own. For example, suppose you wish to create two variables for counting rainfall and snowfall, and to include them in a TSA called "working". You request the TSA with the words "area" and "endarea" thus:

```
area working ok  (Press ENTER after each line.)
 var rainfall ok
 var snowfall ok
endarea ok
```

Note: You can only have one TSA active at a time. Plan carefully! Furthermore, each TSA is limited to 255 bytes; each variable takes two bytes. If you exceed this, you will not get a warning. If you want to reserve more, see chapter 10, Advanced Topics.

If you now type "working," SnapFORTH will attempt to find exactly enough RAM for the two variables "rainfall" and "snowfall." If it succeeds, a 1 (true) will be left on the stack; if it fails, a 0 (false) will be left. Let's allocate the variables "rainfall" and "snowfall" by invoking the name of the TSA:

```
  working .1 ok
```

To initialize "rainfall" to the value 3, type in the following line exactly as it appears in the text. (Don't worry about what this line means just yet.) You need to set an initial value (initialize) a variable right at the beginning; until you do, it will contain a meaningless number.

```
  3 rainfall ! ok
```

The current value of the variable named "rainfall" is now 3.

3.2.1 Examining Variables

The difference between a constant and a variable shows up when you try to get SnapFORTH to print the value you have assigned. Try asking for the value of a variable in the same way you would for a constant:

```
  rainfall .4078 ok
```

What happened? The number you got from SnapFORTH (which might look like the example we provided or might be a similar large and improbable number) is not the value of the variable; rather, it is its address in the TSA in RAM. (an "address" is a location in memory.) Getting the value of the variable takes an additional step:

```
  rainfall @ .3 ok
```

The @ symbol ("fetch") goes to the address that RAINFALL has already put on the stack and puts the value it finds there on the stack, replacing the address. The dot, as usual, prints this number, removing it from the stack. These two operations are combined with a single symbol, the question mark (?).

```
  rainfall ?3 ok
```

Changing the value of a variable is quite simple. You enter the new value, type the name of the variable, and add a new command ! ("store").

Suppose you wanted to change the value of your variable "rainfall" from 3 to 6, for example. Your instructions to SnapFORTH would look like this:

```
  6 rainfall ! ok
```

Be sure you can read and understand this the same way that SnapFORTH does. Typing a 6 puts that number on the stack; typing "rainfall" puts the address of that variable on the stack as well; ! stores the new value at the address on top of the stack, destroying the old value in the process. Now when you ask for "rainfall" you will get the new value:

```
  rainfall ?6 ok
```

Note: Before you can use the ! ("store") command, a value and an address within a TSA must be on the stack; the address (where you will be storing the value) must be on top. "Store" needs this information to finish its task.

3.2.2 Changing Variables

Using the commands you already know from the previous chapter, it is easy to manipulate the value of a variable. For example, suppose you have a meteorological variable called "recordhigh" for temperature. Each day you would like to compare the value of "recordhigh" (the highest temperature recorded so far) with today's peak temperature (which you have already named "todaystemp"). If the value of "todaystemp" is greater than the value of "recordhigh," then the value of "recordhigh" should be changed to reflect this. Your problem has three parts: you must first create and initialize "recordhigh" and "todaystemp"; next, you must determine which of the values is greater; and finally, you must order the numbers on the stack so that this value will be stored in "recordhigh."

First, you need to create and allocate a TSA for the new variables. But wait: you already have one, and one is the limit. You have to get rid of "working" before proceeding. Do it this way:

```
  forget working ok
```

Now start with a new TSA (go ahead and use the same name again if you like; SnapFORTH has already forgotten the old TSA, so it won't be confused):

```
  area working ok
  var recordhigh ok
  var todaystemp ok
  endarea ok
  working.1 ok
```

Next, let's initialize "recordhigh" to 98 and "todaystemp" to 102.

```
  98 recordhigh ! ok
  102 todaystemp ! ok
```

Finally, let's check "recordhigh"

against today's temperature:

```
  recordhigh @ todaystemp @ max
    recordhigh ! ok
```

Make sure you understand how SnapFORTH reads your instructions at each point: The command "recordhigh @" puts the current value of "recordhigh" (98) on the stack. The command "todaystemp @" puts today's peak temperature (102) on the stack as well. The word "max" compares the two and eliminates the smaller value from the stack, leaving 102. Typing "recordhigh" puts the address of that variable on top of the stack. The "store" command (!) stores the value 102 at the address specified on the stack, destroying the earlier value (98) in the variable. What is the current value of "recordhigh?"

```
 recordhigh ?102 ok
```

Variables are particularly useful when keeping track of running totals. Suppose you are working with two variables, "totalrain" and "todaysrain." As of yesterday, "totalrain" is 15 and "todaysrain" is 2. You want to add "todaysrain" to "totalrain" to update your running total. Again, your problem is in three parts: you must allocate and initialize the variables, add the two values, and store the total correctly. First you must forget the old TSA, set up a new one, and allocate the new variables. Next, initialize "totalrain" to 15 and "todaysrain" to 2. We assume you know how to allocate and initialize variables. If you are still confused, please reread the material above.

Now study the following example:

```
 totalrain @ todaysrain @ +
   totalrain ! ok
```

Walk through this example to be sure you understand it. Typing "totalrain @" puts the value of this variable (15) on the stack. Next, "todaysrain @" puts the value of this variable (2) on the stack as well. The operator + adds the two values and puts the result on the stack, eliminating the two arguments. Typing "totalrain" pushes the address of this variable on the stack. The command ! stores the total of 15 plus 2 at the address of "totalrain." Check to see what the current value of "totalrain" is:

```
 totalrain ?17 ok
```

Since keeping running totals is such a common task, SnapFORTH has a special command just for this purpose, written +! ("plus-store"). To use "plus-store," push the value to be added onto the stack. Then push the address of the variable onto the stack. The command +! adds the value on the stack to the one at the address indicated and stores the result at the address. First, let's set the variable values back to the ones you started with:

```
 2 todaysrain ! 15 totalrain ! ok
```

The problem you just solved becomes shorter and simpler using +! :

```
 todaysrain @ totalrain +! ok
```

Check to be sure that the result is the same with this method:

```
 totalrain ?17 ok
```

When you want to change the value of a variable, the new value does not need to come from other already-established variables. To SnapFORTH, a value on the stack is a value on the stack, regardless of where it came from.

Therefore, if you wanted to add 4 to the value of your current variable "totalrain," you would follow exactly the same procedure as you used in the example above:

```
 4 totalrain +! ok
```

Four is the new value; typing "totalrain" pushes its address onto the stack; entering +! adds 4 to the old value and stores the total at "totalrain's" address. Check your result:

```
 totalrain ?21 ok
```

3.2.3 Byte Operators

Special byte operators are available to manipulate 8-bit numbers. They work just like conventional operators. The word "cvar" will establish an 8-bit variable in a TSA, "c@" will fetch an 8-bit number, and "c!" will store an 8-bit number. Unsigned byte values that you can manipulate with byte operators range from 0 to 255.

3.3 VECTORS

Now that you understand how constants and variables work separately, it's time to learn how to use them together. A logical grouping of identically-sized variables is called a vector. Each variable is differentiated from the rest by a number called an index. One of the best reasons for using vectors is saving space. It takes a relatively large amount of SnapFORTH space to name and store variables, and when they are clustered, more space can be freed for other tasks. As an example, suppose you are keeping track of your daily caloric intake and compiling the results weekly. If you set up seven different variables, one for each day of the week, a great amount of computer space would be taken up with the "bookkeeping" tasks of naming, locating, and storing so many separate pieces of information.

To continue our caloric example, let's create a vector called "cal/day," meaning calories per day. (Long variable names also take up unnecessary computer space; abbreviate when you can, but don't be too cryptic.) Vectors are created and allocated in much the same way that variables are (remember to forget the old TSA "working" first):

```
area working ok
7 vector cal/day ok
endarea ok
working .1 ok
```

The vector "cal/day" is created with enough RAM area to store seven 2-byte numbers. The values are numbered or "indexed." The first value is at index 0 and the last value is at index 6. (By the way, to avoid the common error of trying to use the value in the seventh position [which doesn't exist], you could allocate one extra value to the vector with "8 vector cal/day.")

You can visualize "cal/day" as a large box divided into seven smaller two-byte boxes, each of which stores a day's caloric total. The address of any box can be found by pushing the index of that box on the stack, followed by the name of the vector, "cal/day." Once you have the address, you can add a value to the box. For example, to initialize the first box (at index 0) to 2000, type:

```
2000 0 cal/day ! ok
```

To examine the first box, type:

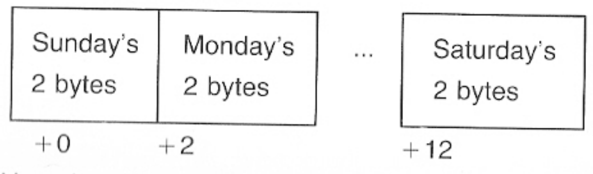
```
0 cal/day ?2000 ok
```

We can give each box a name by creating constants for each index. (This is not the most efficient use of computer space, but it is a good example of how to use constants for clarity.)

```
0 constant sunday ok ( Sunday's index )
1 constant monday ok ( Monday's index )
2 constant tuesday ok ( Tuesday's index )
3 constant wednesday ok ( Wednesday's index )
4 constant thursday ok ( Thursday's index )
5 constant friday ok ( Friday's index )
6 constant saturday ok ( Saturday's index )
```

Now the long box looks (diagrammatically) like this:

7 VECTOR CAL/DAY



How does this work? Study the following example. Suppose you consumed 1000 calories on Tuesday; your input would be:

```
1000 tuesday cal/day ! ok
```

Entering 1000 puts that number on the stack. Typing "tuesday" puts the index 2 on the stack. Typing "cal/day" replaces the index 2 with the address of the second box, and "store" (!) stores 1000 there ("inside" the vector "cal/day").

Now how do we get Tuesday's caloric record out? Watch:

```
tuesday cal/day ?1000 ok
```

Review this carefully. Typing the constant "tuesday" puts its value (the index 2) on the stack. Next, "cal/day" replaces the 2 with the address of the second box. The command ? goes to that address and fetches the value, putting it on the stack and then printing it.

For larger vectors, it is not practical to make each offset a constant. Instead, the index should be given directly. For example, suppose you were interested in daily caloric intake for a month. You would first need to forget the old TSA and start over. When you retype "working," include the vector:

```
31 vector cal/day/month ok
```

Remember, you can only have one TSA at a time, and it is limited to 255 bytes. This will be your last reminder!

To enter 1000 calories on the 23rd day, type the following:

```
1000 23 cal/day/month ! ok
23 cal/day/month ?1000 ok
```

3.3.1 Vectors of Bytes

One byte of computer memory can hold any number from zero to 255. If the values you put into an vector are never outside this range, you can effectively double the storage of an vector by storing one-byte numbers rather than two-byte numbers. To do this, you need to allocate a "cvector" instead of a "vector." Study the following example:

```
7 cvector meals/day ok ( Within a new TSA )
```

This sets up a vector of seven bytes. The bytes within the vector are indexed as usual (from 0 to 6). To store into this byte-vector, however, we use the command " c! " ("c-store"), which is the byte-equivalent of " ! " ("store.") (Historically, the "c" stands for "character." Commands which begin with "c" are often used to manipulate characters, such as "a" or "b," which occupy one byte of memory each.) For example, to specify four meals on the third day, type:

```
4 2 meals/day c! ok
```

(Remember, the vector actually begins with day zero, so day 3 has the index 2).

To fetch the value of the third day, use " c@ " ("c-fetch"), which is the byte-equivalent of " @ " ("fetch"):

```
 2 meals/day c@ .4 ok
```

Numbers on the stack are normally two-byte numbers. The word " c! " serves to pack these numbers into one-byte form, while " c@ " serves to unpack them into two-byte form again.

TABLE 3.1 CONSTANTS, VARIABLES and VECTORS

WORD	STACK	ACTION
constant	<name> (n—>)	Creates constant of name <name> and value n.
area	<name> (—>ok?)	Creates a temporary storage area (TSA) in RAM. Returns 1 if successful, 0 otherwise.
endarea	(—>)	Marks the end of the TSA. Variables and vectors between area and endarea will be allocated when the named TSA is executed.
var	<name> (—>)	Creates variable of name <name> within a TSA.
cvar	<name>	Creates 8-bit variable (ie, one byte or character) of name <name> within a TSA.
@	(address—>n)	Replaces address with the value n found at that address.
c@	(address—>n)	Replaces address with the byte value n found at that address. (The high-order byte is set to 0.)
?	(address—>)	Prints the value found at address.
!	(n address—>)	Stores n at address.
c!	(n address—>)	Stores the (low-order) byte of n at address. The byte at address + 1 is not disturbed.
vector	<name> (n-size—>)	Creates vector <name> within a TSA. n-size 2-byte elements will be allocated when the named TSA is executed.
	(n-index—>n-adr)	In use, the address of the n-index element will be left on the stack.
cvector	<name> (n-size—>)	Creates byte-array <name>. n-size bytes will be allocated within the TSA.
	(n-index—> n- adr)	In use, the address of the n-index byte will be left on the stack.

CHAPTER 4: THE EDITOR

Programming with SnapFORTH is never a one-shot operation. Inevitably you will want to add or change something later as you think about your work a second time or devise a new purpose for your program. To keep you from having to type the entire program again, SnapFORTH allows you to use the editor directly from the SnapFORTH capsule. This is the same editor described in the File System chapter of the instruction manual for your HHC. The editor lets you type commands and definitions into a more permanent storage.

Two kinds of memory are available to you for storage of your programs—intrinsic and extrinsic. Intrinsic memory, as its name implies, is built in. Extrinsic memory is contained in programmable memory peripherals which you can attach to your HHC. SnapFORTH allows you to save programs in memory in the form of files.

You can select the memory area that you want to save your program in. You are now operating in intrinsic memory, the default condition. To switch, press the I/O key, and you will see these lines in turn (the numbers you see may vary):

```
1=INT RAM,1060 FREE
2=EXT RAM,2038 FREE,SLOT=0
```

If you have no extrinsic memory, you will see only line 1. If you have extrinsic memory, select option #2 by pressing that number; you will see line #2 briefly in inverse characters (white on black). Return to intrinsic memory by pressing key number 1. Now press the I/O key (a toggle key) again, and SnapFORTH will return you to the program you were in.

4.1 CREATING A NEW FILE

Let's create a new file, one that you will call "utility." First, get back to the main menu by hitting CLEAR twice. Choose option 3, the file system. Within this option, choose option 1, NEW FILE. SnapFORTH will prompt you:

```
TYPE FILE NAME, THEN ENTER
```

Let's call your file "utility." After you ENTERed, the file was created; you can now input material. ask for the file system menu again, and it will now look like this:

```
1=NEW FILE
2=COPY FILE
3=utility
```

(You may have a few more files than this which may occupy other file numbers.) To get back into the file (open the file) for more input or to edit what you have already input, you need only to select the number of your file.

A simpler way to create or edit a file directly from SnapFORTH is to use the word "edit." Simply type the word "edit" followed by the name of the file. As in the earlier method, the file is created after you ENTER and is then ready for input. (If the file already existed, this command would open it for editing.)

File names can be no longer than 80 characters. Do not use blanks within the name of your programs.

Let's create a program using variables you have seen before in chapter 3, Constants and Variables. This program will create variables for rainfall and snowfall within the TSA called "working."

Enter this information line by line, making sure that you press ENTER after each line.

```
area working
   var rainfall
   var snowfall
endarea
working drop
4 rainfall ! 6 snowfall !
```

After you have ENTERed the last line, exit the file by typing a CLEAR. Now examine "rainfall" and see what you get.

But SnapFORTH doesn't seem to recognize your new word:

```
rainfall ?
ERROR: CAN'T FIND rainfall
```

What happened? As it turns out, even though you typed ENTER for each line, your program has not really been processed by SnapFORTH; instead, it is now contained in what is called a source file. When you want the file contents to be processed, you need to type the command "load" followed by the name of the file:

```
load utility
```

The file will now be processed by SnapFORTH. Now if you examine "rainfall," you will get the result you were expecting:

```
rainfall ?4 ok
```

Now go back to the main menu. (remember, two CLEARs) and ask for the file system with option 3. What do you get?

```
1=NEW FILE
2=COPY FILE
3=mydown
4=utility
```

This is yet another proof that your file has been properly entered. Select "utility." The word "utility" will appear in inverse characters to show that it is an "old" file.

4.2 EDITING COMMANDS

To see your file's contents, you now need to know how to move around within the file. First, let's establish basic cursor movement commands.

Going Down

To move the cursor down one line, type a down arrow. Repeat the command to move more than one line. The HHC will beep at you and stop if you try to go below the last line.

Going Up

To move the cursor up a line, type an up arrow. You can repeat this command also, but the HHC will beep at you and stop if you try to go above the first line.

Going to the Right

To move the cursor to the right one character at a time, type a right arrow. As with the up arrow, the HHC will beep and stop when you hit the end of the line.

Going to the Left

To move the cursor to the left one character at a time, type the left arrow. As with the other control commands, the HHC will beep and stop when you hit the end of the line.

Before you go any further, try out these commands on your utility file.

One important point: When you ENTER a normal line, only those characters to the left of the cursor are interpreted. When you ENTER a line from within the editor, however, it doesn't matter where the cursor is at that moment.

Now that you know how to get to the places you want to correct in your file, here's how to do the corrections.

Inserting Characters

Suppose you had typed the word

```
workng
```

and needed to fix it. First move the cursor so it is on top of the "n." Type the insert key. (Notice the "checkerboard" cursor.) Now type the missing "i" SnapFORTH will automatically move the "ng" over to make room for the new letter.

If you wanted to insert several characters at a time—say a whole word—press the lock key before inserting. Now you won't have to hold down the insert key for each new character inserted. To get out of the locked insert mode, press the insert key again. Remember that SnapFORTH can handle lines of no more than 80 characters. If you insist on filling a line with more than this, the HHC will beep at you and stop accepting the characters.

Deleting Characters

This time, suppose you had typed the word

```
woorking
```

and didn't think that acceptable either. Move the cursor so it is above the unwanted character. Press the delete key (notice the "hollow" cursor) and then a left arrow. The word will close up from the right, erasing the extra "o" in the process. The cursor will now be over the character that was to the left of the one you deleted. Again, if you want to delete several characters in a row, press the lock key first. Get out of the delete mode by pressing the delete key again.

Inserting Lines

What if you wanted to add a new variable in "working" that would occupy a whole new line? Let's try it with a variable called "nightfall" that will appear after "snowfall" in the TSA "working." Position the cursor on the line below the one where "nightfall" will appear—the line that says "endarea." Type an insert key and then an up arrow. A new empty line above "endarea" will now open up. Type the new line

```
var nightfall
```

and ENTER. The same effect could be accomplished by positioning the cursor on "snowfall" (the line above), typing an insert, and pressing the down arrow. If you wanted to insert several lines at a time, press the lock key first. In that case, press an insert again to get out of the insert mode.

Deleting Lines

Suppose you've decided that "nightfall" is a rather silly variable and you'd just as soon be rid of it. Position the cursor on the line "var nightfall" and press the delete key, followed by a down or up arrow. The next or previous line will now move up to close up the space.

Note: Even if you deleted all the lines in a file this way, you would still have the name of the file in the menu. To delete an entire file, see the section on deleting files.

4.3 COPYING FILES

When you are satisfied (for the moment) with the file you have created, you may want to make a backup copy or save it for future reference. The memory area within the HHC (intrinsic RAM) may be too small for such permanent storage; in this case, an external storage device such as a programmable

memory peripheral (extrinsic RAM) provides mass storage in much the same way that a floppy disk does on a desk-top computer.

Return to the file system and select option "2 = COPY FILE." After the prompt "SELECT FILE," you will be shown the names of all the files in the active memory space (intrinsic or extrinsic, whichever was selected last):

```
SELECT FILE
3=mydown
4=utility
```

Let's make a backup copy of "utility" (press "4") in extrinsic memory:

```
utility
SELECT DESTINATION RAM
1=INT RAM,2099 FREE
2=EXT RAM,6507 FREE,SLOT=0
```

Select extrinsic memory (press "2"):

```
COPY DONE
```

You may now press CLEAR, OFF, and disconnect the programmable memory peripheral for safekeeping. If you wish to reedit "utility," reconnect the memory peripheral, select extrinsic memory (with the I/O key), and use the COPY command with intrinsic memory as the destination.

Note: It is a good idea to have a copy of your files in extrinsic memory. You can then make a working copy in intrinsic memory and, when you are happy with the results, recopy the file to extrinsic memory.

4.4 RENAMING AND DELETING A FILE

When you call for a file and see its name in inverse characters, you can edit that name just as you would any line in SnapFORTH, except that you cannot have any spaces in the new name. Before attempting to delete a file, make sure it is in active memory by selecting its area. Let's get rid of "utility" in intrinsic memory for now. First, select "utility" for editing. As before, its name will appear in inverse characters, signifying that the file is indeed in memory. Now press the delete key. and immediately press the down arrow. Return to the file system and you will see that "utility" is no longer there:

```
1=NEW FILE
2=COPY FILE
3=mydown
```

CHAPTER 5: WORDS & THE DICTIONARY

As you have already seen, words are essential to the workings of SnapFORTH. All commands in SnapFORTH are accomplished with words, each of which has a very specific meaning and must be used in the right place in exactly the right order. Where do these words come from? Basically, from you. A certain number of words are provided for you in the HHC itself. More are provided in the SnapFORTH capsule, and the rest are written by the user. SnapFORTH is extensible, in other words; you are given the basic tools with which to create new words or delete old ones, and then you are left with the freedom to define only those that suit your own purposes.

As with any language, spoken or computed, words can be combined to form more complex sentences. It is to your advantage to plan ahead when you define words so as to include words you have already defined. This is called "modular programming," and besides being logically and aesthetically pleasing, it saves computer space. Be alert for examples of this as you learn about defining words.

5.1 PRINTING MESSAGES

Before learning to define new words, you need to learn a different type of print command. The dot you were using to print numbers from the stack is one type of print command; to print messages requires a different type. Perhaps you will define a word called "greetings" which will respond with the typed message:

```
hello! i am a message!
```

To make SnapFORTH print this message, it must be surrounded by special print commands:

```
." hello! i am a message!"
```

Notice that the message is preceded by the SnapFORTH command ." ("dot-quote"). Furthermore, a space separates the ." from the "hello". This space is necessary for SnapFORTH to recognize the beginning of the printed message, but it does not move the message over one space. Notice also that this space is not required before the end quotation mark.

5.2 DEFINING WORDS

The simplest kind of word neither requires arguments from the stack nor leaves any results on the stack. The word "greetings," described above, is an example of such a word. Let's see how we might define it.

Definitions always start with a colon (:). The colon's meaning is that a definition follows immediately; like any other word, it is followed by at least one space. The next element must be the name of your new word. So far you have entered the following:

```
: greetings
```

Since your message starts on a new line, you must first tell SnapFORTH to wait and start a new blank display line. You already know the word that performs this function,"cr." Next, you want to tell SnapFORTH what the printed message will be and surround this by the print command you have just learned. Your definition now looks like this:

```
: greetings cr ." hello! i am
  a message!"
```

The last instruction to be added is a semicolon (;), which lets SnapFORTH know that the definition is over. Again, the semicolon is a word and must have a space before it. Your final definition looks like this:

```
: greetings cr ." hello! i am
  a message!" ;
```

When you press the ENTER key, SnapFORTH will compile the new definition, responding with its usual "ok." (By "compile," we mean that SnapFORTH has now stored the word in its dictionary and will execute the word when you ask for it again.) Now when you ask for "greetings," SnapFORTH will type the message as you intended.

5.3 VLIST

All the words in SnapFORTH, including the one you just compiled, exist in a long list called, naturally enough, a dictionary. You will often want to check this dictionary to see whether a particular word has been defined yet. To do this, simply type "vlist". The name of the most recently defined word is displayed; press any key to display the next word. Press LOCK, right arrow to scroll the words in your dictionary on the display. Your vlist may look like this:

```
vlist
GREETINGS SNOWFALL
RAINFALL WORKING FORTH
ASSEMBLE^ FORWARD DROP
...
```

Notice that the word most recently defined ("greetings") appears first—at the "top" of the dictionary. In many ways, the dictionary works like a special kind of stack. New entries are always stored at the top of the dictionary and are removed in that order.

5.4 FORGET

Perhaps you don't like a word you have just defined. You might have a better one now, or you might have found a lot of errors and just want to start over. Maybe you were just experimenting anyway and never intended to keep the word permanently. The word "forget" is for you; your word will be erased from the dictionary with this command. Why don't we get rid of "greetings":

```
forget greetings ok
```

Check to see that greetings no longer appears in the dictionary:

```
vlist
SNOWFALL RAINFALL
WORKING FORTH ASSEMBLE
FORWARD DROP @ ! C@ C!
```

Special note: When you pick a word and ask SnapFORTH to forget it, it will erase that word and every word defined after it as well. In other words, if you wish to add a variable to the TSA "working" in the file "utility" (see Chapters 3 and 4):

```
forget working ok
load utility ok
```

The command "forget working" erases "snowfall" and "rainfall" as well. Words that turn out to produce errors can also haunt you later; be sure to forget all of them.

5.5 MORE DEFINITIONS

This section introduces the SnapFORTH command "random," which will be used within other definitions. Carefully type the following lines into the file "utility." Don't worry about what they mean just yet.

```
area working
  var seed
endarea
working drop
: rnd seed @ 13 u* drop
  flip 7 + dup seed ! ;
: random abs rnd u* swapdrop ;
```

If you have extrinsic memory, use the copy command from the file system to save a backup copy of "utility." We will shortly be giving you other useful words to add to "utility."

Next, load the utility file into the dictionary. Don't forget to "forget" the previous TSA "working" if it is already in the dictionary:

```
forget working ok
load utility ok
```

If you make a mistake, "forget working," reedit "utility," and "load utility" again.

Now test "random" by typing:

```
10 random .
```

After you press ENTER, SnapFORTH will print a number between 0 and 9. (Specifically, n random will generate a random number between 0 and n-1, replacing n with the number.) Try this several times. The number printed will vary in an apparently random manner.

Sometimes a word does not need arguments on the stack but does leave results on the stack. For example, such a word might use the command "random," which generates random numbers. The word "random" can be used to simulate chance events, such as you might find in gambling. You can define the word "dice," for example, which will show you the numbers on top of two dice each time you throw them. Think about your problem for a minute. You must generate two random numbers between 1 and 6; it will not do to generate 0, as might happen with "random," so you must think of a way to eliminate this possibility. Here is one way:

```
: dice 6 random 1+ . 6 random 1+ . ;
```

: dice

6	puts 6 on the stack.
random	generates a number between 0 and 5, replacing the 6 with this number.
1+	adds 1 to the number you just generated; instead of a number from 0 to 5, you now have a number from 1 to 6.
.	prints the resulting number.
6	puts another 6 on the stack.
random	generates yet another random number between 0 and 5.
1+	adds 1 to this number.
.	prints this number.
;	ends the definition.

Now type in "dice" directly from the keyboard and try it out:

```
dice 1 4 OK
dice 3 2 OK
dice 1 6 OK
```

The word "dice" can be embellished by adding a message with the print command you learned earlier (."):

```
: dice
    cr ." the throw of the dice is "
    6 random 1+ . 6 random 1+ . ;
```

When you compile dice a second time, SnapFORTH will warn you that you have used a word twice with the message:

```
REDEFINING DICE
```

You can still proceed with your second definition; both will appear in the vlist, but only the most recent definition (the second one) will actually be used when you call for it. Try your new word dice to see what you get:

```
dice
the throw of the dice is 4 1 ok
dice
the throw of the dice is 1 3 ok
dice
the throw of the dice is 2 5 ok
```

If you now forget dice, only the most recent definition (and any words defined after it) will be forgotten:

```
forget dice ok
dice 1 4 ok
```

Another kind of definition both requires arguments on the stack and leaves results on the stack. The arithmetic operators you learned in chapter 2 (+, -, *) are examples of this type. A more complex example might be a word like "dozens," which will take a number you provide and tell you how many dozens are in it with what remainder. Like the preceding definition, this one will incorporate a message. After you type a number and "dozens," you want SnapFORTH to start a new line, print the number of dozens followed by the message "dozen plus" and then type the remainder. You already know all the words necessary to create this definition:

```
: dozens 12 /mod cr
    . ." dozen plus " . ;
```

Be sure that you can identify the purpose of all the elements in this definition:

: dozens	
12	puts 12 on the stack.
/mod	divides the number you specify by 12 and puts the quotient on top of the stack with the remainder second, replacing the original number.
cr	starts a new line.
.	prints the top number on the stack, which is the quotient.
." dozen plus"	prints the message "dozen plus."
.	prints the second number on the stack, which is the remainder.
;	ends the definition.

Now try out your new problem-solver:

```
367 dozens
30 dozen plus 7 OK
```

5.6 MULTIPLE DICTIONARIES

All of the words you have been creating have gone into the dictionary "myown," which you created in Chapter 1. A dictionary is a file too, consisting of all the words you have defined in it. When you reenter SnapFORTH from the main menu, it will display the names of all the dictionaries currently stored (in this case, we only have "myown") and give you the option of choosing one. When you choose a dictionary, all the words in it will become available to you for use. You can create separate dictionaries with option "1 = NEW FILE" at the start of a SnapFORTH session.

To "empty" a dictionary of all words, select the dictionary from the SnapFORTH menu and use vlist to determine which word immediately precedes the word "FORTH." This is the first word in the selected dictionary. Forgetting this word forgets all the words in the dictionary.

TABLE 5.1 WORDS AND THE DICTIONARY

WORD	STACK	ACTION
:	: <name> ... (—>)	Creates dictionary entry for the word <name>. Begins compilation.
;	<name> ... ; (—>)	Terminates dictionary entry for the word <name>. Stops compilation.
."	."cccccc" (—>)	Prints text cccccc (up to "). If ." is included in the definition of a word, printing will take place when the word is later executed.
forget	: <name> ... (—>)	Deletes the word <name> from the dictionary. Also deletes all words added after <name>.
vlist	(—>)	Lists the words currently in your dictionary, starting with the one most recently defined.
(This command must be added to SnapFORTH. See the text.)		
random	(n-limit—>n)	Returns a random number ranging from 0 to n-limit minus 1.

CHAPTER 6: FLOW OF CONTROL

Now that you have learned how to create simple definitions, you can a) build into them ways to make decisions based on criteria you supply and b) repeat operations within limits you specify. These are called flow of control operations; they can only be used within definitions.

6.1 DEALING WITH DECISIONS

Many games, for example, use structured operations. Games involving dice often have elaborate branched structures which depend on the throw of the dice at each branch. Think of the decisions involved in playing craps. Your first throw of the dice could be a 2 or a 12 (snake eyes or boxcars), in which case you would lose. Or it could be a 7 or an 11 (craps), in which case you would win. If neither, the sum of the throw would be recorded and you would continue to throw. If the throw equaled your previous sum, you would win; if it equaled 7, you would lose. If neither, you would keep rolling until you won or lost. Programming SnapFORTH to play craps would first involve checking the sum of the dice for 2 or 12, 7 or 11, and deciding what to do based on the result. SnapFORTH would save the result, if neither condition were filled, and continue rolling the dice until certain other conditions were filled, checking each time before rerolling. Each time it checked it would get a "yes" or "no" answer on which it would base its next action.

In executing the decisions and limits you establish in your structure, how does SnapFORTH deal with yes and no answers? By means of "logical operators" or "truth functions." Before you proceed with flow of control problems, you must understand how these operators work and how they differ from each other.

6.1.1 Logical Operators

All the decisions that SnapFORTH makes are expressed as true or false, yes or no, zero or non-zero. Logical operators allow you to "test" numbers for various properties—same, different, above zero, below zero—and always return an answer of true or false.

Before we begin, let's define a word which will print "true" if the value on top of the stack is true and "false" if the value is false, removing the value from the stack in the process. Type the following (don't worry what it means just yet; it will be explained below):

```
 : truth if ." true " else ."
    false " then ;
```

Now try out some values with this word.

```
0 truth false ok
1 truth true ok
50 truth true ok
-3 truth true ok
```

Any non-zero number is considered true; zero is the only number that will give you a false response.

The operator "and" can be used to test whether the two top numbers on the stack are both true.

Note: The word "and" is a "bitwise" word, so it is not enough for the numbers to be non-zero, as in truth; they must both be ones. Since most logical operators leave a zero (false) or a one (true) on the stack, this is seldom a problem:

```
1 0 and truth false ok
0 1 and truth false ok
0 0 and truth false ok
1 1 and truth true ok
```

(but)

```
2 4 and truth false ok
```

The word "or" will give a true response if either of the arguments is non-zero. Try 1 and 0 again:

```
1 0 or truth true ok
```

2 and 3 would yield the same answer:

```
2 3 or truth true ok
```

because at least one argument is non-zero. The only way to get a false response would be to try two zeros.

```
0 0 or truth false ok
```

Incidentally,"or" is a "bitwise" "or."

The commands < ("less than") and > ("greater than") will test two arguments for their value relative to each other. The first argument entered is compared to the second. Try 2 and 4:

```
2 4 < truth true ok
```

SnapFORTH is telling you that 2 is indeed less than four. Try the greater than symbol:

```
2 4 > truth false ok
```

Two is not greater than 4.

The word = ("equals") will test for the equality of two arguments. Test this with 1 and 3:

```
1 3 = truth false ok
```

and 3 and 3:

```
3 3 = truth true ok
```

The word " <> " ("unequal") will test for inequality, leaving a 1 on the stack if they are unequal. Try this out with some pairs of numbers:

```
3 6 <> truth true ok
4 4 <> truth false ok
```

The word 0= is tricky at first because it reverses the answers you have come to be familiar with. This word will tell you whether an argument is equal to zero; if it is, your answer will be true. You will want to be able to do this in future flow of control operations when it is necessary to check for the presence of a zero on the stack. See how this works:

```
4 0= truth false ok
-15 0= truth false ok
0 0= truth true ok
```

You can also use "not," which is a synonym for 0= :

```
4 not truth false ok
0 not truth true ok
```

The operator 0< will tell you if an argument is less than zero—a negative number. Negative numbers, then, give a true response; numbers 0 and above give a false answer.

```
5 0< truth false ok
-5 0< truth true ok
```

Finally, SnapFORTH includes the command 0> , which tests to see whether its argument is one or greater:

```
-1 0> truth false ok
0 0> truth false ok
10 0> truth true ok
```

6.1.2 Conditional Structures

Within conditional structures, SnapFORTH will make decisions according to criteria you specify and perform operations based on those decisions. SnapFORTH first looks at the argument (condition) on top of the stack and examines it for its truth value; if true, it does one thing, if false, it does another, and then it moves on. The operations SnapFORTH performs could be anything; if an argument is true, you could command SnapFORTH to print the word "true" (or the word "panamahat") or store the value at a certain address or multiply it by 6...anything you want.

The form that this structure must take is strictly ordered, however:

```
: definition condition "if" one
   thing "else" the other thing
   "then" move on ;
```

Be sure you know what is happening at each stage of this structure:

: definition

condition	you enter the starting information at this point plus any manipulations that will lead to a zero (false) or non-zero (true) result on top of the stack.
if	SnapFORTH tests the condition for truth value. If true, it goes straight on to "one thing." If false, it checks to see if there is an "else" (there may not be) and executes it if there is. If no "else" is present, a false value will make the structure skip to "then."
one thing	SnapFORTH executes the command.
else	if the condition was false, SnapFORTH skips to this point.
other thing	SnapFORTH executes this second command rather than the first.
then	after SnapFORTH has considered the "if/else" choice, it stops the process.
move on	SnapFORTH moves on to other commands (if any).

Notice that the "else" section is optional; if the condition turns out to be false, you may want SnapFORTH to ignore it entirely, in which case SnapFORTH would simply move to "then" and end the structure.

The word "truth" that you have been using up to now is a good example of a conditional structure. Look at the definition again:

```
: truth if ." true " else
." false " then ;
```

Examine this definition one step at a time:

: truth

(condition)	the condition, in this case, is the argument to truth, which is assumed to be on the stack.
if	SnapFORTH tests the condition for truth value. If it is true, it will execute whatever appears after "if"; if it is false, it will proceed to "else" or "then". The condition is removed from the stack.
." true"	if the condition was true (non-zero), SnapFORTH prints "true".
else	if the condition was false (zero), SnapFORTH comes to this section...
." false"	...and prints "false".
then ;	the structure is finished.

A more complicated type of "if...else...then" structure involves two or more mutually exclusive choices. It is quite possible to order each choice independently, that is, to have SnapFORTH look for one condition, execute a command you specify, then look for another condition with a new operation, and so on. However, with mutually exclusive choices, the first "if...else...then" structure limits the need for further checks. For example, if you are checking to see if a value is 5, 3, or 7 and the value happens to be 5, there is no need to ask whether it is 3 or 7. For maximum efficiency, you would want SnapFORTH to skip the next two questions. If you use three independent "if...else...then" structures, this economy is not possible. Therefore, you should learn to use a "staircase" or "case" structure for these situations. If you diagrammed the process SnapFORTH uses to implement such a structure, it would look something like this:

```
if   ( yes? then execute following command and go to
     outermost 'then' )
else  ( no? then try second question )
    if   ( yes to second question? then execute second
         command and move to innermost 'then' )
    else  ( no? then execute third command and move
          to innermost 'then' )
    then  (stop innermost structure)
then  (stop outermost structure)
```

An interesting example of such a structure might be the word ".dice." This word, like the word "dice" that you defined in chapter 5, throws dice but also checks them for certain combinations—snake eyes (1 and 1) and boxcars (6 and 6). If the answer to either of these checks is 1 (true), it will tell you by printing "snake eyes" or "boxcars." If neither is true, SnapFORTH will do nothing except print the throws themselves. The throws of the dice will be left on the stack for possible future use.

As you think about how to order your definition of .dice , remember that many of the operations (arithmetic operators, logical operators, "if") that you will perform on the numbers you generate will destroy these numbers. You must think ahead of time about duplicating these numbers so that they will not be lost. Also, it is a good idea to form the habit of "commenting" on your definition. These comments will include notes on what is added to and left on the stack and summaries of what each section of the definition does. Comments should begin with the word (("paren"). Because (is a SnapFORTH word, it must be followed by at least one space.

In addition, you can use a simple backslash (\) to simplify the addition of comments. Anything to the right of a backslash is considered to be a comment. If the comment occupies more than one line, a backslash must start each new line.

Now try formulating a definition based on a new version of "dice." It might look like this:

```
  : dice
    6 random 1+ 6 random 1+ ;

 : .dice ( --> die-1 die-2 )
      dice 2dup cr . .
      2dup + ( form sum )
      dup 2 =
    if ( snake eyes? )
      ." snake eyes " drop
    else 12 =
      if ( boxcars? )
        ." boxcars "
      then
    then ;
```

Check to be sure you know the purpose of each word in the definition:

: .dice

dice	generates two random numbers between 1 and 6 and places them on the stack.
2dup	copies the two numbers; now you have two sets of two random numbers, each in the original order.
cr	starts a new display line.
. .	prints the first set of numbers, destroying them.
2dup	creates another duplicate pair of numbers.
+	adds the top two numbers, destroying them and leaving the sum on the stack.
dup	makes a copy of the top number (the sum).
2 =	checks to see if the sum is equal to 2; if yes, then a true condition is placed on the stack, destroying the sum and the number 2.
if	removes the condition from the stack, checking to see whether it is true. If yes, it executes the next commands (skipping "else," if it appears) and moves to the outermost "then." If no, it proceeds to "else."
."snake eyes"	prints this message.
drop	drops the top sum from the stack, leaving only the original throw. SnapFORTH now proceeds directly to the outermost "then."
else 12 =	checks the sum to see if it is equal to 12, destroying the sum and the 12 in the process.
if	If yes, the next command is executed and SnapFORTH moves to the innermost "then"; if no, SnapFORTH moves directly to the innermost "then."
."boxcars"	prints this message.
then then	stops the innermost and outermost structures.
	stops the definition.

Now try out your new word and add two dots each time so as to clear the stack:

```
.dice . .
1 1 snake eyes 1 1 ok
.dice . .
2 4 2 4 ok
.dice . .
3 5 3 5 ok
.dice . .
6 6 boxcars 6 6 ok
```

6.2 LOOPS

Loops come in two basic types: finite and indefinite. Finite loops are set up to repeat a given number of times; indefinite loops continue until a certain condition is fulfilled or a given event occurs, regardless how many repetitions this takes.

6.2.1 Finite Loops: "do...loop."

This control structure will repeat a given set of commands as many times as you specify—from an initial value to an upper limit.

For SnapFORTH to perform the usual stack operations and print your commands as well as keep track of how many times it has repeated a given operation, another type of stack is required. What we have been calling the "stack" is correctly termed the "parameter stack"; the new stack we are introducing now is usually called the "temporary stack" ("T-stack").

One of the simplest kinds of finite loops is one that will print a series of numbers between limits you specify. You could decide to print a list of numbers from 1 to 10, for example. To do this, you will need to do the following:

1. Name the definition.
2. Set up the initial loop value and upper limit.
3. Use "do" to move these loop parameters to the T-stack and start the loop.
4. Move a copy of the top number on the T-stack, called the "index", to the parameter stack.
5. Print this number.
6. Use "loop" to add one to the index on the return stack.

   If the index is now equal to the upper limit, "loop" will automatically remove the index and upper limit from the T-stack and exit.
   Otherwise, the loop will be continued (at step 4).

For aesthetic purposes, it is also useful to start a new line before printing each number, by adding a "cr", to make a vertical list. Your result should look like this:

```
: numberlist 11 1 do
   cr i . loop ; ok
```

Why is your limit 11 instead of 10? SnapFORTH will stop executing the commands within the loop when the current number on top of the T-stack (the index) equals or exceeds the upper limit. Always remember to add one to the limit when setting up a list like this.

Just to check, walk through the definition to be sure you know what each element does.

: numberlist

11	puts 11 on the parameter stack.
1	puts 1 on the parameter stack.
do	moves the loop parameters to the T-stack and starts the loop.
cr	starts a new line to begin the list.
i	copies the index from the top of the T-stack to the parameter stack.
	prints whatever is on the parameter stack.
loop ;	increments the index by 1. If the index equals 11, exits the loop and removes the index (11) and the upper limit (11) from the T-stack. Otherwise, starts the process over.

Try your loop and see what you get:

```
numberlist
1
2
3
4
5
6
7
8
9
10 ok
```

A further variation on the "do...loop" procedure involves use of the word "+loop." Instead of incrementing the index by one each time the loop repeats, you can determine the increment yourself. For instance, you could do the list from 1 to 20 by threes. The definition would be only slightly changed:

```
: numberlist 21 1 do
   cr i . 3 +loop ; ok
```

Note that the increment should immediately precede the "+loop." Your results will now look like this:

```
numberlist
1
4
7
10
13
16
19 ok
```

A more complicated kind of loop is a "nested loop"—a loop within a loop. One thing such a structure could do is take a list and compare each item in that list to all the items in a second list. One word that uses such nested loops is "odds," which examines probabilities in dice-throwing. Taking the sum of the numbers on the two dice, it is clear that some sums will occur more often than others because there are more combinations of numbers that can equal them. A sum like 2 could only be formed one way (1 + 1); on the other hand, 7 might be formed several ways. The word "odds" will tell you how many possible ways there are to make a given sum out of the 36 possible combinations of numbers (6 x 6) on the two dice.

The word "odds" must do several things, then. It must first compare a sum that you provide to every possible sum of dice values. The first time it finds a combination that leads to the same sum as yours, it will put a 1 (true) on the stack. Each successive equality it finds will increment this stack number by one. The final count on the stack will be the total number of possible combinations. After you print this, there will be nothing left on the stack.

How do you use nested loops in this word? You have two lists, one for each die, each from 1 to 6. You must compare a 1 on the first die to every possible number on the second die, then a 2 to every possible number, and so forth. You cannot increment the first loop until you have completed the second loop entirely. Therefore, the second loop is "nested" inside the first. To handle two lists at the same time, you will need one additional command. The word "i," which you already know, copies the index of a loop to the parameter stack for manipulation; if you have two loops being operated on at the same time, however, you must have another command to get at the index of the first loop, which is put on the T-stack first and is therefore further down. Stack manipulation commands do not work on the T-stack. The command "j" will work for this second outermost

loop. In this case, "j" is the index of the first die; "i" will index the innermost loop (the second die).

Now you are ready to write the definition for "odds":

```
: odds ( sum of throw --> )
    0 swap
   7 1 do ( j loop )
    7 1 do ( i loop )
       dup i j + =
     if swap 1+ swap
     then
    loop
   loop
     drop cr . ." out of
      36 ways " ;
```

Several elements of this definition are worthy of note:

: odds

0 swap	puts the count (0) on the stack and SWAPs it with the sum you are looking for so that the sum is again on the top of the stack. The count (0) is the number of combinations found so far. This count will be incremented by the number of possible combinations you find later.
7 1 do	establishes the parameters of and starts the outermost ("j") loop
7 1 do	establishes the parameters of and starts the innermost ("i") loop
dup	copies the sum you are seeking.
i j +	takes the first die ("i") and second die ("j") and adds them.
=	checks if this is the sum you are looking for (true or false).
if	tests the condition and removes it from the stack. If true, continues with the next statement. If false, skips to the nearest "then".
swap 1 +	swaps the sum and count. Adds 1 to the count, which is now on top of the stack.
swap	moves the sum you are looking for back to the top.

then	ends conditional structure.
loop loop	ends "i" and "j" loops.
drop cr	drops original sum and starts a new line.
	prints number of possible combinations.
."out of 36 ways"	prints this message.
;	ends definition.

What do you get when you try "odds"?

```
4 odds
3 out of 36 ways ok

2 odds
1 out of 36 ways ok
```

6.2.2 Indefinite Loops

Indefinite loops also come in two types: those with a test performed at the beginning of a loop ("begin...until") and those with a test performed at the end of the loop ("begin...while...repeat"). Because this latter kind of loop waits to perform the test, it always executes at least once.

6.2.3 "begin...until"

This loop will repeat a given operation until a certain condition is "met"—that is, until it sees a true (non-zero) condition on the stack. Between the words "begin" and "until" are words which manipulate the stack contents and ultimately lead to this true or false result. The word "until" removes this condition from the stack and either repeats or goes on.

An example of a task this loop might perform is something we will call "doubles," which will "throw dice" (generate random numbers) until it gets doubles and will then stop. The word will first announce its intention (by printing "let's throw doubles"') and then start generating and printing pairs of random numbers. After each throw it will check to see if the two numbers left on the stack are equal; if yes, it will stop, and if no, it will return to the beginning of the loop. Here is the definition of doubles:

```
: doubles ( --> )
  cr ." let's throw doubles! "
  begin .dice = until ;
```

Note that the message is not included in the loop. If it were, it would be printed each time a new pair of numbers were generated; all you want it to do is print at the beginning of the search. Now try out "doubles":

```
doubles
let's throw doubles!
4 3
1 2
1 5
5 6
5 4
6 1
2 2 ok
doubles
let's throw doubles!
6 3
4 3
1 2
6 6 boxcars ok
```

A slightly more complicated task for "begin...until" to accomplish is to check the throw of the dice to see if their sum is an even number. See if you understand all the elements in the definition:

```
: evens ( throw dice until throw is even )
  begin
    dice
    cr 2dup . . 2dup + 2 mod 0=
  until
    cr ." even throw is " . . ;
```

Note that "checking for evenness" can be expressed as "2 mod 0 = "—that is, dividing by two and checking to see if the remainder is 0. Now try out your new definition:

```
evens
5 2
4 3
4 4
even throw is 4 4 ok
```

It's a good habit to check the stack after you define a word to see what is left there. Type a new word, ".s ," to display the contents of the stack without destroying them:

```
.s 4 3 5 2 ok
```

Where did all these numbers come from? They must be all the dice throws that were rejected, the ones that added up to odd numbers. The definition works, of course, but you don't want to clutter up the stack with rejected numbers. The parameter stack is quite small, and if it "overflows," you will get an error message. Something should be built into the definition to drop these numbers.

then	ends conditional structure.
loop loop	ends "i" and "j" loops.
drop cr	drops original sum and starts a new line.
	prints number of possible combinations.
."out of 36 ways"	prints this message.
;	ends definition.

What do you get when you try "odds"?

```
4 odds
3 out of 36 ways ok

2 odds
1 out of 36 ways ok
```

6.2.2 Indefinite Loops

Indefinite loops also come in two types: those with a test performed at the beginning of a loop ("begin...until") and those with a test performed at the end of the loop ("begin...while...repeat"). Because this latter kind of loop waits to perform the test, it always executes at least once.

6.2.3 "begin...until"

This loop will repeat a given operation until a certain condition is "met"—that is, until it sees a true (non-zero) condition on the stack. Between the words "begin" and "until" are words which manipulate the stack contents and ultimately lead to this true or false result. The word "until" removes this condition from the stack and either repeats or goes on.

An example of a task this loop might perform is something we will call "doubles," which will "throw dice" (generate random numbers) until it gets doubles and will then stop. The word will first announce its intention (by printing "let's throw doubles"') and then start generating and printing pairs of random numbers. After each throw it will check to see if the two numbers left on the stack are equal; if yes, it will stop, and if no, it will return to the beginning of the loop. Here is the definition of doubles:

```
: doubles ( --> )
  cr ." let's throw doubles! "
  begin .dice = until ;
```

Note that the message is not included in the loop. If it were, it would be printed each time a new pair of numbers were generated; all you want it to do is print at the beginning of the search. Now try out "doubles":

```
doubles
let's throw doubles!
4 3
1 2
1 5
5 6
5 4
6 1
2 2 ok
doubles
let's throw doubles!
6 3
4 3
1 2
6 6 boxcars ok
```

A slightly more complicated task for "begin...until" to accomplish is to check the throw of the dice to see if their sum is an even number. See if you understand all the elements in the definition:

```
: evens ( throw dice until throw is even )
  begin
    dice
    cr 2dup . . 2dup + 2 mod 0=
  until
    cr ." even throw is " . . ;
```

Note that "checking for evenness" can be expressed as "2 mod 0 = "—that is, dividing by two and checking to see if the remainder is 0. Now try out your new definition:

```
evens
5 2
4 3
4 4
even throw is 4 4 ok
```

It's a good habit to check the stack after you define a word to see what is left there. Type a new word, ".s ," to display the contents of the stack without destroying them:

```
.s 4 3 5 2 ok
```

Where did all these numbers come from? They must be all the dice throws that were rejected, the ones that added up to odd numbers. The definition works, of course, but you don't want to clutter up the stack with rejected numbers. The parameter stack is quite small, and if it "overflows," you will get an error message. Something should be built into the definition to drop these numbers.

That something is the "detour" section of the structure, called "while." It functions like "else" in the "if...else...then" structure; it performs a second command whenever the condition does not fulfill your criteria. This command could be to drop the offending numbers, to print out an explanatory message, or to perform any other task.

There are a few notable differences in this new structure, called "begin...while...repeat." Most important, the test is performed at the beginning of the loop. The "while" section will be executed (and the loop repeated) only if the condition puts a true condition on the stack. Like "begin...until," the condition is always removed from the stack.

You can use "while" to dispose of the unwanted (odd) throws in "evens." You might remove the offending numbers from the stack by printing them; you might also simply drop them. In this case, you will print them with an explanatory message. Study the following definition carefully:

```
: evens2 ( throw dice until throw is
           even )
   begin
     dice
     2dup + 2 mod
   while
     cr ." odd throw is " . .
   repeat
     cr ." even throw is " . . ;
```

Be sure you understand what is on the stack at each point. After the "2 mod" section, there is either a one or a zero on the stack. If it is one (meaning an odd number), the "while" section will be executed and the loop repeated. If zero (meaning an even number), the structure will skip the "repeat" section and print the message following it.

Now try out the definition and check the stack to see if anything is left there.

```
evens2
even throw is 5 3 ok

evens2
odd throw is 5 4
even throw is 5 1 ok

.s EMPTY ok
```

It works! Congratulations on fixing your first bug.

6.3 ANOTHER DUP

Because the word "loop" (or "+loop") lies at the end of the "do...loop" (or "+loop") structure, you will always execute a

loop at least once, even if the initial lower limit already equals (or exceeds) the upper limit:

```
: test 0 0 do i . loop ; ok
test 0 ok
```

To prevent this, you can usually rewrite your loop so that if the initial lower limit of the loop is zero, the loop is skipped. For example, the following word "countup" counts (and prints) up to whatever limit you push on the stack, starting with one. If the limit is zero, it does nothing:

```
: countup ( n-limit --> )
    dup ( if not zero )
   if 1+ 1
    do I .
     loop
   else drop
   then ; ok

3 countup 1 2 3 ok
0 countup ok
```

Notice the need for an "else" clause to "drop" the unwanted initial zero. This construction, and its variations, are so common in SnapFORTH that a special command is provided to optimize them. This command, called "?dup" ("query-dup"), duplicates the number on top of the stack, provided that it is not a zero. The previous example can now be simplified:

```
: countup2
   ?dup
   if 1+ 1
    do i .
     loop
   then ; ok

3 countup2 1 2 3 ok
0 countup2 ok
```

The word "?dup" can be used in other flow of control situations, such as "begin...repeat."

6.4 THE RETURN STACK

You should now know how to use the parameter stack easily with "drop," "dup," "swap," and so forth. You have also learned that another stack (the T-stack) is used by "do...loop" to keep indices and limits. You can also use a third stack, the "return stack," with the SnapFORTH words >r ("to-r"), r ("r"), and r> ("r-from"). The command >r removes the top item from the parameter stack and pushes it onto the return stack. The command r> removes the top item from the return stack and pushes it onto the parameter stack. These commands should only be used within definitions. For example, one way to swap the second and third item on a parameter stack is this:

```
: swap23
   >r swap r> ; ok
1 2 3 swap23 ok
. . . 3 1 2 ok ( instead of 3 2 1 )
```

The command "r" copies the item on top of the return stack and pushes it onto the parameter stack. A word such as "doit," which adds 3 to a number and multiplies the result by itself, would use all three return stack commands:

```
  : doit
     >r 3 r + r> * ; ok
 2 doit . 10 ok
```

This is useful if you need a number several times in a definition but do not want to keep it on top of the parameter stack. Also remember to drop any numbers you push on the return stack before ending a definition (using the sequence "r> drop" as necessary).

6.5 FINISHING EARLY

6.5.1 Leaving a "do...loop"

What do you do if you are in the middle of a "do...loop" and decide that you are done? For example, suppose you are searching an array for a certain element and have found it. Why should you have to continue looking? In fact, you can exit a "do...loop" at any time with the command "leave." "Leave" sets the upper limit of a loop to its current index (the index is not changed). When you reach the end "loop" or "+loop," you will exit the loop. For example:

```
: early 100 0
   do i 5 =
     if leave
     else i .
     then
   loop ; ok

early 0 1 2 3 4 ok
```

When "i" equals 5, you exit the loop via "leave."

6.5.2 Exiting a Word

SnapFORTH also provides for immediately finishing the execution of a word with the command "exit." "Exit" is especially useful if you encounter a condition while executing a word

which makes executing the rest of the word pointless. For example, you may be deep inside your flow of control structure when you detect that you are about to divide by zero. What you would like to do is to print an error message and exit the word:

```
: calculate
   begin
     ...code...
   if
     ...more code...
   else
     if ( n1 and n2 are now on
          the stack )
       dup 0= ( divide by zero? )
      if cr ." wrong! "
       drop drop exit
      else /
        ...continue...
      then
     then
   then
  until ;
```

Be sure to clear the parameter stack and return stack before you "exit." You must "leave" a "do...loop" rather than "exit" it.

6.6 CASE STATEMENTS

"Case" statements resemble "staircases" of dependent "if...else...then" clauses, such as we have seen in section 6.1.2. Case statements are easier to read than multiple "if...else...then" statements, especially if there are more than two dependent clauses.

The case structure begins with the word "docase" and ends with the word "endcase." Each dependent clause within the case begins with a "selector" (a number), followed by the word "case" and ends with the word "else." If the argument that is on the stack when "docase" is entered equals one of the selectors, the dependent clause of that selector is executed, and SnapFORTH then skips to the "endcase" word. If the argument fails to match any of the selectors, execution will continue at the first SnapFORTH word (if any) following the last "else" (sometimes called an "otherwise" clause). Here is a simple example:

```
: spellnumber ( n1 --> )
  docase
     0 case ." zero " else
     1 case ." one " else
     2 case ." two " else
       ." too big "
   endcase ;
```

Now try "spellnumber":

```
 0 spellnumber zero ok
 1 spellnumber one ok
 5 spellnumber too big
```

There are also ways in which to use "if" within case statements. In the following example, for instance, you want SnapFORTH to check the temperature of some water and display a message telling you what form the water is in—ice, water, or steam. You could accomplish it this way:

```
 : h2o ( temp --> )
   docase
      32 < if ." ICE " else
     212 > if ." STEAM " else
              ." WATER "
   endcase ;
```

Give "h2o" some temperatures and see what happens:

```
 -14 h2o ICE ok
 33 h2o WATER ok
 213 h2o STEAM ok
```

TABLE 6.1 FLOW OF CONTROL

WORD	STACK	ACTION
and	(n1 n2—> n3)	Leaves bitwise "and" of n1 and n2.
or	(n1 n2—> n3)	Leaves bitwise "or" of n1 and n2.
<	(n1 n2—> t/f)	True if n1 is less than n2.
>	(n1 n2—> t/f)	True if n1 is greater than n2.
=	(n1 n2—> t/f)	True if n1 equals n2.
0=	(n1—> t/f)	True if n1 equals zero.
0<	(n1—> t/f)	True if n1 is less than zero.
0>	(n1—> t/f)	True if n1 is one or greater.
if else then	(t/f—>)	Used in a definition in the form t/f "if...else...then" or simply t/f "if...then." If condition is true, the words following "if" are executed (but the words following "else" are skipped). If false, the words following "else" are executed (if the "else" part exists).
do	(n1 n2—>)	Used in a definition in the form "do...loop" or "do...+loop." The words within the loop will be repeatedly executed (in order) until the loop index (initially n2) equals or exceeds the loop upper limit.
loop		If the word "loop" ends the the loop, the index will be incremented by 1. If the word
+loop	(n—>)	"+loop" ends the loop, the index will be incremented by whatever number is currently on top of the stack. If the increment is negative,"+loop" will decrease the index until it is less than the the limit.

Now try "spellnumber":

```
 0 spellnumber zero ok
 1 spellnumber one ok
 5 spellnumber too big
```

There are also ways in which to use "if" within case statements. In the following example, for instance, you want SnapFORTH to check the temperature of some water and display a message telling you what form the water is in—ice, water, or steam. You could accomplish it this way:

```
 : h2o ( temp --> )
   docase
      32 < if ." ICE " else
     212 > if ." STEAM " else
           ." WATER "
   endcase ;
```

Give "h2o" some temperatures and see what happens:

```
 -14 h2o ICE ok
 33 h2o WATER ok
 213 h2o STEAM ok
```

TABLE 6.1 FLOW OF CONTROL

WORD	STACK	ACTION
and	(n1 n2—> n3)	Leaves bitwise "and" of n1 and n2.
or	(n1 n2—> n3)	Leaves bitwise "or" of n1 and n2.
<	(n1 n2—> t/f)	True if n1 is less than n2.
>	(n1 n2—> t/f)	True if n1 is greater than n2.
=	(n1 n2—> t/f)	True if n1 equals n2.
0=	(n1—> t/f)	True if n1 equals zero.
0<	(n1—> t/f)	True if n1 is less than zero.
0>	(n1—> t/f)	True if n1 is one or greater.
if	(t/f—>)	Used in a definition in the form t/f "if...else...then" or simply
else then		t/f "if...then." If condition is true, the words following "if" are executed (but the words following "else" are skipped). If false, the words following "else" are executed (if the "else" part exists).
do	(n1 n2—>)	Used in a definition in the form "do...loop" or "do...+loop." The words within the loop will be repeatedly executed (in order) until the loop index (initially n2) equals or exceeds the loop upper limit.
loop		If the word "loop" ends the the loop, the index will be incremented by 1. If the word
+loop	(n—>)	"+loop" ends the loop, the index will be incremented by whatever number is currently on top of the stack. If the increment is negative,"+loop" will decrease the index until it is less than the the limit.

Now try "spellnumber":

```
 0 spellnumber zero ok
 1 spellnumber one ok
 5 spellnumber too big
```

There are also ways in which to use "if" within case statements. In the following example, for instance, you want SnapFORTH to check the temperature of some water and display a message telling you what form the water is in—ice, water, or steam. You could accomplish it this way:

```
: h2o ( temp ---> )
  docase
     32 < if ." ICE " else
    212 > if ." STEAM " else
             ." WATER "
  endcase ;
```

Give "h2o" some temperatures and see what happens:

```
-14 h2o ICE ok
33 h2o WATER ok
213 h2o STEAM ok
```

TABLE 6.1 FLOW OF CONTROL

WORD	STACK	ACTION
and	(n1 n2—> n3)	Leaves bitwise "and" of n1 and n2.
or	(n1 n2—> n3)	Leaves bitwise "or" of n1 and n2.
<	(n1 n2—> t/f)	True if n1 is less than n2.
>	(n1 n2—> t/f)	True if n1 is greater than n2.
=	(n1 n2—> t/f)	True if n1 equals n2.
0=	(n1—> t/f)	True if n1 equals zero.
0<	(n1—> t/f)	True if n1 is less than zero.
0>	(n1—> t/f)	True if n1 is one or greater.
if	(t/f—>)	Used in a definition in the form t/f "if...else...then" or simply t/f "if...then."
else then		If condition is true, the words following "if" are executed (but the words following "else" are skipped). If false, the words following "else" are executed (if the "else" part exists).
do	(n1 n2—>)	Used in a definition in the form "do...loop" or "do...+loop." The words within the loop will be repeatedly executed (in order) until the loop index (initially n2) equals or exceeds the loop upper limit.
loop		If the word "loop" ends the the loop, the index will be incremented by 1. If the word
+loop	(n—>)	"+loop" ends the loop, the index will be incremented by whatever number is currently on top of the stack. If the increment is negative,"+loop" will decrease the index until it is less than the the limit.

TABLE 6.1 FLOW OF CONTROL (continued)

WORD	STACK	ACTION
i	(—> n)	Used within a "do...loop" or "do...+loop." The word "i" leaves a copy of the loop index on the parameter stack.
j	(—> n)	Used within a nested "do...loop" or "do...+loop." The word "j" leaves a copy of the index of the next-outermost loop on the parameter stack.
k	(—> n)	Used within a nested "do...loop" or "do...+loop." The word "k" leaves a copy of the index of the third-outermost loop on the parameter stack.
begin		Used in a definition in the form "begin...until" or "begin...while...repeat." The words following the "begin" will be repetitively executed (in order) and must leave a t/f condition on stack.
until	(t/f—>)	If the loop ends with "until," the loop will terminate when a true condition is left on the stack.
while repeat	(t/f—>)	If it ends with "while," the loop will terminate when a false condition is left, and the words between "while" and "repeat" will be skipped. Otherwise, the words between "while" and "repeat" will be executed and the loop will repeat. If you use "while," the test will occur before the loop is entered. If you use "repeat," the loop will always be entered at least once.
?dup	(n—> n (n))	Duplicates n if it is not zero.
>r	(n—>)	Transfers n to the return stack.
r	(—> n)	Copies the number on top of the return stack to the parameter stack.

TABLE 6.1 FLOW OF CONTROL (continued)

WORD	STACK	ACTION
r>	(—> n)	Transfers n from the return stack to the parameter stack.
leave		Forces termination of a "do...loop" or a "do...+loop" by setting the upper limit of a loop to its current index. The index is unchanged. Execution continues until the end "loop" or "+loop."
exit		Forces termination of a word. Should not be used within a "do...loop" or "do...+loop."
docase	(n—>)	Used in a definition in the form: docase
case		selector1 case words1 else selector2 case words2 else otherwise-words
endcase		endcase. A selector is an expression leading to a number. If the selector equals the number n, the words within the respective "case...else" will be executed, and execution will continue following the "endcase.' If no selector matches n, the otherwise-words (between the last "else" and "endcase") will be executed.

Now try "spellnumber":

```
0 spellnumber zero ok
1 spellnumber one ok
5 spellnumber too big
```

There are also ways in which to use "if" within case statements. In the following example, for instance, you want SnapFORTH to check the temperature of some water and display a message telling you what form the water is in—ice, water, or steam. You could accomplish it this way:

```
: h2o ( temp ---> )
  docase
     32 < if ." ICE " else
    212 > if ." STEAM " else
             ." WATER "
  endcase ;
```

Give "h2o" some temperatures and see what happens:

```
-14 h2o ICE ok
33 h2o WATER ok
213 h2o STEAM ok
```

TABLE 6.1 FLOW OF CONTROL

WORD	STACK	ACTION
and	(n1 n2—> n3)	Leaves bitwise "and" of n1 and n2.
or	(n1 n2—> n3)	Leaves bitwise "or" of n1 and n2.
<	(n1 n2—> t/f)	True if n1 is less than n2.
>	(n1 n2—> t/f)	True if n1 is greater than n2.
=	(n1 n2—> t/f)	True if n1 equals n2.
0=	(n1—> t/f)	True if n1 equals zero.
0<	(n1—> t/f)	True if n1 is less than zero.
0>	(n1—> t/f)	True if n1 is one or greater.

if
else
then
	(t/f—>)	Used in a definition in the form t/f "if...else...then" or simply t/f "if...then." If condition is true, the words following "if" are executed (but the words following "else" are skipped). If false, the words following "else" are executed (if the "else" part exists).

do	(n1 n2—>)	Used in a definition in the form "do...loop" or "do...+loop." The words within the loop will be repeatedly executed (in order) until the loop index (initially n2) equals or exceeds the loop upper limit.
loop		If the word "loop" ends the the loop, the index will be incremented by 1. If the word
+loop	(n—>)	"+loop" ends the loop, the index will be incremented by whatever number is currently on top of the stack. If the increment is negative,"+loop" will decrease the index until it is less than the the limit.

CHAPTER 7: MORE ABOUT NUMBERS

So far we have been working with the familiar decimal (base 10) number system. In the decimal system, each digit in a number can have one of ten values: 0 1 2 3 4 5 6 7 8 or 9. This system seems natural to us. After all, we each have ten fingers (and toes). When you first begin a session with the SnapFORTH language, the numeric base is initialized to decimal. You can return to decimal at any time with the word "decimal." Computers, however, have essentially one finger (or toe) each and count in the binary (base 2) system. In this system, each digit can have one of two possible values: 0 or 1.

7.1 INTERNAL REPRESENTATION

Each zero or one in computer memory is called a "bit" or is said to contain one "bit" of information. But, practically speaking, working with individual bits is both tedious and confusing. Instead, computer memory is organized into groups of eight bits called "bytes" of information. Each byte in memory has a location or "address." You can think of each byte as a cubbyhole or post office box containing eight zeroes or ones. But please don't confuse the address of a byte with its contents. The byte at address 27 could contain the number 15.

7.2 HEXADECIMAL NUMBERS

The hexadecimal (base 16) number system (usually abbreviated "hex") lets you work with groups of four zeroes or ones. Each of the 16 possible groups corresponds to a single hexadecimal digit:

HEX	BINARY	DECIMAL
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
a	1010	10

HEX	BINARY	DECIMAL
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

The word "hex" lets you work in the hexadecimal system:

```
decimal 14 hex .e ok
decimal ok
```

You remain in whatever number system you typed last (in this case, decimal); it is a good idea to finish in decimal whenever possible. The contents of any byte can be represented by two hexadecimal digits. Hex numbers are more readable if you precede them with an extra zero: "0ab" instead of just "ab."

7.3 BYTES AND WORDS

How many different patterns can you store in each byte? Well, the smallest pattern you can store is hex 0 (decimal 0). The largest pattern you can store is hex 0ff (decimal 255). This makes 256 possible patterns. Each pattern can represent a decimal integer, but it can also represent almost anything else. For example, the hex pattern 41 can mean "A" or even "don't forget to see your dentist tomorrow." As long as everyone agrees what each pattern means in a given situation, there will be no confusion.

Since 256 possible patterns just aren't enough, two adjacent bytes are usually combined into one 2-byte "memory cell." One of the two byte addresses, usually the lower one, is chosen to be the address of the memory cell. The rightmost byte is called the "least significant byte" and is usually found at the lower or cell address. The leftmost byte is called the "most significant byte."

A 2-byte memory cell can hold one of 65536 possible patterns. This normally gives you a possible address range from 0 to 65535. (With the HHC's "bank-switching," you can reach a much larger memory. For details, consult the ***SnapFORTH Reference Manual***.) The meaning of each pattern depends on the context in which it is used. For example, the contents of one 2-byte cell might represent the address of another 2-byte cell (or single byte). This gives you a possible address range from 0 to 65535. Since many microcomputers use 2-byte addresses, they are limited to 65536 bytes of usable memory and are said to have a "65K address space."

By the way, ten bits of memory can hold 1024 different patterns:

```
hex 03ff decimal .1023 ok
```

The number 1024 is so close to 1000 that it is usually called "1K." You will often find that 65536 bytes of memory are simply called "64K" of memory.

7.4 SIGNED INTEGERS

Another way of interpreting the 65536 possible 2-byte patterns is to divide them into half positive and half negative integers. To accomplish this, you will have to "sacrifice" one of your 16 binary bits, giving you a reduced range from +32767 to -32768. That 16th bit (which would have given you 65536 positive bits in an unsigned system) will now be used to mark the number for positiveness or negativeness. If the bit farthest to the left (the most significant bit or MSB) is zero, then the remaining 15 bits are interpreted as positive integers. If the MSB is one, the numbers are negative integers. The largest possible positive integer is then:

```
hex 07fff decimal . 32767 ok
```

which would look like this in binary:

```
0111111111111111
```

(This number is also often called "32K.")

To form negative integers, we can subtract positive integers from zero:

```
decimal 0 1 - .-1 ok
```

Notice what the result would have been if you had typed that in binary:

```
0000000000000000 - 0000000000000001 =
    1111111111111111
```

Now you can see that -1 (decimal) equals 0ffff in hex. Check and see if it's true:

```
hex 0ffff . -1 ok
decimal ok
```

Even though you "sacrificed" the 16th bit as a sign marker, it is still included in any computations. This can lead to strange results if you happen to exceed the allotted 15-bit range. For instance, if you try to create a positive integer larger than 32767, this is what will happen:

```
32767 1 + .-32768 ok
-32768 hex .-8000 ok
decimal ok
```

Any 2-byte number (or "single-number integer") larger than 32767 will have a MSB of one and will be treated as a negative number. In fact, by adding one to the largest positive number (32767), we have "wrapped around" within the number space to the largest negative number (-32768). In SnapFORTH, this "wraparound" is simply the normal consequence of having 2-byte "signed" integers and is not considered to be an error (by SnapFORTH).

SnapFORTH actually allows you to enter much larger numbers, but this can lead to some surprising results:

```
 100000 .-31072 ok
```

Also surprising are the results of arithmetic calculations producing intermediate values of more than 16 signed bits:

```
 30000 10 * 10 / .-2768 ok
```

7.5 RATIONAL ARITHMETIC

Many operations with numbers require multiplication by fractions or ratios. In other words, these operations can be done as a multiplication followed by a division. For example, to find 2/3 of 1000, you can type

```
 1000 2 * 3 / .666 ok
```

To multiply 20 by pi to two decimal digits of accuracy (3.14), try

```
 20 314 * 100 / .62 ok
```

For greater accuracy (3.1415) try

```
 20 31415 * 10000 / .-2 ok
```

Woops! What happened? The intermediate product of 20 * 31415 was larger than 16 signed bits, and the extra bits were discarded.

To protect these intermediate products, you can use the SnapFORTH operator "*/mod" ("star-slash-mod"):

```
 20 31415 10000 */mod .62 ok ( the answer )
 .8300 ok ( the remainder )
```

This operator multiplies the second and third stack items to form a larger intermediate product (20 31415 * yields 628300). (This product, called a "double number," will be discussed in the next section.) A "/mod" is then done between the intermediate product and the number on top of the stack to leave the single-number remainder (second item) and quotient (top item) on the stack. Using "*/mod," you are guaranteed that your calculations will be fully accurate, as long as the final result does not exceed the limits of a single-number integer. The "*/mod" operator is especially

useful in conversions, such as feet to meter and Fahrenheit to Centigrade.

7.6 DOUBLE NUMBERS

Sometimes even single-number results are too small. For example, if you want to keep track of a checking account in pennies, you would be limited to amounts no larger than 327 dollars and 67 cents. Because of this, SnapFORTH provides "double-number" 4-byte signed-integer numbers and operators. With them, you may now put up to $21474836.54 into your checking account.

Any number ending with a comma is considered to be a double number. To print a double number on the stack, use the word "d." as follows:

```
 200000, ok
 d.200000 ok
 2, d.2 ok
```

SnapFORTH actually splits the double number into two single numbers: the most significant number is kept on the stack above the least significant number. To see how the two numbers divide, display the result with a conventional print command instead of a "d.":

```
 hex ok
 02468ac, cr . .
 24 68AC, ok
 decimal ok
```

This suggests that a single-number integer can be increased to a double-number integer by pushing a second number over it on the stack. Positive numbers can be extended by pushing a zero; negative numbers by pushing a -1. The SnapFORTH word which does this is "s>d" (single to double number):

```
 123 s>d d.123 ok
 -49 s>d d.-49 ok
```

To convert a double number back to a single number, simply "drop" off the most significant part:

```
 4, drop .4 ok
 -1, drop .-1 ok
```

Of course, if the double number doesn't fit the single-number format, the extra bits will be lost.

Most of the single-number arithmetic and stack words have a double-number equivalent. For example, "2drop" drops a double number from the stack. These double-number words are briefly described in table 7.1. The double-number stack operators can also be used to manipulate single-number numbers two at a time—for example, the phrase "over over" becomes "2dup."

7.7 UNSIGNED INTEGERS

Sometimes it is better to use a number as a 2-byte unsigned number (ranging from 0 to 65535) rather than a 2-byte signed number (ranging from -32768 to +32767). For example, you may need to check if one address in memory is lower than another:

```
  hex ok
  36f9 7aab < .1 ok ( true )
```

but

```
  7aab 9072 < .0 ok ( false! )
  decimal ok
```

Why did SnapFORTH return a false in the second example? The hexadecimal number 9072 is treated as a negative number by the signed comparator <. SnapFORTH provides a handful of arithmetic words (u* u< and u.) which treat numbers as unsigned 2-byte numbers. SnapFORTH also provides an unsigned mixed-number word "m/mod" ("m-slash-mod"), which is briefly described in table 7.2. Try the above comparison again with the unsigned comparator "u<":

```
  hex ok
  7aab 9072 u< .1 ok ( true )
  decimal ok
```

7.8 MORE MIXED-NUMBER OPERATORS

You may have noticed that SnapFORTH provides no double-number multiplication command. This would lead to double-double-number results, which would be difficult to use. SnapFORTH does however provide two mixed-number words, "m*" and "m/," which allow you to obtain double-number results from mixed-number arithmetic.

The word "m*" multiplies two signed single-number integers to form a signed double-number product:

```
  1000 1000 m* d.1000000 ok
```

The word "m/" divides a signed double-number divisor by a signed single-number dividend to produce two signed single-number results: the quotient (on top of the stack) and the remainder (underneath it). The remainder takes its sign from the dividend:

```
  1000000, 1000 m/ . .1000 0 ok
```

7.9 FLOATING POINT

Using a new set of floating point commands, you can deal with numbers other than integers—entering, transforming, and computing with them. By adding a dot to a number (within it or at the end), it automatically becomes a floating point number. Each floating point number uses eight bytes apiece, twice that of a double number. Numbers entered in scientific notation automatically become floating point numbers. Here are some valid floating point numbers:

```
  3. 3.14159 2.5e10
```

To display them, you need the new command " g. "

```
  3. g.3 ok
  3.14159 g.3.14159 ok
  2.5e10 g.25000000000 ok
```

Simple arithmetic commands have their floating point analogs—"f+," "f-," "f*," and "f/." Try a few examples to see what you get:

```
  1. 2.5 f+ g. 3.5 ok
```

A floating point variable also exists—"fvar." It is created within a TSA just like any other variable, except that it allocates eight bytes instead of two. The floating point commands "f@" and "f!" operate on such variables. Floating point constants, on the other hand, must be contained within definitions. For example, if you wanted a constant for the value of pi, you could create a word which would contain the floating point value thus:

```
  : pi 3.14159 ; ok
```

Then you could call for "pi" and use it as you would any constant:

```
  pi g. 3.14159 ok
```

SnapFORTH already includes a floating point comparator, "f<." You can define other comparators yourself, such as "f=":

```
  : f= ( fp#1 fp#2 --> t/f )
    f- >r drop 2drop r> 0= ;
```

Finally, four stack manipulation analogs exist for floating point numbers, "fswap," "fdup," "fdrop," and "fover." These work just like their conventional counterparts.

TABLE 7.1 DOUBLE-NUMBER WORDS

WORD	STACK	ACTION
d+	(d1 d2—> d-sum)	Adds d1 to d2.
dabs	(d1—>ud2)	Forms the absolute value of d1.
dnegate	(d1—>d2)	Reverses the sign of d1.
d.	(d1—>)	Prints a double number.
d-	(d1 d2—>d-diff)	Subtracts d1 from d2.
d=	(d1 d2—>t/f)	True if d1 equals d2.
2!	(dn address—>)	Stores dn at the address.
2@	(address—>dn)	Fetches dn from the address.
2drop	(dn—>)	Drops dn from the stack.
2dup	(d1—>d1 d1)	Duplicates d1 on the stack.
2over	(d1 d2 —>d1 d2 d1)	Leaves a copy of the second double number on the stack.
2rot	(d1 d2 d3 —>d2 d3 d1)	Rotates the third double number to the top of the stack.
2swap	(d1 d2—>d2 d1)	Swaps d1 and d2.
dvar		The double-number equivalent of "var," used only within TSAs.

TABLE 7.2 MORE ARITHMETIC WORDS

WORD	STACK	ACTION
decimal		Sets the SnapFORTH number base to 10.
hex		Sets the SnapFORTH number base to 16.
base		Character variable containing the current numeric base.
*/mod	(n1 n2 n3—> n4 n5)	Multiplies n1 by n2, then does a "/mod" by n3. The intermediate result of the multiplication is a double number.
u*	(un1 un2—>un3)	Performs unsigned multiplication of un1 by un2.
u<	(un1 un2—>t/f)	True if unsigned un1 is less than un2.
u.	(un1—>)	Prints an unsigned number.
m/mod	(ud1 un2 —>un3 ud4)	Performs unsigned division of double number ud1 by single number un2, leaving the single-number remainder un3 and double-number quotient un4.
m*	(n1 n2—> d-prod)	Multiplies n1 by n2, leaving a double-number product.
m/	(dn1 n2—> n3 n4)	Divides double number dn1 by single number n2, leaving single-number remainder n3 and single-number quotient n4.

TABLE 7.3 FLOATING POINT WORDS

WORD	STACK	ACTION
g.	(fp#—>)	Prints a floating point number.
f+	(f1 f2—>f3)	Adds two floating point numbers.
f-	(f1 f2—>f3)	Subtracts f2 from f1.
f*	(f1 f2—>f3)	Multiplies f1 and f2.
f/	(f1 f2—>f3)	Divides f1 by f2.
f<	(f1 f2—>t/f)	Leaves a true if f1 is less than f2; otherwise leaves a false.
fswap	(f1 f2—>f2 f1)	Swaps the two top floating point numbers on the stack.
fdup	(f1—>f1 f1)	Duplicates the top floating point number on the stack.
fdrop	(f1 f2—>f1)	Removes the top floating point number on the stack.
fover	(f1 f2—> f1 f2 f1)	Pushes a copy of the second floating point number on the stack onto the stack.
fvar	<name> (—>)	Creates floating point variable of name <name> within a TSA.
f@	(addr—>f-val)	Replaces address with the value f-val found at that address.
f!	(f-val addr—>)	Stores f-val at address.

(This command must be added to SnapFORTH. See the text.)

WORD	STACK	ACTION
f=	(f1 f2—>t/f)	Leaves a true if f1 is equal to f2; otherwise leaves a false.

CHAPTER 8: STRINGS

Strings allow you to enter, manipulate, store, move, and print characters in almost all the ways you do numbers. You must learn a new and slightly different set of rules to master strings.

8.1 STRING CONSTANTS

Before a string is used or stored, it must be created and named. One way to do this is to put the characters into a string constant whose contents do not change. Enter the word "string" followed by a quotation mark and a space, type your string plus quotation mark, space, and give the string a name. Follow this example:

```
string" abc" alphabet ok
```

What exactly have you done? Well, you now have a "box" of sorts called "alphabet" in memory that contains several elements:

```
3 a b c
```

First, your constant has a name, "alphabet." Second, it has a length, 3. Each character is one byte long, so you have three bytes in your string. Notice, however, that the number 3 takes up its own byte; your total string "box" is actually four bytes long.

Now that you've put the string into a constant, how can you get the information back out again? Invoke "alphabet" and see what happens:

```
alphabet . 1000 ok
```

The number you got (we're using 1000 as an example) is certainly not the string that you typed in; it is the address of "alphabet." (Specifically, it is the address of the constant's first element which, as you remember, is not the first letter of the string—but rather the current length, 3.) You have to do a little more before you can see the contents of "alphabet."

First, you need to use the word "count." Try typing it after "alphabet" with two dots:

```
alphabet count . . 3 1001 ok
```

Now what's on the stack? The first (top) number, 3, is the current length of "alphabet"; the second is the address of "alphabet" plus one, which is the address of the first character ("a") of the string. The initial address, 1000, is no longer on the stack. These two arguments, in this order, are required for another word, "type," to display the contents of "alphabet." That is, with these instructions, "type" knows to display three characters starting at address 1001. See if this is true:

```
alphabet count type abc ok
```

Often you will want to know just the length of a string. Here's one way to write a word ("len") that would accomplish this:

```
: length ( addr len --> len )
   swapdrop ;
```

Try it out:

```
alphabet count length . 3 ok
```

How does it work? You invoke the name of your constant, which leaves the initial address of its string on the stack. The word "count" increments the address by 1, giving you the first-character address, and puts the current length of the string on the stack above it. These two arguments are passed to the word "length." The word "swapdrop" reverses the position of the length and address and drops the top element (the address), leaving the length.

8.2 STRING VARIABLES

For strings that will be changing frequently, you will need to employ string variables, which are contained in TSAs just like ordinary variables. Let's set up a new TSA called "strings" which will contain a string variable (called "letters") of no more than 20 characters:

```
area strings
   20 string letters
endarea
strings drop ok
```

You will probably want to type these lines into a source file for future use, using the commands given in chapter 4, "The Editor." Don't forget to "load" the file.

8.3 STRING MANIPULATION COMMANDS

8.3.1 s! and cmove

One way to put information into "letters" is with a string constant, using the new command "s!" ("s store"). You should add this definition to the same source file that you used for the TSA "strings":

```
: s! ( addrfrom lenfrom addrto
     lento --> )
   drop 2dup 1- c! swap cmove ;
```

To use "s!," you must provide a "from" string (address and length) and a "to" string (address and length). Now try to store the contents of "alphabet" into "letters":

```
alphabet count letters count s! ok
```

Check the contents of "letters" to see if the operation worked, using the same commands you did with the string constant "alphabet":

```
letters count type abc ok
```

The command you have just used in "s!"—"cmove"—has other independent uses. It will move sequences of bytes (such as strings) from one address to another, starting with the leftmost byte (with the lower address). All you need to tell SnapFORTH is the from-address, the to-address, and the number of bytes being moved.

8.3.2 S=

The command s= ("s equals") will compare two strings (whether in variables or constants) for equality. It needs the usual current length and first-character address for each string to make the comparison. Depending on the result, a 1 or a 0 will be left on the stack, and all the other arguments will be destroyed. Let's compare "letters" and "alphabet":

```
letters count alphabet count
   s= . 1 ok
```

8.3.3 S+

The command s+ ("s plus") is useful when you want to get at just part of a string and ignore the rest of it. For example, suppose you wanted to see just a few letters, say the last two, of "letters." The word s+ requires first-character address, current length, and an offset as arguments to single out part of the string. Your operation would look like this:

```
letters count 1 s+ type bc ok
```

Check to be sure you know what each element in this operation did. Typing "letters count" put 3 and 1001 on the stack. Typing "1" added an offset. Typing "s+" incremented the first-character address by the offset, and "type" displayed the characters beginning at that address.

If you wanted to get at just the middle character, the operation would be slightly more complicated. Let's try a longer string to show how this might work. First, set up a string constant called "days$" containing abbreviations for the seven days:

```
string" montuewedthufrisatsun" days$
```

Now you want to define a word (we'll call it "dayofweek" for "day of week") that will accept a number from the user (1 for Monday, 2 for Tuesday, etc) and create a string containing the abbreviation for that day. The word must get at the abbreviation and also cut off any abbreviation that appears later in the string. Here's how:

```
 : dayofweek ( n --> )
   days$ count rot 1- 3 * s+ drop 3 ;
```

Walk through this definition (paying particular attention to the stack contents at each point) to be sure you know what each element does. We're assuming that the address of "days$" is 1000:

3		user enters offset for "Wednesday".
	: dayofweek	starts definition.
1000 3	days$	puts initial address of constant on stack.
21 1001 3	count	adds first-character address and current length to stack.
3 21 1001	rot	moves third element (offset) of stack to the top.
2 21 1001	1-	subtracts 1 from offset. (You must do this because the constant is numbered starting with 0 but we number the days of the week starting with 1.)
3 2 21 1001	3	puts 3 on the stack.
6 21 1001	*	multiplies top two numbers, replacing them with the answer. (There are three letters for each abbreviation, so s+ needs an offset of 6 to get to the third abbreviation.)
21 1007	s+	accepts offset, number of characters, and first-character address and increments the latter by the offset (destroying the offset in the process).
1007	drop	eliminates the number of characters from the stack.
3 1007	3	puts a 3 on the stack to take its place (the number of characters in the abbreviation "wed").
	;	ends definition.

You now have the requisite arguments on the stack to display, move, or store the string "wed."

```
 3 dayofweek type wed ok
```

8.4 CHARACTER & STRING CONVERSIONS

Single characters are internally represented by the ASCII standard. (A chart listing ASCII character codes is in the ***SnapFORTH Reference Manual***. To convert a character to its ASCII equivalent, immediately precede the character with an ampersand ("&"). The ASCII value is left on the stack:

```
 &a . 97 ok
 &A . 65 ok
 &1 . 49 ok
```

Be sure not to use a space between the ampersand and the character. If you wish to use the ASCII value for a blank, use the constant "bl" ("blank"), which obviates the need for a separate ampersand:

```
 bl . 32 ok
```

Note for the advanced user: preceding the character with a caret ("^") instead of an ampersand gives you the corresponding control-character:

```
 ^a. 1 ok
 ^g. 7 ok
```

8.4.1 Strings to Numbers

A string may be converted to a double-number with the word "val." "Val" needs a string variable as a working area. You should add such a variable (we'll call it "scratch") to the TSA "strings" when you enter the definition of "val" into your source file:

```
 ( area strings ... )
   81 string scratch
 ( endarea ... )
```

```
: val ( addr len --> dn )
  scratch 2dup + over c! swap cmove
  scratch dconvert drop 0= if
    s>d then ;
```

Conversion operates in the current numeric base. It will stop when it encounters the first non-numeric character (exceps sign). Notice how the second example is altered because of the comma:

```
string" 123" alpha
alpha count val d. 123 ok

string" 123,45" beta
beta count val d. 123 ok

string" -12345678" gamma
gamma count val d. -12345678 ok
```

8.4.2 Numbers to Strings

Suppose you have to print the number in a left-justified field or with an imbedded decimal point? Here's how you can completely control the format of the number as you convert it to a string.

First, you should "dup" the number and put it in a safe place (in a variable or deeper down on the stack). Next convert the number to a double number if necessary (see chapter 7). Now use "dabs" to leave the unsigned double number on the stack. Finally, execute the word "<#" ("less sharp"). This word has no effect on the stack; it sets up a temporary area to store the characters you form as you convert the number for printing.

The word "#" ("sharp") converts one digit of the number in the current base and saves it in the temporary holding area. This digit is "removed" from the number. For example, suppose the double number 123 is on the stack. After the first "#", the character 3 would be added to the holding area and the number 12 would be left on the stack. The word "#S" ("sharp s") converts all of the remaining digits of a number to characters in the holding area, leaving a double-number zero on the stack.

The word "sign" adds the character - ("minus") to the holding area if the number on top of the stack is negative. The number is removed. Remember the original copy of the number kept in a safe place? Now is the time to push it on the stack, right over the double-number zero. "Sign" checks (and destroys) this number, leaving the double-number zero.

Finally, the word "#>" ("sharp greater") ends the conversion process. The double-number zero is dropped, and the starting address and current length of the holding area are left on the

stack. These two arguments look like a source string ("from" string) and can be used by any of the string functions. This string is stored in a volatile holding area and should be immediately "type"d or else saved in a string variable.

Here is the definition of a simple word ("string dollar") which converts a double number to a string:

```
: str$ ( dn --> addr len )
  swap over dabs <# #S rot sign #> ;
```

Try it out on some double numbers:

```
1234, str$ type 1234 ok

50000 str$ scratch count s! ok
scratch count type 50000 ok
```

At any time in the conversion process you can insert special characters into the holding area with the word "hold." For example, to insert a decimal point use " &. hold." Suppose you want to print a decimal point in the hundred's position:

```
: hundred ( by hundreds )
  ( requires unsigned double-number
    argument )
  <# # # &. hold #s #> ;
```

It would work like this:

```
45993, hundred type 459.93 ok
```

Perhaps you have some money in the bank and want to see the amounts in a readable form:

```
: money ( convert money )
  2dup dabs
  <# # # &. hold ( pennies )
  #s &$ hold
  2drop dup sign ( minus money? )
  #> ;

  12345, money type $123.45 ok
  -9999, money type -$99.99 ok
```

With a little more work, you could expand this to put a comma in the thousand's position, and so forth.

TABLE 8.1 STRINGS

WORD	STACK	ACTION
string	"cccc" <name> (—>addr)	Creates string constant <name>. When executed, the starting address of the beginning of the string memory area (the length byte) will be left on the stack.
string	<name> (n-max-length—>)	Creates a string variable with <name> and maximum length n-max-length.
<name>	(—>st-adr n-len)	When executed, the starting address of the beginning of the string memory area (the length byte) will be left on the stack.
type	(st-adr n-len—>)	Types the character sequence at starting address st-adr and current length n-len on the display.
s+	(addr1 len1 pos —>addr2 len2)	Forms a substring from the string specified by addr1 len1, starting at position pos. The new string has address addr2 (=addr1 + pos) and length len2 (=len1-pos).
s=	(addr1 len1 addr2 len2—>t/f)	Returns "true" if the first string (addr1 len1) equals the second (addr2 len2). Otherwise returns false.
<#		Prepares a temporary holding area to use for converting a number to a string of characters.
#	(ud1—>ud2)	Converts a digit to a character. The ud1 is then divided by "base" to produce ud2.
#s	(ud—>0 0)	Converts all remaining digits of ud until only a double-number zero is left on the stack.
sign	(n—>)	Adds the ASCII char "-" to the holding area if n is negative.
#>	(dn—>string)	Ends conversion. Drops dn, leaving the starting address and length of the character string in the holding area.

(These commands must be added to SnapFORTH. See the text.)

length	(str—>n-len)	Finds current string length.
s!	(addr1 len1 addr2 len2—>)	Stores source string (addr1 len1) into target string (addr2 len2).
val	(addr len—>dn)	Converts a string to a double number.
str$	(d-num—>str)	Converts a double number to a temporary string held in the string variable "scratch."

CHAPTER 9: USER INTERFACE

Consider the user, who will be sitting at the keyboard at some later date, trying to use the program you have written. Often you will need to communicate with him or her through the words in your programs. In addition, your programs may need instructions from the user. "How many pages should I print?" or "Which item are you looking for?" are typical questions a program might need to ask. This means the program must allow you to enter numbers and strings directly from the keyboard.

9.1 SINGLE-CHARACTER INPUT AND OUTPUT

The SnapFORTH command "key" accepts a character from the keyboard. When "key" is executed, SnapFORTH will patiently wait for you to type a character on your keyboard. As soon as you do, SnapFORTH will translate this key into its ASCII value and push it on the stack. For example, try typing

```
"key"
```

and ENTER. The cursor will wait patiently for the next key you press. Notice that you do not see the usual "ok." Now press "a" and ENTER. You should now see:

```
key ok
```

The "a" is not printed, but its ASCII value is now on the stack—check just to see:

```
.97 ok
```

You can now write a program that shows you the ASCII value for any key on your keyboard:

```
: seekey
  cr ." KEY?" key . ; ok

seekey
KEY?98 ok ( "b" )
seekey
KEY?49 ok ( "1" )
seekey
KEY?13 ok ( ENTER )
```

Since "key" does not display, you could use it for entering secret passwords. You can also use "key" to select among different action alternatives:

```
: ready? ( Are you ready for this? )
    cr ." READY (ENTER)?"
    key 13 = ; ( 13 = ASCII
      value of ENTER key )
```

```
ready? ( type "n" when asked )
READY (ENTER)? ok
. 0 ok
ready? ( press ENTER when asked )
READY (ENTER)? ok
. 1 ok
```

The word "ready?" leaves a "true" result on the stack if you press ENTER. If you hit any other key, it leaves a "false" result.

The word "emit" is opposite in action to "key." "Emit" removes an ASCII value from the stack and prints it:

```
 &a emit a ok
```

Emitting a character of your choice can be very useful:

```
 : bell ^g emit ;
```

"Bell" emits a "bell character" (control-g) which causes your HHC to "beep." This word is already defined in SnapFORTH, where it is called "beep." It can also be useful to be able to emit a tone of a pitch and duration of your choosing. To accomplish this, you need the word "squeak ," whose arguments are a numbered pitch from 0 to 36 (approximately by half tones) and numbered duration (about 200 per second). Try this:

```
 32 100 squeak ok
```

If you want, you can compose little tunes by stringing together "squeaks."

The word "key" can be effectively combined with "emit." Here's a simple word you can use to get single-digit numbers from the user:

```
 : get# ( ---- # )
     cr ." HOW MANY?" key dup
        emit &0 - ;

get# . ( press 3 when asked )
HOW MANY?3 ok
.3 ok
```

Check to see if you understand each element in the definition of "get#." "Key" accepts the next kets ASCII value on the stack. "Dup emit" displays the key and retains a copy of it on the stack at the same time. If the user types a "numbered" key (from 0 to 9), the ASCII value of the key will be converted to the actual number and left on the stack. This is done by the phrase "&0 -," which subtracts the ASCII value of 0 (48) from the key. For example, typing a 4 puts the ASCII value 52 on the stack; 52 48 - then yields 4.

9.2 INPUT STRINGS AND NUMBERS

Sometimes questions can't be answered with single characters. What if you want to get a user's name or a number larger than nine? You would want to allow the user to type a string of characters or numbers, perhaps backspacing to make corrections. When the user was finished, he or she would hit the ENTER key. SnapFORTH would then accept the entire string or number and continue execution.

The clue to making this operation successful is the word "get$." It will wait for and then accept a response from the user, leaving this response as a temporary string on the stack. Type the code for "get$" at this point (and put it in the same file that you used to load your string words from in chapter 8):

```
 : get$ ( ---> addr len )
     cr query 1 word here count ;
```

Next, define a word which will use "get$" to get the user's name—we'll call it "signature":

```
 : signature ( ---> addr len )
     cr ." SIGN IN PLEASE: " get$ ;
```

Now play user yourself with the new word. First type "signature" and ENTER. The window will now display the prompt, and the cursor will be waiting at the end for your answer.

```
 SIGN IN PLEASE:
```

Second, type your name:

```
 George Washington ok
```

Notice that as soon as you started to type your name, SnapFORTH erased the prompt and moved to a fresh line. What is on the stack now? The address and length of the temporary string with your name in it. With this address and length, you can store your name in another variable—let's use "scratch," which you still have from chapter 8.

```
 signature scratch count s! ok
```

Check to see if your name is actually there:

```
 scratch count type George
    Washington ok
```

With your knowledge of "get$" (and "val" from chapter 8), you can rewrite "get#" so as to accept a larger number from the user. "Val" will convert the string obtained by "get$" to a double number, which is then left on the stack. Try it this way:

```
 : get# ( ---> dn )
     cr ." HOW MANY? " get$ val ;
```

Now play user again and try out your word:

```
get#
HOW MANY?
33 ok
```

The word "get#" takes a single number from the user, converts it to a double number, and leaves it on the stack. Use a "d." to check what's on the stack:

```
d. 33 ok
```

9.3 FILE STORAGE

It is helpful when collecting data to have a system for organizing it and storing it all in one place. It can then be used by your programs. Let's suppose, for example, that you needed to set up a file of all your valuable household goods and their value for homeowner's or renter's insurance. You will first need to create a list of the goods themselves—with the help of the following commands.

Note: Before proceeding, decide which memory bank you want to store the file in, and make sure that's what you have by using the I/O key. The dictionary must be in the same bank as the file.

9.3.1 Creating a File

To create a file, you need the word "make." The address and length of a string containing the name of the file are necessary arguments for "make." The string can come from a string constant, a string variable, or perhaps even directly from the user via the word "get$" (see above). For now, we'll use a string constant to create the file "inventory":

```
string" inventory " inventory ok
inventory count make .1 ok
%text get-type c! ok
```

The phrase "%text get-type c!" informs SnapFORTH that we will be using this file to store text strings. The word "make" leaves either a true (1) or a false (0) on the stack. A 1 means that the file has successfully been created; a 0 means that there was not enough room for it. This flag is useful to check if anything went wrong.

9.3.2 Opening a File

The command "open" will make an existing file the "current file"; the file manipulation commands you are about to learn

will apply only to a current file. This command also requires address and current length as arguments; the name of the file, "inventory," supplies this information. Let's open "inventory":

```
inventory count open ok
```

At this point you have a true or false on the stack; true if the file has been successfully opened, and false if SnapFORTH could not find such a file.

9.3.3 Accessing a File

The contents of a file are stored in separate lines, or "records." Each record is numbered, and you must specify where you want your information to go by calling for this number. The way to get the number is by using a word called "rec-cnt"; it finds the number of lines currently in the file and adds 1 to that number, leaving it on the stack. (The 1 is necessary because lines are numbered beginning with zero.) If you ask for a "rec-cnt" on "inventory" now, you will get this:

```
inventory count rec-cnt .1 ok
```

You are now in a position to use the word "insert." This command will take your string and put it in the first record, which is record #0. Let's invent a string constant for one of your household goods, say your television, and add it to record #0:

```
string" television" tv ok
tv count rec-cnt insert .1 ok
```

The word "insert" leaves a 1 or a 0 on the stack—-1 if the insertion was successful, and 0 otherwise. Let's add a few more names to the file:

```
string" microwave" microwave ok
microwave count rec-cnt insert
  drop ok
string" painting" painting
painting count rec-cnt insert
  drop ok
string" stereo" stereo ok
stereo count rec-cnt insert
  drop ok
rec-cnt .4 ok
```

You can read any record of the file with the word "read." You must provide "read" with the address and length of a buffer which it can use to save the record. You can use the string variable "scratch," provided no record in the file is more than 80 characters long. We can assure SnapFORTH that there are 80 characters in "scratch" with this phrase:

```
scratch 1+ 80
```

Let's read the second record of inventory into scratch:

```
  scratch 1+ 80 2 read .1 ok
```

"Read" returns a "true" if successful and a "false" otherwise. Furthermore, if "read" succeeds, it will also leave the length of the record on the stack. You should save this length where it belongs—at the beginning of the string "scratch":

```
  scratch c! ok
  scratch count cr type
  painting ok
```

Here is a word which will list the contents of a text file. See if you can figure out how it works:

```
  : tlist
     rec-cnt 0 do
        cr i . space
        scratch 1+ 80 i read drop
           scratch c!
        scratch count type loop
           space ; ok

  tlist
  0 television
  1 microwave
  2 painting
  3 stereo ok
```

The command "space" is used to display a blank between the number and the record. "Space" is equivalent to "bl emit."

9.3.4 Altering a File

Now suppose that you want to add something to the file that will be between "microwave" and "painting." How would you do this, and what would happen to the information already in that position? The word "insert" can also take care of this for you. If you give as an argument the number of the record where you want the information to go, "insert" will move all the records of equal or higher number out of the way, renumbering them in the process, and make space for your record. Now you can set up your new string constant and insert it as record #2:

```
string" camera" camera ok
camera count 2 insert .1 ok
```

Check "painting" to see what its new record number is:

```
tlist
0 television
1 microwave
2 camera
3 painting
4 stereo ok
```

Maybe you don't have a painting anymore and want to replace its record with another possession—perhaps a dishwasher. For this you will need a new word, "write." Like "insert," this command also requires the string (address and length) you wish to write and the record number you wish to replace:

```
  string" dishwasher" dishwasher ok
  dishwasher count 3 write .1 ok

  tlist
  0 television
  1 microwave
  2 camera
  3 dishwasher
  4 stereo ok
```

The word "write" will leave a 1 or a 0 on the stack—1 if the operation was successful, and 0 if SnapFORTH could not find such a record number.

9.3.5 Deleting From a File

Maybe you don't have a dishwasher anymore either, but you have no possession in mind to replace it with. You would like to delete its record and move up the ones above it to fill its space, renumbering them in the process. In that case, use the word "delete." It only needs to know the record number:

```
  3 delete .1 ok

  tlist
  0 television
  1 microwave
  2 camera
  3 stereo ok
```

As with the other file manipulation commands, "delete" leaves either a 1 or a 0 on the stack, depending on whether it found a record of that number.

If you're tired of "inventory" now, let's get rid of it with "delete-file." As with all other file manipulation commands, "delete-file" will work only on a file that is already open, as "inventory" is now. It requires as its argument the address of the current file, which you can provide with the word "cfile" (the "c" stands for "current"):

```
  cfile delete-file .1 ok
```

Now if you go back to the file system you will see that "inventory" is no longer there.

Not all files contain text. Some, which are used just to store numeric data, are called "binary files." Full details on this type of file can be found in the ***SnapFORTH Reference Manual***.

9.4 GETTING IT TOGETHER

Now it's time to put together what you know about user interface and the file system. Just to show what you can do with this knowledge, why don't you design a word (called "getitem") which will ask the user for the names of particular items, accept as many as he or she enters until the word "stop" is typed, and will store each as a record in a file. Before starting, remember to recreate the file "inventory" which you just learned how to delete:

```
string" inventory" inventory
```

```
inventory count make drop
%text get-type c!
```

You also need to establish a string constant called "stop."

```
string" stop " stop
```

Now you're ready to write the definition for "getitem":

```
: getitem
  inventory count open drop
  begin cr ." ITEM:" get$
    2dup stop count s= 0=
  while rec-cnt insert drop
  repeat 2drop ;
```

You're ready now to play user again and see what your new word does. When you ask for "getitem," here's what you'll see:

```
getitem
ITEM:
```

Enter your first item and continue to make a list as SnapFORTH prompts you until you are ready to say "stop":

```
ships
ITEM:
sealing wax
ITEM:
cabbages
ITEM:
kings
ITEM:
stop ok
```

Now ask for a "tlist" to see what's in "inventory":

```
tlist
0 ships
1 sealing wax
2 cabbages
3 kings ok
```

CHAPTER 10: ADVANCED TOPICS

In this final chapter, we will touch on some of the more powerful but complex aspects of the SnapFORTH language. To treat each of these in detail would require an additional volume as large as this one. If you are interested in continuing your study of SnapFORTH, we urge you to read carefully the ***SnapFORTH Reference Manual.***

10.1 EXCEPTIONAL CONDITIONS

You can terminate the execution of a word at any time with the command "exit." On a somewhat larger scale, you can terminate the execution of all words and return to the execution (keyboard entry) mode with the command "clear." "Clear" clears the return stack, changes to execution mode (see below), and returns control to SnapFORTH.

The command "abort," like "clear," clears the return stack, changes to execution mode, and returns control to SnapFORTH. "Clear" is normally used to terminate execution of a program when a task is completed (or determined to be unnecessary). "Abort" is normally used to terminate execution of a program which has encountered a serious error.

10.2 CREATE AND DOES>

One of SnapFORTH's most powerful features is the ability to create new classes of objects with the commands "create" and "does>." These objects can be viewed as intelligent data structures, or perhaps as words with data storage. The command "create" specifies what actions will be taken when an object of a given class is created. The command "does>" specifies what actions will be taken when an object of a given class is executed. You can find a detailed discussion of "create" and "does>" in "Forth extensibility: Or how to write a compiler in twenty-five words or less" by Kim Harris in ***BYTE*** (August 1980, pp 164-184).

Consider the definition of "cvector" which was used in chapter 3 to create vectors of bytes:

```
: cvector ( create objects of class
             cvector )
    create allot
    does> + ; ok
```

Using 1000 as an example of an address for the first byte, you would get:

```
 10 cvector items ok

 0 items .1000 ok
 3 items .1003 ok
```

The word "create" in the definition of "cvector" creates objects of class "cvector." Whenever the word "cvector" appears, a new word is created in the dictionary whose name is the word which follows "cvector." In this case, the word is named "items." The words which follow "create" are then executed. In this case, the single word "allot" is executed. "Allot" takes the number currently on the stack (10) and allocates that many bytes of storage at the end of the dictionary. So "10 cvector items" puts a ten on the stack and creates the word "items." The "allot" then executes and creates storage space for 10 bytes.

The command "does>" ensures that when any word created by the preceding "create" is executed, the address of the parameter field of that word will be pushed on the stack and control will transfer to the SnapFORTH words following the "does>." In the above example, "3 items" will first push a three and then the parameter field address of "items" on the stack. Control then transfers to the + operator, which adds the two arguments. The result is the address of the third byte in the byte vector "items."

10.3 EXECUTION AND COMPILATION

When you are typing commands from the keyboard (or LOADing them from files), SnapFORTH executes each word as soon as you hit the return key. This is called the "execution" mode. If you enter a definition by typing a ":" ("colon"), SnapFORTH will compile the words you type until you finish the definition with a ";" ("semicolon"). This is called "compilation" mode. The words within the definition are not executed until you type the name of the definition (in execution mode). In fact, the ":" changes the mode from execution to compilation and the ";" changes it back again.

A third class of words in SnapFORTH is called "compiler words." These words are executed instead of compiled when used within a definition (that is, in compilation mode). SnapFORTH uses these words to control the compilation process itself. Words like "if," "begin," "do," and so forth are examples of compiler words. You can turn any word into a compiler word by following its definition with the command "immediate."

FRIENDS AMIS, INC.

An important note: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or programs are satisfactory.

USA

Panasonic Company
Division of Matsushita Electric Corporation of America
One Panasonic Way,
Secaucus, New Jersey 07094

Panasonic Hawaii Inc.
91-238 Kauhi St. Ewa Beach
P.O. Box 774
Honolulu, Hawaii 96808-0774

Panasonic Sales Company
Division of Matsushita Electric of Puerto Rico, Inc.
Ave. 65 De Infanteria, KM 9.7
Victoria Industrial Park
Carolina, Puerto Rico 00630

CANADA

Panasonic Canada
Division of Matsushita Electric of Canada Limited
5770 Ambler Drive, Mississaugo,
Ontario L4W2T3

OTHERS

Matsushita Electric Trading Co., Ltd.
32nd floor, World Trade Center Bldg.,
No. 4-1, Hamamatsu-Cho 2-Chome,
Minato-Ku, Tokyo 105, Japan
Tokyo Branch P.O. Box 18 Trade Center