FORTH PROGRAMMING

LEO J. SCANLON

BLACKSBURG CONTINUING EDUCATION SERIES™
edited by Larsen, Titus & Titus

The Blacksburg Continuing Education™ Series

The Blacksburg Continuing Education Series™ of books provide a Laboratory—or experiment-oriented approach to electronic topics. Present and forthcoming titles in this series include:

- Advanced 6502 Interfacing
- Analog Instrumentation Fundamentals
- Apple II Assembly Language
- Apple Interfacing
- Basic Business Software
- Basic Robotics Concepts
- BASIC Programmer's Notebook
- Circuit Design Programs for the Apple II
- Circuit Design Programs for the TRS-80
- Computer Assisted Home Energy Management
- Computer Communication Techniques
- Design of Active Filters, With Experiments
- Design of Op-Amp Circuits, With Experiments
- Design of Phase-Locked Loop Circuits, With Experiments
- Design of VMOS Circuits, With Experiments
- 8080/8085 Software Design (2 Volumes)
- 8085A Cookbook
- Electronic Music Circuits
- Fiber Optics Communications, Experiments, and Projects
- 555 Timer Applications Sourcebook, With Experiments
- FORTH Programming
- Guide to CMOS Basics, Circuits, & Experiments
- How to Program and Interface the 6800
- Introduction to Electronic Speech Synthesis
- Introduction to FORTH
- Microcomputer—Analog Converter Software and Hardware Interfacing
- Microcomputer Data-Base Management
- Microcomputer Design and Maintenance
- Microcomputer Interfacing With the 8255 PPI Chip
- NCR Basic Electronics Course, With Experiments
- NCR EDP Concepts Course
- PET Interfacing
- Programming and Interfacing the 6502, With Experiments
- Real Time Control With the TRS-80
- 16-Bit Microprocessors
- 6502 Software Design
- 6801, 68701, and 6803 Microcomputer Programming and Interfacing
- The 68000: Principles and Programming
- 6809 Microcomputer Programming & Interfacing, With Experiments
- STD Bus Interfacing
- TEA: An 8080/8085 Co-Resident Editor/Assembler
- TRS-80 Assembly Language Made Simple
- TRS-80 Color Computer Interfacing, With Experiments
- TRS-80 Interfacing (2 Volumes)
- TRS-80 More Than BASIC
- Wordprocessing for Small Businesses

In most cases, these books provide both text material and experiments, which permit one to demonstrate and explore the concepts that are covered in the book. These books remain among the very few that provide step-by-step instructions concerning how to learn basic electronic concepts, wire actual circuits, test microcomputer interfaces, and program computers based on popular microprocessor chips. We have found that the books are very useful to the electronic novice who desires to join the "electronics revolution," with minimum time and effort.

Jonathan A. Titus, Christopher A. Titus, and David G. Larsen
"The Blacksburg Group"

FIRST EDITION
SECOND PRINTING—1983

International Standard Book Number: 0-672-22007-5
Library of Congress Catalog Card Number: 82-060875

Edited by: *C. Herbert Feltner*
Illustrated by: *David K. Cripe*

Printed in the United States of America.

Preface

Ever since the advent of FORTRAN in the 1950s, so many high-level languages have been introduced that few people can keep track of them. Some of these languages generated initial excitement, then slowly faded from the scene. Other languages were well conceived, but were so difficult to use that they, too, disappeared. Still other languages attracted a dedicated group of advocates, but were too specialized to gain general acceptance. However, every so often a new language has features which so appeal to the programming public that the language is accepted. FORTH* is such a language.

The most attractive feature of FORTH is that it can be *extended.* That is, if you need to perform some function that is not already included in the language, you can add it! FORTH consists of a set of predefined commands, called *words.* Each word performs one specific task, such as adding two numbers or storing a number in memory. If you wish to perform more than one of these functions, you can do so by having the computer execute the appropriate sequence of words. For example, to add two numbers and store the sum in memory, you would execute the "add" word, then the "store" word.

If your application calls for many "add-then-store" sequences, you can define this two-word sequence as a new, single word, and add it to FORTH's word set, its *dictionary.* Thereafter, any time you need to do an add-then-store, you simply use the new word from the dictionary. In this way, FORTH programming involves defining word after word, with each new word at a higher level than the words used to construct it.

This building-block approach offers several advantages. First, since new words are always constructed from previously defined, error-free, older words, FORTH programs (that is, *words;* they are equivalent) are inherently easy to debug. In

*FORTH is a trademark of FORTH, Inc., 2309 Pacific Coast Highway, Hermosa Beach, CA 90254.

most cases, debugging a new word is simply a matter of determining whether the word operates as expected, not whether its construction is valid. Most word definitions are short, too, since you are building with lower-level words, which also makes troubleshooting easier.

Moreover, the modularity of FORTH words allows them to be used in a variety of programs. This means that you only need to program a given function *once*. Thereafter, it becomes part of a permanent "toolbox" in your system.

Ease of programming and debugging ensures that even novice FORTH programmers can develop software quickly and efficiently. Besides this cost-effectivity, system developers are also impressed with the compactness of FORTH-based software in memory and the fact that programs execute faster than with most other high-level languages—*often at speeds approaching that of assembly language!*

This book describes FORTH "from the ground up." It starts with the most fundamental concepts and gradually introduces more complex topics, thereby helping you learn the language in an orderly manner. If you have already done some FORTH programming, using any of the standard software packages, this book will clarify many points that may not be fully explained (or perhaps not even mentioned) in the manufacturer's documentation.

Most FORTH packages are based on either of two popular "dialects," FORTH-79 or fig-FORTH. This book describes *both* dialects, and identifies programming differences where they exist. Therefore, users of virtually any FORTH package will find material directly applicable to their system.

This book has 13 chapters. Chapter 1 introduces the basic concepts of FORTH and shows you how to perform some simple operations. Chapter 2 discusses addition, subtraction, multiplication, and division.

In Chapter 3 you are shown how to manipulate the *stack,* the memory structure on which most FORTH operations take place. Data transfers between the stack and other portions of memory are described in Chapter 4.

Chapter 5 introduces a capability that makes FORTH different from most other high-level languages—adding new operations (words) to the language. Once added, these new words become as much a part of FORTH as the words provided in the original software package.

Chapters 6 and 7 describe the control structures DO-LOOP, BEGIN-UNTIL, and IF-ELSE-THEN; structures that alter the flow of the program based on pre-established conditions.

Constants, variables, arrays, and tables are discussed in Chapter 8. Chapter 9 tells how to switch from one numbering system to another for input and output.

Chapter 10 discusses how to *interact* with the FORTH system while it is running. That is, it tells you how to make the computer wait for information from the terminal or print information in a form you have specified.

Chapter 11 explains how to process strings of text in memory; Chapter 12 continues with a related topic—how to transfer strings, tables, and other "nonprograms" to and from the disk.

Finally, Chapter 13 discusses logical, shift, and rotate operations, which are essential if you plan to manipulate the binary patterns in which data is stored in memory.

This book also has three appendices, for quick reference. Appendix A provides hexadecimal/decimal conversion tables and a listing of the ASCII character codes; Appendix B summarizes the standard word sets for both FORTH-79 and fig-FORTH; and Appendix C describes FORTH-79's Double Number Extension Word Set and defines those words, so you can add them to your system, if desired.

May you have as much satisfaction developing FORTH software as I have had writing this book.

LEO J. SCANLON

ACKNOWLEDGMENTS

I would like to express thanks to Mr. Eric Rehnke of Rehnke Software (Corona, CA), Mr. Bill Exberger of E-Systems/ECI Division (St. Petersburg, FL), and Mr. Wayne Witt of Interactive Computer Systems, Inc. (Tampa, FL), for their support of this project.

Contents

CHAPTER 11

CHAPTER 12

CHAPTER 13

APPENDIX A

APPENDIX B

APPENDIX C

CHAPTER 1

Introduction to FORTH

FORTH was developed by one man, Charles H. Moore (Fig. 1-1), over a period of time starting in the early 1960s.[1] Working on a variety of programs for such diverse applications as satellite orbits, chromatography and business systems, Moore felt hindered by the amount of time it took to develop programs in FORTRAN, ALGOL, and other languages of the day, and decided to invent a tool to help increase his productivity. Through

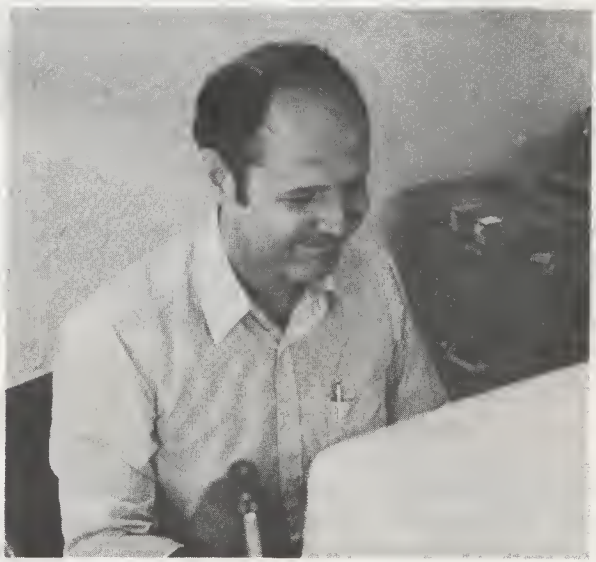


Fig. 1-1. Charles H. Moore, Inventor of FORTH. *(Courtesy FORTH, Inc.)*

the years he added one element after another, as the need arose, until finally, in 1968, he programmed an entity known as FORTH on an IBM 1130 computer. (In those days of "third-generation" computers, Moore saw his accomplishment as a *fourth*-generation language, but labeled it FORTH, rather than FOURTH, because the IBM 1130 permitted only five-character identifiers!)

Three years later, while writing a radio-telescope data acquisition program for the National Radio Astronomy Observatory, Moore added a compiler to the system, followed in 1973 by multiprogramming capability. To this day, Charles Moore's FORTH system runs a radio telescope at the NRAO station at Kitt Peak, Arizona. In great demand by observers, this instrument is responsible for discovering half of the interstellar modules ("space dust") that are known to exist.

Spurred by interest from astronomers, Moore and a few other FORTH enthusiasts left NRAO in 1973 to form FORTH Inc. Although initially dedicated to astronomical applications, the company has since diversified into general-purpose commercial FORTH systems.

Beginning with astronomers, FORTH was quickly "discovered" by individuals and groups around the world. Finally, in 1976, the European FORTH Users' Group (EFUG) was formed. Out of their first meetings grew an international FORTH Standards Team, who drafted a formal specification describing a set of FORTH commands (called *words*) that were to be included in every system, for compatibility. In 1978, the FORTH Interest Group (FIG) was founded by FORTH programmers to encourage use of the language by interchanging ideas through seminars, conventions, and publications. In 1982, FIG had over 2400 members worldwide. Membership in FIG also includes a subscription to its excellent bimonthly magazine, *FORTH DIMENSIONS*. For information contact: FORTH Interest Group, P.O. Box 1105, San Carlos, CA 94070.

FORTH-79 AND fig-FORTH

The FORTH Standards Team is still in existence, and meets periodically to review the Standard. This book describes the 1979 version of the Standard (the most recent version as of this writing), which was released in October 1980. The contents of this *FORTH-79 Standard* are discussed in more detail in the next section.

FORTH-79 is one of two primary "dialects" of FORTH. The other, fig-FORTH, was developed by the FORTH Interest Group. Its "standard" is the *fig-FORTH Installation Manual,* written by William F. Ragsdale. (This manual and the FORTH-79 Standard are both available from the FORTH Interest Group.) There are a number of differences between FORTH-79 and fig-FORTH, both in content and how certain words operate. These differences are identified in the course of the book.

OVERVIEW OF FORTH PROGRAMMING

Every computer language has its own term to describe what you must type in to make things happen. In assembly language, it's called an "instruction." In BASIC and Pascal, it's called a "statement." In FORTH, it's called a "word."

It's All a Matter of Words

A *word* is a sequence of one or more characters that identifies an execution procedure. That is, it is a label for what should happen when the word is executed. Every FORTH system has a built-in set of words. Each of these words makes one specific thing happen.

For example, the FORTH word + causes two numbers in memory to be added, and the sum returned to memory. Another word, TYPE, causes a string of characters in memory to be printed or displayed.

If you want to do something more complex than a single word will provide, you can write several words consecutively to form a *program.* That's really all there is to FORTH programming: combining the words at your disposal in a way that will do something you want done.

The sum total of all words in a FORTH system is contained in what is called, appropriately enough, its *dictionary.* When you first buy a FORTH system, the dictionary contains only the words that have been provided by the manufacturer. If the package you bought conforms to the FORTH-79 Standard, it will contain the FORTH-79 Required Word Set (description upcoming). Otherwise it will contain a fig-FORTH word set, or a polyFORTH word set, or the word set of some other FORTH "dialect."

In any case, *the dictionary is not limited to the words that are built into the system you have purchased.* FORTH, by its nature,

Chart 1-1. Required Word Set

Nucleus Words

! * */ */MOD + +! +loop – / /MOD 0< 0= 0> 1+ 1– 2+ 2– < = > >R ?DUP @ ABS AND begin C! C @ colon CMOVE constant create D+ D< DEPTH DNEGATE do does> DROP DUP else EXECUTE EXIT FILL I if J LEAVE literal loop MAX MIN MOD MOVE NEGATE NOT OR OVER PICK R> R@ repeat ROLL ROT semicolon SWAP then U* U/ U< until variable while XOR

Note: Lower-case entries refer only to the run-time code corresponding to a compiling word.

Interpreter Words

#> #S ' (–TRAILING . 79-STANDARD <# >IN ? ABORT BASE BLK CONTEXT COUNT CR CURRENT DECIMAL EMIT EXPECT FIND FORTH HERE HOLD KEY PAD QUERY QUIT SIGN SPACE SPACES TYPE U. WORD

Compiler Words

+LOOP , ." : ; ALLOT BEGIN COMPILE CONSTANT CREATE DEFINITIONS DO DOES ELSE FORGET IF IMMEDIATE LITERAL LOOP REPEAT STATE THEN UNTIL VARIABLE VOCABULARY WHILE [[COMPILE]]

Device Words

BLOCK BUFFER EMPTY-BUFFERS LIST LOAD SAVE-BUFFERS SCR UPDATE

quired Word Set operate on *numbers*, values which can be contained in 16 bits of a computer's memory. Because 16 bits can represent values no larger than 65,535, the FORTH Standards Team defined a set of words that could operate on *double numbers,* values that occupy 32 bits in memory. These words, which comprise the *Double Number Word Set,* are listed in Chart 1-2. Each word will be described in following chapters.

The *Assembler Word Set* allows you to include assembly language programs within a FORTH program, to perform tasks that cannot be done efficiently in FORTH. As you can see in Chart 1-2, the Assembler Word Set consists of only four words, three that delineate the assembly language code and one that calls the assembler. Of course, the actual assembly language instructions will vary from system to system (depending on which microprocessor is involved), as will the details of the assembler, so these four words are all that are needed in the Standard.

Chart 1-2. Extension Word Sets

Double Number Word Set

2! 2@ 2CONSTANT 2DROP 2DUP 2OVER 2ROT 2SWAP 2VARIABLE
D+ D− D. D.R D0= D< D= DABS DMAX DMIN DNEGATE DU<

Assembler Word Set

;CODE ASSEMBLER CODE END-CODE

The Reference Word Set

Within the FORTH-79 Standard, the FORTH Standards Team included a group of words to be used strictly for reference. Some of the words in this *Reference Word Set* are words that appeared in earlier versions of the Standard and have approved Standard Word Definitions. Others have "uncontrolled" definitions; these have widespread usage among software vendors and/or are candidates for future standardization. Both types of Reference Words are listed in Chart 1-3.

This book contains descriptions of many Extension Words and Reference Words, as well as Required Words. To avoid confusion, Extension Words and Reference Words will be so identified.

What You VLIST Is What You Get

If you ever want to see a list of all the words available in your particular FORTH system, simply type in the word

VLIST

Chart 1-3. Reference Word Set

Standard Word Definitions

--> AGAIN BL BLANKS DUMP EDITOR END ERASE HEX OFFSET SP@ U.R

Uncontrolled Word Definitions

!BITS ** +BLOCK −' −MATCH −TEXT .R /LOOP 1+! 1−! 2* 2/ ;: ;S <> <BUILDS <CMOVE >MOVE< @BITS ABORT" AGAIN ASCII ASHIFT B/BUF BELL C, CHAIN COM CONTINUED CUR DBLOCK DPL FLD FLUSH H. I' IFEND IFTRUE INDEX INTERPRET K LAST LINE LINELOAD LOADS MAP0 MASK MS NAND NOR LUMBER O. OCTAL OTHERWISE PAGE READ-MAP REMEMBER REWIND ROTATE S0 SET SHIFT TEXT THRU USER VLIST WHERE \ LOOP

and press the Return key on your computer's keyboard. This will cause FORTH to list the words in its dictionary, one by one, until the dictionary has been exhausted or you (exhausted by the listing) press ESC, Break, or Reset.

If you have added any words of your own to the dictionary, these, too, will be included in the VLIST. In fact, the listing will start with the most recently defined word, if any, and work down from there.

STACKS

To do any programming in FORTH, you must know what a *stack* is. Why? Because virtually all FORTH operations involve a stack in some way or other. For instance, before adding two numbers, both must be on the stack—and the result is returned on the stack. The same applies to subtraction, multiplication, division, printing numbers, and just about everything else. In FORTH, the stack is where the action is!

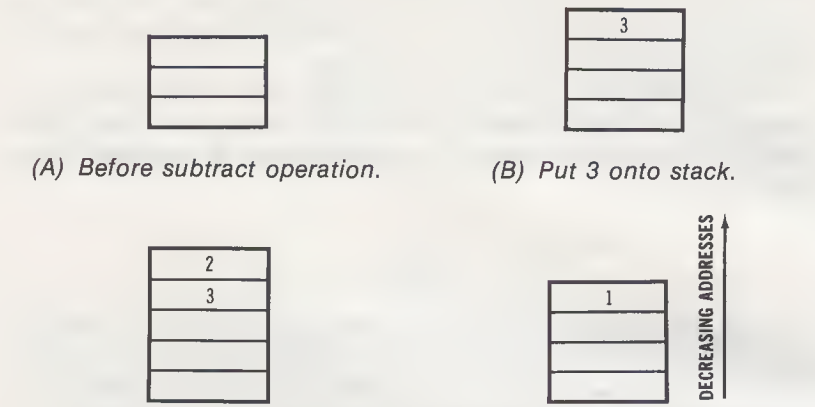
A stack is not some mystical piece of hardware, but is just an area of the computer's memory in which numbers are kept temporarily. In this area of memory, numbers are stacked on each other just as plates are stacked in a kitchen. That is, the first number goes on the bottom of the stack, and each new number is deposited directly above (or on top of) the previous number. And like a stack of plates, the last number to be placed on the stack will be the first number to be removed *from* the stack.

This type of stack is usually categorized as a "last in, first out" (or LIFO) stack. The last number in is always the first number out.

Fig. 1-2 illustrates how a simple subtract operation (3−2) affects the stack. In Fig. 1-2A, three numbers are on the stack, but we don't care what those numbers are. In Figs. 1-2B and 1-2C, the two operands are "pushed" onto the stack. In Fig. 1-2D, following the subtract operation, the operands have disappeared and just the result remains on the stack. Note that *the stack "builds" in the direction of low memory;* each number is pushed onto the stack at a lower address than the preceding number.

Pushing Numbers Onto the Stack

How do you put numbers onto the stack in FORTH? Does it require some exotic command such as PUT 3 ONSTACK? No,



(A) Before subtract operation.

(B) Put 3 onto stack.

(C) Put 2 onto stack.

(D) Subtract. Result is left on stack.

Fig. 1-2. How a subtract operation affects the stack.

nothing as complicated as that. Since FORTH is a stack-oriented language, nearly every operation implies "do *this* to the stack." Therefore, *to push a number onto the stack, simply type in the number and press the Return key* (or, on some computers, the Enter key).

For example, if you type in the digits

```
123
```

and press Return, the decimal value 123 will be pushed onto the top of the stack. If more than one number is to be pushed onto the stack, you can simply type them all on one line (separated by at least one space) and press Return when the final number has been entered. For example, if you type in

```
123 456 789
```

and press Return, three decimal values will be pushed onto the stack: 123, followed by 456, followed by 789. The value 789 will be on top of the stack, because it was specified last.

Displaying Stack Values

How can you find out what values are on the stack at any given time? FORTH provides a word that displays the top number on the stack, by writing it out to your system's active

output device (screen and/or printer). This word is, quite simply, a "dot" [.]; that is, a "period" on your keyboard.

To use an earlier example, if you type in the digits

```
123
```

and press Return, the decimal value 123 will be pushed onto the top of the stack. If instead you type

```
123 .
```

and press Return, the value 123 will go onto the stack, but then will be immediately pulled off and displayed. The display will be in this format:

```
123 OK
```

It is important to remember, though, that *anything displayed with a "dot" is no longer on the stack.* There are ways to display stack values nondestructively, and we'll treat these later in the book.

The dot can also be used to display several values on the stack, by repeating the dot word. For example, if you type in

```
123 456 789 . .  .
```

and press Return, you will receive this display:

```
789 456 OK
```

What is the top value on the stack after this sequence? It is the number 123, because the numbers 456 and 789 were pulled off the stack by the two display operations.

At this point, the stack holds just one number, 123. You can enter one more dot to display that number, but what happens if you enter *two* dots? If you enter two dots:

```
. .
```

and press Return, you will have attempted to display a nonexistent value, so FORTH will print an error message such as

```
? STACK EMPTY
```

REVERSE POLISH NOTATION

In most computer languages, such as BASIC, the preceding subtract operation would be written in the form

```
3 - 2
```

just as it is written using pencil and paper. However, because FORTH always operates on numbers that are already on the stack, it is more efficient to place the operator *after* the operands, rather than between them. Therefore, our subtract operation would appear like this:

```
3 2 -
```

in a FORTH program!

Note that we've actually shown three separate operations here:

1. The number "3" pushes the value 3 onto the stack.
2. The number "2" pushes the value 2 onto the stack, on top of the value 3.
3. The word [−] subtracts the top item on the stack (the value 2, in this case) from the second item on the stack (the value 3), and leaves only the result (the value 1) on the stack.

This peculiar operator-last notation is called *reverse Polish notation* (RPN). This notation is commonly used in stack-oriented machines. Hewlett-Packard's line of programmable calculators are stack-oriented, for example, and require operations to be typed in using reverse Polish notation.

Although RPN is bound to feel somewhat "funny" after a lifetime of exposure to standard operand-operator-operand notation, you should begin to feel comfortable with it after running just a few FORTH programs, You must only remember that *something* must be on the stack before you apply an operator and that the operand(s) must be in the intended order. That is, the top number is the number that FORTH will add to, subtract from, divide into or multiply by the second number.

HOW FORTH WORDS ARE DESCRIBED IN THIS BOOK

In the remaining chapters we will describe each of the words defined in the FORTH-79 Standard. These include Required Words, Extension Words, and Reference Words (both Standard and uncontrolled).

To help you understand how these words *relate* to each other, words that have similar functions will be described together, in groups. That is, those that perform arithmetic operations (such as add, subtract, multiply, and divide) will be described in one group, those that perform memory operations

(such as store, fetch, and block move) will be described in another group, and so on.

Each of the functional group descriptions will be accompanied by a table that summarizes the words in that group. This table will list the syntax of the word, what effect the word has on the stack and the action taken by the word. Further, for non-Required words, the table will include a note that tells whether the word is defined in an Extension Word Set, or is a Standard or uncontrolled word defined in the Reference Word Set.

For example, the table entry for the uncontrolled word 2* will look like this:

WORD	STACK	ACTION	NOTES
2*	n---2*n	Multiplies number by 2.	(1)

The WORD column gives the syntax of the word, the STACK column shows that some number *n* on the top of the stack will be replaced by the value *2*n* after the word is executed (the three dashes, "---", are the normal way "before" and "after" values are separated in FORTH literature), the ACTION column tells what the word does, in plain English, and the NOTES column is keyed to a note at the end of the table. In this case, the appropriate note for the word 2* will read:

(1) Included in Reference Word Set as an uncontrolled word definition.

REFERENCE

1. Moore, Charles H. "The Evolution of FORTH, an Unusual Language." *BYTE*, August 1980, pp. 76-92. This article provides a fascinating insight into the development of FORTH, by its inventor.

CHAPTER 2

Arithmetic Operations

In addition to the four standard arithmetic operations—add, subtract, multiply, and divide—FORTH provides words that provide such handy tasks as finding the larger or smaller of two numbers, or determining an absolute value. Since many of these operations are available for both single-precision numbers and double-precision numbers, we should begin by describing these two types of numbers.

NUMBERS AND DOUBLE NUMBERS

In FORTH, the basic unit of data is called a *number*. A number can be any integer value between −32,768 and 32,767. Readers who have been involved with the technicalities of computers will recognize these limits as the range of values that can be represented in 16 "bits" (binary digits) of a computer's memory. Of these 16 bits, the lowest 15 bits hold binary data and the highest bit holds a sign indicator, usually 0 = positive and 1 = negative. (Don't worry about this for now. We'll discuss bits in more detail in Chapter 9.)

If we are dealing with absolute quantities, such as memory addresses, the highest bit is also a data bit, so all 16 bits represent data. Now our 16 bits hold *unsigned numbers*, which can have values from 0 to 65,535.

FORTH can also operate on *double numbers*, those that are contained in 32 bits of memory. Signed double numbers range from −2,147,483,648 to 2,147,483,647 and *unsigned double numbers* range from 0 to 4,294,967,295.

Pushing Numbers Onto the Stack

You will recall from Chapter 1 that to push a number onto the stack, you simply type it in and press Return. Unsigned numbers are pushed onto the stack in the same way, but *double numbers must contain a decimal point*, to distinguish them from numbers.

The decimal point may precede the double number, follow it, or fall anywhere within it. As long as the decimal point is there, FORTH will interpret the value as a double number and push that double number onto the stack. For instance, [.600000], [600000.] and [60.000] all cause the same value, 600000, to be pushed onto the stack.

Displaying Numbers

In Chapter 1 you learned that a number at the top of the stack is pulled off and displayed by using the word [.]. Unsigned numbers and double numbers also have their own special display words; they are [U.] and [D.], respectively. Here are some examples:

```
40000 U. 40000 OK          (Unsigned number)
.600000 D. 600000 OK       (Double number)
600000. D. 600000 OK       (Double number)
-30. D. -30 OK             (Double number)
```

Note that the last example number (−30) is specified as a double number (by the presence of the decimal point), although it certainly could have been represented as just a (16-bit) number. This example was included to show that "number-size" values may, at times, have to be specified as double numbers, if they are to be used in an operation where a double number operand is required.

How Numbers Are Stored in Memory

Most of the popular microcomputers are designed around an 8-bit microprocessor, which means that their basic unit of data is eight binary bits, or one *byte*. This means that a FORTH number, which occupies 16 bits, must be stored in two consecutive bytes in memory—or on two consecutive bytes on the stack.

The arrangement of these two bytes will depend on how your FORTH system is implemented. In some systems, the higher

eight bits will be at the lower-addressed location; in other systems, the higher eight bits will be at the higher-addressed location.

But unless you're examining the contents of memory byte by byte, the arrangement of individual bytes is unimportant. In fact, the FORTH-79 Standard totally avoids defining the order of bytes within a 16-bit number. However, in a *double number* the Standard states that the higher 16 bits (with sign) must be stored starting at the lower address in memory, and must be more accessible on the stack.

Fig. 2-1 shows a typical arrangement of numbers and double numbers on the stack or in memory.

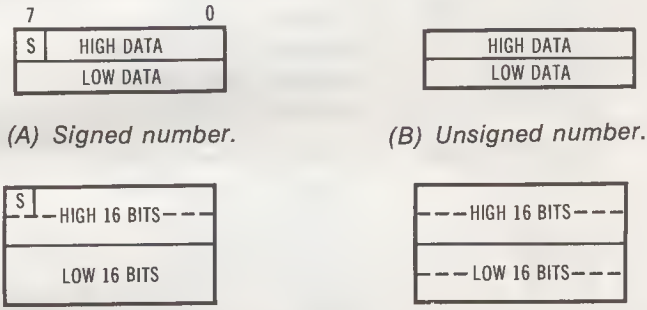


Fig. 2-1. Typical values in memory.

THE ARITHMETIC WORD GROUP

Table 2-1 summarizes the FORTH words that perform arithmetic operations, subdivided by function. At the end of this chapter we will present the equivalent table for fig-FORTH.

ADDITION

The fundamental addition word is the "plus" [+] operator, which adds two numbers at the top of the stack and leaves

Table 2-1. Arithmetic Words

Word	Stack	Action	Notes
.	n ---	Prints number on top of stack.	
D.	d ---	Prints double number on top of stack.	(1)
U.	un ---	Prints unsigned number on top of stack.	
+	n1 n2 --- sum	Adds two numbers.	
1+	n --- n+1	Adds 1 to number.	
2+	n --- n+2	Adds 2 to number.	
D+	d1 d2 --- sum	Adds two double numbers.	
−	n1 n2 --- diff	Subtracts n2 from n1.	
1−	n ---n−1	Subtracts 1 from number.	
2−	n ---n−2	Subtracts 2 from number.	
D−	d1 d2 --- diff	Subtracts d2 from d1.	(1)
*	n1 n2 --- prod	Multiplies signed numbers n1 and n2, leaving single-precision product.	
U*	un1 un2 --- ud	Multiplies unsigned numbers un1 and un2, leaving double-precision product.	
2*	n --- 2*n	Multiplies number by 2.	(2)
/	n1 n2 --- quot	Divides n1 by n2, leaving quotient.	
MOD	n1 n2 --- rem	Divides n1 by n2, leaving remainder with same sign as n1.	
/MOD	n1 n2 --- rem quot	Divides n1 by n2, leaving remainder and quotient.	
U/MOD	ud un --- rem quot	Divides double dividend by single divisor, leaving single remainder and quotient, all unsigned.	
2/	n --- n/2	Divides number by 2.	(2)
*/	n1 n2 n3 --- quot	Multiplies n1 by n2, then divides the result (a 32-bit intermediate product) by n3, leaving single quotient.	
*/MOD	n1 n2 n3 --- rem quot	Same as [*/], but leaves both remainder and quotient.	

Table 2-1—cont. Arithmetic Words

Word	Stack	Action	Notes
**	n1 n2 --- n1**n2	Leaves the value of n1 raised to the power n2.	(2)
NEGATE	n --- −n	Reverses the sense of a number.	
DNEGATE	d --- −d	Reverses the sense of a double number.	
ABS	n ---|n|	Leaves the absolute value of a number.	
DABS	d ---|d|	Leaves the absolute value of a double number.	(1)
MAX	n1 n2 --- max	Leaves the greater of two numbers.	
MIN	n1 n2 --- min	Leaves the lesser of two numbers.	
DMAX	d1 d2 --- dmax	Leaves the greater of two double numbers.	(1)
DMIN	d1 d2 --- dmin	Leaves the lesser of two double numbers.	(1)

Notes: (1) Included in Double Number Extension Word Set.
(2) Included in Reference Word Set, as an uncontrolled word definition.

their sum. (In FORTH nomenclature, the word "leaves" implies "leaves on the top of the stack.")

The sequence

```
3 2 +
```

followed by Return, leaves the number 5. If you wish to use your FORTH computer as a calculator, you can follow the "plus" [+] with a "dot" [.], and have the result printed, but not saved. Therefore, the key sequence

```
3 2 + . RETURN
```

will give this display:

```
3 2 + . 5 OK
```

Although the Return key was shown above, for clarity, it will not be shown throughout the remainder of this book. Instead, regular type will be used to denote operator input and *italics* will be used to denote computer-generated display (or print) information; a "Return" is assumed to separate the two.

Remember, numbers can be negative as well as positive, so an operation such as

```
3 -2 + . 1 OK
```

will work, too!

Adding a Column of Numbers

Being free-form, FORTH does not limit you to a single operation on a line. If you wish to add a column of numbers and print the final result, your line will look something like this:

```
32 64 + 56 + 96 + 124 + . 372 OK
```

Alternatively, you can group all of the "pluses" at the end, and use this variation:

```
32 64 56 96 124 + + + + . 372 OK
```

This second form works fine as long as you're just adding all of the numbers together (and you've specified the correct number of [+] operators). However, it can get you in trouble if there are some subtractions or other arithmetic operations mixed in. All things considered, *the simplest solution is the more "correct" solution*.

Add One or Two to the Stack (1+ and 2+)

The requirement to add a small constant, such as 1 or 2, to the number on the stack is so common that the FORTH-79 Standard has Required Words to perform each of these tasks.

An example of "one-plus" is

```
6 1+ . 7 OK
```

and an example of "two-plus" is

```
6 2+ . 8 OK
```

These words are of dubious value when you are using FORTH to perform simple calculator-like functions, as we have been doing so far. However, if the value at the top of the stack represents a *memory address*, rather than data, [1+] can change that address so that it references (that is, "points to") the next consecutive byte in memory, thereby preparing FORTH to operate on that byte. Similarly, the word [2+] can change an address so that it references the next consecutive (two-byte) number in memory.

Add Double Numbers (D+)

The 32-bit double numbers on the stack can be added just as easily as the 16-bit numbers, by using the word [D+] instead of [+]. A typical example is:

```
100000. 200000. D+ D. 300000 OK
```

SUBTRACTION

The fundamental subtraction word is the "minus" [−] operator, which subtracts the top number on the stack from the second number on the stack and leaves their difference. (Again, "leaves" implies "leaves on the top of the stack.")

The numbers can be both positive, as in

```
3 2 − . 1 OK
```

or both negative, as in

```
−3 −2 − . −1 OK
```

or mixed, as in

```
3 −2 − . 5 OK
```

Further, additions can be mixed with subtractions (or other operations) on a single line, as in

```
32 64 + 56 − 96 + 124 − . 12 OK
```

Subtract One or Two from the Stack (1− and 2−)

Like their addition counterparts, these two words provide a quick way to apply a small constant to the top value on the stack. Examples are:

```
6 1− . 5 OK
1 2− . −1 OK
```

Subtract Double Numbers (D−)

This word, [D−], is the first word we've encountered that is not part of the Required Word Set. However, it is included in the Double Number Extension Word Set, which comes with many FORTH software packages.

A typical example of its usage is:

```
10000. 200000. D− D. −190000 OK
```

MULTIPLICATION

The fundamental word for multiplying two stack values is "times" [*]. This word multiplies the top two numbers on the stack and leaves the product.

For example,

```
3 2 * . 6 OK
```

However, the product must also be a number-sized value. That is, the product *must* be in the range −32,768 to 32,767. Many versions of FORTH will produce a product that has "wrapped-around" to a 16-bit value. For instance, 2 times 30,000 should produce 60,000, but in many FORTH packages, this will happen:

```
30000 2 * . −5536 OK        (Incorrect answer)
```

Students of binary arithmetic will realize that the carry has been ignored, and that the result (minus carry) has been incorrectly interpreted as an "OK" answer!

Unsigned Multiplication (U*)

If the numbers you are multiplying are unsigned, the product is a *double number*, which expands the range of valid answers considerably. For example, our previous out-of-bounds operation

```
30000 2 U* D. 60000 OK
```

now produces the correct answer!

Multiply by Two (2*)

An uncontrolled word definition in the Reference Word Set allows us to double the top number on the stack; that is, multiply this number by two. An example of this word, [2*], is:

```
4 2* . 8 OK
```

DIVISION

FORTH includes three different Required Words that divide the second number on the stack by the top number on the stack. The first of these words, [/], leaves only the *quotient*. An example is:

```
7 4 / . 1 OK
```

Note that although the decimal answer is 1.75, FORTH truncates the fractional portion (the remainder), thereby rounding off the quotient toward zero rather than toward the nearest integer (2).

A second divide word, [MOD] (for "modulo"), performs the same divide operation, but leaves only the *remainder*. An example is:

```
7 4 MOD . 3 OK
```

The remainder will always carry the same sign as the dividend (second number on the stack), as we see here:

```
−7 −4 / . 1 OK      (Quotient)
−7 −4 MOD . −3 OK       (Remainder)
```

The third divide word, [/MOD], leaves both the *quotient and remainder* on the stack, with the quotient on top. Using the previous examples:

```
7 4 /MOD . . 1 3 OK
−7 −4 /MOD . . 1 −3 OK
```

Unsigned Division (U/MOD)

The unsigned division word U/MOD (called U/ in fig-FORTH) divides a double dividend by a single divisor to produce a single remainder and quotient. The double dividend feature makes the following kind of division operation possible:

```
3000000. 460 U/MOD U. U. 6521 340 OK
```

where 6521 is the quotient (since it was left at the top of the stack) and 340 is the remainder.

Divide by Two (2/)

An uncontrolled word in the Reference Word Set allows us to divide the top number on the stack by two. An example of this word, [2/], is:

```
7 2/ . 3 OK
```

MULTIPLY-THEN-DIVIDE

Many applications involve *scaling* a number; that is, multiplying the number by a fraction. The easiest way to do this, with

the words we've studied so far, is to multiply the number by the numerator (the top part of the fraction) and then divide the result by the denominator (the bottom part of the fraction). The following sequence scales the number 32 using the fraction 5/9:

```
32 5 * 9 / . 17 OK
```

Since this task is so common, FORTH provides two Required Words that will do the job. Both words require all three operands to be on the stack: the base number, the numerator, and the denominator, in that order.

The first of these words, [*/], leaves only the *quotient;* that is, the integer portion of the answer. For example,

```
32 5 9 */ . 17 OK
```

The second word, [*/MOD], leaves both the *quotient and remainder*—that is, the integer and fractional portions of the answer—on the stack, with the quotient on top. Therefore,

```
32 5 9 */MOD . . 17 7 OK
```

[*/] and [*/MOD] Give More Precise Answers

Besides the convenience offered by these two combination words, they will actually produce more precise answers if large numbers are involved. This is true because both [*/] and [*/MOD] maintain the intermediate product (number times numerator) as a *32-bit* value, rather than a 16-bit value.

This double-precision feature is valuable in operations involving "pi" (π), for instance. Since FORTH cannot handle fractional numbers directly, a value such as 3.1416 could be represented as the fraction 31416/10000, or as the closer approximation 355/113. Thus, the equation for the circumference of a circle becomes

$$c = \pi d = \frac{355 * d}{113}$$

where,

c is circumference,

d is the diameter.

If the diameter is already on the stack, the circumference can be calculated with the sequence

```
355 113 */
```

Let's see what happens with a diameter of 42:

```
42 355 113 */ . 131 OK
```

So the circumference is equal to 131.

RAISE A NUMBER TO A POWER

The uncontrolled word [**] raises a number to a power. Specifically, the second number on the stack is raised to the power given as the top number on the stack, leaving a result number. For example,

```
9 4 ** . 6561 OK
```

Bear in mind, though, that the result must not exceed the limits −32,768 through 32,767.

NEGATE

Two FORTH words allow you to reverse the *sense* of the value at the top of the stack. That is, these words make positive values negative and negative values positive. They do this by two's complementing the value, by subtracting it from zero.

One of these words, NEGATE (labeled MINUS in fig-FORTH), operates on a number. Two examples are:

```
3 NEGATE . −3 OK
−3 NEGATE . 3 OK
```

The other word, DNEGATE (labeled DMINUS in fig-FORTH), operates on double numbers, such as

```
300000. DNEGATE D. −300000 OK
```

Combine DNEGATE and D+ to Subtract Double Numbers

If your FORTH does not have the Double Number Extension Word Set, which includes the "subtract-double" word D−, you can still subtract double numbers, by combining Required Words DNEGATE and D+.

Assuming that the two double operands are already on the stack, the sequence to subtract the top value from the second value is:

```
DNEGATE D+
```

Using some actual numbers, an example is:

```
200000. 10000. DNEGATE D+ D. 190000 OK
```

ABSOLUTE VALUE

If you care only about the magnitude of a number, and not whether it's a negative or positive value, you can derive its *absolute value* with two FORTH words.

One of these words, ABS, takes the absolute value of a number. That is,

```
3 ABS . 3 OK
-3 ABS . 3 OK
```

Similarly, DABS, a Double Number Extension Word, takes the absolute value of a double number, such as

```
-300000. DABS D. 300000 OK
```

MAXIMUM AND MINIMUM

There are times in which two numbers are on the stack, but we only want to use the larger number, or the smaller number. FORTH has words that compare these two numbers, then leave only the larger or the smaller, depending on which was specified. The two words that do this job are MAX and MIN, respectively. Here are examples of both:

```
6 -6 MAX . 6 OK
-64 -20 MAX . -20 OK
6 -6 MIN . -6 OK
-64 -20 MIN . -64 OK
```

The Double Number Extension Word Set includes equivalent words for finding the larger or smaller double number. Examples of these words, DMAX and DMIN, are:

```
300000. 100000. DMAX D. 300000 OK
-200000. 200000. DMIN D. -200000 OK
```

fig-FORTH ARITHMETIC WORDS

Table 2-2 summarizes the arithmetic words included in Release 1 of fig-FORTH. A comparison with Table 2-1, which lists

the FORTH-79 arithmetic word group, will reveal that many of the definitions are identical. To avoid repetition, the remainder of this chapter will describe only the words that are unique to fig-FORTH.

Addition and Subtraction

Although fig-FORTH includes all four of the FORTH-79 addition words, it has only the basic subtraction word, [−]. Fortunately, the three missing subtraction words, [1−], [2−], and [D−], are easy to simulate. We showed the simulation of [D−] earlier, in our discussion of DNEGATE. However, in fig-FORTH DNEGATE is labeled DMINUS, so the simulation sequence for [D−] becomes:

```
DMINUS D+
```

Multiplication

For multiplication fig-FORTH includes the two multiplication words [*] and [U*], and adds one more valuable word, [M*]. This word is similar to [*], but leaves a double-number product rather than a single-number product. Therefore, [M*] eliminates the possibility of generating an out-of-range product by multiplying two signed numbers. With [M*] you can make this calculation

```
30000 30000 M* D. 900000000 OK
```

which would have failed using [*].

Division

For division, fig-FORTH includes all four of the FORTH-79 Required Words (although the FORTH word U/MOD is called U/ in fig-FORTH), and is missing only the uncontrolled word 2/, which is not much of a loss. In addition, fig-FORTH has two very useful new division words.

The first of these words, M/, acts like the /MOD, but operates with a *double-number dividend.* This feature makes the following kind of calculation possible:

```
3000000. 5000 M/ . . 600 0 OK
```

where 600 is the quotient and 0 is the remainder. As usual, the remainder will have the same sign as the dividend.

At this point we know of two fig-FORTH words that use a double-sized dividend, U/ (for unsigned numbers) and M/ (for

Table 2-2. fig-FORTH Arithmetic Words

Word	Stack	Action
+	n1 n2 --- sum	Adds two numbers.
1+	n --- n+1	Adds 1 to number.
2+	n --- n+2	Adds 2 to number.
D+	d1 d2 --- dsum	Adds two double numbers.
–	n1 n2 --- diff	Subtracts n2 from n1.
*	n1 n2 --- prod	Multiplies numbers n1 and n2, leaving single-precision product.
M*	n1 n2 --- dprod	Multiplies numbers n1 and n2, leaving double-precision product.
U*	un1 un2 --- ud	Multiplies unsigned numbers n1 and n2, leaving double-precision product.
/	n1 n2 --- quot	Divides n1 by n2, leaving quotient.
MOD	n1 n2 --- rem	Divides n1 by n2, leaving remainder with sign of n1.
/MOD	n1 n2 --- rem quot	Divides n1 by n2, leaving remainder and quotient.
M/	d n --- rem quot	Divides double dividend by single divisor, leaving single remainder and quotient.
U/	ud un --- rem quot	Divides unsigned double dividend by unsigned single divisor, leaving single remainder and quotient.
M/MOD	ud un --- rem dquot	Same as [U/], but leaves single remainder and double quotient.
*/	n1 n2 n3 --- quot	Multiplies n1 by n3, then divides the result (a 32-bit intermediate product) by n3, leaving a single quotient.
*/MOD	n1 n2 n3 --- rem quot	Same as */ , but leaves both quotient and remainder.
MINUS	n --- –n	Reverses the sense of a number, leaving its two's complement.

Table 2-2.—cont. fig-FORTH Arithmetic Words

Word	Stack	Action
DMINUS	d --- −d	Reverses the sense of a double number.
+−	n1 n2 --- n3	Applies the sign of n2 to n1, leaving it as n3.
D+−	d1 n --- d2	Applies the sign of n to d1, leaving it as d2.
S−>D	n --- d	Sign-extends a number to form a double number.
ABS	n --- n	Leaves the absolute value of a number.
DABS	d --- d	Leaves the absolute value of a double number.
MAX	n1 n2 --- max	Leaves the greater of two numbers.
MIN	n1 n2 --- min	Leaves the lesser of two numbers.

signed numbers). As valuable as these words are, however, both are inadequate if the dividend is considerably larger than the divisor, because both can return only a 16-bit quotient. The second fig-FORTH word, M/MOD, eliminates this problem for unsigned numbers by dividing an unsigned double dividend by an unsigned single divisor to produce a single remainder and a *double quotient*. Thus, M/MOD permits this kind of calculation to take place:

```
3000000. 7 M/MOD D. U. 428571 3 OK
```

Multiply-Then-Divide, Absolute Value, Max and Min

In each of these categories fig-FORTH offers the same words as FORTH-79, except that fig-FORTH does not include the double number maximum and minimum words, DMAX and DMIN. However, these words can be easily formed with a double number compare operation, which will be discussed in a later chapter.

Change a Negative Number to Positive, or Vice Versa

As was mentioned previously in this chapter, fig-FORTH includes the two's-complementing words NEGATE and DNE-

GATE, but has them labeled MINUS and DMINUS, respectively. In addition, fig-FORTH provides words that apply the sign of the top value (number or double number) on the stack to the second value on the stack.

These words, [+−] and [D+−], operate on numbers and double numbers, respectively. In many cases, this top, determining number will be a *flag*. A flag is the result of a compare operation, which we will discuss in Chapter 7.

Convert a Number to a Double Number

Rather than just changing the sign of a number, the fig-FORTH word [S−>D] actually converts the number to a double number. This conversion is made by extending the sign of the number from 16 bits to 32 bits, but don't worry about the mechanics of the operation just yet. We will discuss the physical details of numbers in a later chapter.

[S−>D] is an extremely valuable word, because it permits numbers to be used in double number operations. For instance, it permits a number to be added to a double number, with the sequence (S−>D D+).

SUMMARY

In this chapter we examined the basic data units in FORTH—numbers and double numbers—and then discussed the various words that perform arithmetic operations on them. Besides the usual add, subtract, multiply, and divide operations, FORTH provides such special-purpose tasks as multiply-then-divide, negate, absolute value, maximum and minimum.

In addition to the Required Words, which primarily operate on 16-bit numbers, there were a variety of Double Number Extension Words, which operate on 32-bit double numbers. There were also two "uncontrolled words," [2*] and [2/], which multiply or divide the top number on the stack by two.

The chapter concluded with a summary of the words that are available with Revision 1 of fig-FORTH, accompanied by a discussion of how these words relate to the arithmetic words in the FORTH-79 Standard.

CHAPTER 3

Stack Manipulation

As you may have noticed, the programming examples in the last chapter were all exceptionally "clean." The operand words were right where we wanted them on the stack and in the proper order (since they were entered from the keyboard in most cases), and almost all examples involved just one operation, which left the result on the top of the stack; again, just where we wanted it for printing. Ah, if all programming could be that simple!

In real life, where most tasks involve many operations, things don't normally work out that conveniently. At times, a number to be printed will end up as the second item on the stack or a divisor-dividend combination will occur in reverse order. For these situations, and similar mismatches, we need to be able to manipulate the stack. Table 3-1 summarizes the FORTH-79 stack manipulation words, subdivided by function.

DUPLICATE TOP ITEM

Two Required Words cause the top number on the stack to be duplicated, leaving two copies of that number. The word DUP always duplicates the top number, regardless of its value, whereas ?DUP (called −DUP in fig-FORTH) only duplicates the number if it is nonzero.

The word DUP is often used when a number is to operate on itself. Some common applications of DUP are:

1. To double a number (DUP +)
2. To square a number (DUP *)
3. To cube a number (DUP DUP * *)
4. To quadruple a number (DUP * DUP *)

Table 3-1. Stack Manipulation Words

Word	Stack	Action	Notes
DUP	n --- n n	Duplicate top number of stack.	
?DUP	n --- n (n)	Duplicate top number only if it is nonzero.	
2DUP	d --- d d	Duplicate top double number of stack.	(1)
OVER	n1 n2 --- n1 n2 n1	Leave copy of second number on top of stack.	
2OVER	d1 d2 --- d1 d2 d1	Leave copy of second double number on top of stack.	(1)
PICK	n1 --- n2	Leave copy of n1-th number on top of stack.	
DROP	n ---	Delete top number from stack.	
2DROP	d ---	Delete top double number from stack.	(1)
SWAP	n1 n2 --- n2 n1	Exchange top two numbers on stack.	
2SWAP	d1 d2 --- d2 d1	Exchange top two double numbers on stack.	(1)
ROT	n1 n2 n3 --- n2 n3 n1	Rotate third number to top of stack.	
2ROT	d1 d2 d3 --- d2 d3 d1	Rotate third double number to top of stack.	(1)
ROLL	n --- (n)	Rotate n-th number to top of stack.	
DEPTH	--- n	Count numbers on stack.	
S0	--- addr	Leaves the address of the bottom of the stack.	(2)
SP@	--- addr	Leaves the address of the top of the stack.	(3)

Notes: (1) Included in Double Number Extension Word Set.
(2) Included in Reference Word Set, as an uncontrolled word definition.
(3) Included in Reference Word Set, as a Standard Word Definition.

Another common usage for DUP is to print a number without losing it from the stack. For example, the sequence

```
42 DUP * DUP . 1764 OK
```

squares the number 42, then duplicates the result (1764) before printing it. The result is still on the top of the stack, for subsequent processing.

One more use for DUP is to save a copy of a result that must be used immediately, then used again later. We'll see how to access that saved result when we get to words like SWAP and ROT, later in this chapter.

Our second Required Word, ?DUP, normally precedes the construct IF . . . THEN, so we'll postpone the discussion of ?DUP until we cover that construct.

Table 3-1 also shows the Extension Word 2DUP, which is the counterpart of DUP for duplicating double numbers.

COPY AN ITEM ONTO TOP

Sometimes we will want to duplicate the second value on the stack, rather than the top number. This can be done for numbers with the word OVER and for double numbers with the word 2OVER; the latter is a Double Number Extension Word.

For example, the sequence

```
20 30 OVER
```

leaves the numbers 20, 30, and 20 at the top of the stack. You might want to do this type of operation if you had just made two calculations and wanted to do something with the first result immediately, and do something else with it later. Remember, since the 20 was entered first, it will end up as the second number on the stack when 30 is entered.

PICK Any Number in the Stack

FORTH-79 also allows you to copy any number in the stack onto the top, using the word PICK. To select the number to be PICKed, put its depth level on top of the stack and execute PICK. A 1 on the stack selects the top number (1 PICK = DUP), a 2 on the stack selects the second number (2 PICK = OVER), a 3 on the stack selects the third number, and so on.

For example,

```
6 PICK
```

copies the sixth number onto the top of the stack. As with OVER, after PICK, the selected number is in two places on the stack: its original position and on top of the stack.

Fig. 3-1 shows the numbering system for numbers on the stack. Double numbers are also shown, for future reference.

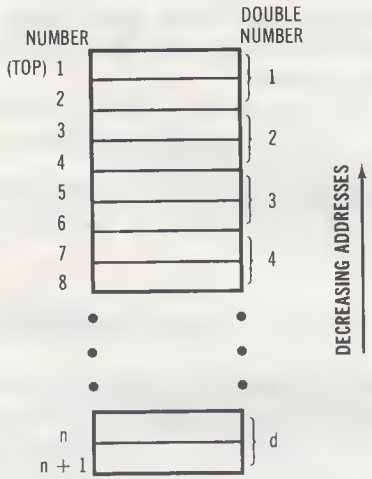


Fig. 3-1. Numbers and double numbers on the stack.

DELETE TOP ITEM

There are also times in which we won't need the top value on the stack, and wish to delete it. For these times, FORTH-79 offers the Required Word DROP, to delete a number, and the Double Number Extension Word 2DROP, to delete a double number.

MOVE AN ITEM TO THE TOP

The DUP and OVER word groups allowed us to *copy* a value onto the top of the stack, but leave the original value in its former position. Those words are useful for applications in which we will need to re-access the original value. However, and more often the case, we will want to access some previous result from down in the stack, but we won't need it again. The words in this group give us that capability, by moving a selected value from its present position to the top of the stack.

SWAP the Top Two Items

The Required Word SWAP and the Double Number Extension Word 2SWAP perform the simplest kind of stack move operation; they just *exchange* the top two numbers or double numbers, respectively.

Such an exchange is handy if you've just performed a divide operation, for instance, with a /MOD or */MOD word, and wish to operate on the remainder. As you may recall, both words leave the quotient on top, followed by the remainder. However, a simple application of SWAP will put the remainder on top and the quotient in the second position. You're now set up to operate on the remainder!

ROTate the Third Item Up to the Top

The FORTH Standards Team realized that the *third* item on the stack is also one that is often needed, and provided two words that *move* this third item up to the top of the stack. As before, there is a Required Word, ROT, which accesses the third number, and a Double Number Extension Word, 2ROT, which accesses the third double number.

For example, if the top three numbers look like this before ROT:

150	(Top number)
250	(Second number)
350	(Third number)

they will look like this after [ROT] has been executed:

350	(Top number)
150	(Second number)
250	(Third number)

ROLL Any Number to the Top

FORTH-79 also allows you to move *any* number to the top of the stack, using the word ROLL. To select the number to be ROLLed, put its depth level on top of the stack, then execute ROLL. A 2 on the stack selects the second number (2 ROLL = SWAP), a 3 on the stack selects the third number (3 ROLL = ROT), a 4 on the stack selects the fourth number, and so on.

For example,

```
6 ROLL
```

moves the sixth number onto the top of the stack, and moves numbers one through five down one level.

Delete Any Item From the Stack

Since the words in this group physically displace a number or double number from its original position in the stack, these words can also be used to *delete* a value from the stack, in the same way DROP and 2DROP deleted the top value. All you need to do is add a DROP or a 2DROP, depending on whether you are deleting a word or a double word. Therefore, the following operations are readily available:

1. Drop the second number from the stack (SWAP DROP).
2. Drop the second double number from the stack (2SWAP 2DROP).
3. Drop the third number from the stack (ROT DROP).
4. Drop the third double number from the stack (2ROT 2DROP).
5. Drop the nth number from the stack (n ROLL DROP).

FIND THE BOUNDS OF THE STACK

The three last words in Table 3-1 return information about the current status of the stack. The Required Word DEPTH leaves a count of the 16-bit values contained on the stack. If the stack holds 16-bit *numbers,* the value returned will be a count of those numbers. If the stack holds 32-bit *double numbers,* the value returned will be a count of those double numbers, multiplied by two.

For example,

```
DEPTH . 8 OK
```

informs us that there are currently eight 16-bit values on the stack. That is, the stack holds either eight numbers or four double numbers.

FORTH maintains a special memory address pointer called a *stack pointer,* which holds the address of the top number on the stack. Upon entering FORTH, the stack pointer contains an implementation-dependent address called the "bottom" of the stack. With each number pushed onto the stack, the stack pointer gets decremented by two and thereby holds the address of the current "top" of the stack. Two FORTH words, S0 and SP@, can be used to find the memory address of the bot-

tom and top of the stack, respectively. The addresses returned by these words are illustrated in Fig. 3-2.

Knowing either the bottom or top address of the stack, you can compute the address of any item on the stack. For example, since SP@ leaves the address of the top number, then

```
SP@ 2+
```

will leave the address of the second number,

```
SP@ 4 +
```

will leave the address of the third number, and so on.

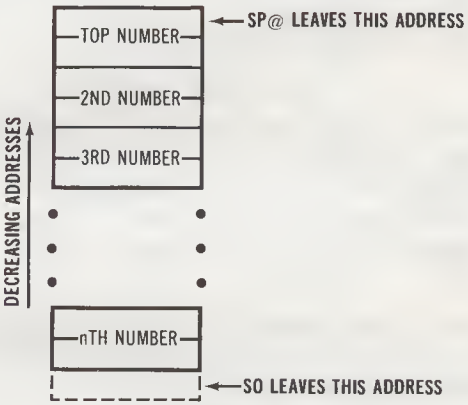


Fig. 3-2. Stack boundaries (FORTH-79).

DERIVING DOUBLE-NUMBER EQUIVALENTS OF PICK AND ROLL

The Double Number Extension Word Set provides double-number equivalents to most of the words that manipulate numbers on the stack. These include 2DUP, 2OVER, 2DROP, 2SWAP, and 2ROT. However, there are no double-number equivalents to PICK, the word that copies the nth number to the top of the stack, or ROLL, the word that rotates the nth number to the top of the stack. In this section we will show you how to derive double-number equivalents for both PICK and ROLL, using the available Required Words.

As you know, both PICK and ROLL based their operation on a "stack-depth" index at the top of the stack. If the index was 2, the second number was copied or rotated; if the index was 3, the third number was copied or rotated, and so on. If we think of a number as a 16-bit value (which it is), and a 32-bit double number as two 16-bit values (high-order 16 bits and low-order 16 bits), the problem of copying or rotating a double number becomes a matter of applying PICK or ROLL *twice,* once for each half of the double number.

The double number copy and exchange operations we'll be giving here also begin with having an index on the top of the stack, but in this case the index will identify the *double number* to be operated on. Fig. 3-1 shows the relationship between number indexes and double number indexes. As you can see, for any double number index *d,* the single number indexes *n* and *n+1* can be derived using these equations:

$$n = 2d - 1 \text{ and } n+1 = 2d$$

However, if we apply these equations directly to access a double number, we will be "off" by one stack value, because they don't account for the fact that there are *two* indexes at the top of the stack, rather than just one. To accommodate this difference, just add one to each equation, to give:

$$n = 2d \text{ and } n+1 = 2d + 1$$

At this point we are ready to look at two program sequences, one that will copy the nth double number onto the top of the stack (a 2PICK sequence, if you will) and another that will rotate the nth double number onto the top of the stack (a 2ROLL sequence). These sequences are shown in Examples 3-1 and 3-2, respectively. In a later chapter we will show how sequences such as these can be defined as words; they are put in the dictionary to be thereafter used like the predefined words.

Incidentally, the text enclosed in parentheses in these examples is comments, to describe what's taking place. This is the standard way comments are shown in FORTH. Note, too, that there is a space between the left parenthesis and the beginning of the comment. *This space is required,* because [(] is actually a Required Word, rather than simply a punctuation mark! No space is required between the end of the comment and the right parenthesis, however.

These examples also illustrate a good FORTH programming practice: *keep the lines short!* Example 3-1 could have been written like this:

```
2 * DUP 1+ PICK SWAP PICK
```

Example 3-1. Copy a double number onto the top of the stack

```
( This sequence copies the d-th double number onto the )
( top of the stack.                                    )
( Stack: d --- d1 )
2 *         ( Leave index to high-order16 bits)
DUP 1+      ( Leave index to low-order 16 bits)
PICK        ( Copy low-order 16 bits to top of stack)
SWAP        ( Put high-order index on top of stack)
PICK        ( Copy high-order 16 bits to top of stack)
```

Example 3-2. Rotate a double number onto the top of the stack

```
( This sequence rotates the d-th double number onto the )
( top of the stack.                                     )
( Stack: d --- d1 )
2 *         ( Leave index to high-order 16 bits)
DUP 1+      ( Leave index to low-order 16 bits)
ROLL        ( Rotate low-order 16 bits to top of stack)
SWAP        ( Put high-order index on top)
ROLL        ( Rotate high-order 16 bits to top of stack)
```

but that would have made it diabolically difficult to understand, and virtually impossible to comment.

fig-FORTH STACK MANIPULATION WORDS

Table 3-2 summarizes the stack manipulation words included in fig-FORTH. A comparison with Table 3-1, which summarizes the FORTH-79 stack manipulation words, makes it apparent that fig-FORTH offers only the fundamental words that duplicate, copy, delete, and rotate numbers. Operations on double numbers are totally omitted.

Note that the descriptions of S0 differ between Table 3-1 and Table 3-2. This occurs because in FORTH-79, S0 returns the address of the bottom of the stack, whereas in fig-FORTH, S0 returns a *pointer* to the address of the bottom of the stack. That is, the fig-FORTH S0 returns the address of a memory location,

Table 3-2. fig-FORTH Stack Manipulation Words

Word	Stack	Action
DUP	n --- n n	Duplicate top number of stack.
-DUP	n --- n (n)	Duplicate top number only if it is nonzero.
OVER	n1 n2 --- n1 n2 n1	Leave copy of second number on top of stack.
DROP	n ---	Delete top number from stack.
SWAP	n1 n2 --- n2 n1	Exchange top two numbers on stack.
ROT	n1 n2 n3 --- n2 n3 n1	Rotate third number to top of stack.
S0	--- addr	Points to the address of the bottom of the stack.
SP!		A user-supplied procedure to initialize the stack pointer from S0.
SP@	--- addr	Leaves the address of the top of the stack.

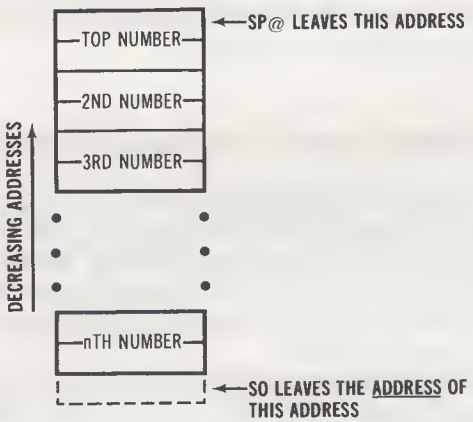


Fig. 3-3. Stack Boundaries (fig-FORTH).

and that memory location contains the address of the bottom of the stack. Fig. 3-3 illustrates the relationship of the SP@ and S0 addresses to the stack. You might wish to compare this diagram to its FORTH-79 counterpart, in Fig. 3-2.

Also fig-FORTH provides a new word, SP!, that initializes the stack pointer using the address pointed to by S0.

SUMMARY

This chapter showed us how items on the stack could be manipulated, thereby bringing a value to the top, where it could be processed. Included were words that duplicated or deleted the top item, words that brought an item to the top from a lower level (by either copying it or rotating it) and words that returned the address boundaries and the number count.

Some simple examples showed how the words learned so far could be combined, to define new operations. These examples also demonstrated the proper way of adding comments to FORTH programs—with a left parenthesis and a blank preceding the commentary text.

CHAPTER 4

Memory Operations

Chapters 2 and 3 dealt exclusively with operations on the stack. This is a proper place to begin, since virtually all FORTH words affect the stack in one way or another. However, the stack is just a *temporary* depository for information. In all computer systems, most information is held in *memory*, including programs being executed.

There are essentially two types of memory: random access memory and mass storage. *Random access memory* can be either read/write memory (usually called RAM, although this is a misnomer) or read-only memory. Read/write memory serves as "working storage" for data and programs while the system is running; all information in read/write memory is lost when the power is turned off. Read-only memory (ROM) is permanent storage that is usually used to hold system programs; information stays in read-only memory "forever," even when the power is off. As the names imply, you can transfer information into or out of read/write memory, but you can only transfer information out of read-only memory. All operations in this chapter are performed on read/write memory, or RAM.

Mass storage is memory that holds information that will later be transferred to or from read/write memory. Floppy disks, hard disks, reel tapes, and cassette tapes are common examples of mass storage devices. We will cover mass storage operations later in this book.

THE MEMORY WORD GROUP

Table 4-1 summarizes the words that operate on memory. Each of these words requires a memory address to be on the

Table 4-1. Memory Words

Word	Stack	Action	Notes
@	addr --- n	Fetches number at address.	
!	n addr ---	Stores number at address.	
C@	addr --- byte	Fetches byte at address.	
C!	n addr ---	Stores least-significant byte of n at address.	
2@	addr --- d	Fetches double number at address.	(1)
2!	d addr ---	Stores double number at address.	(1)
?	addr ---	Displays number at address.	
DUMP	addr n ---	Displays the contents of n memory locations, starting at addr.	(3)
+!	n addr ---	Adds n to the number at address.	
MOVE	addr1 addr2 n ---	Copies n numbers starting at addr1 to memory starting at addr2. The move proceeds from low memory to high memory.	
CMOVE	addr1 addr2 n ---	copies n bytes starting at addr1 to memory starting at addr2. The move proceeds from low memory to high memory.	
<CMOVE	addr1 addr2 n ---	Copies n bytes starting at addr1 to memory starting at addr2. The move proceeds from high memory to low memory.	(2)
FILL	addr n byte ---	Fills n consecutive bytes in memory with the value byte, starting at addr.	
ERASE	addr n ---	Fills n consecutive bytes with 0, starting at addr.	(3)
BLANKS	addr n ---	Fills n consecutive bytes in memory with the ASCII value for "blank," starting at addr.	(3)

Notes: (1) Included in Double Number Extension Word Set.
(2) Included in Reference Word Set, as an uncontrolled word definition.
(3) Included in Reference Word Set, as a Standard Word Definition.

stack and some require an additional parameter, such as a number.

Memory Is Arranged in Bytes

In FORTH, addresses are unsigned numbers, which can have values between 0 and 65,535. This is the normal addressing range of 8-bit microprocessors, since these microprocessors have 16-line *address buses*.

FORTH also assumes that memory is arranged in *bytes* (8 bits of data), the smallest memory unit that is addressable by a microprocessor. Therefore, we see words in Table 4-1 that reference bytes, as well as numbers and double numbers. A single byte can represent unsigned values between 0 and 255 and signed values between −128 and 127.

Know Your Memory Map

To use the FORTH memory words intelligently, you must understand how the memory in your particular system is arranged. That is, you must know which portions of memory hold RAM, which hold ROM and, if your microcomputer uses *memory-mapped* I/O (with peripheral devices addressed like a standard memory location), which portions of memory are connected to peripherals.

All of these questions can be answered by studying the *memory map* of your system. Become familiar with the memory map. Even if you don't see any reason to do so now, it will pay off in the future.

FETCH AND STORE

FORTH-79 provides four Required Words that transfer numbers and bytes between the stack and memory. The words [@]and [C@] fetch (read) a number or a byte from a specified location in memory and leave it on the top of the stack. The words [!] and [C!] store (write) a number or a byte into a specified location, taking the address and the value from the stack. In the case of [C!], only the least-significant byte of the 2-byte number is transferred; the most-significant byte is ignored.

Here are examples of these four words:

```
500 200 ! OK      ( Store the number 500 at address 200)
200 @ . 500 OK ( Read and display the number stored at address 200)
```

```
50 200 C! OK    ( Store the byte 50 at address 200)
200 C@ . 50 OK( Read and display the byte stored at address 200)
```

Keep in mind that a read operation simply co*pies* the contents of a memory location onto the stack; it does not affect the memory location contents in any way.

The Double Number Extension Word Set includes the words [2@] and [2!], which fetch and store double numbers, respectively. Using the preceding parameters:

```
500. 200 2! OK  ( Store the double number 500 at
                  address 200)
200 2@ D. 500 OK( Read and display the double number
                  stored at address 200)
```

Of course, for numbers and double numbers, the specified location does not hold the *entire* value, but just one of its bytes. Numbers occupy two bytes in memory and double numbers occupy four bytes in memory.

Deriving [2@] and [2!] Using [@] and [!]

If your FORTH does not have the words [2@] and [2!], you can derive these double number functions using the Required Words [@] and [!]. It simply involves operating on each half of the double number (the high-order 16 bits and the low-order 16 bits) individually. Examples 4-1 and 4-2 show the sequences to perform the [2@] and [2!] functions, respectively.

Example 4-1. Fetch a double number from memory

```
( This sequence fetches a double number from memory.)
( Stack: addr --- d)
DUP 2+ ( Leave addr+2)
@      ( Fetch low-order 16 bits)
SWAP   ( Put addr on top of the stack)
@      ( Fetch high-order 16 bits)
```

DISPLAY THE CONTENTS OF MEMORY

In the course of debugging a program, you may need to find out what a certain memory location contains, without reading its contents onto the stack. The FORTH word [?] displays the

Example 4-2. Store a double number in memory

```
( This sequence stores a double number into memory.)
( Stack: d addr ---)
ROT     ( Rotate low-order 16 bits to top of stack)
OVER    ( Copy addr to top of stack)
2+      ( and change it to addr+2)
!       ( Store low-order 16 bits)
!       ( Store high-order 16 bits)
```

number stored at a selected address. For example, the sequence

```
200 ? 500 OK
```

shows that memory starting at address 200 contains the number 500.

There are no standard FORTH words to display a byte or a double number in memory, but these operations are nothing more than the sequences

```
C@ .      ( Display a byte)
```

and

```
2@ D.     ( Display a double number)
```

Again, if your FORTH does not have the [2@] word, you can substitute the sequence shown in Example 4-1, and follow it with [D.].

Use DUMP To Display Consecutive Bytes in Memory

An even more useful debugging word is DUMP, which is a Standard Word in the Reference Word Set. DUMP displays a selected number of memory locations, starting at a specified address. Most FORTHs, including fig-FORTH, display several values on a line, and precede the first value with its address.

For example, the sequence

```
220 6 DUMP
```

may produce a listing such as

```
220   0 0 60 0
224 136 4
```

INCREMENT A NUMBER IN MEMORY

In Chapter 2 we discussed adding values to numbers on the stack, using the words [1+] and [2+]. FORTH offers a similar, but more versatile word called [+!], which adds a number on the stack to a number in memory. For example,

```
5 200 +!
```

adds the value 5 to the number stored at address 200. Similarly,

```
-5 200 +!
```

subtracts 5 from the number stored at address 200.

Other Arithmetic Operations on Memory

Clearly, it is also possible to perform a multiplication or a division on a number in memory, by simply fetching the number onto the stack, operating on it, then returning the result to memory. Before fetching the number, however, you must duplicate the address, so that it is available for the store operation.

The normal sequence for a multiply or divide operation on memory is:

```
DUP @      ( Fetch the number from memory)
ROT        ( Rotate operand to top of stack)
( Perform the operation)
SWAP       ( Put the address on top)
!          ( Return the result number to memory)
```

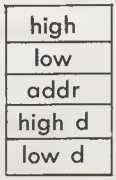
Adding a value to a double number in memory is somewhat more complex, though, due to the amount of stack manipulation that is required. Example 4-3 shows a sequence that will do the job, along with a "snapshot" of the stack after the memory contents have been fetched. The snapshot includes a number on each level of the stack, which will help you understand what the first ROLL is doing. You should be able to follow the sequence from there.

MOVE A BLOCK OF DATA IN MEMORY

Many applications require blocks of information to be copied from one part of memory to another. For instance, data processing often involves copying a table of numbers from one file

Example 4-3. Add d to a double number in memory

```
( This sequence adds d to the double number starting at addr.)
( Stack: d addr --- )
DUP     ( Leave copy of addr)
2@      ( Fetch double number at addr)
5 ROLL  ( Rotate low operand to top)
5 ROLL  ( Rotate high operand to top)
D+      ( Add the two double numbers)
ROT     ( Rotate addr to top)
2!      ( Store result in memory)
```



Stack After [2@]

to another and word processing involves duplicating a phrase, a sentence, or a paragraph in two different parts of a text. Such tasks can be readily accomplished with two Required block-move words, MOVE and CMOVE.

Both words accept three arguments from the stack: a source address (addr1), a destination address (addr2) and a count of the items to be moved (n). Starting with the number at addr1, MOVE copies n numbers to the block of memory starting at addr2. Similarly, CMOVE copies n bytes from addr1 to the block of memory starting at addr2.

For example,

```
200 400 20 MOVE OK
```

copies 20 numbers that start at address 200 to the portion of memory that starts at address 400. Similarly,

```
200 400 20 CMOVE OK
```

performs the same operation, except that 20 bytes are copied from address 200 to address 400. These examples copied information into a higher part of memory, but they could have also copied it into a lower part of memory. Therefore,

```
400 200 20 MOVE OK
```

is an equally valid operation.

If you wish to check that the move was made correctly, just apply the word DUMP. For example, the sequence

```
400 20 DUMP
```

will list the contents of locations 400 through 419.

<CMOVE Solves Overlap Problems

The Required words MOVE and CMOVE perform their copy operations by working from the beginning of the source block to the end; that is, the first item (number or byte) is copied first. The FORTH-79 Standard also includes an uncontrolled word, <CMOVE, that works in the opposite order, from the end of the block to the beginning, with the *last* byte being copied first.

Why would you ever need two separate byte-copying words? Well, if the source and destination blocks lie in two distinct areas of memory, either copy sequence (first-to-last or last-to-first) has the same effect, so either approach will have the same effect. However, if the destination block *overlaps* the source block, only one sequence will get the job done; the opposite sequence will wipe out one or more needed items.

To illustrate this point, Fig. 4-1 shows how CMOVE and <CMOVE each affect an operation in which four bytes (the values 1, 2, 3, and 4) are displaced one byte position backward in memory. In Fig. 4-1A, CMOVE starts the copy operation with the first byte, and produces the desired result. However, in Fig. 4-1B, <CMOVE copies the source bytes in the reverse order and ends up over-writing the destination block with 4's.

Similarly, if the source block was copied forward (to a higher address), instead of backward (to a lower address), <CMOVE would produce the proper result but CMOVE would not. To be safe in your block-copy operations, use this rule: *if a block is being copied to lower-addressed memory, use CMOVE; if a block is being copied to higher-addressed memory, use <CMOVE.*

FILL A BLOCK OF MEMORY

If your application involves processing a data table or an array in memory, you may wish to *initialize* that area of memory with some known value—perhaps zero—before storing anything else there. The easiest way to do this is with the word FILL.

Before executing FILL, the stack must hold three parameters:

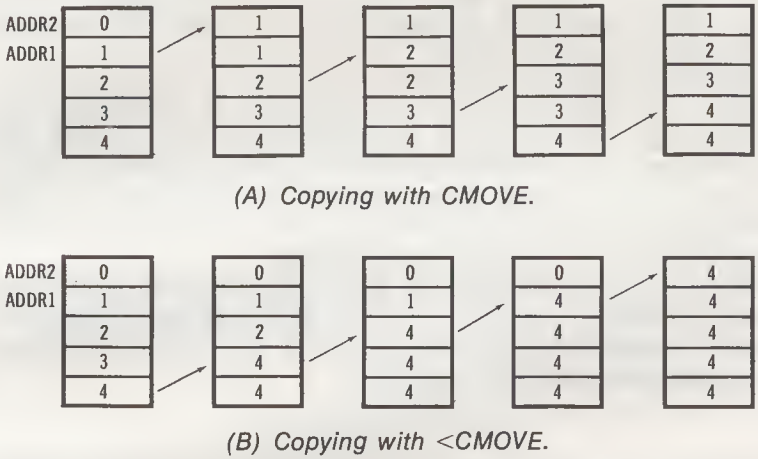


Fig. 4-1. Two block-copy operations.

the starting address of the block to be filled, the length of the block (in bytes), and the byte value to be stored into the block. For example,

```
200 50 0 FILL OK
```

stores a 0 into locations 200 through 249, inclusive. Remember, a byte can hold any value between 0 and 255.

Zero is such a commonly used filler value that the Reference Word Set includes a Standard Word, ERASE, that fills the specified block with the byte value 0. Thus, the ERASE equivalent of the preceding sequence is:

```
200 50 ERASE OK
```

Fill a Text Block With Blanks

In word processing and other text-oriented applications, the counterpart of zero is an ASCII blank or space. Therefore, in these applications you will often want to fill memory with blanks, usually to wipe out whatever text is currently in that block.

In the computer field, blanks and all other text characters are usually represented using an industry-standard code called *ASCII* (for American Standard Code for Information In-

terchange). Every time you press a key at the keyboard, the ASCII value for that key will be entered into the computer. Similarly, every character transmitted to a printer or a display screen will also be in ASCII.

The ASCII standard provides value for 127 separate characters, including letters of the alphabet (both upper case and lower case), numeric digits, punctuation marks, symbols, and control characters (backspace, delete, etc.). Appendix A shows the ASCII value in two different bases, decimal (base 10) and hexadecimal (base 16). Don't worry about hexadecimal numbers just yet; we will discuss them later.

At any rate, the Reference Word Set includes a Standard Word, BLANKS, that fills memory with the ASCII value for "blank" (decimal 32, or SP, in Appendix A). This word is handy for eradicating text. For instance,

```
400 40 BLANKS OK
```

eradicates the text in locations 400 through 439, by filling it with ASCII blanks.

Of course, the computer doesn't know whether 32, the value you've stored into memory, represents the number 32 of the ASCII value for "blank." This strictly depends on how your program makes use of the information!

Incidentally, ERASE, when coupled with CMOVE, implies some possible text processing capability with FORTH. Notice that the sequences

```
200 400 20 CMOVE OK
200 20 ERASE OK
```

are effectively moving a string of characters—a sentence, perhaps—from address 200 to address 400, then blanking out the source block. We will discuss the impressive string processing capabilities of FORTH later.

fig-FORTH MEMORY WORDS

The fig-FORTH memory word set, summarized in Table 4-2, is a subset of the FORTH-79 memory word set. As usual, fig-FORTH lacks the double number operators—the words [2@] and [2!] in this case—but includes all other words except [+!]. You'll see a <CMOVE definition in Chapter 11. (Double number words [2@] and [2!] were provided earlier in this chapter, as Examples 4-1 and 4-2, respectively.)

Table 4-2. fig-FORTH Memory Words

Word	Stack	Action
@	addr --- n	Fetches number at address.
!	n addr ---	Stores number at address.
C@	addr --- byte	Fetches byte at address.
C!	n addr ---	Stores least-significant byte of number at address.
?	addr ---	Displays number at address.
DUMP	addr n ---	Displays the contents of n memory locations, starting at addr. Both addresses and contents are shown in the current number base.
+!	n addr ---	Adds n to the number at address.
MOVE	addr1 addr2 n ---	Copies n numbers starting at addr1 to memory starting at addr2.
CMOVE	addr1 addr2 n ---	Copies n bytes starting at addr1 to memory starting at addr2.
FILL	addr n byte ---	Fills n consecutive bytes in memory with the value byte, starting at addr.
ERASE	addr n ---	Fills n consecutive bytes with 0, starting at addr.
BLANKS	addr n ---	Fills n consecutive bytes in memory with the ASCII value for "blank," starting at addr.

SUMMARY

This chapter described how FORTH can be used to operate on memory. We began with a brief overview of memory, including the concept of byte addressing, then discussed the FORTH words that have memory-related functions. Besides the basic fetch and store operations, there were words to display the contents of memory (either individually or in blocks), add a value to a number in memory, move blocks and fill a selected portion of memory.

The chapter also included program sequences to fetch or store a double number and to add a value to a number or double number in memory.

CHAPTER 5

Add Your Own Words to FORTH

After reading to this point in the book, you are probably wondering what is so special about FORTH. Based on what we've learned so far, that is a reasonable question. Indeed, from all appearances FORTH seems to be a rather ordinary high-level language. It has its share of good features (words can be combined in any order), bad features (programs tend to be hard to follow without copious comments) and features of dubious merit (reverse Polish notation, stack orientation).

Well, before dismissing FORTH as "just another language," you might consider reading about the most unique aspect of FORTH. Specifically, *FORTH allows you to add your own words to the language!*

That is, the FORTH dictionary can be physically extended to include words that you have defined. Once "compiled," these newly defined words become as much a part of the dictionary as [+], [DUP], [SWAP], and the other words that came with your FORTH package. Because these new words are held in the computer's read/write memory, they will disappear when you turn the power off. To solve this problem, FORTH provides words that allow you to save the definitions of these words on disk or cassette.

Table 5-1 summarizes the words we will describe in this chapter, words that are used to define new words and words that are used to communicate with the disk or cassette.

HOW TO DEFINE A NEW FORTH WORD

Defining a new word to be added to the FORTH dictionary is extremely easy. All you have to do is enter your definition in

Table 5-1. Word-Defining Words and Disk Words

Word	Stack	Action	Notes
:	---	Used in the form : name . . . ; to begin colon-definition of the word name.	
;	---	Ends colon-definition.	
(	---	Used in the form (comment) to begin a comment that will be ended by a right parenthesis on the same line.	
."	---	Used in the form ." message" to begin an output message that will be ended by a double quote on the same line.	
CR	---	Transmits a carriage return and line feed to the current output device.	
VLIST	---	Lists the word names in the dictionary, starting with the most recent definition.	(1)
FORGET	---	Used in the form FORGET name to delete the most recent definition of name from the dictionary.	
'	--- addr	Used in the form ' name to leave the dictionary address of the word name.	
EDITOR		Invokes the user-defined editor vocabulary.	(2)
EMPTY-BUFFERS		Marks all block buffers as "empty," without necessarily affecting their contents. Usually the first word executed in an editing session.	
UPDATE		Marks the most recently referenced block as	

Table 5-1—cont. Word-Defining Words and Disk Words

Word	Stack	Action	Notes
		"modified." This block will be automatically transferred to disk if its memory buffer is needed for storage of another block, or upon execution of SAVE-BUFFERS.	
SAVE-BUFFERS		Writes all UPDATEd blocks to disk.	
FLUSH		Synonym for SAVE-BUFFERS.	(1)
INDEX	n1 n2 ---	Displays the first line of screens n1 through n2. The first line usually contains a title.	(1)
LIST	n ---	Lists the ASCII symbolic contents of screen n on the current output device, setting SCR to contain n.	
LOAD	n ---	Loads block n (compiles or executes), reading the block from disk if it is not already in memory.	
SCR	--- addr	Leaves the address of a variable containing the number of the screen most recently LISTed.	
-->		Directs FORTH to continue interpretation on the next screen.	(2)
;S		Stops interpretation of a screen.	(1)

Notes: (1) Included in Reference Word Set, as an uncontrolled word definition.
(2) Included in Reference Word Set, as a Standard Word Definition.

this form:

```
: name word(s) ;
```

That is, a word definition is comprised of four parts:

1. The FORTH word [:], pronounced "colon."
2. The *name* of the new word. The FORTH-79 Standard specifies that a name can be up to 31 characters long and may not contain an ASCII null, blank, or "return."

3. The sequence of previously defined *words* that tell what operation the new word is to perform. The word(s) used here can be drawn from the dictionary that came with the FORTH package, or from words that you have previously defined (and thereby added to the dictionary) using the procedure we are describing.
4. The FORTH word [;], pronounced "semicolon."

Some Simple Examples

To see how a new word can be defined, let's assume that we want to have a word that squares the number at the top of the stack. From Chapter 3 we know that squaring a number can be performed with the sequence

```
DUP *
```

Therefore, if we call our new word SQUARE, its *colon-definition* (the standard FORTH nomenclature) would be

```
: SQUARE DUP * ;
```

After typing in this line and pressing Return, SQUARE is in the dictionary, ready to be used. To verify that it works, you can try sequences such as these:

```
2 SQUARE . 4 OK
-8 SQUARE . 64 OK
17 SQUARE . 289 OK
```

With SQUARE now defined and compiled (in the dictionary), it can serve as a building block for more new words. For instance, the definition of a word that cubes a number on the stack might be

```
: CUBE DUP SQUARE * ;
```

Applying CUBE to our three preceding examples:

```
2 CUBE . 8 OK
-8 CUBE . -512 OK
17 CUBE . 4913 OK
```

Now *both* words can be used to raise a number to the sixth power, with the colon-definition

```
: N**6 SQUARE CUBE ;
```

which allows us to do this:

```
2 N**6 . 64 OK
```

(The example values −8 and 17 cannot be demonstrated with [N**6], because they produce a result that is not representable as a *number;* the result exceeds the upper limit of 32,767.)

By now you have a clear idea of the "extensibility" of FORTH, so we'll forego additional examples. Hopefully, though, you also have an appreciation for the potential of this very unique feature. Admittedly, this building block approach can be applied in other computer languages—typically as "nested subroutines"—but not with the programming ease provided in FORTH.

DEFINITIONS ARE COMPILED

As mentioned in Chapter 1, every FORTH system includes two special programs that translate the character combinations which form FORTH words into electrical patterns the computer (or, more precisely, the computer's central processing unit) can understand. These two programs are called the interpreter and the compiler.

An *interpreter* is a program that translates and executes each line as it is typed in from the keyboard. Once executed, the commands (words) on the line are scrapped, and the interpreter sits and waits for another line. By using FORTH's interpreter—as you have been doing in the preceding chapters—you are, in effect, employing the computer as a calculator.

By contrast, a *compiler* is a program that translates an entire program into "machine code" patterns, and stores these patterns in memory, for later execution. So that the program (a colon-definition, in FORTH) can be found when it is to be executed, the compiler registers the word's name and the memory address of its machine code in the dictionary.

To execute a compiled program, you simply type in its word name. FORTH will take that name and search the dictionary for it. Then, upon finding the name, FORTH uses the accompanying address to locate and execute that particular name's machine code program. The compiler also performs the same kind of dictionary search each time it translates a colon-definition that includes that word name.

ADDING COMMENTS TO DEFINITIONS

If a colon-definition is to be printed out, for use by someone other than yourself (or by yourself, at a later date), it is worthwhile to *document* the definition, so it remains understandable.

FORTH allows you to add *comments* anywhere within the definition, by entering them in this format:

```
( comment)
```

Unlike most languages, the left parenthesis is a *word,* rather than simply a punctuation mark. This word, [(], which must be followed by at least one blank space, tells FORTH that the text between the left parenthesis and the right parenthesis—or a Return—represents a nonexecutable comment.

Comments should be used after a defining word name, to tell what the word does and what effect it has on the stack. Comments should also be used after most lines of the definition, to tell what that particular line is accomplishing.

If you studied the examples in the preceding chapters, you should already have a good idea of what kinds of comments should accompany a definition.

INCLUDING MESSAGES IN DEFINITIONS

All of the preceding programs in this book have been innocuous little sequences that accepted just a few parameters as "input," and produced just one or two numbers as "output." These kinds of programs require very little mental effort to interpret the display; the number that precedes FORTH's ubiquitous *OK* is inevitably the result.

With programs that produce more than a single-number result, or programs that must be interpreted by an untrained operator, things can get confusing. To help alleviate these problems, FORTH provides a word that outputs a user-defined message to the display and/or printer. This word, [."], requires the format

```
." your message here"
```

Being nothing more than printed comments, messages follow the rules of comments. That is, the beginning word [."] must be followed by at least one space, but no space is needed between the end of the message and the terminator ("). Messages can only be used within colon-definitions; they cannot stand alone.

The colon-definition

```
: PRINT-N**2      ( Print N squared )
  ( n --- )
    ." N SQUARED IS "
    SQUARE . ;
```

combines the previously defined word SQUARE with a message, to produce a more meaningful output. A sample run is:

```
3 PRINT-N**2 N SQUARED IS 9 OK
```

Well, this does the job, but the message is crammed onto the same line as the input value and the word name. Let's pretty-up the printout with a *carriage return* or two.

Carriage Returns Make Printouts More Readable

Realizing that FORTH programmers are tidy people who would want to format their output, the FORTH-79 Standards Team included a carriage return/line feed as a Required Word. This word, CR, causes the display screen cursor or the printer head to advance to the leftmost position of the next line.

Now we can add a couple of carriage returns to [PRINT-N**2] —and have it print out the input—to give this new definition:

```
: PRINT-N**2      ( Print N squared )
    ( n ---)
DUP CR.         ( Print N)
." SQUARES IS "   ( Message)
SQUARE . CR ;   ( Print N**2)
```

Now our printout is of the form:

```
3 PRINT-N**2
3 SQUARED IS 9
OK
```

TAKE INVENTORY WITH VLIST

As was mentioned in Chapter 1, if you ever want to see a list of all the words that are in your FORTH dictionary at any given time, simply type in the word

```
VLIST
```

and press Return.

Unlike the stack, which "grows" toward lower memory, the dictionary "grows" toward higher memory. Each word name that you add with a colon-definition will be recorded in the dictionary at a higher address than the previously defined word. Since VLIST starts at the top of the dictionary and works

downward, the listing will show the most recently defined word and work backwards, chronologically.

Most FORTHs provide the memory address along with the word name, so after defining SQUARE, CUBE, AND N**6, the beginning of the listing may look like this:

```
VLIST
  336  N**6
  325  CUBE
  316  SQUARE
  309  EDITOR
   .
   .          ( Continue with lower entries)
   .
```

FORGET A WORD TO DELETE IT

If you find that one of your recently defined words is no longer needed, or contains errors, you can delete that word from the dictionary with FORGET. The required format for FORGET is:

FORGET name

However, FORGET deletes not only the named word, but also every word that was defined thereafter. Therefore, if you decide to delete SQUARE, the command sequence

FORGET SQUARE OK

will delete CUBE and N**6 as well. In this case, the additional two deletions are desirable, since both include [SQUARE] in their definitions.

REDEFINING A WORD

When it comes to colon-definitions, FORTH does not differentiate between new words that are being added to the dictionary and those that are already in the dictionary. Besides allowing you to define new, uniquely named words, FORTH allows you to *redefine* existing words, if you so choose. If you enter a colon-definition for a name that is already in the dictionary, FORTH will issue a warning message of the form

name NOT UNIQUE OK

but (as the "OK" implies) the new definition is still entered into the dictionary.

Since FORTH itself provides no protection against multiple definitions (although some FORTH packages do), *any* word in the dictionary can be redefined. For instance, someone who's really weird may decide to redefine [+] as a subtraction word, via the definition

```
: + - ;
```

Thereafter, that person's FORTH system would produce operations such as

```
3 2 + 1 OK
```

Needless to say, you shouldn't allow a first-grader around such a person. (Come to think of it, please don't allow *me* around such a person!)

FORTH Definitions Are Cumulative

What about earlier words whose definitions *include* the newly redefined word? Are those words affected by the redefinition? No, they are not affected; they will operate exactly as they did before the redefinition! The reason for this is that FORTH definitions are *cumulative*. Once a word name is in the dictionary, its operation will not change unless that word itself is redefined.

What happens to an earlier definition when a word name is redefined? Nothing. The earlier definition remains in the dictionary—as you will see if you run a VLIST—and will again become the active definition if you ever FORGET the redefinition.

In practice, try to avoid redefining words (especially FORTH's Required Words) unless it's absolutely necessary. You are just courting disaster.

['] Discovers Whether a Word Is in the Dictionary

When in doubt, you can find out whether or not a word name is already in the dictionary without running a VLIST. The word that lets you do this is ['], pronounced "tick," which has the form

```
['] name
```

['] leaves the dictionary address of the named word on top of

the stack, so if we want to find out whether SQUARE is in the dictionary, the sequence

```
' SQUARE OK
. 316 OK
```

shows that SQUARE is in the dictionary, at address 316.

Incidentally, applying ['] or FORGET to a name that is not in the dictionary results in an *error*, which causes the stack to be cleared. Most FORTHs will also issue an error message, so if SQUARE were deleted, you would receive an indication such as

```
SQUARE ?
```

RENAMING A WORD

A word can be *renamed*, of course, which is entirely different than redefining it. If a word is renamed, the dictionary will contain two different names that have the same definition, and either name can be used in a subsequent definition. If a word is redefined, the dictionary will contain one word name that has two different definitions, of which only the most recently defined definition applies.

Renaming is almost exclusively used to make a FORTH program compatible with some other version of FORTH. For instance, in Chapter 3 we discussed a word that duplicates the top number on the stack only if that number is nonzero. This word is called [?DUP] in FORTH-79 and [−DUP] in fig-FORTH. Before a fig-FORTH program that contains [−DUP] can be run on a FORTH-79 compatible system, the definition

```
: −DUP ?DUP ;
```

must be added to the FORTH-79 dictionary.

THE DISK AND THE EDITOR

A program isn't very useful if it disappears when you turn off the power, so you must know how to preserve it on some "mass storage" media; either disk or tape. FORTH has words to communicate with both media, but since most FORTH packages come on disk, our discussion here will be limited to disk-related words.

Besides saving the new dictionary words (that is, the words that comprise your program), you will also want to save a copy

of the definition text for those words—the *source code*—in case you want to alter and recompile them later. Virtually all FORTH packages include a text-altering program called an *editor*. An editor is not a required element of the FORTH system, however, so neither the FORTH-79 Standard nor the fig-FORTH Installation Manual has any editor specifications. As a result, editor commands vary from one FORTH package to another. Since this is a general-purpose book, we will not describe the details of any particular package, but instead investigate features found in most FORTH editors and take a brief look at a typical editor command set.

DISK TERMINOLOGY

The easiest way to remember FORTH's disk-related words is to introduce them in the order in which they would be used. Before discussing disk operations, however, we must define some terms that will appear throughout these discussions.

In FORTH, data is transferred to and from the disk in *blocks*, where a block consists of 1024 bytes. The number of blocks that can be held on any one disk depends on the storage capacity of that particular disk. For example, a single 5¼-inch, 13-sector floppy diskette holds about 110 blocks of data.

Disks can also be used to save source programs—ASCII text you typed in using the editor. In this case, the FORTH literature refers to the 1024-byte mass storage unit as a *screen* rather than a block. Thus, you will encounter FORTH words that cause a "screen" to be compiled and others that cause a "block" to be executed. Screens are usually organized as 16 lines of 64 characters ($16 \times 64 = 1024$). However, some display screens are organized as 24 lines of 40 characters; these use only 960 of the screen's 1024 bytes.

Each block or screen being transferred to or from disk is maintained in an area of memory called a *block buffer*, which is also 1024 bytes long. Fig. 5-1 shows the transfer paths between disk and a computer terminal, via a block buffer.

Unlike most systems, FORTH automatically does all the buffer management, so you rarely need to worry about where a particular block is stored in memory—or whether it's even in memory at all! To access any given block, you simply "load" that block by specifying its number. If the block is already in memory, FORTH will immediately display the OK message. If the block is on disk, FORTH will read it into a block buffer, then display the OK message.

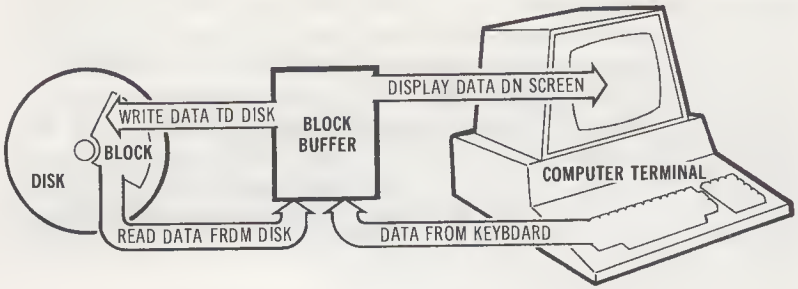


Fig. 5-1. Data transfers to and from disk.

How FORTH Manages Block Buffers

Some FORTH systems have many block buffers, others have as few as two. Associated with each block is a system identifier containing the number of the block and a flag that indicates whether the block has been modified (or *updated*) since being read into memory.

When you wish to load in a block, FORTH checks whether the block is in memory. If the block is in one of the buffers in memory, FORTH does nothing but issue the OK message. If the block is still on disk, FORTH will read it in to the most recently referenced block buffer. However, if that buffer has been updated, its contents are first written to disk before the new block is read in. This technique ensures that updated data in memory is never "clobbered" by new data. As you can see, then, the only difference between a system with many buffers and a system with two buffers is that the two-buffer system will be slower, because it will require more disk accesses.

DISK OPERATIONS

Now that you are familiar with the basic terminology of disk operations, we can introduce the FORTH words that transfer data to and from the disk. Rather than simply describe these words one by one, we will present a scenario of a typical programming session, from creating the *source program* to executing the compiled *object program,* and interject descriptions of the appropriate words in the order you would use them.

STANDARD SEQUENCE OF DISK OPERATIONS

There are nine fundamental steps for creating a FORTH program using disk. They are:

1. Empty all block buffers.
2. Select a screen to hold the new words.
3. Execute the editor program.
4. Type in the definitions for the new words.
5. Edit the text on this screen, as required.
6. Write the source program onto the disk.
7. Compile the source text from disk. This enters the new words into the dictionary.
8. Execute the new words.
9. If all definitions are correct, you are done. If not, list the appropriate screen text on the display and return to Step 3.

Emptying the Block Buffers

The first step here, emptying the block buffers, should be the initial step in *every* editing session. Just as pressing the Reset key on a computer keyboard puts the processor in a known state, emptying the block buffers puts those buffers in a known state. The FORTH word that does this,

```
EMPTY-BUFFERS
```

does not necessarily affect the contents of the block buffers (that's implementation-dependent), but marks each buffer as "empty," which tells the system "this buffer contains no useful information, and is available for storing disk data."

Selecting a Screen

With the block buffers initialized, we can now select one block on the disk as a "screen" to hold the text for our new definitions. The active screen is identified by the value of the user variable SCR, so screen 50 can be selected by entering the sequence

```
50 SCR !
```

Using an Editor

The words that allow us to enter text and alter that text are contained in the e*ditor program*. An editor has its own diction-

ary, separate from FORTH's main dictionary, and is invoked with the FORTH word EDITOR.

Editors differ from one system to another, but all editors have commands to select a screen, to move the cursor, and to add and delete specific lines. Most editors also allow you to change strings of characters within a line, so you don't have to retype the entire line. Table 5-2 lists the command for the editor shown in the fig-FORTH Installation Manual. Developed by William F. Ragsdale of the FORTH Interest Group, this editor is typical of those included in the popular FORTH packages on the market.

Note there is no EDITOR command in Table 5-2, nor is there a command to change the variable SCR. These commands are not listed because they are built into the definitions of LIST (which edits a previously created screen) and CLEAR (which creates a new screen). Some other FORTH words described in this section are also built into editor commands, but they will be covered separately for the sake of completeness.

The organization of text on a screen is, of course, up to you. However, *standard FORTH practice is to use the top line to hold a comment describing the contents of that screen.* For instance, the top line may read (DOUBLE-WORD MULTIPLY & DIVIDE) if the screen holds those word definitions. Comments are always ignored by the compiler, so you are encouraged to comment your programs liberally.

UPDATE Edited Block Buffers

Once you have completed editing an entire screen, you must "tell" FORTH that this screen has information that must be saved on disk the next time a disk transfer takes place. You can so mark the current screen as "saveable" by executing the word

```
UPDATE
```

(The editor in Table 5-2 automatically marks the screen as UPDATEd at the end of every editing command.)

Writing Blocks on Disk

The final command in any editing session is SAVE-BUFFERS (called FLUSH in fig-FORTH). This command takes all screens or blocks that have been marked as UPDATEd and writes them to disk. Since all buffers carry the screen/block number, there is no need to specify these numbers with the SAVE-BUFFERS command.

Putting the Source Program Into the Dictionary

At this point, the text (or *source program*) for the new words is saved on disk. To enter these new words into the FORTH dictionary, you must execute the word LOAD. For example, the sequence

```
50 LOAD OK
```

compiles the words in screen 50 into the dictionary, exactly as if you had typed them in from the keyboard. That is, once LOADed, the new words can be executed by typing in their names, as usual.

Making Changes to a Program

Because your source text is saved on disk, you can easily change the definitions and recompile them, if you ever wish to do so. To make these changes, LIST the source program on the screen, then invoke the editor once again to start a new editing session. For example,

```
50 LIST OK
```

will list the source program we just saved on disk. Before LOADing the updated source program, you may wish to FORGET the original definitions in the dictionary.

SOME MORE TEXT-PREPARATION WORDS

Many programs will occupy more than one screen. In standard FORTH-79, you can compile these programs with a series of LOADs. For instance,

```
50 LOAD 51 LOAD 52 LOAD OK
```

will cause screens 50, 51, and 52 to be loaded in from the disk and compiled into the dictionary. However, the Reference Word Set contains a Standard Word Definition that does the same job. This word,

```
-->
```

placed at the end of a screen, tells FORTH to continue interpretation at the next sequential screen. Therefore, if screens 50 and 51 are each terminated with [-->], all three screens (50, 51, and 52) will be compiled when you execute the sequence

```
50 LOAD OK
```

Table 5-2. A Typical Editor Command Set

Command	Description
Screen Editing Commands	
s LIST	List screen s and select it for editing.
s CLEAR	Clear screen s and select it for editing.
s1 s2 COPY	Copy screen s1 to screen s2.
L	Re-list the current screen. The cursor line is displayed after the screen listing, to show the cursor position.
FLUSH	Used at the end of an editing session to write all updated text to disk.
Cursor Movement Commands	
TOP	Move the cursor to the start of the screen.
c M	Move the cursor c characters to the right (if c is positive) or to the left (if c is negative).
Line Editing Commands	
P next	Used after LIST or CLEAR to input new text to screen s. Example: 0 P THIS IS HOW 1 P TO INPUT TEXT 2 P TO LINES 0, 1 AND 2 OF THE SCREEN.
n D	Delete line, but save it in the pad.
n T	Type line n and save it in the pad.
n R	Replace line n with the text in the pad.
n I	Insert the text from the pad at line n, moving the previous line n and subsequent lines down. Line 15 is lost.
n E	Erase line n with blanks.
n S	Spread at line n. Line n becomes blank; previous line n and subsequent lines move down. Line 15 is lost.
String Editing Commands	
F text	Find the next occurrence of the string *text*, starting at the current cursor position. If the string is found, the cursor is moved to the end of the string and the line is printed. If the string is not found, an error message is printed and the cursor is moved to the top of the screen.
B	Used after F to back up the cursor to the beginning of the text string, to prepare for editing this string.

Table 5-2.—cont. A Typical Editor Command Set

Command	Description
N	Used after F to find the next occurrence of the text string.
X text	Find and delete the next occurrence of the string *text*.
C text	Insert the string *text* at the current cursor position.
TILL text	Delete characters on the current line, from the cursor position to the end of the string *text*.
C DELETE	Delete c characters to the left of the cursor.

The Reference Word Set contains another compiler-controlling word, the uncontrolled word

```
;S
```

This word *stops* interpretation of a screen, often at the end of a [--->] sequence. The word [;S] can also be used to separate word definitions from comments at the end of a screen.

Finally, the uncontrolled word INDEX displays the first line (as mentioned previously, usually a comment) of a specified range of screens. For example, the sequence

```
50  52  INDEX
```

will cause the first lines of screens 50, 51, and 52 to be displayed. INDEX is useful for finding a screen whose number you don't remember.

fig-FORTH DEFINITION WORDS AND DISK WORDS

Except for EDITOR, fig-FORTH has all the words summarized in Table 5-1. In fig-FORTH, SAVE-BUFFERS is called FLUSH.

In addition, fig-FORTH has a variable called FENCE that prohibits FORGETting below a user-specified word in the dictionary. For example, to protect the word MYWORD and its predecessors, execute this sequence:

```
' MYWORD FENCE !
```

to store the address of MYWORD into the variable FENCE. Thereafter, any attempt to FORGET the word MYWORD, or a word defined previous to MYWORD, will produce an error condition. If you ever *do* wish to permit FORGETting below the

MYWORD level, you will need to change the value of FENCE. To eliminate the FENCE effect entirely, execute

```
0 FENCE !
```

Table 5-3. Words Added to FORTH in Chapter 5

Word	Stack	Action
SQUARE	n --- n**2	Squares a number.
CUBE	n --- n**3	Cubes a number.
N**6	n --- n**6	Raises a number to the sixth power.

SUMMARY

This chapter described how to add words to the FORTH dictionary using colon-definitions, including some remarks about comments and messages. We also discussed the implications of redefining, renaming and deleting words, and showed how word definitions could be saved on disk.

Table 5-3 summarizes words defined in this chapter that can be added to FORTH.

Table 2-2. fig-FORTH Arithmetic Words

Word	Stack	Action
+	n1 n2 --- sum	Adds two numbers.
1+	n --- n+1	Adds 1 to number.
2+	n --- n+2	Adds 2 to number.
D+	d1 d2 --- dsum	Adds two double numbers.
–	n1 n2 --- diff	Subtracts n2 from n1.
*	n1 n2 --- prod	Multiplies numbers n1 and n2, leaving single-precision product.
M*	n1 n2 --- dprod	Multiplies numbers n1 and n2, leaving double-precision product.
U*	un1 un2 --- ud	Multiplies unsigned numbers n1 and n2, leaving double-precision product.
/	n1 n2 --- quot	Divides n1 by n2, leaving quotient.
MOD	n1 n2 --- rem	Divides n1 by n2, leaving remainder with sign of n1.
/MOD	n1 n2 --- rem quot	Divides n1 by n2, leaving remainder and quotient.
M/	d n --- rem quot	Divides double dividend by single divisor, leaving single remainder and quotient.
U/	ud un --- rem quot	Divides unsigned double dividend by unsigned single divisor, leaving single remainder and quotient.
M/MOD	ud un --- rem dquot	Same as [U/], but leaves single remainder and double quotient.
*/	n1 n2 n3 --- quot	Multiplies n1 by n3, then divides the result (a 32-bit intermediate product) by n3, leaving a single quotient.
*/MOD	n1 n2 n3 --- rem quot	Same as */ , but leaves both quotient and remainder.
MINUS	n --- –n	Reverses the sense of a number, leaving its two's complement.

As you know, both PICK and ROLL based their operation on a "stack-depth" index at the top of the stack. If the index was 2, the second number was copied or rotated; if the index was 3, the third number was copied or rotated, and so on. If we think of a number as a 16-bit value (which it is), and a 32-bit double number as two 16-bit values (high-order 16 bits and low-order 16 bits), the problem of copying or rotating a double number becomes a matter of applying PICK or ROLL *twice,* once for each half of the double number.

The double number copy and exchange operations we'll be giving here also begin with having an index on the top of the stack, but in this case the index will identify the *double number* to be operated on. Fig. 3-1 shows the relationship between number indexes and double number indexes. As you can see, for any double number index *d,* the single number indexes *n* and *n+1* can be derived using these equations:

$$n = 2d - 1 \text{ and } n+1 = 2d$$

However, if we apply these equations directly to access a double number, we will be "off" by one stack value, because they don't account for the fact that there are *two* indexes at the top of the stack, rather than just one. To accommodate this difference, just add one to each equation, to give:

$$n = 2d \text{ and } n+1 = 2d + 1$$

At this point we are ready to look at two program sequences, one that will copy the nth double number onto the top of the stack (a 2PICK sequence, if you will) and another that will rotate the nth double number onto the top of the stack (a 2ROLL sequence). These sequences are shown in Examples 3-1 and 3-2, respectively. In a later chapter we will show how sequences such as these can be defined as words; they are put in the dictionary to be thereafter used like the predefined words.

Incidentally, the text enclosed in parentheses in these examples is comments, to describe what's taking place. This is the standard way comments are shown in FORTH. Note, too, that there is a space between the left parenthesis and the beginning of the comment. *This space is required,* because [(] is actually a Required Word, rather than simply a punctuation mark! No space is required between the end of the comment and the right parenthesis, however.

These examples also illustrate a good FORTH programming practice: *keep the lines short!* Example 3-1 could have been written like this:

```
2 * DUP 1+ PICK SWAP PICK
```

INCREMENT A NUMBER IN MEMORY

In Chapter 2 we discussed adding values to numbers on the stack, using the words [1+] and [2+]. FORTH offers a similar, but more versatile word called [+!], which adds a number on the stack to a number in memory. For example,

```
5 200 +!
```

adds the value 5 to the number stored at address 200. Similarly,

```
-5 200 +!
```

subtracts 5 from the number stored at address 200.

Other Arithmetic Operations on Memory

Clearly, it is also possible to perform a multiplication or a division on a number in memory, by simply fetching the number onto the stack, operating on it, then returning the result to memory. Before fetching the number, however, you must duplicate the address, so that it is available for the store operation.

The normal sequence for a multiply or divide operation on memory is:

```
DUP @      ( Fetch the number from memory)
ROT        ( Rotate operand to top of stack)
( Perform the operation)
SWAP       ( Put the address on top)
!          ( Return the result number to memory)
```

Adding a value to a double number in memory is somewhat more complex, though, due to the amount of stack manipulation that is required. Example 4-3 shows a sequence that will do the job, along with a "snapshot" of the stack after the memory contents have been fetched. The snapshot includes a number on each level of the stack, which will help you understand what the first ROLL is doing. You should be able to follow the sequence from there.

MOVE A BLOCK OF DATA IN MEMORY

Many applications require blocks of information to be copied from one part of memory to another. For instance, data processing often involves copying a table of numbers from one file

3. The sequence of previously defined *words* that tell what operation the new word is to perform. The word(s) used here can be drawn from the dictionary that came with the FORTH package, or from words that you have previously defined (and thereby added to the dictionary) using the procedure we are describing.
4. The FORTH word [;], pronounced "semicolon."

Some Simple Examples

To see how a new word can be defined, let's assume that we want to have a word that squares the number at the top of the stack. From Chapter 3 we know that squaring a number can be performed with the sequence

```
DUP *
```

Therefore, if we call our new word SQUARE, its *colon-definition* (the standard FORTH nomenclature) would be

```
: SQUARE DUP * ;
```

After typing in this line and pressing Return, SQUARE is in the dictionary, ready to be used. To verify that it works, you can try sequences such as these:

```
2 SQUARE . 4 OK
-8 SQUARE . 64 OK
17 SQUARE . 289 OK
```

With SQUARE now defined and compiled (in the dictionary), it can serve as a building block for more new words. For instance, the definition of a word that cubes a number on the stack might be

```
: CUBE DUP SQUARE * ;
```

Applying CUBE to our three preceding examples:

```
2 CUBE . 8 OK
-8 CUBE . -512 OK
17 CUBE . 4913 OK
```

Now *both* words can be used to raise a number to the sixth power, with the colon-definition

```
: N**6 SQUARE CUBE ;
```

which allows us to do this:

```
2 N**6 . 64 OK
```

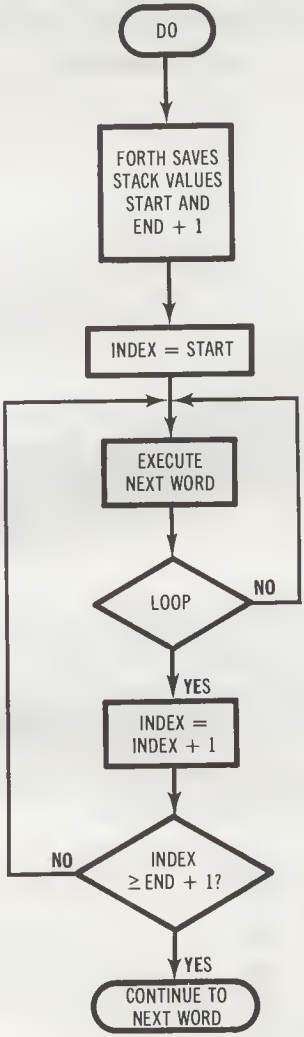


Fig. 6-1. DO-LOOP operation.

index is used to number the printout. The colon-definition for this new word, PRINT8-NUMBERED, is:

```
: PRINT8-NUMBERED
   9 1 DO CR I . . LOOP ;
```

Now, if the stack holds the same eight numbers as before, 11 through 18, PRINT8-NUMBERED will generate this printout:

```
PRINT8-NUMBERED
1 18
2 17
3 16
4 15
5 14
6 13
7 12
8 11 OK
```

The Index As An Operand

The index value can also be used as an operand within a DO-LOOP. For instance, in Chapter 2 we presented this equation for the circumference of a circle:

$$c = \pi d = \frac{355^{*}d}{113}$$

and showed how the word [*/] could be used to make such a calculation. Example 6-1 formalizes the FORTH calculation sequence as a colon-definition for the word CIRCUM, and includes a colon-definition for a second word, PRINT-CIRCUM, which prints a list of diameters and corresponding circumferences.

For example, we can get a list of circumferences for diameters between 60 and 69, as follows:

```
70 60 PRINT-CIRCUM
60 188
61 191
62 194
63 197
64 201
65 204
66 207
67 210
68 213
69 216 OK
```

Indent Lines Within a DO-LOOP

You should note that the line

```
CIRCUM .
```

Example 6-1. Circumference of a circle

```
: CIRCUM
    ( Calculate circumference of a circle from its diameter )
    ( Stack: diam --- circum )
    355 113 */ ;

: PRINT-CIRCUM
    ( Print a list of diameters and their corresponding )
    ( circumferences, incrementing diameter by one each time. )
    ( Stack inputs end+1 and start are the first diameter and )
    ( the last diameter plus one. )
    ( Stack: end+1 start --- )
    DO CR I DUP .          ( Print diameter)
     CIRCUM .              ( Print circumference )
    LOOP ;
```

in the PRINT-CIRCUM definition in Example 6-1 is *indented* one space to the right of DO. This is the standard way of delineating the boundaries and range of influence of DO-LOOPs and other control structures, and shows clearly which instructions are included in the structure. As a matter of good programming practice, you should get into the habit of aligning the DO and LOOP lines, and indenting all lines between these two words.

+LOOP ADDS ANY NUMBER TO THE INDEX

FORTH provides an alternate type of DO-LOOP that allows the index to be incremented or decremented by *any* number value, not just +1. Like the standard DO-LOOP, this type,

limit start DO . . . n +LOOP

uses the top number on the stack (*start*) as the initial value of the index, but changes the index by *n* each time +LOOP is encountered. A DO−+LOOP stops looping when the index equals or exceeds *limit*.

Put more concisely, *if n is positive,* limit must be greater than start and looping continues until the index has been increased to a value equal to or greater than limit. *If n is negative,* limit must be less than start and looping continues until the index

has been increased to a value equal to or less than limit. Thus, a sequence of the form

```
10 0 DO . . .  2 +LOOP
```

will cause the enclosed operation (shown as ". . ." here) to be repeated five times, with index values of 0, 2, 4, 6, and 8, respectively. Similarly, a sequence of the form

```
0 10 DO . . .  -2 +LOOP
```

will also cause the specified operation to be repeated five times, but with index values of 10, 8, 6, 4, and 2, respectively.

Clearly, the +LOOP form is useful for applications in which certain index values are to be skipped, because it gives you every "nth" value and bypasses all others. For instance, our PRINT-CIRCUM word could be redefined to print out a list of circumferences in which the diameter increases by two with each loop. This is shown in Example 6-2.

Example 6-2. Print alternate circumferences

```
: PRINT-CIRCUM-2
    ( Print a list of diameters and circumferences, increasing )
    ( the diameter by two each time. )
    ( Stack: limit start --- )
    DO CR I DUP .        ( Print diameter )
     CIRCUM .            ( Print circumference )
     2                   ( Add two to index )
    +LOOP ;
```

Now, if we execute PRINT-CIRCUM-2 for diameters between 60 and 70, we get:

```
70 60  PRINT-CIRCUM-2
60 188
62 194
64 201
66 207
68 213 OK
```

The upper limit diameter, 70, was not processed because at that point the index equaled *limit*.

+LOOP IS USEFUL FOR MEMORY OPERATIONS

Operations on consecutive numbers or double numbers in memory often employ the +LOOP variant to increment the address. Since each number has an address that is two greater than the preceding number, and each double number has an address that is four greater than the preceding double number, consecutive items can be referenced by applying an increment of two or four to +LOOP.

Example 6-3 defines the word AVG-MEMORY, which takes the average of the numbers contained between addresses addr1 and addr2. This sequence begins by calculating a byte count (addr2 − addr1) and converting it to a number count. At this point the stack contains (addr1 count), where count is the number that will become the divisor in the averaging calculation.

Knowing that we will need to add "count" numbers that are two bytes apart, we form the DO-LOOP's upper limit by multi-

Example 6-3. Average the numbers in memory

```
: PICK
    ( Leove copy of n1-th number on top of the stock )
    ( n1 --- n2 )
    2 *              ( Address index = 2*n1)
    SP@ +            ( Address of n2 = top of stock + index)
    @ ;              ( Fetch n1-th number )

: AVG-MEMORY
    ( Average the numbers contained in memory, )
    ( from oddr1 to oddr2.                     )
    ( oddr1 addr2 --- overoge )
    OVER −           ( Colculote byte count)
    2 / 1+           (  ond convert it to number count)
    DUP 2 *          ( DO-LOOP limit = 2*count)
    0                ( To stort, totol = 0)
    SWAP 0           ( Put DO-LOOP limits on top)
    DO
     3 PICK I +      ( Address = oddr1 + I)
     @ +             ( Add next number to totol)
     2               ( I = I + 2)
    +LOOP
    SWAP /           ( Average = totol/count)
    SWAP DROP ;      ( Delete oddr1)
```

plying count by two (DUP 2 *). Then, after initializing the "total" to zero, the stack holds the values (addr1 count limit 0). With the line (SWAP 0), we put the DO-LOOP parameters limit and start (0) on top of the stack, and we're ready to execute the DO-LOOP.

The DO-LOOP removes the top two numbers, limit and 0, from the stack, then performs this sequence:

1. Copy addr1 and add I to it, to form the effective address of a number (3 PICK I +).
2. Fetch the number from memory and add it to the total (@ +).
3. Increment I by two and compare it to limit.
4. If I equals limit, terminate the loop; otherwise, return to Step 1.

Upon completion of this loop, the stack will hold the values (addr1 count total). The average is obtained by swapping count and total, then performing a divide. To "clean up" the stack, addr1 is deleted with (SWAP DROP), leaving just the average on the stack.

The word PICK is a Required Word in FORTH-79, but is not included in fig-FORTH.

NESTED DO-LOOPS

DO-LOOPs can also be placed *inside* other DO-LOOPs. That is to say, DO-LOOPs can be *nested*. Nesting allows you to execute a series of repetitive operations in an "inner" DO-LOOP each time the "outer" DO-LOOP is executed.

Moreover, the two DO-LOOP types just mentioned can be nested in any combination, so it is possible to imbed a [DO . . . LOOP] within a [DO . . . +LOOP], or a [DO . . . +LOOP] within another [DO . . . +LOOP], or whatever.

To delineate the two DO-LOOPs, the inner loop is usually indented one space to the right, so that on paper a nested DO-LOOP structure should have the overall appearance of Fig. 6-2. DO-LOOPs can be nested to any practical level, but it's doubtful whether you will ever need to nest them deeper than three or four levels.

Indexes for Nested DO-LOOPs

Recall that the word I leaves the current value of a DO-LOOP index on the top of the stack. In nested DO-LOOPs, I will leave the index of the loop in which it appears.

```
: NESTED-DO-LOOPS
  limit1 start1          ( Limits for outer loop)
  DO . . .               ( Start of outer loop)
   . . .
   . . .
    limit2 start2        ( Limits for inner loop)
    DO . . .             ( Start of inner loop)
     . . .
     . . .
     n2
    +LOOP                ( End of inner loop)
   . . .
   . . .
   n1
  +LOOP                  ( End of outer loop)
```

Fig. 6-2. Nested DO-LOOPS.

FORTH provides a similar word, J, that leaves the index of the next *outer* loop. It is used within nested DO-LOOPs of the form

DO . . . DO . . . J . . . LOOP . . . LOOP

To illustrate nested DO-LOOPs, Example 6-4 defines the word SALES-TAXES, which prints taxes at four different rates (4, 5, 6, and 7%) over a range of dollar amounts, in $20.00 increments. For example, the input sequence

```
200 100 SALES-TAXES
```

will produce this printout:

```
100 4 5  6  7
120 4 6  7  8
140 5 7  8  9
160 6 8  9 11
180 7 9 10 12 OK
```

LEAVE TERMINATES A DO-LOOP

You can force a DO-LOOP to terminate at any time with the word LEAVE, which sets the loop limit equal to the current value of the index. Since LEAVE is usually executed as the re-

Example 6-4. Print sales taxes at four rates

```
: SALES-TAXES
    ( Print sales taxes at 4, 5, 6, and 7% far a range af )
    ( amaunts, in $20 increments.                         )
    ( high-amaunt law-amaunt --- )
    DO
     CR I .            ( Print amaunt)
     8 4
     DO
       J I 100 */ .    ( Calculate tax)
       .               ( and print it)
     LOOP
     20                ( Increment by 20)
    +LOOP ;
```

sult of some conditional test, we will postpone discussing it further until Chapter 7.

THE RETURN STACK

The stack we have been referring to throughout this book is called the *data stack* (or sometimes the computation stack) in the FORTH literature. Every FORTH system contains a second stack, called the *return stack,* for keeping track of DO-LOOP limits and for various other "housekeeping" tasks.

The return stack, like the data stack, operates on a "last-in, first-out" basis. The last number pushed onto the return stack will be the first number pulled from it. Moreover, the return stack, like the data stack, grows toward low memory; each number pushed onto the stack will be stored at a lower-valued address than its predecessor.

For DO-LOOP operations, FORTH uses the return stack to hold boundary values. For example, Fig. 6-3A shows how these values are arranged on the return stack for a single (unnested) DO-LOOP application. At the end of each loop, the top value ("start," initially) is incremented and then compared to the second value ("limit"), to determine whether looping is to be continued or terminated.

Fig. 6-3B shows the return stack arrangement for a nested DO-LOOP application. The bounds of the inner (I) loop are on top, followed by the bounds of the outer (J) loop.

(A) Single DO-LOOP. *(B) Nested DO-LOOPS.*

Fig. 6-3. DO-LOOP bounds on the return stack.

Manipulating the Return Stack

Although the return stack is intended for the use of the FORTH system software, you can also make use of it from time to time—*but* you should do so with utmost caution, to avoid crashing the system!

Table 6-2 summarizes the FORTH words that can be used to manipulate the return stack. Also included are some alternate descriptions of the DO-LOOP index words I and J—and one new word, I'—related to the internal organization of the return stack.

Table 6-2. Return Stack Manipulation Words

Word	Stack	Action	Notes
>R	n ---	Transfers n from the data stack to the return stack.	
R>	--- n	Transfers n from the return stack to the data stack.	
R@	--- n	Copies the top number of the return stack onto the data stack.	
I	--- n	Copies the top number of the return stack onto the data stack. Same as R@.	
I'	--- n	Copies the second number of the return stack onto the data stack.	(1)
J	--- n	Copies the third number of the return stack onto the data stack.	

Note: (1) Included in Reference Word Set, as an uncontrolled word definition.

The word >R transfers the top number on the data stack to the return stack. Conversely, the word R> transfers the top number on the return stack to the data stack. A third word, R@ (called R in fig-FORTH), makes a copy of the top number on the return stack, and copies it onto the data stack.

The return stack is particularly useful for holding numbers from the top of the data stack while you operate on a number that was previously "buried" in the data stack. Example 6-5 demonstrates this technique in a fig-FORTH definition for the FORTH-79 word ROLL. As you will recall from Chapter 3, ROLL rotates any number in the data stack to the top of the stack.

Example 6-5 may look complex, but isn't. It simply involves moving all but the nth number (the ROLL "target") to the return stack, then bringing these numbers, one by one, back to the data stack and swapping them with the target number. The added return stack instructions are required because, in each case, numbers moved to the return stack must be stored beneath the DO-LOOP limits (which are at the top of the return stack).

Example 6-5. ROLL, in fig-FORTH

```
: ROLL
    ( Rotate nth number to top of stack )
    ( n --- (n))
    DUP 1 =
    IF
     DROP            ( If 1 ROLL, you are done)
     ELSE
      DUP 1
      DO
       SWAP              ( Otherwise put next number on top)
       R> R>             ( Fetch loop limits from return stack)
       ROT >R            ( Move next number to return stack )
       >R >R             (  then return loop limits          )
      LOOP
      1
      DO
       R> R>             ( Fetch loop limits)
       R>                (  and previously saved number)
       ROT ROT >R >R     ( Return loop limits)
       SWAP              ( Put ROLL target on top)
      LOOP
     THEN ;
```

fig-FORTH DO-LOOP AND RETURN STACK MANIPULATION WORDS

Table 6-3 summarizes the fig-FORTH words associated with DO-LOOPs and those that are used to manipulate the return stack. As you can see, fig-FORTH includes all of the FORTH-79 Required Words except J. As mentioned earlier, R in fig-FORTH is identical to R@ in FORTH-79.

Table 6-3. fig-FORTH DO-LOOP and Return Stack Manipulation Words

Word	Stack	Action
DO LOOP	end+1 start ---	Used in a colon-definition in the form DO . . . LOOP to begin a loop which will terminate when a loop index equals or exceeds end+1. The loop index begins with the value start, and is incremented by one each time LOOP is encountered.
DO +LOOP	limit start ---	Used in a colon-definition in the form DO . . . +LOOP to begin a loop which will terminate when a loop index equals or exceeds limit. The loop index begins with the value start, and is incremented by n each time +LOOP is encountered.
I	--- index	Copies the current loop index onto the stack. May only be used in the form DO . . . I . . . LOOP or DO . . . I . . . +LOOP
LEAVE		Forces a DO-LOOP to terminate at the next LOOP or +LOOP, by setting the loop limit equal to the index.
>R	n ---	Transfers n from the data stack to the return stack.
R>	--- n	Transfers n from the return stack to the data stack.
R	--- n	Copies the top number of the return stack onto the data stack.
R0	--- addr	Points to the address of the bottom of the return stack.
RP !		A user-supplied procedure to initialize the return stack pointer from R0.

Example 6-6 is the definition of the missing word, J. From Fig. 6-3B, you would expect to pull just two numbers—the inner loop bounds—off the return stack in order to make the outer loop index (that is, J) available for reading, but this definition pulls *thr*ee numbers off the return stack. This extra pull (R>) is needed because when FORTH encounters the new word J in a colon-definition, it saves a return address at the top of the return stack while it looks up and executes J's definition in the dictionary. Thus, the first R> removes the return address from the return stack, the next two R>'s remove the inner loop bounds, and then R reads the outer loop index, J.

Example 6-6. Return the index of the next outer loop in a nested DO-LOOP

```
: J
    ( Return index of the next outer loop of a nested DO-LOOP )
    ( --- n )
    R>              ( Fetch return address)
    R> R>           ( Fetch I loop limits)
    R               ( Read J index)
    SWAP >R         ( Restore I loop limits)
    SWAP >R         (  and return address )
    SWAP >R ;       (  on return stack    )
```

Two additional return stack words, R0 and RP! are also contained in fig-FORTH. R0 returns a *pointer* to the address of the bottom of the stack. That is, it returns the address of a location in memory, and that location holds the address of the bottom of the return stack. The second word, RP!, initializes the return stack pointer using the address pointed to by R0. Note that these words are return stack equivalents of the data stack words S0 and SP!, described in Chapter 3 (Table 3-2).

SUMMARY

In this chapter we studied a repeating control structure called a DO-LOOP. Included were discussions of the two basic forms of DO-LOOPs: (1) DO . . . LOOP, which increments the loop count by one, and (2) DO . . . +LOOP, which increments the loop count by a specified number. We also discussed how

Table 6-4. Words Added to FORTH in Chapter 6

Word	Stack	Action
CIRCUM	diam --- circum	Return circumference of a circle.
PRINT-CIRCUM	end+1 start ---	Print diameters and circumferences.
PRINT-CIRCUM-2	limit start ---	Print alternate diameters and circumferences.
AVG-MEMORY	addr1 addr2 --- average	Average the numbers in memory, from addr1 to addr2.

to access the index value (I, J, or K) and how DO-LOOPs can be nested, to imbed DO-LOOPs within other DO-LOOPs.

This chapter also introduced the *return stack*. Although primarily intended to hold DO-LOOP boundaries and other system information, the return stack is usable (with caution) as temporary storage for user parameters.

Table 6-4 summarizes words defined in this chapter that can be added to FORTH. The next chapter will discuss additional kinds of control structures, those that base their operation on some pretested condition.

CHAPTER 7

Conditional Control Structures

There are many applications in which you will want to control the actual execution path of a program, by having some sequence of words executed in one situation and another sequence of words executed in some other situation. For instance, if a program is controlling the sprinkler system around your home, you may want the lawn sprinklers turned on every day, the garden sprinklers turned on every other day, and none of the sprinklers turned on if it is raining. (From what I've seen, many sprinkler controllers lack the latter feature!) That is, you would want the program to make a "decision," based on one or more conditions. The control structures described in this chapter provide that decision-making capability.

Table 7-1 summarizes the control structures we'll cover in this chapter. This table also includes a summary of the *comparison words* that provide the basis upon which a control structure makes its execution "decision." Since a comparison word always precedes a conditional control structure, let's begin by describing FORTH's comparison words.

COMPARISON WORDS

FORTH's conditional control structures base their "decisions" on a true/false indicator at the top of the stack. That is, these control structures operate one way if the indicator is "true" and another way if the indicator is "false." This indicator, called a *flag*, is considered to be true if it has a value of 1 and false if it has a value of 0.

The FORTH words that govern the state of a flag are called *comparison words*, because they set the flag to either true or false based on the result of a comparison operation. These

Table 7-1. Comparison Words and Conditional Control Structures

Word	Stack	Action	Notes
0<	n --- flag	Sets flag true if n is less than zero.	
0=	n --- flag	Sets flag true if n is equal to zero.	
0>	n --- flag	Sets flag true if n is greater than zero.	
<	n1 n2 --- flag	Sets flag true if n1 is less than n2.	
=	n1 n2 --- flag	Sets flag true if n1 is equal to n2.	
>	n1 n2 --- flag	Sets flag true if n1 is greater than n2.	
<>	n1 n2 --- flag	Sets flag true if n1 is not equal to n2.	(2)
U<	un1 un2 --- flag	Sets flag true if un1 is less than un2. Both are unsigned numbers.	
D0=	d --- flag	Sets flag true if double number d is equal to zero.	(1)
D<	d1 d2 --- flag	Sets flag true if d1 is less than d2.	
D=	d1 d2 --- flag	Sets flag true if d1 is equal to d2.	(1)
DU<	ud1 ud2 --- flag	Sets flag true if ud1 is less than ud2. Both are unsigned double numbers.	(1)
NOT	flag1 --- flag2	Reverses the value of a flag. This is identical to 0=.	
?DUP	n --- n (n)	Duplicates n if it is nonzero.	

BEGIN
UNTIL
 	
WHILE
REPEAT
	
	UNTIL: flag ---
	
	WHILE: flag ---
		Used in a colon-definition in the form:
		BEGIN . . . flag UNTIL or
		BEGIN . . . flag WHILE . . . REPEAT
		A BEGIN-UNTIL loop will be repeated until flag is true. A BEGIN-WHILE-REPEAT loop will be repeated until flag is false.	

END		A synonym for UNTIL.	(3)
AGAIN		Used in a colon-definition in the form:	(3)

Table 7-1—cont. Comparison Words and Conditional Control Structures

Word	Stack	Action	Notes
		BEGIN . . . AGAIN to force an unconditional jump back to BEGIN.	
IF ELSE THEN	IF: flag ---	Used in a colon-definition in the form: flag IF . . . (true). . . THEN or flag IF . . . (true). . . ELSE . . . (false). . . THEN If flag is true, the words following IF are executed and the words following ELSE are skipped. If flag is false, words between ELSE and THEN are executed and the words between IF and ELSE are skipped. IF-ELSE-THEN conditionals may be nested.	
IFTRUE OTHER- WISE IFEND	IFTRUE: flag ---	Used during interpretation in the form: flag IFTRUE . . . (true). . . OTHERWISE . . . (false). . . IFEND These conditional words operate like IF-ELSE-THEN, except that they cannot be nested.	(2)
LEAVE EXIT		Forces a DO-LOOP to terminate at the next LOOP or +LOOP, by setting the loop limit equal to the index. When compiled within a colon-definition, forces the definition to terminate at that point. May not be used within a DO-LOOP.	

Notes: **(1) Included in Double Number Extension Word Set.**
(2) Included in Reference Word Set, as an uncontrolled word definition.
(3) Included in Reference Word Set, as a Standard Word Definition.

comparison operations are essentially subtractions, except that they produce a qualitative result—1 or 0 (true or false)—rather than a quantitative result. In this way, comparison words operate like teachers who grade students with either "pass" or "fail" instead of with a letter grade or a number grade.

As you can see from Table 7-1, there are two types of comparison words: those that compare the top item of the stack with zero and those that compare the second item on the stack (n1) with the top item on the stack (n2). In all cases, the operation removes the compared words from the stack and replaces them with the flag value. For example, the sequence (3 2 >) leaves a flag of 1, since 3 is greater than 2, whereas (2 3 >) leaves a flag of 0, since 2 is not greater than 3.

Incidentally, you may wonder why the word ?DUP is included in Table 7-1, since it is neither a flag-related word nor a conditional control structure. The word ?DUP, called −DUP in fig-FORTH, is relevant here because it is primarily used to preserve a nonzero stack value before that value is compared with zero ([0<], [0=] or [0>]).

BEGIN-UNTIL LOOPS

BEGIN-UNTIL loops are simply DO-LOOPs in which the number of repetitions is unspecified. They will repeat an operation continually until some predefined condition occurs.

BEGIN-UNTIL loops come in two forms:

```
BEGIN . . . flag UNTIL
BEGIN . . . flag WHILE . . . REPEAT
```

where *flag* is a true/false indicator on the top of the stack. The form BEGIN-UNTIL repeats the operation between BEGIN and UNTIL until the flag becomes *true*. The form BEGIN-WHILE-REPEAT repeats the operation between BEGIN and REPEAT until the flag becomes *false* (at which time, FORTH, continues at whatever word follows REPEAT).

As you can see, a BEGIN-UNTIL loop always executes an operation at least once, because its flag is not tested until the end of the operation. However, a BEGIN-WHILE-REPEAT loop allows you to bypass the main operation (the one enclosed by WHILE and REPEAT) entirely, and *never* execute it, because the flag is tested prior to the operation, at WHILE.

To see how BEGIN-UNTIL might be used, consider a FORTH program that controls a chemical plant, and must monitor a

mixing tank that is being filled with a chemical solution. If VALVE-ON and VALVE-OFF are words that control the filler valve, and FULL? is a word that sets a flag to true when the tank is full, you might expect to see this sequence in the program:

```
..                ( Previous operations)
..
VALVE-ON          ( Turn valve on)
BEGIN
  FULL?           ( Wait for tank to fill,)
UNTIL
VALVE-OFF         ( then turn valve off )
..                ( Continue the program)
..
```

Derive a Square Root with BEGIN-UNTIL

BEGIN-UNTIL loops are also ideal for making repetitive calculations that converge to a desired result. A case in point is Newton's method for calculating the square root of a number. Essentially, Newton's method states: If *A* is an approximation for the square root of *n*, then

$$A1 = (n/A + A)/2$$

is a better approximation. That is, each time a new approximation is calculated using the preceding equation, you will get closer and closer to the actual value of the square root.

Writing a program to calculate the approximations is a simple task. All you need to do is start with some arbitrary initial approximation (the value 1 is as good as any), then use a BEGIN-UNTIL loop to grind out subsequent approximations. The only problem is knowing when to stop. We could have the program make some given number of iterations, but any number we choose may be too few for large values or too many for small values. An alternate approach—the one we'll adopt here—is to continue looping until a calculation produces the same result as the preceding calculation, because from that point the approximations will not change.

Example 7-1 shows the colon-definition for SQRT, a word that calculates the square root of a number using Newton's method. All of the OVERs make the program look somewhat complex, but it's quite easy to understand if you draw a few stack pictures as you go along. (That's a*lways* a good approach with FORTH programs!)

Just prior to the BEGIN-UNTIL loop, the stack holds the base number and the first approximation (un 1). The next approximation, A1, is begun by copying un and A onto the top of the stack (OVER 0 3 PICK), with a 0 to make un a double number, then dividing A into n (U/MOD) and dropping the remainder (SWAP DROP). The second term, A, is added to the result (OVER +) and the sum averaged (0 2 U/MOD). After dropping this second remainder (SWAP DROP), there are three values on the stack (un A A1) and we are prepared to check whether the top two, A and A1, are equal.

The comparison word we will be using, = , always destroys the top two numbers on the stack and replaces them with a flag value. Knowing that, should we duplicate both A and A1? No, we only need to duplicate A1, because the previous approximation (A) is no longer needed, regardless of the compare result. The operation (SWAP OVER =) moves A to the top, puts a

Example 7-1. Square root, using Newton's method

```
: SQRT
    ( Calculate the square root of an unsigned number using )
    ( Newton's method.                                       )
    ( un --- sqrt(un) )
    1                        ( To begin, assume A = 1)
    BEGIN
      OVER 0 3 PICK U/MOD    ( A1 = (n/A + A)/2 )
      SWAP DROP
      OVER + 0 2 U/MOD
      SWAP DROP
      SWAP OVER =            ( Test for A1 = A)
    UNTIL
    SWAP DROP ;              ( Delete un)
```

copy of A1 on top of it, and replaces the two with a true or false flag.

If A and A1 had different values, UNTIL removes the flag and transfers control back to the line following BEGIN. Otherwise, UNTIL removes the flag and allows control to "drop through" to the last line (SWAP DROP), where un is deleted and only the final square root value remains on the stack.

Infinite BEGIN-UNTIL Loops

Some FORTHs, including fig-FORTH, provide the form

BEGIN . . . AGAIN

in which AGAIN forces the processor to return to BEGIN after each loop, unconditionally. Thus, the loop repeats until either a reset or an interrupt occurs. BEGIN-AGAIN loops are nearly always used to produce a wait state, in which the processor waits for an interrupt from some external device in the system. In FORTH-79, you can produce the [AGAIN] function with [0 UNTIL].

IF-THEN CONTROL STRUCTURES

The next kind of control structure, IF-THEN, also comes in two different forms:

```
flag IF . . . (true) . . . THEN
flag IF . . . (true) . . . ELSE . . . (false) . . . THEN
```

With both forms, the IF takes a flag from the top of the stack and makes an execution decision based on whether the flag is true or false.

If the flag is true, an IF-THEN structure will execute the words between IF and THEN, and an IF-ELSE-THEN structure will execute the words between IF and ELSE. *If the flag is false*, an IF-THEN structure will simply skip the words between IF and THEN, but an IF-ELSE-THEN structure will execute the words between ELSE and THEN before continuing on in the program.

For example, the sequence

```
10 >
IF
  CR . " Number is greater than 10."
THEN
```

prints the message only if the number at the top of the stack was greater than 10 (the past tense "was" applies here because comparison words delete the operands from the stack).

However, the sequence

```
10 > CR
IF
  . " Number is greater than 10."
ELSE
  . " Number is less than or equal to 10."
THEN
```

prints a message regardless of the value of the operand.

Incidentally, *don't* make the mistake of assuming the IF-THEN sequence

```
10 > CR
IF
." Number is greater than 10."
THEN
." Number is less than or equal to 10."
```

performs the same task as the preceding IF-ELSE-THEN sequence, because if the stack number happens to be greater than 10 you will receive *both* messages:

```
Number is greater than 10.
Number is less than or equal to 10.
```

You can stay out of trouble by remembering that an IF-THEN will only do something if the flag is true, whereas an IF-ELSE-THEN will do something in either case, flag true or flag false.

Nested IF-THENs and IF-ELSE-THENs

Like all control structures, IF-THENs and IF-ELSE-THENs can be nested to any practical depth. Be aware, though, that every comparison word will remove one or two operands from the stack, so you must ensure that the appropriate stack operands are available for each level of nesting.

To demonstrate nesting, let's take our compare-with-10 testing one step further, to find out whether the number at the top of the stack is less than, equal to, or greater than 10. Thus, we want a FORTH program that will make a *three-way decision*. Example 7-2 shows a definition, COMPARE-WITH-10, that will do the job.

Note that the DUP at the beginning of the definition saves a copy of the stack number for the possible use of the second-level IF-ELSE-THEN. If the number is greater than 10, a DROP deletes that copy.

Using a False Flag With IF-THEN

Thus far we have discussed only the standard form of IF-THEN, in which you execute a sequence if a flag is true and skip the sequence if that flag is false. Is it possible to use IF-THEN in the opposite way—execute if false, skip if true? Well, you can certainly do this by using the IF-ELSE-THEN variant

Example 7-2. A three-way compare in FORTH

```
: COMPARE-WITH-10
    ( Compare the number on the top of the stack with 10, )
    ( and print an appropriate message. The number itself  )
    ( is deleted.                                          )
    ( n --- )
    DUP 10 > CR
    IF
      ." Number is greater than 10."
      DROP
    ELSE
      10 =
      IF
        ." Number is equal to 10."
      ELSE
        ." Number is less than 10."
      THEN
    THEN ;
```

with nothing between IF and ELSE. For example, to execute a sequence if a stack number is less than or equal to 10, you could use:

```
10 >
IF
ELSE
. " Number is less than or equal to 10."
THEN
```

However, this approach includes an IF that does nothing more than allow you to get to the "false" case between ELSE and THEN.

Would the following approach do the same job?

```
10 <
IF
 . " Number is less than or equal to 10."
THEN
```

No, this is insufficient because the case of number = 10 would cause the message to be *skipped*. What we need is a conditional that causes the sequence to be executed if the stack number is *not* greater than 10; something that reverses the flag established by the sequence (10 >).

FORTH-79 has a Required Word, called NOT, that reverses the state of a flag. If a flag is true, NOT changes it to false; if a flag is false, NOT changes it to true. Therefore, the sequence to use in the preceding application is:

```
10 > NOT
IF
  ." Number is less than or equal to 10."
THEN
```

NOT is extremely convenient for this kind of situation, because it allows you to use IF-THEN in a sequence that would normally call for the form IF-ELSE-THEN.

Incidentally, fig-FORTH does not include the word NOT, but fig-FORTH users can use the word [0=], which is equivalent to NOT.

Mixing IF-THENs and DO-LOOPs

Any of the control structures can be mixed as long as none of their individual rules are violated. Example 7-3 shows how IF-ELSE-THENs can be used in a DO-LOOP oriented program, to process some possible inputs that cannot easily be incorporated into the DO-LOOP.

Example 7-3. Raise number to a power

```
: **
    ( Raise n1 to the power n2. If n2 is negative, the )
    ( result is zero.                                  )
    ( n1 n2 --- n1**n2 )
    DUP 1 >
    IF                          ( n2 > 1)
      OVER SWAP                     ( n1 n2 --- n1 n1 n2)
      1                             ( DO-LOOP start = 1)
      DO OVER * LOOP                ( Multiply current product by n1)
      SWAP DROP                     ( Delete n1)
    ELSE ?DUP 0=
      IF DROP 1                 ( n2 = 0, so answer = 1)
      ELSE 0<
        IF DROP 0               ( n2 < 0, so answer = 0)
        THEN
      THEN                      ( n2 = 1, so answer = n1)
    THEN ;
```

This example is a colon-definition for the FORTH-79 uncontrolled word [**], which was described in Chapter 2. Here, the DO-LOOP calculates n1**n2 for values of n2 that are greater than one, and nested IF-ELSE-THENs take over if n2 is equal to zero, equal to one or negative. Note that the DO-LOOP is presented first in the definition, because it is most likely to be executed; this saves FORTH some searching time.

An IF-ELSE-THEN for the Interpreter

The FORTH-79 Standard provides for an interpreted (rather than compiled) form of IF-ELSE-THEN, called IFTRUE-OTHERWISE-IFEND. This uncontrolled definition has the general form

flag IFTRUE . . . (true) . . . OTHERWISE . . . (false) . . . IFEND

However, because IFTRUE-OTHERWISE-IFEND is interpreted, it cannot be nested, whereas IF-ELSE-THEN can be nested, as we have seen.

DISPLAY THE CONTENTS OF THE STACK

With the background provided by this chapter, we are now able to develop an extremely useful word that is mentioned in neither the FORTH-79 Standard nor the fig-FORTH Installation Manual, yet it is provided in virtually every FORTH package. This word, [.S], displays the contents of the stack without altering the stack in any way. As any FORTH programmer will tell you, [.S] is absolutely the most powerful debugging tool at your disposal.

Example 7-4 presents the colon-definition for [.S] in both FORTH-79 and fig-FORTH versions. Both versions are similar, but whereas the FORTH-79 is based on the words DEPTH and SP@, the fig-FORTH version is based on words S0 and SP@. The definitions shown here are designed to display the stack numbers one per line, in top-to-bottom order.

Either version should produce this kind of display:

```
10 20 30 40 .S
40
30
20
10 OK
```

And if the stack is empty, you should see this display:

```
.S
STACK EMPTY OK
```

Example 7-4. Display the contents of the stack

```
: .S
  ( Display the contents of the stack, withaut altering them. )
  ( --- )
  DEPTH 2 *                ( Byte count = 2*DEPTH)
  ?DUP 0=                  ( Check far empty stack)
  IF
  CR ." STACK EMPTY "      ( If empty, display message)
  ELSE
  SP@ 2+                   ( DO-LOOP start = original SP)
  DUP ROT +                ( DO-LOOP limit = SP + bytes)
  SWAP                     ( start limit --- limit start)
  DO
    CR I ?                 ( Display next number)
    2
  +LOOP
  THEN ;
```

(a) FORTH-79 versian

```
: .S
  ( Display the cantents af the stack, withaut altering them. )
  ( --- )
  SP@ S0 @ -               ( -Byte caunt = SP@ - (S0) )
  -DUP 0=                  ( Check for empty stack)
  IF
  CR ." STACK EMPTY "      ( If empty, display message)
  ELSE
  S0 @                     ( DO-LOOP limit = (S0) )
  DUP ROT +                ( DO-LOOP start = (S0) - bytes)
  DO
    CR I ?                 ( Display next number)
    2
  +LOOP
  THEN ;
```

(b) fig-FORTH versian

TO FINISH EARLY, JUST LEAVE OR EXIT

Two Required Words can force a DO-LOOP or a colon-definition to be prematurely terminated. The word LEAVE causes a DO-LOOP to be terminated at the next occurrence of LOOP or +LOOP, by setting the loop limit equal to the current value of the index. The word EXIT causes a colon-definition to be terminated immediately; that is, it acts like an imbedded [;]. EXIT can be used anywhere within a colon-definition, except within a DO-LOOP; but, of course, we have LEAVE for that purpose.

Both LEAVE and EXIT are usually preceded by a comparison, to test for the terminate condition. However, EXIT is generally used to abort a colon-definition on an error, whereas LEAVE is often activated when a *desired* event occurs.

For instance, the FIND-NUMBER program in Example 7-5 searches a specified block of memory for a certain number

Example 7-5. Find a number in memory

```
: FIND-NUMBER
  ( Search memory between addr1 and addr2 for the first )
  ( occurrence of the number n.                          )
  ( If n is found, its address is returned on the top    )
  ( of the stack; otherwise, 0 is returned on the stack. )
  ( If found: addr1 addr2 n --- addrn )
  ( If not found: addr1 addr2 n --- 0 )
  0                       ( For now, assume "not found")
  4 ROLL ROT              ( and rearrange stack as:     )
  4 ROLL                  ( 0 addr1 n addr2             )
  3 PICK -                ( Calculate byte count)
  2+                      ( DO-LOOP limit = byte count + 2)
  0                       ( DO-LOOP start = 0)
  DO
    OVER I +              ( Address = addr1 + I)
    @                     ( Fetch next number from memory)
    OVER =                ( Number = n?)
    IF
      ROT 3 PICK + I +    ( If so, make stack read)
      SWAP ROT            ( addrn n addr1       )
      LEAVE
    THEN
  2
  +LOOP                   ( If not, go fetch next number)
  DROP DROP ;             ( Delete top two numbers)
```

Table 7-2. fig-FORTH Comparison Words and Control Structures

Word	Stack	Action
0<	n --- flag	Sets flag true if n is less than zero.
0=	n --- flag	Sets flag true if n is equal to zero.
<	n1 n2 --- flag	Sets flag true if n1 is less than n2.
=	n1 n2 --- flag	Sets flag true if n1 is equal to n2.
>	n1 n2 --- flag	Sets flag true if n1 is greater than n2.

BEGIN
UNTIL
WHILE
REPEAT
	
	UNTIL: flag ---
	WHILE: flag ---
		Used in a colon-definition in the form:
		BEGIN . . . flag UNTIL or
		BEGIN . . . flag WHILE . . . REPEAT
		A BEGIN-UNTIL loop will be repeated until flag is true. A BEGIN-WHILE-REPEAT loop will be repeated until flag is false.

END		A synonym for UNTIL.
AGAIN		Used in a colon-definition in the form: BEGIN . . . AGAIN to force an unconditional jump back to begin.
-DUP	n --- n (n)	Duplicates n if it is nonzero.

IF
ELSE
ENDIF
	IF: flag ---	Used in a colon-definition in the form:
		flag IF . . . (true). . . ENDIF or
		flag IF ELSE . . . (false). . . ENDIF
		If flag is true, the words following IF are executed and the words following ELSE are skipped.

Table 7-2—cont. fig-FORTH Comparison Words and Control Structures

Word	Stack	Action
		If flag is false, words between ELSE and ENDIF are executed and the words between IF and ELSE are skipped. IF-ELSE-ENDIF conditionals may be nested.
THEN		A synonym for ENDIF.
LEAVE		Forces a DO-LOOP to terminate at the next LOOP or +LOOP, by setting the loop limit equal to the index.

value, and uses LEAVE to exit the DO-LOOP if the number is found. On a successful search, the matching memory address is returned on the stack, but if the number is not found, a zero is returned on the stack.

Upon entry, it is assumed that the number will not be found, so a zero is placed at the bottom of the stack for possible future use. After calculating the DO-LOOP limits, the program fetches a number from memory and compares it with the search value, *n*. If the two numbers are identical, an IF-THEN replaces the stack's zero value with the effective address of the matching number in memory, then a LEAVE forces the DO-LOOP to terminate at +LOOP. Otherwise, the DO-LOOP continues searching memory, one number at a time, over the selected range.

Upon completion of the DO-LOOP, whether due to match or no-match, the top two numbers on the stack, addr1 and n, are deleted, leaving only the matching address (if the number was found) or zero (if the number was not found).

Example 7-5 includes the words PICK and ROLL. If you have fig-FORTH, or some other FORTH that does not have these words, refer to Examples 6-3 and 6-5, respectively.

As mentioned earlier in this section, EXIT is normally used to terminate a colon-definition on an error condition, particularly invalid input from the user. For instance, based on the fact that you can't take the square root of a negative number, EXIT could be included in the word SQRT (Example 7-1), to terminate the execution if the stack number is negative. With the

addition of this safeguard, the definition of SQRT would begin like this:

```
SQRT
 DUP 0<        ( These two lines guard against )
 IF EXIT THEN  ( a negative input.              )
 1
 BEGIN
  OVER OVER /
 ..
 ..
 etc.
```

fig-FORTH COMPARISON WORDS AND CONDITIONAL CONTROL STRUCTURES

Table 7-2 summarizes the comparison words and conditional control structures provided by fig-FORTH. As you can see, fig-FORTH has just a minimal complement of comparison words; those that compare just signed numbers (although the 0> function is missing). There are no comparison words for unsigned numbers or double numbers. However, fig-FORTH *does* have FORTH-79's required conditional control structures. The forms BEGIN-UNTIL, BEGIN-WHILE-REPEAT, IF-THEN and IF-THEN-ELSE are all provided, along with the alternate forms IF-ENDIF and IF-THEN-ENDIF. The DO-LOOP terminate word LEAVE is also included in fig-FORTH, but the colon-definition terminate word is not.

SUMMARY

This chapter covered conditional control structures and comparison words that provide the basis for an execution decision. One kind of conditional control structure, BEGIN-UNTIL, repeats an operation until a flag is true. A variation, BEGIN-WHILE-REPEAT, repeats an operation until a flag is false.

Another kind of control structure, the IF-THEN conditional, executes an operation only if a flag is true; otherwise that operation is skipped. A variation, IF-ELSE-THEN, executes the IF-ELSE part if a flag is true and executes the ELSE-THEN part if it is false.

Table 7-3. Words Added to FORTH in Chapter 7

Word	Stack	Action
SQRT	un --- sqrt (un)	Returns integer square root of an unsigned number.
.S		Displays contents of the stack, without altering the stack.
FIND-NUMBER	addr1 addr2 n --- addrn	Returns address of a number, if it occurs between addr1 and addr2. If the number is not found, zero is left on the stack.

Finally, we examined two words that are not conditional themselves, but are normally executed conditionally. LEAVE forces a DO-LOOP to terminate and EXIT forces a colon-definition to terminate.

This chapter also included definitions of three new words (Table 7-3) that you may wish to add to FORTH, as well as the definition of [**], an uncontrolled FORTH-79 word that raises a number to a power.

CHAPTER 8

Constants, Variables, Arrays, and Tables

From time to time, you will find it convenient to reference data values by name, so you don't have to bother remembering what the actual values are. FORTH provides two types of a named data, called constants and variables. A *constant* is a named *value* that usually remains unchanged throughout a program (although it can be changed, with some difficulty) and a *variable* is a named *location* that can be easily changed whenever you choose to do so.

In this chapter we will discuss the FORTH words CONSTANT and VARIABLE, plus some other *defining words* that are used to establish arrays and tables in memory. The term defining word applies to FORTH words that create a new entry in the dictionary. We encountered one defining word earlier, without calling it by that name. It was the word [:], the beginning word of colon-definitions.

Table 8-1 summarizes the words that will be discussed in this chapter.

CONSTANTS

Setting up a constant is simply a matter of specifying a number, typing the word CONSTANT, and then giving the number a name, in this general form:

```
n CONSTANT name
```

As with colon-definitions, constant names can be up to 31 characters long.

For example, the sequence

```
53 CONSTANT X1
```

Table 8-1. Constant- and Variable-Defining Words

Word	Stack	Action	Notes
CONSTANT	n ---	A defining word used in the form: n CONSTANT name to create a dictionary entry for name, leaving n in its parameter field. When name is later executed, n will be left on the stack.	
2CONSTANT	d ---	Double-number equivalent of CONSTANT.	(1)
VARIABLE		A defining word used in the form: VARIABLE name to create a dictionary entry for name and allot two bytes for storage in the parameter field. The application must initialize the stored value. When name is later executed, the address of its parameter field is left on the stack.	
2VARIABLE		Double-number equivalent of VARIABLE. Allots four bytes in the parameter field.	(1)
ALLOT	n ---	Adds n bytes to the parameter field of the most recently defined word.	
,	n ---	Allots two bytes in the dictionary, storing n there.	
C,	n ---	Allots one byte in the dictionary, storing the low-order 8 bits of n there.	(2)
CREATE		A defining word used in the form: CREATE name to create a dictionary entry for name, without allocating any parameter field memory. When name is later executed, the address of its parameter field is left on the stack.	

Table 8-1—cont. Constant- and Variable-Defining Words

Word	Stack	Action	Notes
DOES>		Defines the run-time action of a word created by a high-level defining word. Used in the form: : name CREATE . . . DOES> . . . ; and then name namex DOES> marks the termination of the defining word name, and begins the definition of the run-time action for words that will later be defined by name. Upon execution of namex, the sequence of words between DOES> and [;] will be executed, with the address of namex's parameter field on the stack.	
<BUILDS		A synonym for CREATE when used in the form: : name <BUILDS . . . DOES> . . . ;	(2)

Notes: (1) Included in Double Number Extension Word Set.

(2) Included in Reference Word Set, as an uncontrolled word definition.

will cause the value 53 to be pushed onto the stack each time the word X1 is executed. The colon-definition

```
: X1 53 ;
```

achieves the same result, but takes up more memory and more time to be executed, than the CONSTANT form.

Changing the Value of a Constant

Once assigned, constants normally retain their value permanently. However, you may encounter a situation in which a constant has inadvertently been given the wrong value, and must be changed.

Changing the value of a constant takes this kind of operation:

```
n1 ' name !
```

where *n1* is the new value to be assigned to the constant *name.* This involves three steps: (1) put the new value, n1, on the stack; (2) fetch the address of the parameter field of name from the dictionary with the sequence [' name]; and (3) store n1 at that address with the word [!].

For example, to change the value of X1 from 53 to 54 requires the sequence

```
54 ' X1 !
```

Subsequent references to X1 will return the value 54, rather than 53.

VARIABLES

To set up a variable with FORTH-79, you simply enter the word VARIABLE, followed by the variable's name. This sequence,

```
VARIABLE name
```

registers *name* in the dictionary and allocates two bytes to hold whatever value you will eventually give that name. The contents of those two bytes in memory are as yet undefined; they may contain *anything.*

For example, the assignment

```
VARIABLE HI-SCORE
```

enters the name HI-SCORE into the dictionary, followed by two memory bytes. To assign a value to HI-SCORE (say, "85"), just enter

```
85 HI-SCORE !
```

HI-SCORE will have the value 85 until you change it.

In fig-FORTH, the variable is initialized at the same time it is defined, using the general form

```
n VARIABLE name
```

Thus, with fig-FORTH you don't run the risk of forgetting to initialize the variable, as you do with FORTH-79. To define HI-SCORE in fig-FORTH, you would enter

```
85 VARIABLE HI-SCORE
```

which does the same job as the two preceding FORTH-79 sequences.

Variables Return an Address

There is an important difference between how constants and variables operate. Whereas, a constant returns its value on the stack, a variable returns the *address* of its value on the stack. Therefore, it takes two operations to access the value of a variable: reading the address and fetching the number at that address. For example, to fetch the value of HI-SCORE you must enter

```
HI-SCORE @ OK
```

To *change* the value of HI-SCORE, enter

```
90 HI-SCORE ! OK
```

and to *examine* the current value of HI-SCORE, enter

```
HIGH-SCORE ? 90 OK
```

Variables Can Keep Running Totals

When used in conjunction with the word [+!], variables can be used to maintain running totals of things being counted. As you recall from Chapter 4, [+!] adds a number on the stack to another number in memory, based on an address at the top of the stack.

As an example of a running total, consider a FORTH-based quality control system in a factory that has a variable called REJECTS, which holds a count of the faulty parts turned out in a single day (or week, or month, or quarter). At the beginning of each monitoring period, REJECTS would be initialized with

```
0 REJECTS !
```

Then, with each bad part culled out of the system, the control program could update the count with

```
1 REJECTS +!
```

Thus, at the end of the monitoring period, REJECTS holds the accumulated bad-parts count.

SUPER VARIABLES: ARRAYS

Some applications include several data parameters that are related in one way or another. For instance, a manufacturer of widgets may have a FORTH system with which he'd like to keep

track of the number of widgets produced each day of the week. Due to a sudden, unexplained popularity for widgets, the plant must operate around the clock, seven days a week. (Rumor has it that widgets are being used for various and sundry diabolical purposes in obscure "emerging nations." The CIA, KGB, and Interpol refuse to comment.)

As the resident FORTH expert at Widgets Semi-Limited, your first inclination is to assign seven separate variables to this task (WIDGETS-MONDAY, WIDGETS-TUESDAY, etc.), one variable for each day of the week. However, you discover that seven different variables with seven different names occupy a lot of space in memory and, noting their similarity, wonder whether you can somehow group them into one super-variable. As a matter of fact, you can. All seven variables can be grouped into an *array.*

An array is a logical grouping of identically sized variables. Assuming that the widget output for any given day does not exceed 32,767, we could define an array in which the "identically sized variables" are numbers. With seven days in the week, we will need 14 data bytes, two bytes for each widget count number.

Based on what we already know, we could define a variable called WIDGETS as

```
VARIABLE WIDGETS
```

This enters the name WIDGETS into the dictionary and allocates two bytes of parameter space to that name, as shown in Fig. 8-1A.

But the WIDGETS array needs 14 bytes of data storage, so we are 12 bytes short. How can the extra 12 bytes be added to WIDGETS' dictionary entry? They can be added with ALLOT, a word that adds a specified number of bytes to the parameter field (that is, the data field) of the most recently defined word in the dictionary. Therefore, we must follow the WIDGET definition with the sequence

```
12 ALLOT
```

Now there are 14 data bytes allocated to WIDGETS (Fig. 8-1B).

Accessing Numbers in an Array

The 14 bytes allocated to WIDGETS provide memory space for seven 2-byte numbers. How can numbers be stored into this

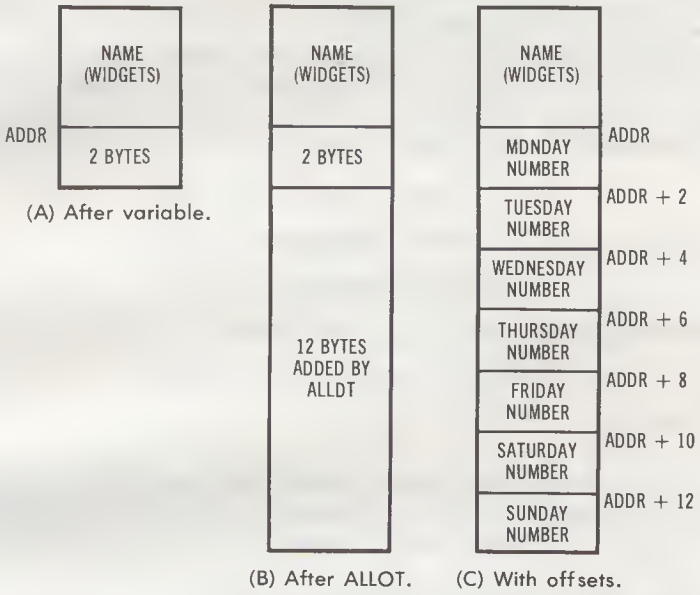


Fig. 8-1. Building up the WIDGETS array.

space, and once stored, how can those numbers be retrieved? Well, knowing that the numbers lie two bytes apart in memory, we can get the address of any given number by adding some multiple of two to the "base address" of the data area; the address put on the stack by the word WIDGETS.

If we call this address *addr,* the address of the first number is addr, the address of the second number is addr+2, the address of the third number is addr+4, and so on. That is, each number is located at some constant *offset* from the base address.

The key word here is "constant." Since each number in WIDGETS reflects the widget production count for a particular day of the week, we can have seven different constants, one for each day's offset value. Specifically, the constants are:

```
 0 CONSTANT MONDAY
 2 CONSTANT TUESDAY
 4 CONSTANT WEDNESDAY
 6 CONSTANT THURSDAY
 8 CONSTANT FRIDAY
10 CONSTANT SATURDAY
12 CONSTANT SUNDAY
```

With these constants assigned, the data area of WIDGETS is effectively partitioned into seven different two-byte number "boxes," as shown in Fig. 8-1C.

The constants allow the widget production number for any given day to be accessed by adding that day's constant to the base address on the stack. For example, to store the number 1315 into the Thursday count, enter

```
1315 WIDGETS THURSDAY + ! OK
```

Similarly, to add 12 to the Thursday count, enter

```
12 WIDGETS THURSDAY + +! OK
```

and to display the Thursday count, enter

```
WIDGETS THURSDAY + ? 1327 OK
```

Large Arrays

For very large arrays, such as an array that holds a widget count for an entire year, it is impractical to set up a constant for each day, because that would require too many constants and take up a lot of memory. For such arrays, it is much better to access a given number by adding its offset to the base address.

For example, a 365-day WIDGETS array could be established with sequences

```
VARIABLE WIDGETS OK( Define the variable WIDGETS      )
364 2 * ALLOT OK     ( and make it a 365-number array )
```

This new version of WIDGETS holds 365 numbers, one for each day of the year. As before, each number has an address that is two greater than the number that precedes it. Therefore, the number for January 1 is located at addr, the number for January 2 is located at addr+2, the number for January 3 is located at addr+4, and the address for the final day, December 31, is located at addr+728. The general form for the offset is, then:

```
Offset = 2*(day-number) − 2
```

where day-number is 1 for January 31 and 365 for December 31.

Thus, to store the value 1430 into the number for day 230, enter the sequence

```
1430 WIDGETS 230 2 * 2− + ! OK
```

and to display the value for day 230, enter

```
WIDGETS 230 2 * 2− + ? 1430 OK
```

Arrays of Bytes

If your array values never exceed 255, the storage limit of a single byte, it is extremely inefficient to store these values into two bytes (as we have been doing thus far). Is it possible to set up an array of bytes, instead of an array of numbers? Yes, it is. Moreover, it is done with the *same* procedure used for number arrays!

Regardless of whether an array is to hold bytes or numbers (or double numbers, for that matter), you will use VARIABLE to assign the array name and allocate the first two bytes, then use ALLOT to allocate additional bytes as needed. Both of these operations involve reserving some number of bytes in memory for later use; neither "tells" FORTH whether the reserved memory will be used to hold byte values or number values.

The data type held in an array is purely a function of the words used to access the array. To access an array of numbers, you would use [!] and [@] as the store and fetch words, and offsets in multiples of two. To access an array of bytes, you would use [C!] and [C@] as the store and fetch words, and offsets in multiples of one.

For example, if Widgets Semi-Limited had a secondary product called frammets, but never produced more than 255 frammets per day, you could set up a seven-day frammet production array as:

```
VARIABLE FRAMMETS OK
5 ALLOT OK
```

and the offsets as

```
0 CONSTANT MON
1 CONSTANT TUES
2 CONSTANT WED
3 CONSTANT THURS
4 CONSTANT FRI
5 CONSTANT SAT
6 CONSTANT SUN
```

Now, to store the value 43 into the Thursday count, enter

```
43 FRAMMETS THURS + C! OK
```

and to display the Thursday count, enter

```
FRAMMETS THURS + C@ . 43 OK
```

note that the sequence (C@ .) had to be used here because there is no byte equivalent of [?] in either FORTH-79 or fig-FORTH.

Initializing an Entire Array

Some arrays hold data that is primarily used for reference, such as dimensions, conversion factors, rates, prices and the like. Typically, this kind of array needs to be entirely initialized at the same time it is defined.

One way of initializing an entire array is by executing a series of store operations, one for each element (number or byte) in the array. For example, our fictional company, Widgets Semi-Limited, may wish to set different widget production quotas for each day of the week, due to variances in the work force. As their FORTH programmer, you could define a quota array as

```
VARIABLE WIDGET-QUOTA
12 ALLOT
```

and initialize it with this series of store operations:

```
1500 WIDGET-QUOTA !          ( Monday quota = 1500)
1450 WIDGET-QUOTA 2 + !      ( Tuesday quota = 1450)
1455 WIDGET-QUOTA 4 + !      ( Wednesday quota = 1455)
1500 WIDGET-QUOTA 6 + !      ( Thursday quota = 1500)
1425 WIDGET-QUOTA 8 + !      ( Friday quota = 1425)
1300 WIDGET-QUOTA 10 + !     ( Saturday quota = 1300)
1200 WIDGET-QUOTA 12 + !     ( Sunday quota = 1200)
```

As tedious and time-consuming as this is with just seven numbers, imagine what it would be like if the array had 1000 numbers! Is there no easier way to initialize an array?

As a matter of fact, there *is* an easier way to initialize an array—using the word [,] (that is, "comma"). The word [,] is a combination ALLOT-and-store word. It allocates two bytes in the dictionary, just like (2 ALLOT), then stores the top number on the stack into those two bytes, just like [!]. Therefore, using [,] the preceding nine-line operation reduces to these four lines:

```
VARIABLE WIDGET-QUOTA
1500 WIDGET-QUOTA !
1450 , 1455 , 1500 ,
1425 , 1300 , 1200 ,
```

For arrays of bytes, there is a similar word, [C,], which allocates one byte in the dictionary, then stores the low-order eight bits of the top number of the stack into that byte. Thus, [C,] is equivalent to the sequence (HERE C! 1 ALLOT).

CREATE an Array

If your application involves only one or two arrays, the VARIABLE-and-ALLOT sequences just described can be used to create those arrays. However, if several arrays are needed, you're probably better off automating the process, by defining a new FORTH word that will create arrays.

This new word—called ARRAY, perhaps—should take just two parameters, a name and a size (number count), and use those parameters to add an array to the dictionary. For example, if you type in

```
10 ARRAY ITEMS
```

a 10-number (20-byte) array called ITEMS should be added to the dictionary.

Well, I have good news! FORTH contains a word combination, CREATE and DOES> (<BUILDS and DOES> in fig-FORTH) that permits you to define ARRAY, or any other new operation or data type. Used alone, CREATE enters a specified name into the dictionary, without allocating any parameter space to that name. (That is, CREATE operates like VARIABLE, minus the two data bytes.) For example

```
CREATE ITEMS
```

enters the name ITEMS into the dictionary, but reserves no parameter space for ITEMS.

However, the combination of CREATE and DOES>, in a colon-definition of the form

```
: name CREATE . . . DOES> . . . ;
```

creates a new, "intelligent" data structure called *name*. This definition has two distinct parts. The words between CREATE and DOES> specify what happens when the colon-definition is compiled and the words between DOES> and [;] specify what happens when an object of the class name is executed. Confused? An example should make this clearer.

Example 8-1 is a colon-definition for a new data type called ARRAY. This word, ARRAY, is designed to create an array in the dictionary that has a specified name and size (number count). The general form for using ARRAY is

```
n ARRAY name
```

so if you enter

```
10 ARRAY ITEMS
```

Example 8-1. Create a number array

```
: ARRAY
  ( Create an array labeled name that has space for n )
  ( numbers, using the general form                   )
  (     n ARRAY name                                  )
  ( Thereafter, fetch the address of any element in   )
  ( name onto the stack by entering                   )
  (     element# name                                 )
  CREATE                ( Enter name into dictionary  )
   2 * ALLOT            (  and allocate 2n bytes to it )
  DOES>
   SWAP                 ( Swap element# and base-addr)
   2 *                  ( Offset = 2*element#)
   + ;                  ( Addr = base-addr + offset)
```

an array named ITEMS, which has space for 10 numbers (that is, 20 bytes), will be entered into the dictionary. In Example 8-1, the word CREATE enters the *name* (ITEMS, in this case) into the dictionary and the sequence (2 * ALLOT) doubles the number count *n* (10, in this case), then allocates that number of bytes to its parameter field.

Once an array is set up in the dictionary, you need to be able to access its individual elements (numbers) in order to initialize them, display them, change them, or whatever. As with FORTH's built-in data type VARIABLE, the standard way to gain access to any particular element is to read its *address* onto the stack. In the case of our array ITEMS, for example, typing in

```
8 ITEMS
```

should return the address of element 8 on the stack. (The elements are assumed to be numbered 0 through 9, so element 8 is actually the next-to-last element in the array.)

The second half of Example 8-1, from DOES> to [;], does the address fetching when you reference an element of the created array. The reference

```
element# name
```

puts the value element# onto the stack, then the address of the array parameter field (called base-addr in Example 8-1). Following DOES>, the SWAP exchanges these two values so that a subsequent [2 *] operation can convert element# into an ad-

dress offset. A simple add operation, base-addr + offset, leaves the address of the referenced element on the stack.

In summary, then, the command that creates our sample array ITEMS is:

```
10 ARRAY ITEMS
```

and some typical operations on element 8 of ITEMS are:

```
3500 8 ITEMS ! OK        ( Store 3500 into element 8)
8 ITEMS ? 3500 OK        ( Display value of element 8)
2 8 ITEMS +! OK          ( Add 2 to element 8)
8 ITEMS ? 3502 OK        ( Display new value of element 8)
```

Example 8-2 shows the colon-definition for the equivalent byte array-creating word, CARRAY. Note that this definition is much simpler than the definition of ARRAY, because the size specifier, *b,* is already a byte count and *element#* is already an offset.

Example 8-2. Create a byte array

```
: CARRAY
  ( Create an array labeled name that has space for b )
  ( bytes, using the general form                     )
  (     b CARRAY name                                 )
  ( Thereafter, fetch the address of any element in   )
  ( name onto the stack by entering                   )
  (     element# name                                 )
  CREATE                ( Enter name into dictionary)
   ALLOT                (   and allocate by bytes to it)
  DOES>
   + ;                  ( Addr = base-addr + element#)
```

Using CARRAY, the command that creates a 10-byte array called B-ITEMS is:

```
10 CARRAY B-ITEMS
```

and typical operations on element 8 of B-ITEMS are:

```
200 8 B-ITEMS C! OK      ( Store 200 into element 8)
8 B-ITEMS C@ . 200 OK ( Display value of element 8)
```

If you plan to conduct many byte operations, you will probably want to have a few byte operators that FORTH doesn't provide. Example 8-3 defines [C?], which displays the contents

of a byte in memory, and Example 8-4 defines [C+!], which adds a value to a byte in memory. These words are byte equivalents of the FORTH words [?] and [+!], respectively. Here is how these new words can be used with the B-ITEMS array:

```
8 B-ITEMS C? 200 OK     ( Display value of element 8)
2 8 B-ITEMS C+! OK      ( Add 3 to element 8)
8 B-ITEMS C? 202 OK     ( Display new value of element 8)
```

Example 8-3. Display a byte in memory

```
: C?
  ( Display the byte at addr. )
  ( addr --- )
  C@ . ;
```

Example 8-4. Add n to a byte in memory

```
: C+!
  ( Add n to the byte at addr. )
  ( n addr --- )
  DUP @                 ( Fetch byte at addr)
  ROT +                 (  and add n to it  )
  SWAP C! ;             ( Return sum to memory)
```

SUPER CONSTANTS: TABLES

If you have a set of related data that will not change, you may wish to put it in a *table,* rather than in an array. Tables are often used to replace complicated or time-consuming operations, such as calculating the square root or cube root of a number, or to hold conversion factors, such as the sines or cosines of a range of angles. Tables are especially efficient when a function is limited to a very small range of arguments, because they alleviate the need to perform complex calculations each time a function is obtained. However, since tables usually require large amounts of memory storage space, they are most efficient in applications where storage space can be sacrificed for execution speed.

Examples 8-5 and 8-6 show the definitions for TABLE and CTABLE, the words that create tables of numbers or tables of

bytes, respectively. Note that both of these definitions are similar to those of their array-creating counterparts, ARRAY and CARRAY, except that:

1. Due to the "permanent" nature of a table, elements are initialized as they are added, using either [,] (number table) or [C,] (byte table). Therefore, the CREATE portion of the definition is devoid of additional words.
2. The DOES> portion of the definition has an [@] or [C@] word, to put the *value* of the element onto the stack, rather than its address.

Example 8-5. Create a number table

```
: TABLE
  ( Create a table labeled name, using the general form )
  (     TABLE name                                      )
  ( Elements are added with "comma".                    )
  ( Thereafter, fetch the value of any element by       )
  ( entering                                            )
  (     element# name                                   )
  CREATE
  DOES>
   SWAP                ( Swap element# and base-addr)
   2 *                 ( Offset = 2*element#)
   +                   ( Addr = base-addr + offset)
   @ ;                 ( Fetch value at addr)
```

Example 8-6. Create a byte table

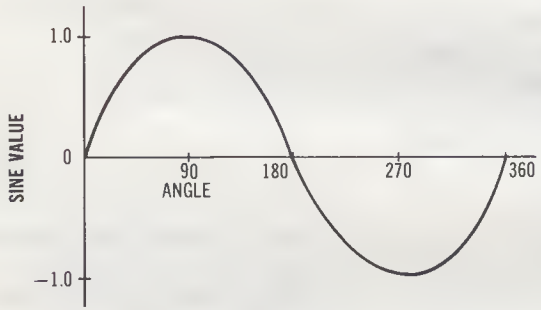
```
: CTABLE
  ( Create a byte table labeled name, using the general )
  ( form                                                )
  (     CTABLE name                                     )
  ( Elements are added with "c-comma".                  )
  ( Thereafter, fetch the value of any element by       )
  ( entering                                            )
  (     element# name                                   )
  CREATE
  DOES>
   +                   ( Addr = base-addr + element#)
   C@ ;                ( Fetch value at addr)
```

Sine of an Angle

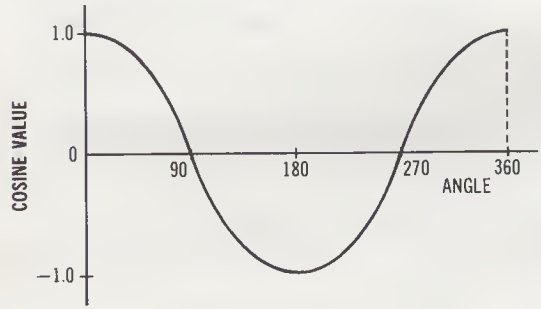
For illustration purposes, let's develop a FORTH program that finds the sine of a specified angle, by looking up that sine in a table.

As you probably recall from high school trigonometry, the sine of all angles between 0° and 360° can be graphed as shown in Fig. 8-2A. Mathematically, this curve can be approximated by the formula

$$\text{sine}(X) = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \frac{X^9}{9!} \ldots$$



(A) Sines.



(B) Cosines.

Fig. 8-2. The sines and cosines of angles between 0 and 360 degrees.

It is certainly possible to write a program to perform this approximation, but such a program may require a couple of milliseconds to calculate the sine. If your application requires very precise sines, you may be forced to write such a program. However, applications with less stringent requirements can use an angle-to-sine look-up table instead.

If the application needs to be able to obtain the sine of any angle between 0° and 360°, where the angle is an integer, how many sine values must the table contain? Must it contain 360 different sine values? No, we can get by with a table of only 91 sine values, one value for each angle between 0° and 90°, inclusive.

To understand how this can be so, look at Fig. 8-2A once again. If we call the leftmost quarter of the graph (angles from 0° to 90°) Quadrant I, we can see that:

1. Sines in Quadrant II (angles between 91° and 180°) are the "mirror-image" of those in Quadrant I.
2. Sines in Quadrant III (angles between 181° and 270°) are the "negative inverse" of those in Quadrant I.
3. Sines in Quadrant IV (angles between 271° and 360°) are the "negative inverse, mirror-image" of those in Quadrant II.

That is, the sines in all four quadrants are some variation of the sines in Quadrant I!

Confident in this knowledge, we can create a table of sine values. To do this we would enter TABLE (Example 8-5) into the dictionary, followed by this sequence:

```
TABLE SINE-TABLE
0 , 175 , 349 , 523 ,          ( Sines for 0 - 3)
698 , 872 , 1045 , 1219 ,      ( Sines for 4 - 7)
..                             ( Sines for angles from)
..                             ( 8 to 86 go here)
9986 , 9994 , 9998 , 10000 ,   ( Sines for 87 - 90)
```

These sine values (Table 8-2 shows the complete list) are to be interpreted with the decimal point placed four digits to the left. That is, the sine of 1° is entered as 175, but should be interpreted as 0.0175. Similarly, the sines of 89° and 90° are entered as 9998 and 10000, but should be interpreted as 0.9998 and 1.0000, respectively. Therefore, to use these values in a program, they must be divided by 10,000.

Now that we have a sine look-up table, let's use it to develop a program that can look up the sine of any angle between 0°

Table 8-2. A Sine Look-Up Table With Angles in 1° Increments

Angle	Sine		Angle	Sine	
	Decimal	Binary		Decimal	Binary
0.00	.0000	00000000	45.00	.7071	01011010
1.00	.0175	00000010	46.00	.7193	01011100
2.00	.0349	00000100	47.00	.7313	01011101
3.00	.0523	00000110	48.00	.7431	01011111
4.00	.0698	00001000	49.00	.7547	01100000
5.00	.0872	00001011	50.00	.7660	01100010
6.00	.1045	00001101	51.00	.7771	01100011
7.00	.1219	00001111	52.00	.7880	01100100
8.00	.1392	00010001	53.00	.7986	01100110
9.00	.1564	00010100	54.00	.8090	01100111
10.00	.1736	00010110	55.00	.8191	01101000
11.00	.1908	00011000	56.00	.8290	01101010
12.00	.2079	00011010	57.00	.8387	01101011
13.00	.2250	00011100	58.00	.8480	01101100
14.00	.2419	00011110	59.00	.8572	01101101
15.00	.2588	00100001	60.00	.8660	01101110
16.00	.2756	00100011	61.00	.8746	01101111
17.00	.2924	00100101	62.00	.8829	01110001
18.00	.3090	00100111	63.00	.8910	01110010
19.00	.3256	00101001	64.00	.8988	01110011
20.00	.3420	00101011	65.00	.9063	01110100
21.00	.3584	00101101	66.00	.9135	01110100
22.00	.3746	00101111	67.00	.9205	01110101
23.00	.3907	00110010	68.00	.9272	01110110
24.00	.4067	00110100	69.00	.9336	01110111
25.00	.4226	00110110	70.00	.9397	01111000
26.00	.4384	00111000	71.00	.9455	01111001
27.00	.4540	00111010	72.00	.9511	01111001
28.00	.4695	00111100	73.00	.9563	01111010
29.00	.4848	00111110	74.00	.9613	01111011
30.00	.5000	01000000	75.00	.9659	01111011
31.00	.5150	01000001	76.00	.9703	01111100
32.00	.5299	01000011	77.00	.9744	01111100
33.00	.5446	01000101	78.00	.9781	01111101
34.00	.5592	01000111	79.00	.9816	01111101
35.00	.5736	01001001	80.00	.9848	01111110
36.00	.5878	01001011	81.00	.9877	01111110
37.00	.6018	01001101	82.00	.9903	01111110
38.00	.6157	01001110	83.00	.9926	01111111
39.00	.6293	01010000	84.00	.9945	01111111
40.00	.6428	01010010	85.00	.9962	01111111
41.00	.6561	01010011	86.00	.9976	01111111
42.00	.6691	01010101	87.00	.9986	01111111
43.00	.6820	01010111	88.00	.9994	01111111
44.00	.6947	01011000	89.00	.9998	01111111
45.00	.7071	01011010	90.00	1.0000	01111111

and 360°. To do this, we must work out some conversion formulas for each quadrant.

For *Quadrant I,* the sine values are taken directly from the table, so for any angle X,

$$0° \leq X \leq 90°, \text{ take sine}(X)$$

directly from SINE-TABLE.

For *Quadrant II,* the sine values are the "mirror-image" of those in Quadrant I. That is, the sine of 91° is the same as the sine of 89°, and the sine of 179° is the same as the sine of 1°. Therefore,

$$90° \leq X \leq 180°, \text{ take sine}(180° - X)$$

For example,

$$\begin{aligned} \text{sine}(170°) &= \text{sine}(180° - 170°) \\ &= \text{sine}(10°) \end{aligned}$$

Angles in *Quadrants III and IV* have sines with the same magnitude, but the opposite signs, as the angles in Quadrants I and II, respectively. This observation allows us to state the following:

$$\begin{aligned} 180° \leq X \leq 270°, &\text{ take } -\text{sine}(X - 180°) \\ 270° \leq X \leq 360°, &\text{ take } -\text{sine}(360° - X) \end{aligned}$$

For example,

$$\begin{aligned} \text{sine}(190°) &= -\text{sine}(190° - 180°) \\ &= -\text{sine}(10°) \end{aligned}$$

Or, in Quadrant IV,

$$\begin{aligned} \text{sine}(290°) &= -\text{sine}(360° - X) \\ &= -\text{sine}(360° - 290°) \\ &= -\text{sine}(70°) \end{aligned}$$

Now, with an equation to relate Quadrants II, III, and IV to Quadrant I, we have all the tools to write a program that will find the sine of any angle between 0° and 360°. This program is given in Example 8-7, as the colon-definition for a word called SINE. Using two IF-ELSE-THENs and one IF-THEN, SINE finds the proper quadrant by a process of elimination.

When using SINE, keep in mind that the sine is returned as an integer that must be divided by 10,000! For example, the sine of 19° will be returned as

```
19 SINE . 3256 OK
```

but its actual value is 0.3256.

Example 8-7. Sine of an angle, using a table

```
: SINE
  ( Return sine of any integer-valued angle between 0    )
  ( degrees and 360 degrees. To use the result, divide   )
  ( it by 10,000.                                        )
  ( angle --- sine )
  DUP 270 >
  IF
   360 SWAP -              ( For 271-360, leave-sine(360-X))
   SINE-TABLE NEGATE
  ELSE
   DUP 180 >
   IF
    180 -                  ( For 181-270, leave -sine (X-180))
    SINE-TABLE NEGATE
   ELSE
    DUP 90 >
    IF
     180 SWAP -            ( For 91-180, leave sine(180-X))
    THEN
    SINE-TABLE             ( For 0-90, leave sine(X))
   THEN
  THEN ;
```

Cosine of an Angle

As Fig. 8-2B shows, the cosine curve is nothing more than the sine curve displaced one quadrant to the left. Thus, the cosine of any given angle is equal to the sine of an angle that is 90° greater. In equation form:

$$\text{cosine}(X) = \text{sine}(X+90)$$

Knowing this, we can use SINE-TABLE to look up the cosine of an angle as well as its sine. Example 8-8 defines the appropriate word, COSINE. As with SINE, the result of COSINE must be divided by 10,000.

Incidentally, note that both the sine curve and the cosine curve are symmetric about the vertical axis, so negative angles have the same sines and cosines as their positive counterparts. For example, −10° has the same sine and cosine as +10°. This means you can also use SINE and COSINE for angles between −1° and −360°, by supplying the angle's *absolute value* on the stack.

Example 8-8. Cosine of an angle, using a table

```
: COSINE
  ( Return cosine of any integer-valued angle between )
  ( 0 degrees and 360 degrees. To use the result,     )
  ( divide it by 10,000.                              )
  ( angle --- cosine )
  DUP 270 >              ( Is angle greater than 270?)
  IF
   270 -                 ( Yes. Look up Quadrant I sine)
  ELSE
   90 +                  ( No. Use next sine quadrant)
  THEN
  SINE ;
```

SORTING ARRAYS

In many applications, the numbers in an array represent test results, statistical data or some other kind of unordered information. If this information is to be processed or analyzed, you will probably want to rearrange it, or *sort* it, into either increasing or decreasing order. This section describes two common sorting techniques, called *bubble sort* and *insertion sort,* and provides FORTH words to perform each type of operation. Although our discussion will concentrate on increasing-order sorts, the principles also apply to decreasing-order sorts,

Bubble Sort

The bubble sort technique is so named because it causes numbers to rise upward in memory (to higher addresses) just as soap bubbles rise into the sky. During a bubble sort, numbers in an array are accessed sequentially, starting with the first number, and compared to the next number in the list. If a number is found to be greater than its higher-addressed neighbor, the numbers are exchanged. The next two numbers are then compared, exchanged if required, and so on. By the time the microprocessor gets to the last number in the array, the largest number will have "bubbled up" to the last number position in the array.

This algorithm usually requires several passes to completely sort the array, as you can see from the example in Fig. 8-3.

Fig. 8-3. A bubble sort "bubbles" the largest numbers to the end.

Here, the first pass "bubbles" 50 to the end of the array and the next two passes "bubble" 40 and 30 to the next highest positions in the array. So this particular array has been sorted in three passes.

With pass-by-pass "snapshots" of the array, like those shown in Fig. 8-3, it is easy for *you* to know when an array is sorted, but how can a c*omputer* know when an array is sorted? Unless it is given a specific pass count, or told when to stop in some other way, the computer will continue executing pass after pass, ad infinitum. Since the number of sorting passes depends on the initial arrangement of the array, we have no way to provide a pass count in a program. For this reason, we will set up a special indicator, called an e*xchange flag,* that the computer can use to know when to stop sorting.

The exchange flag will be set to 1 before each sorting pass. Any sorting pass that includes a number exchange will cause the exchange flag to be reset to 0. Therefore, after each pass the value of the exchange flag tells the computer whether to continue sorting. A value of 0 signals the need for another pass through the array; a value of 1 indicates that the array is sorted, and tells the computer to stop sorting. Fig. 8-4 is a flowchart of the bubble sort algorithm.

As you can see, even if an array is totally ordered at the outset, it will take one pass to deduce this fact. If one pass is the *minimum* in a bubble sort, what *maximum* number of passes can be expected? The preceding five-number sample array, which was already partially sorted, required three passes to put the numbers in order and one more pass to detect that the array was sorted—four passes in all. If that same array had been initially arranged in descending order (the worst case), the bubble sort algorithm would have required five sorting passes; four passes to sort the data and one more pass to detect that no further sorting was needed. From this observation we can state that an N-number array will take from one to N passes to sort, with (N+1)/2 passes being the average.

Example 8-9 shows the definition of the word BBL-SORT, which sorts an array using the algorithm we just described.

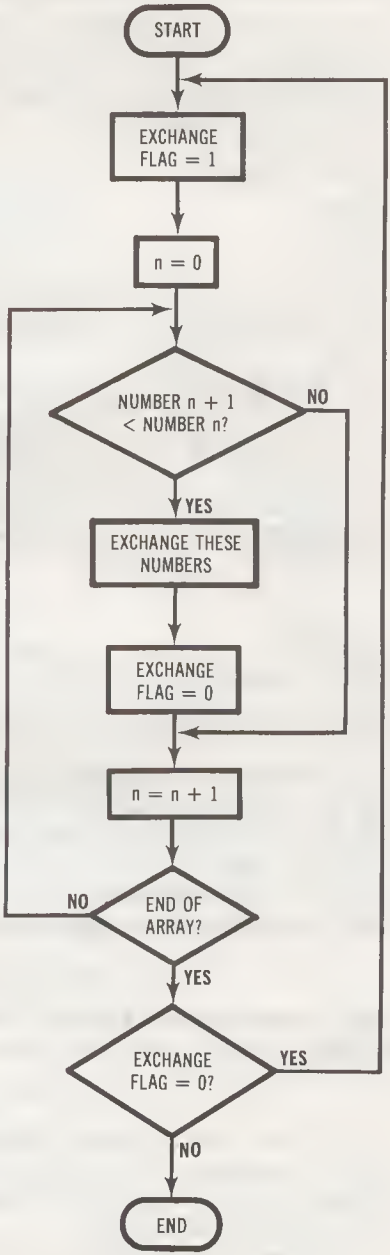


Fig. 8-4. Bubble sort flowchart.

Example 8-9. Sort an array of numbers, using bubble sort

```
: BBL-SORT
  ( Using the bubble sort technique, sort on n-number orroy )
  ( beginning ot oddr.                                       )
  ( oddr n --- )
  DUP + OVER + 2 -      ( DO-LOOP end = oddr + 2n - 2)
  SWAP                  ( DO-LOOP stort = addr)
  >R >R                 ( Sove these limits on return stock)
  BEGIN
   1                    ( To stort, exchonge flog = 1)
   R> R@ OVER >R        ( Fetch DO-LOOP limits)
   DO
    I 2 + @ I @ <       ( Compore next two numbers)
    IF                  ( If second number < first number,)
     I 2+ DUP @         (  exchonge the two numbers      )
     I DUP @
     4 ROLL ! !
     DROP 0             ( Set exchange flog = 0)
    THEN
    2
   +LOOP
  UNTIL                 ( Keep sorting until flog = 1)
  R> R> DROP DROP ;     ( Discord DO-LOOP limits)
```

This definition is straightforward, and follows the flowchart in Fig. 8-4. Note that the DO-LOOP limits are saved on the return stack, because they must be reloaded after each sorting pass. You may also like to study how two numbers can be exchanged. This operation involves first pushing the addresses and number onto the data stack, in this order:

$$\text{oddr}_{I+1}\ n_{I+1}\ \text{oddr}_I\ n_I$$

Next, the sequence (4 ROLL) rotates the address of the second number onto the top of the stack, to produce this arrangement

$$n_{I+1}\ \text{addr}_I\ n_I\ \text{oddr}_{I+1}$$

At this point, we have two pairs of arguments. These are used by the sequence (! !) to store the first number into the second address, then the second number into the first address, which completes the exchange.

The operation performed by BBL-SORT has one deficiency: each execution of the DO-LOOP causes all numbers in the

array to be compared—even the numbers that have already "bubbled up" to their final position! Since there is no need to process the numbers that have already been sorted, we can speed up the bubble sort operation, somewhat, by omitting those numbers from the comparison. That is, in sorting a N-number array, the first pass should process N numbers, the second pass should process N−1 numbers, the third pass should process N−2 numbers, and so on.

To define a new FORTH word with this more efficient algorithm, we need only modify BBL-SORT so that the DO-LOOP "end" limit is decreased by two before each sorting pass. Example 8-10 shows the improved version of BBL-SORT, a word called BSORT. In BSORT, the DO-LOOP end limit is decreased by two immediately after it is pulled from the return stack. To compensate for this before the first pass, the initial DO-LOOP end limit is calculated as

addr + 2n

a value two greater than the initial end limit in the BBL-SORT definition. These two changes are the only differences between BSORT and BBL-SORT; everything else is identical.

How much faster is BSORT than BBL-SORT? To find out, I ran three "worst-case" sorts (numbers initially in decreasing order) using a FORTH system on my Apple II computer, and measured the execution times with a wristwatch. To sort 100 numbers, BBL-SORT took 16 seconds and BSORT took 12 seconds. To sort 200 numbers, BBL-SORT took 66 seconds and BSORT took 47 seconds. To sort 300 numbers, BBL-SORT took 150 seconds and BSORT took 103 seconds. From these results it appears that *BSORT is about 30% faster than BBL-SORT!*

Insertion Sort

The bubble sort technique is easy to understand and easy to implement, but unless the target array is almost sorted at the outset, bubble sort is one of the slowest of all sorting techniques. Another technique that is nearly as simple as the bubble sort, and almost always faster, is the *insertion sort.*

An insertion sort arranges numbers in an array the same way you might arrange a deck of cards. That is, an insertion sort makes one pass through the array, starting with the second number, and compares each number with the numbers preceding it. If a number is less than any of its predecessors, it is moved toward the beginning of the array, to its proper position. If a number is greater than the preceding number (which

Example 8-10. An improved bubble-sort

```
: BSORT
  ( A more efficient version of BBL-SORT, in which only )
  ( previously unsorted numbers are processed.          )
  ( addr n --- )
  DUP + OVER +              ( DO-LOOP end = addr + 2n)
  SWAP                      ( DO-LOOP start = addr)
  >R >R                     ( Save these limits on return stack)
  BEGIN
   1                        ( To start, exchange flag = 1)
   R> 2 - R@ OVER >R        ( Subtract 2 from end limit)
   DO
    I 2 + @ I @ <           ( Compare next two numbers)
    IF                      ( If second number < first number,)
     I 2+ DUP @             (   exchange the two numbers      )
     I DUP @
     4 ROLL ! !
     DROP 0                 ( Set exchange flag = 0)
    THEN
    2
   +LOOP
  UNTIL                     ( Keep sorting until flag = 1)
  R> R> DROP DROP ;         ( Discard DO-LOOP limits)
```

means it is greater than *all* preceding numbers), it is left alone and the process continues with the next number. Fig. 8-5 shows the insertion sort technique applied to the five-number array we bubble-sorted earlier, in Fig. 8-3.

Example 8-11 shows the definition of the word ISORT, which uses the insertion sort technique to sort an n-number array. This definition is comprised of two DO-LOOPs. The outer DO-LOOP sequences through the array, starting with the second number, comparing a number with its immediate predecessor with each pass. If a number is found to be less than the preceding number, the inner DO-LOOP finds the proper position for that number by working backward through the array. The inner DO-LOOP always leaves an address on the stack; the address of the number that is equal to or greater than the search number. (This address can also be the starting address of the array.) The inner DO-LOOP is followed by a sequence that opens up a space for the search number and inserts the number in that space.

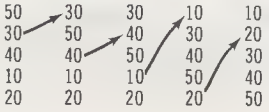


Fig. 8-5. An insertion sort finds the proper place for each number.

Example 8-11. Sort an array of numbers, using insertion sort

```
: ISORT
  ( Using the insertion sort techique, sort an n-number )
  ( array beginning at addr.                             )
  ( addr n --- )
  DUP + OVER +              ( DO-LOOP end = addr + 2n)
  OVER 2+                   ( DO-LOOP start = addr + 2)
  DO
   I @ DUP                  ( Is this number less than)
   I 2 - @ <                (  the preceding number?  )
   IF                       ( Yes. Find its proper place)
    0                       ( Dummy number)
    3 PICK 2 - I 2 -        ( Inner loop limits = addr-2 I-2)
    DO
     DROP                   ( Drop dummy number or last I)
     DUP I @ <              ( Search number < this number?)
     IF
      I                     ( If so, save this address)
     ELSE
      I 2+ LEAVE            ( If not, go insert)
     THEN
     -2
    +LOOP
    DUP DUP 2+              ( Open a gap for the insert,)
    I 3 PICK - <CMOVE !(  then insert the number    )
   ELSE
    DROP                    ( Drop search number)
   THEN
   2
  +LOOP
  DROP ;                    ( Clear the stack)
```

For the "worst case," with all numbers initially in descending order, an insertion sort is slightly slower than a bubble sort.

Table 8-3. fig-FORTH Constant- and Variable-Defining Words

Word	Stack	Action
CONSTANT	n ---	A defining word used in the form: n CONSTANT name to create a dictionary entry for name, leaving n in its parameter field. When name is later executed, n will be left on the stack.
VARIABLE	n ---	A defining word used in the form: n VARIABLE name to create a dictionary entry for name with its parameter field initialized to n. When name is later executed, the address of its parameter field is left on the stack.
ALLOT	n ---	Adds n bytes to the field of the most recently defined word.
,	n ---	Allots two bytes in the dictionary, storing n there.
C,	n ---	Allots one byte in the dictionary, storing the low-order 8 bits of n there.
CREATE		A defining word used in the form: CREATE name to create a dictionary entry for name, without allocating any parameter field memory. When name is later executed, the address of its parameter field is left on the stack.
<BUILDS DOES>		Used in a colon-definition in the form: : name <BUILDS . . . DOES> . . . ; Each time name is executed, <BUILDS defines a new word with a high-level execution procedure. Executing name in the form name namex uses <BUILDS to create a dictionary entry, with a call to the DOES> part for namex. When namex is later executed, the sequence of words between DOES> and [;] will be executed, with the address of namex's parameter field on the stack.

Table 8-4. Words Added to FORTH in Chapter 8

Word	Stack	Action
ARRAY	n ---	Creates an n-number array labeled name, using this form: n ARRAY name Thereafter, fetch the address of any element in name onto the stack by entering element# name
CARRAY	n ---	Same as ARRAY, but creates an n-byte array.
C?	addr ---	Displays the byte at addr.
C+!	n addr ---	Adds the low-order 8 bits of n to the byte at addr.
TABLE		Used in the form: TABLE name to create a dictionary entry for name, without allocating any parameter field memory. Elements are added to the table with [,]. Thereafter, fetch the value of any element by entering element# name
CTABLE		Same as TABLE, but CTABLE is used to create a byte table, in which elements are added with [C,].
SINE	angle --- sine	Returns the sine of any integer-valued angle between 0 and 360 degrees. To use the result, divide it by 10,000.
COSINE	angle --- cosine	Similar to SINE, but returns the cosine of an angle.
BBL-SORT	addr n ---	Sorts an n-number array, starting at addr, using the bubble sort technique.
BSORT	addr n ---	A more efficient version of BBL-SORT, in which only previously unsorted numbers are processed.
ISORT	addr n ---	Sorts an n-number array, starting at addr, using the insertion sort technique.

However, *with random data, an insertion sort is about 20% faster than a bubble sort.*

fig-FORTH CONSTANT- AND VARIABLE-DEFINING WORDS

Table 8-3 lists the fig-FORTH words used to define constants and variables, and to create arrays and tables, Note that fig-FORTH does not include the double-number words 2CONSTANT and 2VARIABLE, but all of FORTH-79's other words are provided.

There are a few differences, however. The primary difference is that fig-FORTH requires a variable to be initialized at the same time it is created, using the form:

```
n VARIABLE name
```

You will recall that FORTH-79 simply allocates memory space for the value, and leaves the initialization responsibility up to you. Thus, fig-FORTH provides a "safer" approach to setting up variables. The only other difference between FORTH-79 and fig-FORTH is that in creating new defining words, FORTH-79 uses the construct CREATE-DOES>, whereas fig-FORTH uses the construct <BUILDS-DOES>. Other than that name change, these constructs are identical between the two FORTHs.

SUMMARY

This chapter discussed *constants,* words that rarely change, and *variables,* words that can be changed as necessary while the program is executing. Referencing a constant leaves its value on the stack, whereas referencing a variable leaves its address on the stack.

We also discussed *tables,* which are groups of constants in memory, and *arrays,* which are groups of variables in memory, and showed how each could be initialized. Finally, we discussed two sorting techniques, *bubble sort* and *insertion sort,* which put data in a form that is more suitable for processing.

Table 8-4 summarizes the words that can be added to FORTH based on the material in this chapter.

CHAPTER 9

Numbering Systems

Until now, all numbers in this book have been given in their decimal form. This has been no hindrance so far because we've been working primarily with data processing types of applications; arithmetic operations, stack manipulations, memory transfers, and the like. However, at some point or other you will probably encounter other numbering systems, those that are more appropriate than decimal when you are dealing with computers.

In this chapter we will present a brief "crash course" on the binary and hexadecimal numbering systems, those most commonly used in computer applications. If you are already familiar with those concepts, feel free to skip the first few sections and proceed to the discussion of BASE, the system variable that FORTH uses to determine which numbering system is in effect at any given time.

THE BINARY NUMBERING SYSTEM

In a computer, all information used in programs (instructions and data) is stored in the computer's *memory*. Memory is comprised of a large number of electrical components that act like light switches. That is, these components have only two possible settings, "on" and "off." However, with just these two settings, combinations of memory components can very effectively represent numbers of any magnitude. How? Read on to find out.

The "on" and "off" settings of the components in memory correspond to the two digits of the *binary numbering system,* the fundamental system for computers. Having only two digits,

1 (on) and 0 (off), the binary numbering system is a *base 2* system. This contrasts with the familiar decimal numbering system, which has 10 digits (0 through 9), making it a base 10 system.

The light switch-like components of memory are called "bits," which is short for *bi*nary digi*ts*. By convention, a bit that is "on" represents the value 1 and a bit that is "off" represents the value 0. This appears to be woefully limiting, until you consider that a decimal digit (no, it's not called a "det") can only range from 0 to 9. Just as decimal digits can be combined to form numbers greater than 9, binary digits can be combined to form numbers greater than 1. In fact, a binary number can represent any value that can be represented by a decimal number; it will just take more binary digits to do the job!

As you know, to represent a decimal number greater than 9 will take one or more extra digits. Numbers between 10 and 99 take an additional "tens position" digit, numbers between 100 and 9999 take a tens position digit and an additional "hundreds position" digit, and so on. Therefore, each decimal digit has a *weight* of 10 times the digit to its immediate right.

For example, the decimal number 324 can be represented as

$$(3 \times 100) + (2 \times 10) + (4 \times 1)$$

or, put another way,

$$324 = (3 \times 10^2) + (2 \times 10^1) + (4 \times 10^0)$$

So, in mathematical terms, each decimal digit is a power of 10 greater than the preceding (less-significant) digit.

A similar rule applies to the binary numbering system: *each binary digit is a power of two greater than the preceding binary digit.* The rightmost digit has a weight of 2^0, the next digit has a weight of 2^1, and so on. Therefore, the binary number 101 has a decimal value of five because

```
101
|||_ = 1 × 2^0 = 1
||__ = 0 × 2^1 = 0
|___ = 1 × 2^2 = 4
      Total = 5
```

Do you understand how binary numbers are constructed? For each binary digit position, you double the weight of the preceding digit. Thus, the first eight binary weights are 1, 2, 4, 8, 16, 32, 64, and 128. These weights are shown in Fig. 9-1.

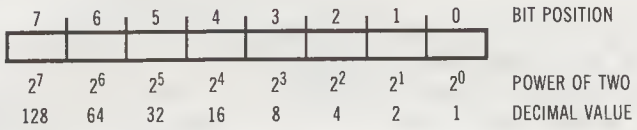


Fig. 9-1. Weights of eight binary digits.

To test your skill at binary numbering, try to construct the binary representations of the decimal values 12, 17, 45, and 79. (You should obtain the binary values 1100, 10001, 101101, and 1001111, respectively.) Conversely, what decimal values are represented by the binary values 1000, 10101, and 11111? (The answers are 8, 21, and 31.)

Eight Bits Form a Byte

The Apple II, the Commodore PET/CBM, the Radio Shack TRS-80 and many other popular microcomputers are designed around an 8-bit *microprocessor.* These microprocessors can process eight bits of information in a single operation. To process more than eight bits requires additional operations.

In computer literature, an 8-bit quantity of information is called a *byte.* With eight bits, a byte can represent decimal integers from 0 (binary 00000000) to 255 (binary 11111111).

Because a byte is the fundamental unit of processing, microcomputers are described in terms of the number of bytes (rather than bits) contained in their memories. Further, microcomputer manufacturers generally construct memory in blocks of 1024 bytes. This particular quantity is an outgrowth of the binary orientation of computers, in that it represents exactly 2^{10} bytes.

The value 1024 has also been given an industry-standard abbreviation; it is referred to by the letter *K.* Thus, when you read an advertisement for a computer that has a "16K RAM," the manufacturer is telling you that this particular product has 16 × 1024 (or 16,384) bytes of programmable memory.

THE HEXADECIMAL NUMBERING SYSTEM

Although binary numbering is an accurate way to represent numbers in memory, strings of nothing but ones and zeroes are

very difficult to work with for any extended period of time. They are error-prone as well, because a number such as 10110101 is extremely easy to write incorrectly as 10110110 (the rightmost two bits have been interchanged).

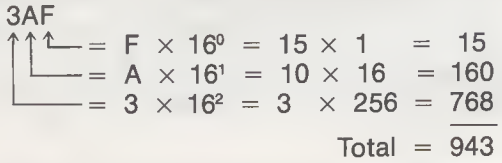
Years ago, programmers realized that they were always required to operate on *groups* of bits, rather than on individual bits. The earliest microprocessors were 4-bit devices (they processed information four bits at a time), so the logical alternative to binary numbering was a system that numbers bits in groups of four.

As you know, four bits can represent the binary values 0000 through 1111, which is equivalent to the decimal values 0 through 15. If each digit of a numbering system is to represent four bits, that numbering system will have digits that can range from 0 to 15 (decimal). Therefore, that numbering system is a *base 16* system.

If the word binary is used to denote a base 2 system and the word decimal is used to denote a base 10 system, what term can be applied to a base 16 system? Well, whoever named the base 16 system took the Greek word "hex" (for six) and the Latin word "decem" (for ten), and combined them to form the word "hexadecimal." Thus, the base 16 numbering system is called the *hexadecimal numbering system.*

The 16 digits of the hexadecimal numbering system are labeled 0 through 9 (decimal values 0 through 9) and A through F (decimal values 10 through 15). The hexadecimal-to-decimal correlations are sumamrized in Table 9-1, for reference purposes.

Like binary and decimal digits, each hexadecimal digit has a "weight" that is some power of its base. Since 16 is the base for the hexadecimal numbering system, each hexadecimal digit has a weight that is a power of 16 higher than the digit to its immediate right. That is, the rightmost digit has a weight of 16^0, the second digit has a weight of 16^1, and so on. For example, the hexadecimal value 3AF has a decimal value of 943, because



To save you the trouble of making this kind of calculation everytime you want to convert a hexadecimal number to a decimal number, or vice versa, a conversion table is given in Appendix A.

Table 9-1. Hexadecimal Numbering System

Hexadecimal Digit	Decimal Value
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Use of Hexadecimal Numbers

For the remainder of this book, and in your own FORTH programming, you will probably want to continue using decimal numbers for most arithmetic calculations, but switch to hexadecimal when specifying memory addresses or operating on individual bits in memory.

The most prevalent use of hexadecimal numbers in computer-related literature is in referring to memory locations. Most 8-bit microprocessors have an addressing range of 65,536 bytes (that is, 64K bytes). This means that memory locations can have addresses from 0 to 65,535 decimal, or from 0 to FFFF hexadecimal. As you can see, then, memory addresses are always two bytes long, or in FORTH terminology, they are *number*-sized.

UNSIGNED AND SIGNED DATA VALUES

Throughout most of this book we have been dealing with *signed* numbers and double numbers, but from time to time we encountered words that can operate on *unsigned* numbers and double numbers. Since this chapter is dedicated to numbering systems, it is an appropriate spot to discuss the details of these two data types. For discussion purposes, we will concentrate

on numbers, but these principles also apply to double numbers—or bytes, for that matter.

Unsigned Numbers

In an *unsigned number,* each bit carries a certain binary weight (as we discussed earlier), according to its position within the number. The least-significant bit has a weight of 2^0 (decimal 1) and the most-significant bit—the sixteenth bit—has a weight of 2^{15} (decimal 32,768). Therefore, if all bits contain zero, the unsigned number has the value 0, and if all bits contain one, the unsigned number has the value 65,535.

Signed Numbers

In a *signed number,* only the 15 least-significant bits represent data; the most-significant bit represents the sign of the number. The most-significant bit is 0 if the number is positive or zero and 1 if the number is negative. Positive signed numbers can have values between 0 (binary 000 . . . 00) and +32,767 (binary 011 . . . 11). Negative signed numbers can have values between −1 (binary 111 . . . 11) and −32,768 (binary 100 . . . 00).

Two's Complement

Why is −1 represented by binary 1111111111111111, rather than by binary 1000000000000001? The answer is that negative signed numbers are represented in their *two's complement form.* The two's complement form was introduced to eliminate the problems associated with allowing zero to be represented in two different forms, all zeroes (the positive form) and all zeroes with a 1 in the sign bit (the negative form). Using two's complement, zero is represented by only one form, all zeroes.

To derive the binary representation of a negative number (that is, its two's complement form), simply take the positive form of the number and reverse the sense of each bit—change each 1 to a 0 and each 0 to a 1—then add 1 to the result. The following example shows the steps required to derive the two's complement binary representation of −32.

```
 00000000 00100000   +32
 11111111 11011111   One's complement
+                1   Add 1
-------------------
 11111111 11100000   Two's complement
```

Fortunately, FORTH has the two's complementing words NEGATE and DNEGATE (called MINUS and DMINUS in fig-FORTH), so you will probably never have to go through this exercise yourself. However, this information may prove to be valuable if your program includes some assembly language routines that manipulate signed numbers.

FORTH AND NUMBER BASES

It is important to realize that the various numbering systems are simply alternate ways for you to *represent* the information being processed by the computer. The fact that you choose to represent that information in binary rather than in decimal, or in hexadecimal rather than in binary, has absolutely no effect on how the computer's *microprocessor* operates on that information. To a microprocessor, the information in memory is nothing more than a series of binary patterns which are processed according to the machine code instruction that is being executed at any given time.

BASE Selects the Numbering System

With that aside, let us say that whereas most computer languages offer a choice of no more than three or four numbering systems (base 2, 10, 16, and occasionally 8), FORTH offers *69 different numbering systems,* ranging from base 2 to base 70. The numbering system that is currently active is determined by the contents of a system variable called *BASE*.

If BASE has a value of 2, FORTH will expect all numbers to be entered in binary form. Similarly, if BASE has a value of 10 or 16, FORTH will expect all numbers to be entered in decimal or hexadecimal form.

Changing the Number Base

When you first enter FORTH, the BASE variable will have the value 10, which selects decimal numbering. If you wish to change to some other number base, simply store that number into BASE. For example, the sequence

```
16 BASE !
```

selects hexadecimal numbering. Once BASE has been changed, FORTH will expect all numbers to be entered in the

selected number base, and will print out all results in that number base.

After entering 16 into BASE, we can operate with hexadecimal numbers directly, such as

```
3AF 6E + . 41D OK
```

This will remain in effect until the value of BASE is altered.

How can we return to the decimal system from hexadecimal? A common error is to attempt this sequence:

```
10 BASE !
```

Why is this sequence erroneous? Because with BASE equal to 16, FORTH assumes that the stack value 10 represents hexadecimal 10 (which is decimal 16) rather than decimal 10. The proper sequence to restore the decimal mode is:

```
A BASE !
```

since hexadecimal A is equivalent to decimal 10.

BASE is one of four FORTH words (see Table 9-2) that can be used to select a number base. Let's look at two of the other words, DECIMAL and HEX.

Table 9-2. Number Base Control Words

Word	Stack	Action	Notes
BASE	--- addr	Leaves the address of a user variable containing the current number base used for input and output conversion.	
DECIMAL		Sets the input/output numeric conversion base to 10.	
HEX		Sets the input/output numeric conversion base to 16.	(1)
H.	n ---	Outputs n as a hexadecimal integer with one trailing blank. The current base is unchanged.	(2)
OCTAL		Sets the input/ output numeric conversion base to 8.	(2)
O.	n---	Outputs n as an octal integer with one trailing blank. The current base is unchanged.	(2)

Notes: (1) Included in Reference Word Set, as a Standard Word Definition.

(2) Included In Reference Word Set, as an uncontrolled word definition.

DECIMAL AND HEX

The Required Word DECIMAL gives you an easy way to exit from a nondecimal numbering system, by storing the decimal value 10 into BASE. You may also want to make a momentary side trip into decimal, to print out some result or to input some decimal data.

Of course, if the decimal mode is just temporary, the original number base must be saved, then stored back into BASE when your decimal work is done. This requires the sequence

```
BASE @          ( Fetch number from BASE)
DECIMAL         ( Call up decimal mode)
..              ( Perform decimal operations)
..
BASE !          ( Restore original number base)
```

For example, if BASE is set to 16 and the stack holds the hexadecimal values 3AF and 6E, the sum of these numbers could be printed out in decimal with the sequences

```
+ BASE @ DECIMAL SWAP . 1053 OK
BASE !
```

The FORTH-79 Reference Word Set contains a similar word, HEX, which stores decimal 16 into BASE, thereby placing FORTH in the hexadecimal mode. Associated with this mode is the uncontrolled word [H.], which displays a number in hexadecimal form without affecting the current base. The definitions of these words are:

```
DECIMAL : HEX 16 BASE ! ;
: H. BASE @ SWAP HEX . BASE ! ;
```

You may be wondering why the word DECIMAL precedes the definition of HEX rather than falls inside the definition. The reason is simple: A BASE-changing word placed outside a definition takes effect when the definition is *compiled,* whereas a BASE-changing word placed inside a definition takes effect when the defined word is *executed.* Therefore, the word DECIMAL preceding the definition of HEX tells the compiler, "The following definition has numbers which are to be interpreted as decimal numbers." By contrast, the word HEX in the definition of [H.] causes FORTH to switch to decimal mode temporarily, before [.] is executed. Recognizing this distinction may help keep you out of trouble when you are developing your own FORTH programs.

Numbering systems can be mixed freely in programs, as long as you keep track of which base is active at any given time. Example 9-1 shows a colon-definition that has both HEX and DECIMAL in the same DO-LOOP. This word, .NUMBERS, displays *n* consecutive numbers in memory, starting at *addr*. Thus, .NUMBERS is similar to DUMP, which displays n consecutive *bytes*. However, .NUMBERS can be more valuable in certain debug operations, because it displays the address in hexadecimal and the number in decimal. (DUMP displays both in the current number base.)

Example 9-1. Display consecutive numbers in memory

```
: .NUMBERS
  ( Display n consecutive numbers in memory, starting  )
  ( at addr. For each number, the address is shown in  )
  ( hexadecimal and the number is shown in decimal.    )
  ( addr n ---- )
  BASE @                  ( Fetch active number base)
  ROT ROT                 ( addr n base ---- base addr n)
  0                       ( DO-LOOP start = 0)
  DO
   CR DUP HEX U.          ( Display address)
   DUP @ DECIMAL .        ( Display number)
   2+                     ( Address next word to be printed)
  LOOP
  DROP                    ( Discard last address on stack)
  BASE ! ;                ( Restore previous number base)
```

While .NUMBERS is executing, the current number base is preserved with the sequence (BASE @) at the beginning of the definition and the sequence (BASE !) at the end. A typical run of .NUMBERS will look like this:

```
HEX OK
B000 5 .NUMBERS
B000 31052
B002 19650
B004 -15743
B006 269
B008 -30019 OK
```

OTHER NUMBER BASES

The FORTH-79 Standard includes one other number base control word, OCTAL, as an uncontrolled definition. Seldom used now is the octal numbering system with the base eight, which means its digits run from 0 to 7. Associated with the octal system is another uncontrolled word, [O.], which displays a number in octal form without affecting the current case. The definitions of these two words are:

```
DECIMAL : OCTAL 8 BASE ! ;
: O. BASE @ SWAP OCTAL . BASE ! ;
```

With BASE, it is possible to define a word for any number base between 2 and 70 (trivia buffs are directed to Table 9-3). However, for most applications, the only other numbering system needed is *binary,* base 2, which can be obtained with this word:

```
DECIMAL : BINARY 2 BASE ! ;
```

The binary numbering system is typically used to operate on individual bits within memory or peripheral control registers. To

Table 9-3. Names of Various Number Bases

Base	Name
2	Binary
3	Ternary
4	Quatenary
5	Quinary
6	Senary
7	Septenary
8	Octal, or octonary
9	Novenary
10	Decimal
11	Undecimal
12	Duodecimal
13	Terdenary
14	Quaterdenary
15	Quindenary
16	Hexadecimal, or sexadecimal
17	Septendecimal
18	Octodenary
19	Novemdenary
20	Vicenary
32	Duosexadecimal, or duotricinary
60	Sexagenary

display a number in its binary form, you could add this word to your dictionary:

```
: B. BASE @ SWAP BINARY . BASE ! ;
```

fig-FORTH NUMBER BASE CONTROL WORDS

fig-FORTH provides only the words BASE, DECIMAL, and HEX, defined as in Table 9-2. However, you can add the other words shown in this table by entering the definitions given in this chapter.

Table 9-4. Words Added to FORTH in Chapter 9

Word	Stack	Action
.NUMBERS	addr n ---	Displays n consecutive numbers in memory, starting at addr. For each number, the address is shown in hexadecimal and the number is shown in decimal.
BINARY		Sets the input/output numeric conversion base to 2.
B.	n ---	Outputs n as a binary integer with one trailing blank. The current base is unchanged.

SUMMARY

This chapter began by describing the way information is stored in a computer's memory; as a series of binary bits, usually addressed eight bits at a time, which is called a byte. From there we moved on to a discussion of the hexadecimal numbering system, which represents four bits as a single digit (0 through 9 and A through F).

These fundamentals paved the way for a description of BASE, the system variable that selects the form in which data will be entered into the system and displayed on the screen or printer. The FORTH words DECIMAL and HEX manipulate BASE to produce decimal or hexadecimal input/output, respectively.

Table 9-4 summarizes the words that can be added to FORTH based on the material in this chapter.

CHAPTER 10

Interacting With FORTH Programs

In the preceding chapters, you and the computer have assumed roles similar to those of a playwright and an actor. As playwright (programmer), you wrote a "script" (program) that informed the actor what to do. With your job done, the actor (computer) read the script and performed it, while you sat back as a passive observer.

However, "real-life" applications often require you to take an active part in the production. That is, you will often wish to assume the additional duties of director, to give the actor (again, the computer) new commands that will affect the outcome of the play.

As an engineer or an accountant, for instance, you may need to halt the computer while you supply a new dimension or a new interest rate. Similarly, as a game-player, you may need to halt the computer while you enter your next move. These applications, and many others, require a certain amount of interaction between the operator/programmer and the computer, while a program is *running*.

This chapter covers the FORTH words you will need to interact with the computer; words that accept information from the keyboard and words that transmit information to a printer or display screen. As we learned in Chapter 4, this kind of information is usually transmitted in a special 8-bit code called *ASCII,* so many of the FORTH words presented here involve operations on ASCII characters. Appendix A contains a summary of the ASCII character set, for reference purposes.

This chapter also covers a related topic: words that are used to format information on a printer or display screen to make that information more understandable. Table 10-1 summarizes all the words that will be discussed here.

Table 10-1. Character and String Input/Output Words

Word	Stack	Action	Notes
KEY	--- char	Leaves the ASCII value of the next available character from the current input device.	
EMIT	char ---	Transmits ASCII character to the current output device.	
BELL		Activates a terminal bell or noise-maker appropriate to the device in use.	(2)
BL	--- char	Leaves the ASCII character for "blank" (decimal 32).	(3)
SPACE		Transmits an ASCII blank to the current output device.	
EXPECT	addr n ---	Transfers characters from the terminal to memory, starting at addr, until a "return" or n characters have been received. One or more nulls are added at the end of the text.	
TYPE	addr n ---	Transmits n characters, beginning at addr, to the current output device.	
COUNT	addr --- addr+1 n	Leaves the address addr+1 and the character count of a string beginning at addr. The byte at addr must contain the character count n. COUNT is usually followed by TYPE.	
PAD	--- addr	Leaves the address of a scratch area used to hold character strings for intermediate processing. The minimum capacity of PAD is 64 characters (addr through addr+63).	
SPACES	n ---	Transmits n spaces to the current output device.	
-TRAILING	addr n1 --- addr n2	Adjusts the character count n1 of a text string beginning at addr to exclude trailing blanks.	
.R	n width ---	Prints the number n, right-justified within the field width.	(2)
D.R	d width ---	Prints the double number d, right-justified within the field width.	(1)
U.R	u width ---	Prints the unsigned number u, right-justified within the field width.	(3)

Table 10.1—cont. Character and String Input/Output Words

Word	Stack	Action	Notes
<#		Used in the form: <# . . . #> to begin a process that converts an unsigned double number into an ASCII character string, stored in right-to-left order.	
#	ud1 --- ud2	Converts one digit of an unsigned double number into an ASCII character, and puts that character into an output character string.	
#S	ud --- 0 0	Converts all remaining digits of an unsigned double number into ASCII characters, and puts those characters into an output character string. Adds a single zero to the output string if the value is zero.	
HOLD	char ---	Inserts an ASCII character into an output character string, at the current position.	
SIGN	n ---	Inserts an ASCII minus sign into the output character string, if n is negative. Usually used immediately before [#>], to produce a leading minus sign.	
#>	d --- addr n	Terminates numeric conversion. Leaves start address and character count of the character string, suitable arguments for TYPE.	
CONVERT	d1 addr1 --- d2 addr2	Converts the text beginning at addr1+1 to a double number, with regard to BASE. The new value is accumulated into d1, and left as d2. addr2 is the address of the first nonconvertible character.	
NUMBER	addr --- d	Converts the text beginning at addr+1 to a double number, with regard to BASE. If numeric conversion is not possible, an error condition exists. The string may contain a leading minus sign.	(2)

Notes: (1) Included in Double Number Extension Word Set.
(2) Included in Reference Word Set, as an uncontrolled word definition.
(3) Included in Reference Word Set, as a Standard Word Definition.

CHARACTER OPERATIONS

The most fundamental kind of ASCII transfer operation is one that involves just one character; either reading a character in from the keyboard or transmitting a character to a printer or display. The word KEY waits for the current input device (usually a keyboard) to supply a character, and then places that character on the top of the stack. Conversely, the word EMIT takes an ASCII character off the top of the stack and transmits it to the current output device (usually a printer or display screen, or both).

Because the word KEY induces a built-in wait until a key is pressed, typing in KEY and Return alone will not produce FORTH's usual OK message; OK will not be displayed until you've pressed another key. Moreover, the single key itself will never be displayed. If you have a FORTH system, type in KEY, then press Return, then press A. This should produce:

```
KEY OK
```

Now try it again, but follow the "A" with a [.], so that the top value is printed. This should produce:

```
KEY OK
. 65 OK
```

where, from Appendix A, "65" is the ASCII representation of the letter A, in decimal.

Granted, a readout of "65" is meaningless unless you're sitting at the terminal with an ASCII table close at hand. To make such ASCII values more readily understandable, FORTH provides a second word, EMIT, which converts this number back to its character form, and displays that character.

Now try the KEY-Return-A sequence again, but follow A with the word EMIT. This should produce:

```
KEY OK
EMIT AOK
```

Here "AOK" is not a message from an astronaut, but rather the printout of our ASCII key character, "A", immediately followed by the FORTH message "OK".

What are some possible uses for single-character transfers? One use is in selecting an option from a "menu" of options. For example, a small business may have disk files containing summary data of accounts receivable, accounts payable, current inventory, payroll information, and so on. If a single digit iden-

tifier is assigned to each of these files, a KEY operation could select one of these files for listing on a printer.

The fact that the response to KEY is not printed suggests another possible use for a single character: as a user access code for a classified system. Entering the correct character will grant the user access to the system; an incorrect character may do anything from print out a message ("Invalid code!") to setting off an alarm. For example,

```
: WAIT-A
  BEGIN KEY 65 = UNTIL ;
```

waits until the letter A is entered at the keyboard before continuing on. If necessary, WAIT-A will loop for years until someone keys in an A! Example 10-1 shows the definition of WAIT-CHAR, a more general-purpose word that will wait for a key whose ASCII value is on top of the stack. For example, this sequence in a program:

```
54 WAIT-CHAR
```

causes the computer to wait until someone presses the "6" key (ASCII 54 in decimal) before continuing.

Example 10-1. Wait for a specified key

```
: WAIT-CHAR
  ( Loop until a character entered from the keyboard )
  ( matches the ASCII value on the top of the stack.  )
  ( char --- )
  BEGIN
    DUP                  ( Duplicate char)
    KEY =                ( Fetch key and compare it to char)
  UNTIL
  DROP ;                 ( Upon match, remove char from stack)
```

Special-Purpose Character Words BELL, BL, and SPACE

FORTH defines three character operations as words because they are used so often. The first, BELL, sends a BEL character (ASCII 7) to the current output device, which should activate a bell, buzzer, or other noise-maker. A handy word for interactive applications, BELL should be used prudently, to preserve the mental health of the user!

The second word, BL, leaves the ASCII value for a blank (decimal 32) on top of the stack. That is, it operates the same as pressing the space bar in response to KEY.

The third word, SPACE, is a variation of BL that *transmits* an ASCII blank to the current output device, without having to push that value onto the stack. That is, SPACE performs the same task as BL followed by EMIT. Like CR, SPACE is useful for separating information being output.

STRING OPERATIONS

Most operations involve a series of characters—that is, a *string* of characters—rather than just a single character. The ASCII characters that comprise the string may represent a number, a portion of text (a phrase, sentence, paragraph, etc.) or a combination of numbers and text. In any case, the interpretation of the string is strictly up to you, because FORTH views a string as simply a series of consecutive bytes in memory, nothing more nor less.

Within a string, each byte holds the ASCII value for the character it represents. You can structure strings any way you want, but FORTH is set up to handle strings as a series of character bytes *preceded* by a byte that holds the count of the number of characters in the string. Because this counter is a *byte* value, FORTH can easily handle strings that have up to 255 characters.

Fig. 10-1 shows how the 11-character string

NO STRINGS!

would be stored in memory. Note that even the blank space between the two words is included in the character count. Abbreviated SP (for space) in Appendix A, it has an ASCII value of decimal 32.

String-Transfer Words

Besides the character transfer words KEY and EMIT, FORTH also has two string transfer words, called EXPECT and TYPE. Both words take two stack parameters: a string starting address (addr) and a character count (n).

The word EXPECT accepts characters from the terminal until *n* characters have been received or until you press the Return key. As each character is received, it is stored into memory at

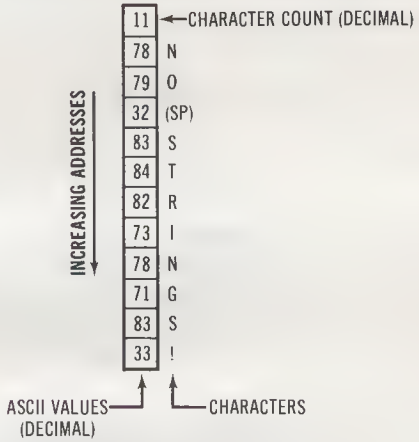


Fig. 10-1. A string in memory.

one byte location higher than the preceding character. Therefore, the first character is stored at addr, the second character is stored at addr+1, and so on. (Unlike KEY characters, most FORTHs display EXPECT characters as they are entered.) When all characters have been received, FORTH adds one or two null characters (ASCII 00) to the end of the text.

For example, the sequence

```
200 25 EXPECT
```

will accept up to 25 characters from the terminal, and store them starting at decimal address 200. You have your choice of entering all 25 characters or entering some lesser number and pressing Return to tell FORTH that you are done. If you are currently using FORTH on a computer, use an EXPECT to put in the string "NO STRINGS!", then examine the contents of memory with DUMP. You should find the byte values shown in Fig. 10-1, minus the count byte, but with one or two null (00) bytes at the end.

Since EXPECT does not produce a count byte, it is not immediately evident how many characters are in the string, unless you've bothered to keep count yourself. However, FORTH gives you a way to determine the length of the string, because it planted one or two null characters at the end. Therefore, the length of the string is simply the difference between the start-

Example 10-2. Length of a string

```
: LEN
  ( Return the character count of the string that starts )
  ( at addr.                                             )
  ( addr --- count )
  255 0                    ( Check 255 characters, maximum)
  DO
   DUP I +                 ( Address = addr + I)
   C@                      ( Fetch next character from memory)
   0=                      ( Is it a NUL?)
   IF
    I LEAVE                ( If so, put count on stack)
   THEN
  LOOP
  SWAP DROP ;              ( Delete addr, leaving only count)
```

ing address of the string and the address of the first null character (ASCII 00).

The word LEN, in Example 10-2, calculates the character count of a string, by checking each byte in the string against the NUL character. LEN is programmed using a DO-LOOP whose index is the offset from the string's start address. Upon encountering NUL, this index is placed on the stack, to reflect the character count.

By combining LEN and EXPECT, you can convert a string received from the terminal into the standard FORTH format, count followed by characters. All you need to do is store the count produced by LEN into the byte that precedes the string in memory; if the string starts at addr, the count should be stored at addr-1. For example, the sequence

```
200 DUP 20 EXPECT NO STRINGS!_OK
                              ↑
                              Return
                              pressed here
```

reads "NO STRINGS!" into memory, starting at location 200. Then, the sequence

```
DUP LEN SWAP 1- C! OK
```

calculates the length of the string, and stores it at location 199. At this point, the pattern of bytes in memory, starting at location 199, should match Fig. 10-1 exactly.

It is certainly possible to combine the EXPECT and LEN tasks

in a single word that transfers a number of keystrokes into memory and precedes it with a count byte. This new word, ACCEPT, is defined in Example 10-3. (Note that ACCEPT requires the address of the count byte, not the address of the first string character.) Now our sample string can be input with the simpler sequence

```
199 20 ACCEPT NO STRINGS! OK
```

Example 10-3. Transfer a string, with length count

```
: ACCEPT
  ( Transfer characters from the terminal until a "return"  )
  ( or a count or n has been received. The characters are )
  ( stored as a packed string, with the length byte at addr. )
  ( addr n --- )
  OVER 1+ DUP ROT .      ( Set up addr+1 for EXPECT and LEN)
  EXPECT                 ( Transfer n keys, starting at addr+1)
  LEN                    ( Calculate the string length)
  SWAP C! ;              (  and store it at addr       )
```

The FORTH word TYPE performs the opposite function of EXPECT. That is, TYPE transmits a string in memory to the current output device, which is normally a printer or a display screen, or both. TYPE takes two stack arguments: the address of the string's first character byte (addr) and a count of characters to be transmitted (n). For example, the sequence

```
200 11 TYPE
```

could be used to print the "NO STRINGS!" string we just entered.

Of course, typing in "11" here presumes you know how many characters are in the string. But even if you don't know how many characters are in the string, you know where that count can be found: in the byte that precedes the first character. If the first character is stored at addr, the count is stored at addr-1.

FORTH has a special word, COUNT, that is designed to fetch the character count of a string. Just give COUNT the address of the count byte and it will return the address of the first character byte and the string's character count—the very parameters required by TYPE! Therefore, the sequence

```
199 COUNT TYPE
```

prints out our "NO STRINGS!" string without having to worry about how many characters are in the string.

Put Temporary Strings in the PAD Area

FORTH systems provide a scratch area in read/write memory that can be used to hold character strings for intermediate processing. This area starts at a fixed distance above the most recently defined entry in the dictionary, so it's starting address will change as words are added to or removed from the dictionary. Normally this poses no problem, because temporary strings are nearly always input and output with two consecutive operations—typically an EXPECT followed by a TYPE—with no intervening dictionary changes.

At any given time the starting address of the scratch area can be placed on the stack by executing the word PAD. The FORTH-79 Standard specifies the minimum capacity of the PAD area to be 64 bytes, which means that it will hold at least 62 ASCII characters, a character count byte, and a null character. The number 64 corresponds to the standard 64-character line length of a FORTH *screen* (you will see more about this later). In fig-FORTH, the PAD area starts 68 bytes above the top of the dictionary.

PAD, therefore, provides an area of memory that you *know* will be free to receive your input strings, making it a convenient place to store them.

An Interactive Game: "Twenty Questions"

To illustrate a possible use of PAD and the other words in this chapter—and to provide some much-needed diversion—we will look at a FORTH program that "plays" the popular parlor game *Twenty Questions*. For the benefit of readers who are unfamiliar with Twenty Questions, it is a game in which one or more players attempt to identify some unknown animal, vegetable, or mineral based on another player's yes or no responses to a series of up to 20 questions.

Here, the "player" that will provide the responses to questions is a *computer*. Your computer! Upon receiving a question typed in at the terminal, the computer will print or display the proper answer to the question, either YES or NO.

Before examining the program that will perform this task, let's look at the responses for a typical Twenty Questions session. Here they are:

20-QUESTIONS

ENTER QUESTION 1
Is it mineral?
NO.

ENTER QUESTION 2
Is it vegetable?
YES!

ENTER QUESTION 3
Does it grow under the ground?
NO.

ENTER QUESTION 4
How about above?
YES!

ENTER QUESTION 5
Are you sure?
YES!

ENTER QUESTION 6
Does it grow on trees?
YES!

ENTER QUESTION 7
Is it green?
NO.

ENTER QUESTION 8
Are people likely to make juice from it?
NO.

ENTER QUESTION 9
Is it a black olive?
YES!

You are certainly curious about the complexity of a program that possesses such intelligence, and must surely wonder how many months, or perhaps years, of programming it took to provide this intelligence. Well, you will be interested to know that the entire Twenty Questions program can by typed into your computer in a matter of *minutes!* In fact, it consists of just a few lines of FORTH code, those shown in Example 10-4.

The "trick" is that the program does not respond to the entire question at all, but merely prints out a message based on

the character at the end of the question—the character that precedes the question mark (?). If this character is an "E" or an "S", the program prints the message "YES!"; otherwise, if the question ends in any other letter, the program prints the message "NO."

This clever idea, adapted from the Atari BASIC and Atari PILOT versions given in COMPUTE! magazine[1], is guaranteed to provide you with hours of pleasure mystifying your friends and relatives. You may also wish to consider adding the letter L to the "YES!" list, so that users will receive a positive response to the questions "Is it an animal?" or "Is it a mineral?"

Example 10-4. "Twenty Questions" Game

```
: 20-QUESTIONS
  ( Provides yes/no answers to up to 20 questions entered      )
  ( from the terminal. Questions can be up to 60 characters    )
  ( long. Uses the word LEN; Example 10-2.                     )
  21 1
  DO
   CR CR." ENTER QUESTION " I . CR
   PAD 60 EXPECT          ( Wait for question)
   PAD LEN                ( Calculate its length)
   PAD + 2 - C@           ( Read next-to-last character)
   CR DUP 69 =            ( Is it an E?)
   IF ." YES!" ELSE       (  If so, print "YES!")
    83 =                  ( Is it an S?)
    IF ." YES!" ELSE      (  If so, print "YES!")
     ." NO."              (  Otherwise, print "NO.")
    THEN
   THEN
  LOOP CR
  ." THAT'S TWENTY QUESTIONS!" CR
  ." TYPE 20-QUESTIONS TO PLAY AGAIN." ;
```

FORMATTING TEXT

In addition to CR, SPACE and the other formatting words we've encountered, FORTH has two other words that are useful

[1]David D. Thornburg, "Computers And Society," *COMPUTE!*, September, 1981, pp. 10-16.

for formatting text strings. One of these, SPACES, transmits multiple blank characters to the current output device, taking the count from the top of the stack. SPACES can be used, for instance, to create a header for a table, as in the following sequence, which writes the labels NUMBER, SQUARE, and CUBE, each separated by six spaces:

```
." NUMBER" 6 SPACES ." SQUARE" 6 SPACES ." CUBE"
```

When executed, this sequence will produce the following printout:

The other useful string formatting word is −TRAILING, which shortens the character count of a string to exclude all trailing blanks, so that they will not be printed. For example, suppose the string

```
THIS IS ALL␣␣␣
```

is stored starting at location 200 in memory (the "␣" symbols represent trailing blanks). A sequence to type out this string is:

```
199 COUNT CR TYPE
THIS IS ALL    OK
```

The three spaces preceding OK reflect the trailing blanks in the string. These spaces in the printout will be eliminated by inserting −TRAILING into the sequence, as follows:

```
199 COUNT −TRAILING OK
CR TYPE
THIS IS ALL OK
```

−TRAILING only adjusts the character count for the print operation, however, the count byte is unaffected.

FORMATTING NUMBERS

Just as text can be displayed in various formats, numbers can also be displayed in various formats, using FORTH words that are especially designed for this task. We will examine two different kinds of number-formatting words here: words that simply display numbers right-justified in a specified character field

and words that can intersperse the numeric digits with dollar signs, colons, decimal points, and other kinds of characters.

Display Numbers Right-Justified

The simplest kinds of number-formatting words are those that cause the stack value to be displayed right-justified in a given field. These words, all variations of the print words we studied in Chapter 2, are [.R], [D.R], and [U.R]. As you may have guessed, [.R] displays a number right-justified, [D.R] displays a double number right-justified, and [U.R] displays an unsigned number right-justified. Each of these words takes two stack parameters: the numeric value itself and a field width specifier. Here are examples of each word, using a six-character field in each case:

```
−25 6 .R −250K  ( Number )
−25. 6 D.R −250K ( Double number, indicated by "." )
350 6 U.R 3500K  ( Unsigned number )
```

These three words can be used to generate a highly readable tabulation, with columns of numbers aligned on the printer paper or on the display screen. For instance, we can now develop a more elegant version of our earlier print-numbers word, .NUMBERS (Example 9-1), one in which the numbers are displayed four per line, and neatly aligned in columns. Example 10-5 shows the definition for this new word, called DUMP-N.

DUMP-N is similar to .NUMBERS, but DUMP-N displays four numbers on a line instead of just one, and [.R] is used to display each number right-justified in a field of eight characters. The construction of DUMP-N is very similar to that of .NUMBERS, too, except that DUMP-N includes a second DO-LOOP, which prints the four numbers on each line. A typical execution run of DUMP-N should look like this:

```
HEX B000 A DUMP-N
B000          31052          19650         −15742            269
B008         −30019            768            127          14780
B010            146            −51            640            310
OK
```

Display Numbers With Imbedded Characters

You will often want to display numbers that represent the time of day, a telephone number, a date, or an amount of money; numbers such as

2:45:55 726-6286 11/30/81 $2500.12

Example 10-5. Formatted dump of numbers in memory

```
: DUMP-N
  ( Display n consecutive numbers in memory, starting )
  ( at addr. Numbers are displayed four per line, in  )
  ( decimal. The address of the first number on each  )
  ( line is also displayed, in hexadecimal.           )
  ( addr n --- )
  BASE @                    ( Fetch active number base)
  ROT ROT                   ( addr n base --- base addr n)
  0                         ( DO-LOOP start = 0)
  DO
   CR DUP HEX U.            ( Display first address on line)
   4 0                      ( Prepare to display 4 numbers)
   DO
    DUP @ DECIMAL 8 .R      ( Formatted display, in fields of 8)
    2+                      ( Address next word to be displayed)
   LOOP
   4                        ( Increment outer loop by 4)
  +LOOP
  DROP CR                   ( Discard last address on stack)
  BASE ! ;                  ( Restore previous number base)
```

That is, you will want to display numbers that have one or more imbedded characters.

To do this, you need to have digit-by-digit control over the way the number is displayed, and have the ability to insert characters of your choice as needed. To provide this control, FORTH has a special number-formatting construct that begins with the word [<#] and ends with the word [#>].

This construct takes a double number from the top of the stack and converts it to a string of ASCII characters in memory. Some of these characters will be digits, but others may be symbols, punctuation marks, or other characters, according to the operations specified within the construct. *The double number is always converted in right-to-left order, with the least-significant digit being converted first.*

At the end of the conversion process, the terminating word #> pushes the string's starting address and character count onto the stack. These parameters generally serve as inputs for the word TYPE, which displays the string on a screen or printer. The general form of the number-formatting construct is:

<# . . . #>

where " . . ." represents one or more special character-generating words. Four such words are available for use in this construct:

1. [#] converts the next digit of an unsigned double number to an ASCII character, and puts that character into the output string. If no digits remain, a zero is added to the output string.
2. HOLD inserts the ASCII value at the top of the stack into the output string, at the current position. For example,

```
47 HOLD
```

   inserts a slash (/) character into the output string.
3. [#S] converts all remaining digits of an unsigned double number to ASCII characters, and puts them into the output string. If no digits remain, a single zero is added to the output string.
4. SIGN inserts a minus sign into the output string if the number at the top of the stack is negative.

The requirements of [#], [#S], and SIGN provide the rules that must be followed in forming the output string. They are:

1. The words [#] and [#S] operate only on *unsigned double numbers*. If the value to be formatted is not an unsigned double number, you must convert it to one.
2. If the value to be formatted is *signed*, its high-order 16 bits must be saved somewhere for later use by SIGN. The logical place to save it is on the stack, beneath the unsigned double number.

Table 10-2 lists the sequence of operations for numbers and double numbers, both signed and unsigned. For both kinds of signed values, the 16 bits containing the sign are placed beneath the double number, then rotated to the top just before SIGN is executed.

Clearly, the simplest use of this construct is to print an un-

Table 10-2. Operations Using the <# . . . #> Construct

Number to be printed	General sequence of operations
Unsigned double number	<# . . . #>
Signed double number	SWAP OVER DABS <# . . . ROT SIGN #>
Unsigned number	0 <# . . . #>
Signed number	DUP ABS 0 <# . . . ROT SIGN #>

signed double number, without modifying it in any way. This gives us the following new FORTH word:

```
: UD. <# #S #> TYPE ;
```

An example of its use is:

```
7266286. UD. 7266286OK
```

This particular definition causes the number to be printed directly beside FORTH's "OK", with no space between them. That's easily remedied by adding a SPACE, as in

```
: UD. <# #S #> TYPE SPACE ;
```

Now we get

```
7266286. UD. 7266286 OK
```

Now we suppose that you want to print the stack value as a telephone number. All this involves is inserting a hyphen character between the third and fourth most significant digits, using the word HOLD. The definition for our phone-number-printing word is:

```
: .PHONE# <# # # # # 45 HOLD # # # #> TYPE SPACE ;
```

where (45 HOLD) puts the ASCII value for hyphen, decimal 45, into the output string. Note that we have seven occurrences of [#] here, four for the least-significant digits and three for the most-significant digits. However, the last three #'s are unnecessary; they can be replaced with [#S]. This allows the shorter definition:

```
: .PHONE# <# # # # # 45 HOLD #S #> TYPE SPACE ;
```

Using our earlier example value with .PHONE#, we get

```
7266286. .PHONE# 726-6286 OK
```

In the examples just given, the definitions for the words UD. and .PHONE#, the results were printed in decimal, with the unstated implication that BASE had the value 10. To be consistent with other FORTH number-printing words, UD. should always print its result in the current number base. However, since phone numbers will, presumably, always be needed in decimal, we should have forced BASE into the decimal mode for .PHONE#. (Do you want to eliminate unwanted phone calls? Have the phone number on your business card printed in *octal*, and reveal the secret only to your friends!) Example 10-6 gives a more fool-proof definition of .PHONE#.

Example 10-6. Print telephone number

```
: .PHONE#
  ( Print the unsigned double number on the stack as a )
  ( telephone number, in the format xxx-xxxx.          )
  ( ud --- )
  BASE @                ( Fetch current base)
  ROT ROT               ( ud base --- base ud)
  DECIMAL               ( Enter decimal mode)
  <# # # # #            ( Convert four low-order digits)
   45 HOLD              ( Insert hyphen in output string)
   #S                   ( Convert remaining digits)
  #>                    ( Leave addr and n for TYPE)
  TYPE SPACE            ( Print phone number)
  BASE ! ;              ( Restore previous base)
```

The definitions for both UD. and .PHONE# required an unsigned double number as the stack argument. Let's examine the remaining features of our number-formatting construct by looking at an application that accepts a *signed number* as an argument—a word that prints out a dollar value, such as $136.55 (a credit) or −$136.55 (a debit).

For this application the <# . . . #> construct must leave the signed number on the stack (for the benefit of SIGN), but put an unsigned double number version of this value above it. From Table 10-2 we know that the required set-up sequence for signed numbers is

```
DUP ABS 0
```

We also know that for dollar amounts, the construct must add two characters to the output string: a decimal point, following the two least-significant ("cents") digits, and a dollar sign, following the most-significant ("dollar") digit. The minus sign, if required, is added by SIGN, which should be the final word in the construct.

With all of these considerations, we can now define the money-printing word .MONEY, as shown in Example 10-7. Note that this word, like the final version of .PHONE#, forces BASE into decimal mode, but preserves the current base on the stack. Sample executions of .MONEY are:

```
13655 .MONEY $136.55 OK
−12366 .MONEY −$123.66 OK
```

Example 10-7. Print dollar amount (−$327.68 to $327.67)

```
: .MONEY
  ( Print the signed number on the stack as a dollar )
  ( quantity, in the format $xxx.xx or −$xxx.xx      )
  (  ---  )
  BASE @                ( Fetch current base)
  SWAP                  ( n base --- base n)
  DUP ABS 0             ( base n --- base n ud)
  DECIMAL               ( Enter decimal mode)
  <# # #                ( Convert cents digits)
   46 HOLD              ( Add decimal point)
   #S                   ( Convert remaining, dollar digits)
   36 HOLD              ( Add dollar sign)
   ROT SIGN             ( Add minus sign, if needed)
  #>                    ( Leave addr and n for TYPE)
  TYPE SPACE            ( Print the string)
  BASE ! ;              ( Restore previous base)
```

The fact that the stack argument is a number limits .MONEY quantities between −$327.68 and $327.67, which is hardly adequate for most monetary applications. Example 10-8 shows the definition of a more useful word, .MONEY-D$, which accepts a signed *double number* argument, and can, therefore, print amounts from −$21,474,836.48 to $21,474,836.47—just what we need to keep track of our Irish Sweepstakes winnings!

With .MONEY-D$, you can expect these kinds of results:

```
211111. .MONEY-D$ $2111.11 OK
0. .MONEY-D$ $0.00 OK
−2145678. .MONEY-D$ −$21456.78 OK
```

And with minimal extra effort you should be able to modify .MONEY-D$ so that it prints a comma between each thousand-dollar division.

CONVERTING TEXT TO NUMBERS

In most cases you enter numbers into the computer by simply typing them in. FORTH interprets this stream of digits as a number, and automatically converts the digits to a numeric value, which it places on the stack. However, if a number is to

Example 10-8. Print dollar amount (−$21,474,836.48 to $21,474,836.47)

```
: .MONEY-D$
  ( Print the signed double number on the stack as a dollar )
  ( quantity, in the format $xxxx.xx or −$xxxx.xx.          )
  ( d --- )
  BASE @                ( Fetch current base )
  ROT ROT               ( d base --- base d)
  SWAP OVER DABS        ( base d --- base n ud)
  DECIMAL               ( Enter decimal mode)
  <# # #                ( Convert cents digits)
   46 HOLD              ( Add decimal point)
   #S                   ( Convert remaining, dollar digits)
   36 HOLD              ( Add dollar sign)
   ROT SIGN             ( Add minus sign, if needed)
  #>                    ( Leave addr and n for TYPE)
  TYPE SPACE            ( Print the string)
  BASE ! ;              ( Restore previous base)
```

be entered interactively, *you* must define the ASCII-to-numeric conversion procedure.

Fortunately, there is a FORTH word that converts a text string in memory to a numeric value. This word CONVERT (or NUMBER in fig-FORTH) takes a double number and an address *from* the stack and returns a double number and an address *to* the stack. Therefore, CONVERT's before-and-after stack description looks like this:

d1 addr1 --- d2 addr2

Here, double number d1 is a "dummy" argument. It does nothing more than reserve 32 bits on the stack, giving CONVERT a place into which it can accumulate the converted string digits. Thus, *d1 is nearly always zero*. The second stack argument, addr1, is the address of the byte that precedes the first string character. CONVERT doesn't use this byte, so it need not contain a length count.

Here is how CONVERT works: Starting at addr1+1, CONVERT converts each ASCII character into a numeric value and accumulates this value into d1. This process continues until CONVERT encounters a nonconvertible (nondigit) character. With the conversion process now finished, the stack

holds the final, accumulated double number (d2) and the address of the nonconvertible character (addr2).

Because CONVERT ignores the contents of the count byte, you can use EXPECT (rather than ACCEPT) to input a number, as in this example:

```
PAD 11 EXPECT 1234 OK
```

The string, now in the pad area of memory, can be converted to a number with the sequence

```
0 0 PAD 1 - CONVERT OK
```

This leaves the address PAD+4 on the stack, with the converted double number 1234 below it. If you don't care about the address, follow the CONVERT sequence with the word DROP, and if your converted value is a 16-bit number, rather than a 32-bit double number, follow the convert sequence with DROP DROP (or just 2DROP, if you have it).

In the preceding example, the "nonconvertible character" happens to be a null character—one of the two nulls appended to the string by EXPECT—but *any* nondigit character can serve as a terminator. By leaving the address of the nonconvertible character, CONVERT gives you the capability of investigating this character and making some decision based on your findings. For example, a decimal point terminator may signify the end of the string, whereas some other character (a colon, perhaps) may signify that a fractional value or a floating point exponent follows, signaling the need for further conversion. As you can see, CONVERT opens up a variety of possible applications.

NUMBER, a Higher-Level CONVERT

Unless you need to do the kind of sophisticated decision-making we've just described for CONVERT, you may prefer to use a higher-level form of CONVERT, called NUMBER. An uncontrolled word in FORTH-79 and a standard word in fig-FORTH, NUMBER performs the same kind of numeric conversion as CONVERT, but can also accept a leading minus sign in the string, allowing you to enter negative numbers. Moreover, NUMBER takes just one stack argument, the address of the string's count byte (which it ignores, as CONVERT does), and leaves only the converted double number on the stack.

A typical example using NUMBER is:

```
PAD 11 EXPECT -1234 OK
PAD 1 - NUMBER OK
D. -1234 OK
```

If you have a FORTH-79 compatible software package, it may not have NUMBER. Example 10-9 provides a definition of NUMBER for your convenience.

Example 10-9. Convert text string to a double number

```
: NUMBER
  ( Converts the number starting at addr+1 to a signed )
  ( double number, using the current base. The string  )
  ( may contain a leading minus sign.                  )
  ( addr --- d )
  0 0 ROT              ( For CONVERT, set d1 = 0)
  DUP 1+ C@            ( Get first string character)
  45 =                 ( If it's a - sign, flag = 1)
  DUP >R               ( Save sign flag on return stack)
  +                    ( Calculate address of first digit)
  CONVERT              ( Convert all digits in the string)
  DROP                 ( Discard address)
  R>                   ( Retrieve the sign flag)
  IF DMINUS THEN ;     ( Negate result if it's negative)
```

Table 10-3. The fig-FORTH Words SIGN, (NUMBER), and NUMBER

Word	Stack	Action
Sign	n d --- d	Inserts an ASCII minus sign into the output character string, if the third number on the stack (n) is negative.
(NUMBER)	d1 addr1 --- d2 addr2	Converts the text beginning at addr1+1 to a double number, with regard to BASE. The new value is accumulated into d1, and is left as d2. addr2 is the address of the first non-convertible character.
NUMBER	addr --- d	Converts the text beginning at addr+1 to a double number, with regard to BASE. If a decimal point is encountered in the text, its position will be given in the user variable DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given.

fig-FORTH CHARACTER AND STRING INPUT/OUTPUT WORDS

All FORTH-79 words listed in Table 10-1 except BELL and [U.R] are contained in fig-FORTH. Moreover, there are these differences between FORTH-79 and fig-FORTH:

1. In fig-FORTH, SIGN assumes the 16-bit number containing the sign is immediately beneath a double number on the stack.

Table 10-4. Words Added to FORTH in Chapter 10

Word	Stack	Action
WAIT-CHAR	char ---	Waits until a key pressed at the terminal matches the ASCII value on the top of the stack.
LEN	addr --- count	Returns the character count of the string that starts at addr.
ACCEPT	addr n ---	Transfers characters from the terminal until a "return" or a count of n has been received. The characters are stored as a packed string, with the length byte at addr.
20-QUESTIONS		Provides yes/no answers for up to 20 questions entered from the terminal.
DUMP-N	addr n ---	Displays n consecutive numbers in memory, starting at addr. Numbers are displayed four per line, in decimal. The address of the first number on each line is also displayed, in hexadecimal.
UD.	ud ---	Prints unsigned double number on top of stack.
.PHONE#	ud ---	Prints unsigned double number in telephone number format, xxx-xxxx.
.MONEY	n ---	Prints signed number as a dollar quantity, in the format $xxx.xx or −$xxx.xx. Can print amounts from −$327.68 to $327.67.
.MONEY-D$	d ---	Prints signed double number as a dollar quantity. Can print amounts from −$21,474,836.48 to $21,474,836.47.

2. The FORTH-79 word CONVERT is called (NUMBER) in fig-FORTH.
3. In fig-FORTH a decimal point is permitted to be included in a number string.

Table 10-3 gives the fig-FORTH definitions for SIGN, (NUMBER) and NUMBER.

SUMMARY

This chapter, probably the most complex yet, introduced words that allow you to *interact* with the computer through your terminal. It began with a discussion of two words that transfer individual characters, KEY and EMIT, and three other character-related words, BELL, BL, and SPACE. We then discussed two string-transfer words, EXPECT and TYPE, and the word COUNT, which sets up the address and character count arguments required by TYPE. This section also included a description of the scratch area called PAD, a portion of memory designed to hold text strings for intermediate processing.

This was followed by a related topic: how to format output information. We learned that text can be formatted with the word SPACES, which outputs a specified number of ASCII blanks, and the word −TRAILING, which excludes trailing blanks from an output string. We also learned that numbers could be formatted in two ways: by displaying them right-justified within a selected character field and by exercising character-by-character control over the output, in order to insert characters between digits. The latter application uses a special <# . . . #> construct to "program" the output with combinations of the words [#], [#S], SIGN and HOLD. The remainder of the chapter dealt with converting text to stack numbers, with either CONVERT or NUMBER.

Table 10-4 summarizes words defined in this chapter that you may wish to add to your FORTH system.

CHAPTER 11

String Processing

If your computer system is to store mailing lists, personnel files, correspondence, or any other kind of text-oriented information, you must not only know how to get this information in and out of the computer (described in Chapter 10), but how to manipulate, or *process,* it. Naturally, each text processing program must be customized to the kinds of text being processed, so we can't hope to provide a "cookbook" of solutions in this kind of book. However, *every* text processing program must provide certain fundamental string operations. These include locating strings, adding, deleting and replacing strings and, in many applications, sorting groups of strings (or files). This chapter covers the fundamental string operations in a general way, which provides you with a basic "tool kit" that can be used to develop your own, specific text application program.

Because there are so many different ways of manipulating strings, neither the FORTH-79 Standard nor the fig-FORTH Installation Manual addresses this subject. In fact, the only FORTH-79 Required Word even remotely connected to string processing is CMOVE, which copies a block of bytes from one part of memory to another. In Chapter 4, we saw two other words that may be used for string processing, but neither of these words (<CMOVE and BLANKS) is "required;" they are simply described in the Reference Word Set. The Reference Word Set also describes two other words, -MATCH and -TEXT, that are useful with strings, but both are uncontrolled words. The five words just mentioned are summarized in Table 11-1.

DEFINING THE FUNDAMENTAL STRING WORDS

Since none of the words in Table 11-1 (except CMOVE) are required, many FORTH packages do not have them. Thus, a

Table 11-1. String Words

Word	Stack	Action	Notes
<CMOVE	addr1 addr2 n ---	Copies n bytes starting at addr1 to memory starting at addr2. The move proceeds from high memory to low memory.	(1)
CMOVE	addr1 addr2 n ---	Copies n bytes starting at addr1 to memory starting at addr2. The move proceeds from low memory to high memory.	
-MATCH	addr1 nl addr2 n2 --- addr3 flag	Attempts to find the n2-character string beginning at addr2 somewhere in the n1-character string beginning at addr1. Returns the address of the last matching character +1 (addr3) and a flag which is zero if the match exists or one if no match exists.	(1)
-TEXT	addr1 n1 addr2 --- n2	Compares two strings over the length n1, beginning at addr1 and addr2. The number returned, n2, is zero if the strings are equal, positive if string 1 is greater than string 2 and negative if string 2 is greater than string 1.	(1)
BLANKS	addr n ---	Fills n consecutive bytes in memory with the ASCII value for "blank," starting at addr.	(2)

Notes: (1) Included in Reference Word Set, as an uncontrolled word definition.
(2) Included in Reference Word Set, as a Standard Word Definition.

logical way to begin our discussion of strings is by defining these fundamental words.

<CMOVE

As you recall from Chapter 4, CMOVE and <CMOVE both copy blocks of byte values (strings, if the bytes are ASCII values) from one part of memory to another. However, CMOVE starts copying from the beginning of the block and works toward the end, whereas <CMOVE starts copying from the end of the block and works toward the beginning. These two different approaches allow us to copy strings to lower memory (with CMOVE) or to higher memory (with <CMOVE) without worrying

about whether the destination string overlaps the source string. Example 11-1 shows a colon-definition for <CMOVE.

However, two separate copy words require us to remember which to use in a given situation. To eliminate this problem, it is worthwhile to add a new FORTH word that makes this decision for us. This new word, CMOVE$ (see Example 11-2), checks the direction of the copy operation, then uses either <CMOVE or CMOVE, depending on whether a string is being copied to a higher address or a lower address.

Example 11-1. Copy a byte string, starting at the end

```
: <CMOVE
  ( Copy n bytes starting at addr1 to addr2. The move )
  ( proceeds from high memory to low memory.          )
  ( addr1 addr2 n --- )
  DUP ROT +              ( Calculate addr2+n                  )
  SWAP ROT               (   and put it on bottom of stack)
  1 -                    ( DO-LOOP end = addr1-1)
  DUP ROT +              ( DO-LOOP start = addr1+n-1)
  DO
   1 -                   ( Decrement last string 2 address)
   I C@                  ( Fetch next byte from string 1)
   OVER C!               (   and copy it to string 2          )
   -1                    ( Decrement index)
  +LOOP
  DROP ;                 ( Discard the string 2 address)
```

Example 11-2. Copy a byte string in either direction

```
: CMOVE$
  ( Copy n bytes starting at addr1 to addr2. If addr2 )
  ( is forward of addr1, the move proceeds from high)
  ( memory to low memory; otherwise the move proceeds)
  ( from low memory to high memory.                   )
  ( addr1 addr2 n --- )
  OVER 4 PICK            ( Copy addr2 addr1 on top of stack)
  >                      ( Copying to a higher address?)
  IF <CMOVE              ( If so, use <CMOVE)
  ELSE CMOVE             ( If not, use CMOVE)
  THEN ;
```

The stack requirements for the preceding words (and for all other words in this chapter) include the two standard string arguments: the address of the first character and a character count. At times you will need to type in both of these arguments from the keyboard, but usually you can use the word COUNT to generate them. As you recall from Chapter 10, COUNT takes a count byte address (addr) from the stack and returns the address of the first character (addr+1) and the contents of the count byte (n). Therefore, if a string's count byte is stored at location 200, and you wish to copy the characters in this string to memory starting at location 400, the sequence to use is:

```
400 200 COUNT CMOVE OK
```

in which COUNT replaces the byte count address (200) with the address of the first character (201) and the byte count value stored at location 200.

-MATCH

Before you can delete, replace, or otherwise process a text string, you must know *where* that string is stored in memory. The word -MATCH searches a string whose text starts at addr1 for the first occurrence of a smaller string (a "substring") whose text starts at addr2. -MATCH requires four stack arguments: the address (addr1) of the first character in the string to be searched (the *target string*), the target string's character count, the address (addr2) of the first character in the string being searched for (the *search string*) and the search string's character count. These arguments must be in the following order on the stack:

```
addr1 n1 addr2 n2
```

-MATCH returns a memory address (addr3) and a flag value. If the search string is *found* in the target string, addr3 points to the byte that follows the matching substring, and flag = 0. If the search string is *not found* in the target string, addr3 points to the byte that follows the target string, and flag = 1.

Usually, -MATCH is used to search a string in memory for the presence of a string typed in at the keyboard, so it is often used in conjunction with a string-input word, such as ACCEPT. For example, Fig. 11-1 shows a string of animal names, starting at location 200 (decimal). To search for CAT in this string, you would execute the sequences

```
PAD 3 ACCEPT CAT OK
200 COUNT PAD COUNT -MATCH OK
```

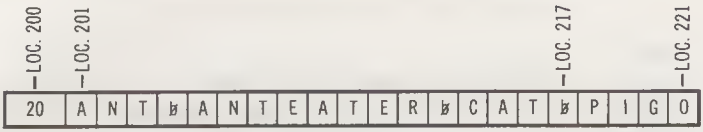


Fig. 11-1. A string of animal names.

The results would be

```
. . 0 217 OK
```

because CAT is in the string (flag = 0), immediately preceding location 217 (remember, addr3 is the last matching address +1).

However, a search for DOG, with this sequence:

```
PAD 3 ACCEPT DOG OK
200 COUNT PAD COUNT -MATCH OK
```

produces the results

```
. . 1 221 OK
```

because DOG is not in the string (flag = 1) and the string ends at location 220 (here, addr3 is the address of the last character +1).

Note that -MATCH never alters the target string, but just produces an address (addr3) that can be used as the basis for a subsequent processing operation. Here are some typical uses for -MATCH:

1. To *insert* a substring following the search string occurrence, execute -MATCH to find the insertion point, then insert the new substring at addr3.
2. To *delete* the search string from the target string, execute -MATCH, then delete n2 characters, starting at location addr3-n2.
3. Similarly, to *replace* the search string occurrence with a new substring, execute -MATCH, then write n2 new characters to memory, starting at location addr3-n2.

The insert, delete, and replace operations will be described in detail later in this chapter.

Example 11-3 shows a definition for -MATCH. This definition consists of two DO-LOOPs. The outer DO-LOOP searches the

Example 11-3. Search for a substring

```
: -MATCH
  ( Attempt to find the n2-character string beginning at )
  ( addr2 somewhere in the n1-character string beginning)
  ( at addr1. Return the last matching address + 1 as )
  ( addr3 and a flag which is zero if a match exists or )
  ( one if no match exists.                              )
  ( addr1 n1 addr2 n2 --- addr3 flag )
  SWAP DUP C@           ( These words change stack to:      )
  5 PICK 5 ROLL +       ( addr1 n2 addr2 byte-1 addr1+n1    )
  DUP 1                 ( To start, assume no match exists)
  SWAP 6 PICK - 1+      ( Final stack is: --- n2 addr2 byte-1)
  7 ROLL                ( addr1+n1 end start                )
  DO
   3 PICK I C@ =        ( Search for byte-1 in target string)
   IF                   ( If found, compare rest of target)
    0                   ( Assume match, set flag = 0)
    6 PICK 1            ( Inner loop limits = n2 1)
    DO
     J I + C@           ( Fetch next target byte)
     6 PICK I + C@      ( and next search byte )
     = NOT              ( Do they match?)
     IF
      DROP 1 LEAVE      ( No. Resume search)
     THEN
    LOOP                ( Yes. Continue comparing)
    IF ELSE             ( If entire string is found,    )
     DROP DROP          ( replace "no match" results)
     I 4 PICK + 0       ( with addr3 and 0,             )
     LEAVE              ( then quit                     )
    THEN
   THEN
  LOOP
  ROT DROP              ( Discard byte-1, addr2 and n2)
  ROT DROP
  ROT DROP ;
```

target string for the first character in the search string. If this character is found, the inner DO-LOOP takes over, comparing the remainder of the search string (n2-1 characters in all) with the next n2-1 characters of the target string. If the inner loop encounters a mismatch, the word LEAVE returns control to the

outer DO-LOOP. Incidentally, the outer DO-LOOP does not need to check every character in the target string, but only up to the last n2 characters, since that's the last possible place the search string will fit. Therefore, the limits of the outer DO-LOOP are:

```
Start of search = addr1
End of search = addr1 + n1 − n2 + 1
```

Of course, the definition of −MATCH must also include provisions to return the proper stack results, addr3 and flag. Initially, a nonmatch is assumed, so the values addr1+n1 (addr3 for a nonmatch) and 1 (flag for a nonmatch) are pushed onto the stack. If a successful match ever occurs, these values are replaced with addr3 = match+1 and flag = 0.

-TEXT

The word -TEXT performs the basic task needed by all sorting programs: it compares the *magnitudes* of two strings, to determine whether the strings are identical or, if not, which has the greater value. Example 11-4 shows the definition of -TEXT. In this definition, two strings of length n1 are compared, byte by byte, by subtracting a byte in string 2 from its counterpart in string 1. The comparison process continues until a subtraction produces a nonzero result or n1 bytes have been compared.

BLANKS

The final word in Table 11-1, BLANKS, is used to erase text that has been deleted or moved to some other part of memory. The definition of BLANKS is:

```
: BLANKS 32 FILL ;
```

where 32 is the ASCII space character.

ADD A NEW SUBSTRING

There are two ways to add a new substring to a string: you can either attach the substring to the end of the string (that is, *append* it to the string) or insert it somewhere within the string. If the string holds information that is unordered, you can just append the substring, but if the string holds personnel records, word processing text or some other kind of ordered informa-

Example 11-4. Compare two strings

```
: -TEXT
  ( Compare two strings over the length n1, beginning at )
  ( addr1 and addr2. The result, n2, is zero if the       )
  ( strings are equal, positive if string 1 is greater    )
  ( than string 2, and negative if string 2 is greater    )
  ( than string 1.                                        )
  ( addr1 n1 addr2 --- n2)
  0                          ( Put dummy number on stack)
  ROT 0                      ( Check n1 bytes)
  DO
   DROP                      ( Discard last result or dummy no.)
   OVER I + C@               ( Fetch next byte of string 1)
   OVER I + C@               (  and next byte of string 2,)
   -                         (  then subtract them       )
   DUP 0= NOT                ( Are these bytes equal?)
   IF LEAVE THEN             ( No. Exit)
  LOOP                       ( Yes. Compare next two bytes)
  SWAP DROP SWAP DROP ;( Discard addresses)
```

tion, you will need to insert the substring in its proper position within the string.

Naturally, appending a substring to the end of a string is much simpler than inserting it within the string. To append a substring, you simply copy the substring to the end of the string, then update the count byte to reflect the increased length. Example 11-5 shows a word called APPEND$ that does precisely those tasks. It copies an n2-character substring to the end of an n1-character string, then stores the sum of n1 and n2 into the string's count byte. The substring is unaffected. Incidentally, instead of copying just the n2 characters of the substring, APPEND$ copies *n2+1* characters, so that the string still terminates with a null character.

Inserting a substring into a string requires one additional stack argument (the address of the insertion point) and one additional operation (opening up a gap in the string, to accommodate the new substring). The word INSERT$, defined in Example 11-6, can be used to make the insertion. INSERT$ performs three operations:

1. It opens up an n2-byte gap in the string, by moving all bytes between addr3 and the end of the string +1 higher in memory by n2 locations.

Example 11-5. Add a substring to the end of a string

```
: APPEND$
  ( Add an n2-character string beginning at addr2 ta the )
  ( end af an n1-character string beginning at addr1.     )
  ( addr1 n1 addr2 n2 --- )
  SWAP 3 PICK 5 PICK +   ( Append string 1)
  3 PICK 1+ CMOVE$       (  ta string 2    )
  + SWAP 1 - C! ;        ( Update string 1's byte caunt)
```

Example 11-6. Insert a substring into a string

```
: INSERT$
  ( Insert an n2-character substring beginning at addr2  )
  ( inta an n1-character string beginning at addr1. The  )
  ( insertian begins at addr3.                           )
  ( addr1 n1 addr2 n2 addr3 --- )
  DUP 6 PICK 6 PICK +    ( Calculate number af bytes )
  1+ OVER -              ( from addr3 ta end af string)
  OVER 5 PICK + SWAP     ( Set up <CMOVE arguments)
  <CMOVE                 (  and apen up the gap  )
  OVER 5 ROLL +          ( Update string 1's byte count)
  5 ROLL 1 - C!
  SWAP <CMOVE ;          ( Make the insertian)
```

2. It changes the string's count byte to the value n1+n2, to reflect the increased length.
3. It copies the substring into the newly created gap in the string.

In actual use, the insertion address, addr3, is usually produced by a preceding -MATCH operation. For example, to insert DOG into our previous list of animal names, following CAT, we could execute these sequences:

```
PAD 3 ACCEPT CAT OK                 ( Find insert paint)
200 COUNT PAD COUNT -MATCH OK
DROP OK                             ( Discard flag)
PAD 4 ACCEPT ϕDOG OK                ( Enter DOG substring)
200 COUNT PAD COUNT 5 ROLL OK ( Set up arguments)
INSERT$ OK                          ( Make the insertian)
```

DELETE A SUBSTRING

Deleting a substring is similar to inserting a substring, except that instead of opening up space in the string, you are shortening it. Example 11-7 shows the definition of a substring delete word called DELETE$. When executed, DELETE$ searches the string for the substring, using -MATCH. If the substring cannot be found in the string, DELETE$ clears the stack and exits. If the substring is found, however, DELETE$ deletes it by moving each remaining byte in the string (and the terminating null character) to a location n2 bytes lower in memory, thereby over-writing the substring. Of course, this also requires the count byte to be changed to reflect the deletion.

For example, to delete CAT from the string of animal names, you can enter the sequences

```
PAD 4 ACCEPT ♭CAT OK
200 COUNT PAD COUNT DELETE$ OK
```

If you then DUMP from location 200, you will see that CAT is no longer in the string and the character count has dropped from 20 to 14.

SORTING TEXT FILES

The string-processing words we've just covered are the foundations of word processing and text editing programs—programs that are useful for preparing correspondence, books, and other documents. Other applications, such as mailing lists, telephone lists, and the like, are also text oriented, but usually involve some processing in addition to manipulation. For these applications, text is grouped in *files,* rather than just strings. Within each file, an individual entry (a name and address, a name and a telephone number, or whatever) is called a *record.* If these records have been entered into the file in random order, as they often are, you may like to sort them alphabetically, to make them easier to process.

We have already been introduced to sorting in Chapter 8, where we encountered two common sorting techniques, called *bubble sort* and *insertion sort,* and implemented each as a FORTH word (BSORT and ISORT, respectively) that could be used to sort numbers in an array. These techniques can also be used to sort records in a file, by making the modifications needed to handle multibyte records rather than two-byte numbers. However, the more generalized nature of file-sorting

routines make them somewhat longer and more complex than array-sorting routines.

For instance, because records may be of various lengths, a file-sorting routine must accept three arguments from the stack (a starting address, a record count, and a bytes per record count) instead of just two arguments (starting address and number count). The third argument, bytes per record, will be used as an offset between records, replacing the constant two-byte offset in array-sorting routines.

Moreover, because records may be many bytes long, if we need to exchange two records (as in bubble sorting) or move a record in memory (as in insertion sorting), we will need to temporarily store a record some place other than on the stack. The pad area is ideal for this purpose.

Finally, comparing two records is also somewhat more complex than comparing two numbers, because numbers can be compared directly with the words [>], [<], and so on, whereas records must be compared with the word -TEXT, which requires you to set up three stack arguments.

Example 11-7. Delete a substring from a string

```
: DELETE$
  ( Delete an n2-character substring beginning at addr2 )
  ( from an n1-character string beginning at addr1.      )
  ( addr1 n1 addr2 n2 --- )
  4 PICK 4 PICK            ( Search for the substring)
  4 ROLL 4 PICK -MATCH
  IF                       ( If substring is not found,)
   DROP DROP DROP          (   clear the stack and exit )
   DROP
  ELSE                     ( If substring is found, )
   DUP 3 PICK -            (   delete it by moving all)
   5 PICK 5 PICK +         (   remaining bytes up by)
   3 PICK - 1+ CMOVE       (   n2 bytes                 )
   - SWAP 1 - C!           ( Update string 1's byte count)
  THEN ;
```

Despite these differences, sorting text files is not too different than sorting number arrays, it just takes more careful programming. Examples 11-8 and 11-9 provide definitions for the file-sorting words BSORT$ and ISORT$. As you can see, these are very similar in structure to their array-sorting counterparts,

BSORT (Example 8-10) and ISORT (Example 8-11). In general, the insertion sort (ISORT$) is usually faster for sorting a file that is randomly arranged, but the bubble sort (BSORT$) can be faster if the file is already nearly sorted.

Example 11-8. Sort records in a file, using bubble sort

```
: BSORT$
  ( Using the bubble sort technique, sort a file beginning )
  ( at addr. No. of records in the file is given by recs;  )
  ( addr recs bytes/rec --- )
  DUP ROT * 3 PICK +            ( DO-LOOP end is one rec. past file)
  3 ROLL                        ( DO-LOOP start is addr)
  >R >R                         ( Save these limits on return stack)
  BEGIN
   1                            ( To start, exchange flag = 1)
   R> 3 PICK - R@ OVER >R       ( Subtract bytes/rec from end limit)
   DO
    OVER DUP I + SWAP I         ( Compare next two records)
    -TEXT 0<
    IF                          ( If 2nd record < 1st record,)
     I PAD 4 PICK CMOVE         (  exchange the two records  )
     I 3 PICK + I 4 PICK CMOVE
     PAD 3 PICK DUP I + SWAP CMOVE
     DROP 0                     ( Set exchange flag = 0)
    THEN
    OVER                        ( Loop increment = bytes/rec)
   +LOOP
  UNTIL                         ( Keep sorting until flag = 1)
  R> R> DROP DROP DROP ;        ( Clear both stacks)
```

File sorting can be much more sophisticated than the level we have addressed here. The words BSORT$ and ISORT$ sort files based on the leading characters in each record. However, in practice, records often hold several different types of information, so the record itself may be subdivided into various *fields*. For example, each record in a mailing list file may be comprised of five different fields: name, street address, city, state, and zip code. To arrange this list geographically, you may wish to sort it by zip code, then sort addresses with the same zip code by city, then perhaps street. That is, your sort would need to extend to a "depth" of three fields, rather than just one. Such considerations are beyond the scope of this book,

Example 11-9. Sort records in a file, using insertion sort

```
: ISORT$
  ( Using the insertion sort technique, sort a file beginning )
  ( at addr. No. of records in the file is given by recs;  )
  ( no. of bytes per record is given by bytes/rec.         )
  ( addr recs bytes/rec ---- )
  DUP ROT * 3 PICK +      ( DO-LOOP end is one rec. past file)
  3 PICK 3 PICK +         ( DO-LOOP start is addr)
  DO
   I OVER I OVER -        ( Is this record less than)
   -TEXT 0<               (  the preceding record?  )
   IF                     ( Yes. Find its proper place)
    0                     ( Dummy number)
    3 PICK 3 PICK -       ( Inner loop limits =          )
    I 4 PICK -            (  addr-bytes/rec I-bytes/rec)
    DO
     DROP                 ( Drop dummy number or last I)
     J OVER I -TEXT 0<    ( Search record < this record?)
     IF
      I                   ( If so, save this address)
     ELSE
      I OVER + LEAVE      ( If not, go insert)
     THEN
     OVER NEGATE
    +LOOP
    I PAD 4 PICK CMOVE    ( Save insert string in the pad)
    DUP DUP 4 PICK +      ( Open up a gap for the insert)
    I 3 PICK - <CMOVE
    PAD SWAP 3 PICK CMOVE (   and insert the string       )
   THEN
   DUP
  +LOOP
  DROP DROP;              ( Clear the stack)
```

but you should be able to apply the principles given here to develop a multilevel sort, if needed.

fig-FORTH STRING WORDS

Only two of the words listed in Table 11-1, CMOVE and BLANKS, are provided by fig-FORTH. However, since the re-

maining words are defined within this chapter, you can easily add them to your system, if you wish.

Table 11-2. Words Added to FORTH in Chapter 11

Word	Stack	Action
CMOVE$	addr1 addr2 n ---	Copies n bytes starting at addr1 to addr2. If addr2 is forward of addr1, the move proceeds from high memory to low memory; otherwise the move proceeds from low memory to high memory.
APPEND$	addr1 n1 addr2 n2 ---	Adds an n2-character substring beginning at addr2 to the end of an n1-character string beginning at addr1.
INSERT$	addr1 n1 addr2 n2 addr3 ---	Inserts an n2-character substring beginning at addr2 into an n1-character string beginning at addr1. The insertion begins at addr3.
DELETE$	addr1 n1 addr2 n2 ---	Deletes an n2-character substring beginning at addr2 from an n1-character string beginning at addr1.
BSORT$	addr recs bytes/rec ---	Uses the bubble sort technique to sort a file beginning at addr. No. of records in the file is given by recs; no. of bytes per record is given by bytes/rec.
ISORT$	addr recs bytes/rec ---	Similar to BSORT$, but uses the insertion sort technique.

SUMMARY

Because there are so many different ways to process text information, both FORTH-79 and fig-FORTH leave this task to the user. For this reason, we used some words suggested in the FORTH-79 Standard to develop new words that perform three fundamental operations: add a substring to the end of a string, delete a substring from a string, and insert a substring into a string. These words can be used as the basis for a string-processing package of your own design. Finally, to aid applications that involve processing files of text records, such as mailing lists, we defined two file sorting words. One of these words was based on the bubble sort technique, the other was based on the insertion sort technique.

Table 11-2 summarizes the words that can be added to FORTH based on the material in this chapter.

CHAPTER 12

More Disk Operations

As you learned in Chapter 5, data is transferred between disk and memory in 1024-byte units called *blocks*. A block may consist of text for a colon-definition (in which case it is referred to as a "screen") or it may consist of numbers in a data base, names and addresses in a mailing list, or various other kinds of nonprograms. If you have been using this book in conjunction with a FORTH system, you should be familiar with the procedures for using the disk to store colon-definitions. However, the disk procedures for storing files of nonprogram information are somewhat different, so they are worth treating individually here. Table 12-1 summarizes the FORTH words that will be introduced in this chapter.

CREATING NEW BLOCKS

Just as you needed to allocate a block buffer in memory to hold each text screen for definitions, you must allocate a block buffer to hold the text or numbers in a file. The FORTH word that does this is BUFFER. BUFFER accepts a block number from the stack and allocates a block buffer (1024 bytes) for that block. If the previous contents of the buffer have been marked as UPDATEd, those contents are first written to disk. To let you know where the block buffer is located in memory, BUFFER leaves the address of its first data storage byte on the stack.

For example, the sequence

```
64 BUFFER U. 42022 OK
```

allocates a block buffer for Block 64 in memory, and shows its starting address as decimal 42022 (hex A426). Since blocks

Table 12-1. The Disk Words BUFFER and BLOCK

Word	Stack	Action
BUFFER	n --- addr	Obtains the next memory buffer and assigns it to block n. If the previous contents of the buffer is marked as UPDATEd, it is written to disk. The block is not read from disk. The address left is the first byte available for data storage within the buffer.
BLOCK	n --- addr	Leaves the address of the first data storage byte in block n. If the block is not already in memory, it is transferred from disk to whichever buffer was least recently accessed. If the block occupying that buffer has been marked as UPDATEd, it is rewritten to disk before block n is read into the buffer.

provide 1024 bytes of storage, you know that the ending address is decimal 43046.

INITIALIZING A BLOCK

At the same time you allocate a block buffer, you will also want to enter some initial data into the buffer. The buffer provides 1024 storage bytes, but the first two bytes should be used to hold a count of the bytes used. Like a string's preceding count byte, a buffer's preceding count bytes must be updated any time you add data to, or delete data from, the buffer. Let's look at the procedures for storing numbers and text into a block buffer and, eventually, onto disk.

If a block is to hold numbers, you need a FORTH word that takes a number count and a starting address from the stack, and uses those arguments to accept numbers from the keyboard and store them into the block buffer. The word !NUMBERS, defined in Example 12-1, can be used to perform this task. For example, the sequence

```
7 64 BUFFER 2 + !NUMBERS 1 2 3 4 5 6 7 OK
```

allocates a block buffer for Block 64 and initializes it with seven numbers, 1 through 7. (Remember, since the numbers are entered with ACCEPT, you must terminate each entry with a carriage return.)

!NUMBERS leaves a byte count on the stack, so this value must be entered into the buffer's count bytes. Can you use the word BUFFER to get the address of the count bytes? No, you cannot, because *BUFFER always allocates a new block buffer,* and you want to store the count in the existing buffer. To do this you will need to use another FORTH word called BLOCK. BLOCK is similar to BUFFER except that BLOCK works with blocks that have already been allocated, and are either in memory or on disk.

Example 12-1. Read numbers into memory from the keyboard

```
: !NUMBERS
  ( Accept n1 numbers from the keyboard and store them into )
  ( memory, starting at addr. Leave byte count n2.          )
  ( n1 addr --- n2 )
  SWAP 2 *              ( Calculate byte count)
  DUP 3 PICK + ROT      ( DO-LOOP limits = addr+n2 addr)
  DO
   PAD 6 ACCEPT         ( Read in a string from the keyboard,)
   PAD NUMBER           (  and convert it to a double number )
   DROP                 ( Convert double number to number)
   I !                  (  and store it in memory          )
   2
  +LOOP ;
```

Now you know how to initialize and save a block of numbers. If a block is to hold a text file, you need a text counterpart of !NUMBERS that takes a record count and a starting address from the stack, then accepts the specified number of records from the keyboard and stores them in the buffer. This particular word must, of course, be designed to accept the kinds of records you wish to store.

Example 12-2 defines a typical text-initializing word, !PHONE-LIST, which can be used to build a telephone list in a block buffer. In this case, each record is 34 bytes long and consists of three fields: a name (up to 22 characters), a three-digit area code, and a phone number in the format xxx-xxxx. The extra byte is a blank between the area code and the phone number. For instance, the sequence

```
3 64 BUFFER 2 + !PHONE-LIST
```

accepts three records from the keyboard and stores them into the buffer allocated to Block 64. With the prompts, the initialization session should look like this:

NAME: PARKER, CHARLIE
AREA CODE: 705
NUMBER: 223-6666

NAME: CHAMBERLAIN, IRV
AREA CODE: 307
NUMBER: 555-2121

NAME: KIRK, CAPTAIN
AREA CODE: 999
NUMBER: 123-4567 OK

Example 12-2. Read phone list into memory from the keyboard

```
: !PHONE-LIST
  ( Accept n1 phone number entries from the keyboard and  )
  ( store them into memory, starting at addr. Each entry  )
  ( consists of a name, up to 22 characters, plus a       )
  ( three-digit area code and an eight-digit number in    )
  ( the format xxx-xxxx. Leave byte count n2.              )
  ( n1 addr --- n2 )
  SWAP 34 *              ( n2 = n1 * 34)
  DUP 3 PICK + ROT       ( DO-LOOP limits = addr+n2 addr)
  DO
   CR CR ." NAME: "
   PAD 22 ACCEPT         ( Read name into pad,       )
   PAD DUP C@ + 1+       (  and add trailing blanks)
   22 PAD C@ - BLANKS
   PAD 1+ I 22 CMOVE     ( Move name chars. to the buffer)
   CR ." AREA CODE: "
   I 22 + 3 EXPECT       ( Read area code into buffer )
   BL I 25 + C!          (  and follow it with a blank)
   CR ." NUMBER: "
   I 26 + 8 EXPECT       ( Read phone number into buffer)
   34
  +LOOP ;
```

!PHONE-LIST, like !NUMBERS, leaves a byte count on the stack. As before, you can record this count in the buffer with

```
64 BLOCK ! OK
```

then save the buffer on disk with either (UPDATE SAVE-BUFFERS) in FORTH-79 or (UPDATE FLUSH) in fig-FORTH.

ACCESSING THE CONTENTS OF A BLOCK

Once a block has been saved on disk, you can bring it back into memory with the word BLOCK. For example,

```
64 BLOCK
```

reads the contents of Block 64 into memory, if it is not already there, and leaves the address of its first storage byte (the first byte of the count) on the stack.

By adding an offset to this starting address, you can access any specific data value in the block. For instance, if a block holds 8-bit *bytes,* you simply add the byte number to the starting address plus two. Therefore, to place the contents of byte 100 of Block 64 onto the stack, execute the sequence

```
64 BLOCK 2 + 100 + C@ OK
```

(This assumes the first data byte in the block is byte 0, so byte 100 is actually the 101st data byte.) Similarly, to change the contents of byte 100 to the value 4, execute the sequence

```
4 64 BLOCK 2 + 100 + C! OK
```

If a block holds 16-bit *numbers* instead of 8-bit bytes, you must double the offset before adding it to the starting address plus two. For example, to place the contents of number 100 of Block 64 on the stack, execute the sequence

```
64 BLOCK 2 + 100 2 * + @ OK
```

and to change the value of that number to 4, execute

```
4 64 BLOCK 2 + 100 2 * + ! OK
```

Of course, you must always "tell" FORTH when a block has been changed, by following the last change with UPDATE.

If a block holds *text,* you can use -MATCH to locate a particular record. For example, to find Captain Kirk's phone number in

the telephone list application in the last section, you might use the sequence

```
64 BLOCK DUP @ OK          ( Address & byte count of block)
PAD 4 ACCEPT KIRK OK       ( Enter search name)
PAD COUNT -MATCH OK        ( Leave flag & end addr + 1)
DROP 4 - OK                ( Leave starting address)
```

Although this procedure gets the job done, it is both cumbersome and error-prone. It is inefficient as well, because -MATCH searches the entire phone file, rather than just the name fields in the file.

Example 12-3 shows a much better search algorithm, and names that algorithm GET-PHONE#. This word searches just the name fields in the file. If the specified name is found, GET-PHONE# leaves its starting address on the stack. An unsuccessful search produces the message "NAME NOT FOUND".

Example 12-3. Search phone list for a name

```
: GET-PHONE#
  ( Search a telephone list for a specified name, up to 22 )
  ( characters, starting at addr1. If the name is found,    )
  ( leave its starting address on the stack. If the name    )
  ( is not found, print a message.                          )
  ( If found: addr1 --- addr2 )
  ( If not found: addr1 ---    )
  PAD 22 ACCEPT PAD COUNT  ( Input search name from keyboard)
  0                        ( Dummy number)
  4 ROLL DUP @ SWAP 2 +( DO-LOOP limits =          )
  DUP ROT + SWAP           (  addr1+2+bytes addr1+2)
  DO
   3 PICK 3 PICK I -TEXT ( Compare next name)
   0=
   IF I LEAVE THEN          ( Exit if match occurs)
   34
  +LOOP
  ?DUP 0=
  IF
   CR ." NAME NOT FOUND "
  ELSE
   SWAP DROP SWAP ROT
  THEN
  DROP DROP ;
```

Once the starting address of a record is on the stack, you can print the entire 34-byte record with the word TYPE. For example:

```
64 BLOCK GET-PHONE# KIRK OK
CR 34 TYPE
KIRK, CAPTAIN          999 123-4567 OK
```

ADDING DATA TO A BLOCK

The data entry words defined in this chapter, !NUMBERS and !PHONE-LIST, can also be used to add new data to an existing block. For example, to add three new values to your block of numbers, you would execute the sequences

```
3 64 BLOCK DUP @ + 2 + !NUMBERS 10 11 12 OK
64 BLOCK +! OK
UPDATE SAVE-BUFFERS OK
```

DUPLICATING A BLOCK

Each block buffer in memory is preceded by two bytes. The first byte holds the block number to which that buffer is allocated. The second byte holds an UPDATE indicator; this indicator is equal to 1 if the block has been marked as UPDATEd and is otherwise equal to 0. By changing the contents of the first byte, the block indentifier, you can *duplicate* a block, thereby creating a backup copy for future reference or an initialized starting point for a new data set.

The procedure for duplicating a block is extremely easy. You simply alter the block identifier, mark the buffer as UPDATEd, then write the buffer to disk. For example, if Block 65 is to hold a duplicate of Block 64, execute the sequences

```
65 64 BLOCK 2 - C! OK
UPDATE SAVE-BUFFERS OK
```

Incidentally, this technique is also convenient for duplicating a *screen,* in case you want to define a word that is similar to a previously defined word. You may wish to do this for debugging purposes, since it allows you to investigate some alternate approach to a definition that is not working quite right, yet save the original definition.

SUMMARY

This chapter described how to create and manipulate data and text files, and how to save these files as "blocks" on disks. A related topic, how to duplicate a block, was also discussed.

CHAPTER 13

Logical, Shift, and Rotate Operations

In the preceding three chapters we concentrated on ways to process ASCII characters, and strings of ASCII characters, because virtually all printers, keyboards, and display terminals are ASCII-based devices. Such standard devices usually come with all the hardware and software necessary to communicate with your computer, so your job becomes one of supplying the correct information and executing a simple "transmit" command, such as KEY, EMIT, EXPECT, or TYPE. Thus, the internal operations are made "transparent" to you. However, if your system includes one or more nonstandard devices (perhaps devices of your own design), or if you wish to alter the internal operation of a standard device, you must develop an appropriate set of programming "tools" that allow you to do so.

To control a device at this more detailed level, you will need to send it specific patterns of binary digits (or *bits*), patterns the device is designed to recognize as commands. Similarly, you will need to accept bit patterns from the device and extract selected bits as status indicators. In both cases you are required to operate on values at the *bit* level, as well as at the number level.

This chapter describes two classes of FORTH words that can manipulate bits within a number. The first class of words perform "logical" operations on numbers. That is, they apply a *mask* to a number, which changes the state of specific bits to produce a new number. The second class of words shift or rotate the bit patterns that represent a number, thereby displacing the entire pattern so many positions to the right or left. Table 13-1 summarizes the words we will describe in this chapter.

Table 13-1. Logical, Shift and Rotate Words

Word	Stack	Action	Notes
AND	n1 n2 --- n3	Leaves the logical AND of n1 and n2.	
OR	n1 n2 --- n3	Leaves the logical inclusive-OR of n1 and n2.	
XOR	n1 n2 --- n3	Leaves the logical exclusive-OR or n1 and n2.	
COM	n1 --- n2	Leaves the one's complement of n1.	(1)
ASHIFT	n1 n2 --- n3	Shifts the value n1 arithmetically by the number of bit positions specified in n2. If n2 is positive, n1 is shifted left and zeroes are shifted into the least-significant bit positions. If n2 is negative, n1 is shifted right and the most-significant bits are replicated.	(1)
SHIFT	un1 n --- un2	Shifts the value un1 logically by the number of bit positions specified in n. If n is positive, un1 is shifted left. If n is negative, un1 is shifted right. Zeroes are shifted into vacated bit positions.	(1)
ROTATE	un1 n --- un2	Rotates the value un1 by the number of bit positions specified in n. If n is positive, un1 is rotated left. If n is negative, un1 is rotated right.	(1)

Note: (1) Included in Reference Word Set, as an uncontrolled word definition.

LOGICAL WORDS

Logical words are so named because they operate according to the rules of formal logic, as opposed to the rules of mathematics. For example, the rule of logic stated "if A is true and B is true, then C is true" has a FORTH counterpart in the word AND, which applies this rule to the 16 pairs of corresponding bits in two numbers. Specifically, for each bit position in which both numbers have a logic 1 (true), the bit position in the result number is set to logic 1. Conversely, for each bit position in which the two numbers have any other combination—both have logic 0, or one has logic 0 and the other logic 1—the bit position in the result number is set to logic 0.

AND is one of three logical words that are required by both the FORTH-79 Standard and fig-FORTH; the other two are OR and XOR. All three words logically combine the 16 bits of two numbers (a source number and a mask) on the stack, and leave the resulting number on the stack.

Since logical operations reference specific bits within a number, you will usually use hexadecimal numbering for these operations (although binary, or base 2, numbering could also be used). Being 16 bits long, a number is represented by 4 hexadecimal digits, the values 0 through FFFF. To help you construct the correct mask value for a logical operation, Table 13-2 shows the hexadecimal representation of a "1" in each of the 16 different bit positions. For example, to select bit position 2, the correct mask value is hex 4; to select bit positions 2 and 3, the correct mask value is hex C (hex 4 + hex 8); and so on.

The Word AND

The word AND is primarily used to mask out (set to zero) certain bits in a number so that some form of processing can be done on the remaining bits. As just mentioned, for each bit position in which both operands contain a logic 1, the corresponding bit position in the result is also set to logic 1; all other operand combinations cause the result bit to be reset to logic 0. Table 13-3 summarizes the AND combinations. Note that *any bit ANDed with 0 will be cleared to zero and any bit ANDed with 1 will retain its original value.*

As an illustration of AND, suppose memory location A000 holds a 16-bit status value from an external device, and bit 6 of this indicates whether device power is on (logic 1) or off (logic 0). If your FORTH program requires device power to be on before continuing, it might include the following sequence:

```
BEGIN
   A000 @          ( Read status value)
   40 AND          ( Isolate the power indicator, bit 6)
   0= NOT          ( Is power on?)
UNTIL              ( If not, keep checking)
. .                ( If so, continue here)
. .
```

The Word OR

The word OR produces a logic 1 result for any bit position in which either the source word or mask (or both) contains a logic 1. Table 13-4 summarizes the OR combinations.

Table 13-2. Hexadecimal Values for Bit Positions

Bit Number	Hex Value	Bit Number	Hex Value
0 (LSB)	0001	8	0100
1	0002	9	0200
2	0004	10	0400
3	0008	11	0800
4	0010	12	1000
5	0020	13	2000
6	0040	14	4000
7	0080	15 (MSB)	8000

Table 13-3. The AND Operation

Operand Bits		Result Bit
A	B	
0	0	0
0	1	0
1	0	0
1	1	1

Table 13-4. The OR Operation

Operand Bits		Result Bit
A	B	
0	0	0
0	1	1
1	0	1
1	1	1

OR is usually used to set specific bits to logic 1. For example, the sequence

```
A200 @ C000 OR A200 !
```

sets the two most-significant bits of location A200 (bits 14 and 15) to logic 1 and leaves all other bits unchanged.

The Word XOR

The word XOR is primarily used to determine which bits differ between two operands, but it can also be used to reverse the state of selected bits. XOR produces a logic 1 result for any

bit position in which the operands differ (one operand contains logic 0, the other contains logic 1). If both operands contain either logic 0 or logic 1, the result bit is cleared to logic 0. Table 13-5 summarizes the XOR combinations.

Table 13-5. The XOR Operation

Operand Bits		
A	**B**	**Result Bit**
0	0	0
0	1	1
1	0	1
1	1	0

The abbreviation XOR stands for *exclusive-OR,* because it functions like OR, but excludes the combination in which both operands hold a logic 1. The way XOR operates on a particular bit can be likened to a radio operator waiting for two messages. If neither message arrives, it is a zero night. If either message arrives, the night is a success. However, if *both* messages arrive at the same time (canceling each other), it also results in a zero night.

As an example, the sequence

```
A200 @ C000 XOR A200 !
```

will reverse the state of the two most-significant bits of location A200 and leave all other bits unchanged.

The Word COM

The FORTH-79 Reference Word Set includes an uncontrolled word that reverses the state of *every* bit in a number. This word, COM (for complement) changes every logic 1 bit to logic 0, and vice versa. Since COM acts like XOR with an all-ones mask, its definition is nothing more than

```
: COM FFFF XOR ;
```

SHIFT AND ROTATE WORDS

Although rather rare, there are some applications that require the capability of displacing an entire bit pattern by one or more positions to the right or left. Serial communications, in which

data is transmitted and received one bit at a time, is a prime example. Because such requirements are relatively uncommon, neither the FORTH-79 nor the fig-FORTH Installation Manual includes standard words for this purpose. However, the FORTH-79 Standard suggests three uncontrolled words that can be used to displace a 16-bit stack operand. Two of these words, ASHIFT and SHIFT, "shift" the operand; the other word, ROTATE, "rotates" the operand.

ASHIFT and SHIFT

In a *shift* operation, bit values that are displaced out of one end of the operand are permanently lost. For a right shift, one or more low-order bits of the operand will be lost. For a left shift, one or more high-order bits will be lost.

There are two different shift words because shifting unsigned numbers is somewhat different than shifting signed numbers. If the operand is *unsigned,* bits vacated during the shift operation are filled with zeroes, as shown in Fig. 13-1A. The same holds

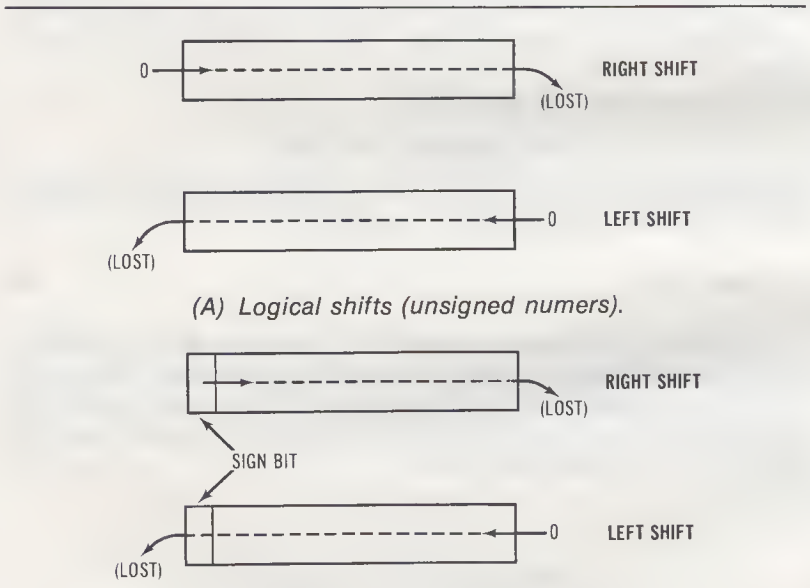


Fig. 13-1. Logical and arithmetic shift operations.

Example 13-1. Shift a number right or left

```
: ASHIFT
  ( Shift n1 arithmetically n2 bit positions. If n2 is )
  ( positive, shift n1 to the left; vacated bit        )
  ( positions receive zeroes. If n2 is negative, shift)
  ( n1 to the right, extending the sign in the most-)
  ( significant bit position.                          )
  ( n1 n2 --- n3 )
  2 OVER ABS **          ( Shift factor = 2**ABS(n2))
  SWAP 0<                ( Determine shift direction)
  IF                     ( For right shift,          )
   /                     (  divide by shift factor)
  ELSE                   ( For left shift,           )
   *                     (  multiply by shift factor)
  THEN ;
: SHIFT
  ( Logical shift un1 by n bit positions; vacated bit  )
  ( positions receive zeroes. If n is positive, shift  )
  ( n1 to the left; if n is negative, shift un1 to the )
  ( right.                                             )
  ( un1 n --- un2 )
  2 OVER ABS **          ( Shift factor = 2**ABS(n))
  SWAP 0<                ( Determine shift direction)
  IF                     ( For right shift,                )
   0 SWAP                (  make un1 a double number    )
   U/MOD                 (  and divide by shift factor )
   DROP SWAP DROP        ( Leave unsigned single quotient)
  ELSE                   ( For left shift,           )
   U*                    (  multiply by shift factor)
   DROP                  ( Leave unsigned single product)
   THEN ;
```

true when a *signed* number is left-shifted. However, when a signed number is right-shifted, vacated bit positions receive the sign bit value, rather than zeroes, and the sign itself is preserved. This is shown in Fig. 13-1B.

The words that perform these shift operations, ASHIFT for signed numbers and SHIFT for unsigned numbers, can be defined based on this simple principle: *each left shift multiplies the operand by two and each right shift divides the operand by two*. Therefore, left-shifting involves multiplying the operand by some power of two and right-shifting involves dividing the

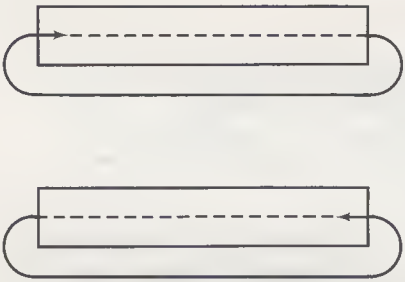


Fig. 13-2. Rotate operations.

Example 13-2. Rotate a number right or left

```
: ROTATE
  ( Rotate un1 by n bit positions. Bits shifted out of )
  ( one end of the number are shifted back in the  )
  ( opposite end. If n is positive, rotate un1 to the  )
  ( left; if n is negative, rotate un1 to the right.  )
  ( un1 n --- un2 )
  OVER OVER              ( Make a copy of un1 and n)
  SHIFT                  ( Shift un1 and save result, un2, )
  >R                     (  on the return stack  )
  DUP 0<                 ( Determine rotate direction)
  IF                     ( For right rotate,  )
   16 +                  (  shift count = n + 16)
  ELSE                   ( For left rotate,  )
   16 -                  (  shift count = n - 16)
  THEN
  SHIFT                  ( Shift vacated bits  )
  R> OR ;                (  and combine the two results)
```

operand by some power of two. In both cases, the exponent of two is nothing more than the number of bit positions the operand is to be shifted. For example, to shift an operand left by three bit positions, multiply it by two to the third power.

Example 13-1 shows the definitions for both ASHIFT and SHIFT. These definitions are similar, except ASHIFT employs the signed multiply and divide words, [/] and [*], whereas SHIFT employs the unsigned multiply and divide words, U/MOD and

U*. Note that the word [**], which was defined in Example 7-3, is used to form the shift factor for both words. You might like to modify the definition of ASHIFT to detect whether a left-shift has altered the sign bit, which produces an invalid result.

ROTATE

A rotate operation is similar to a shift operation except that, with a rotate, bit values displaced out of one end of the operand are entered into the opposite end, to fill vacated bit positions. Thus, no bit values are ever "lost" in a rotate, as they are in a shift. Fig. 13-2 shows how a right rotate and left rotate operate.

Because a sign bit is irrelevant in a rotate operation, we need only one FORTH word here. This word, called ROTATE, is defined in Example 13-2. This definition is comprised of two SHIFT operations, one to shift the number and the other to shift the "vacated" bit positions. The results of these two operations are then combined (ORed) to form the final result.

SUMMARY

This chapter described words that manipulate bits in a number. These included the logical words AND, OR, XOR, and COM, the shift words ASHIFT (arithmetic shift, for signed numbers), SHIFT (logical shift, for unsigned numbers), and the word ROTATE. Only the logical words AND, OR, and XOR are standard in both FORTH-79 and fig-FORTH.

APPENDIX A

Hex/Decimal and ASCII Conversion Tables

Table A-1. Hexadecimal and Decimal Conversion

HEXADECIMAL COLUMNS											
6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
7654		3210		7654		3210		7654		3210	
Byte				Byte				Byte			

Table A-2. Powers of 2

2^n	n
256	8
512	9
1 024	10
2 048	11
4 096	12
8 192	13
16 384	14
32 768	15
65 536	16
131 072	17
262 144	18
524 288	19
1 048 576	20
2 097 152	21
4 194 304	22
8 388 608	23
16 777 216	24

Table A-3. Powers of 16

16^n	*n*
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10
17 592 186 044 416	11
281 474 976 710 656	12
4 503 599 627 370 496	13
72 057 594 037 927 936	14
1 152 921 504 606 846 976	15

Table A-4. ASCII Character Values

Decimal	Hex	Character
0	0	CTRL-@ (Null)
1	1	CTRL-A
2	2	CTRL-B
3	3	CTRL-C
4	4	CTRL-D
5	5	CTRL-E

Table A-4—cont. ASCII Character Values

6	6	CTRL-F
7	7	CTRL-G (Bell)
8	8	CTRL-H (Backspace)
9	9	CTRL-I (Horizontal Tab)
10	A	CTRL-J (Line Feed)
11	B	CTRL-K (Vertical Tab)
12	C	CTRL-L (Form Feed)
13	D	CTRL-M (Carriage Return)
14	E	CTRL-N
15	F	CTRL-O
16	10	CTRL-P
17	11	CTRL-Q
18	12	CTRL-R
19	13	CTRL-S
20	14	CTRL-T
21	15	CTRL-U (Forward Space)
22	16	CTRL-V
23	17	CTRL-W
24	18	CTRL-X (Cancel)
25	19	CTRL-Y
26	1A	CTRL-Z
27	1B	CTRL-SHIFT K (Escape)
28	1C	CTRL-SHIFT L
29	1D	CTRL-SHIFT M
30	1E	CTRL-SHIFT N
31	1F	CTRL-SHIFT O
32	20	Space (Blank)
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	' (apostrophe)
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	, (comma)
45	2D	– (minus or hyphen)
46	2E	. (period)
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9

Table A-4—cont. ASCII Character Values

58	3A	: (colon)
59	3B	; (semicolon)
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	↑
95	5F	_ (underline)
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m

Table A-4—cont. ASCII Character Values

110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	Delete

APPENDIX B

FORTH Word Summaries

This appendix summarizes all of the FORTH-79 Required Words and the most frequently used fig-FORTH words, and identifies some differences between them. For complete descriptions of these words, refer to the FORTH-79 Standard or the fig-FORTH Installation Manual, both available from:

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

FORTH-79 REQUIRED WORDS

Stack Manipulation

DUP	(n --- n n)	Duplicate top number on stack.
DROP	(n ---)	Discard top number on stack.
SWAP	(n1 n2 --- n2 n1)	Exchange two top numbers.
OVER	(n1 n2 --- n1 n2 n1)	Make copy of second number on top.
ROT	(n1 n2 n3 --- n2 n3 n1)	Rotate third number to top.
PICK	(n1 --- n2)	Copy n1-th number to top. (Thus, 1 PICK = DUP, 2 PICK = OVER)
ROLL	(n --- (n))	Rotate n-th number to top. (Thus, 2 ROLL = SWAP, 3 ROLL = ROT)
?DUP	(n --- n (n))	Duplicate only if nonzero. *Called -DUP in fig-FORTH. Called R in fig-FORTH.*
>R	(n ---)	Move top number to return stack.
R>	(--- n)	Retrieve number from return stack.
R@	(--- n)	Copy top of return stack onto data stack.
DEPTH	(--- n)	Count numbers on stack.

Comparison

<	(n1 n2 --- flag)	True if n1 is less than n2.
=	(n1 n2 --- flag)	True if top two numbers are equal.
>	(n1 n2 --- flag)	True if n1 is greater than n2.
0<	(n --- flag)	True if top number is negative.
0=	(n --- flag)	True if top number is zero. (Equivalent to NOT)
0>	(n --- flag)	True if top number is greater than zero.
NOT	(flag1 --- flag2)	Reverse value of flag. (Equivalent to 0=)

D<	(d1 d2 --- flag)	True if d1 is less than d2.
U<	(un1 un2 --- flag)	Compare top two items as unsigned integers.

Arithmetic and Logical

+	(n1 n2 --- sum)	Add numbers.
D+	(d1 d2 --- dsum)	Add double numbers.
−	(n1 n2 --- diff)	Subtract numbers (n1−n2).
1+	(n --- n+1)	Add 1 to top number.
1−	(n --- n−1)	Subtract 1 from top number.
2+	(n --- n+2)	Add 2 to top number.
2−	(n --- n−2)	Subtract 2 from top number.
*	(n1 n2 --- prod)	Multiply.
/	(n1 n2 --- quot)	Divide (n1/n2). Quotient rounded toward zero.

Arithmetic and Logical (cont'd)

MOD	(n1 n2 --- rem)	Leave remainder from division n1/n2. Remainder has same sign as n1.
/MOD	(n1 n2 --- rem quot)	Divide, leaving remainder and quotient.
*/MOD	(n1 n2 n3 --- rem quot)	Multiply, then divide (n1*n2/n3), with double-precision intermediate product.
*/	(n1 n2 n3 --- quot)	Like */MOD, but leave only quotient, rounded toward zero.
U*	(un1 un2 --- udprod)	Multiply unsigned numbers, leaving unsigned double-number product.
U/MOD	(ud un --- urem uquot)	Divide double number by single, giving single remainder and quotient, all unsigned. *Called U/ in fig-FORTH.*
MAX	(n1 n2 --- max)	Leave greater of two numbers.

MIN	(n1 n2 --- min)	Leave lesser of two numbers.
ABS	(n --- |n|)	Absolute value.
NEGATE	(n --- -n)	Leave two's complement. *Called MINUS in fig-FORTH.*
DNEGATE	(d --- -d)	Leave two's complement of double number. *Called DMINUS in fig-FORTH.*
AND	(n1 n2 --- AND)	Logical AND of n1 and n2.
OR	(n1 n2 --- OR)	Logical OR of n1 and n2.
XOR	(n1 n2 --- XOR)	Logical exclusive-OR of n1 and n2.
Memory		
@	(addr --- n)	Fetch number address.
!	(n addr ---)	Store number at address.
C@	(addr --- byte)	Fetch least-significant byte only.
C!	(n addr ---)	Store least-significant byte only.
?	(addr ---)	Display number at addr.
+!	(n addr ---)	Add n to number at addr.
MOVE	(addr1 addr2 n ---)	Copy n numbers starting at addr1 to memory starting at addr2, if n>0.
CMOVE	(addr1 addr2 n ---)	Copy n bytes starting at addr1 to memory starting at addr2, if n>0.
FILL	(addr n byte ---)	Fill n bytes in memory with byte, starting at addr.
Control Structures		
DO . . . LOOP	do: (end+1 start ---)	Set up loop, given index range.
I	(--- index)	Place current DO-LOOP index on data stack.
J	(--- index)	Return index of next outer DO-LOOP.
DO . . . +LOOP	do: (limit start ---)	Like DO . . . LOOP, but adds stack value to index, instead of always 1. Loop terminates

		when index is greater than or equal to limit (n>0), or when index is less than limit (n<0).
LEAVE	(---)	Terminate loop at next LOOP or +LOOP, by setting limit equal to index.
IF . . . (true) . . . THEN	if: (flag ---)	If top of stack is true, execute. *THEN is called ENDIF in fig-FORTH.*
IF . . . (true) . . . ELSE	if: (flag ---)	Same, but if flag is false, execute ELSE clause.
BEGIN . . . UNTIL	until: (flag ---)	Loop back to BEGIN until flag is true at UNTIL.
BEGIN . . . WHILE . . . REPEAT	while: (flag ---)	Loop while flag is true at WHILE; REPEAT loops unconditionally to BEGIN. When flag is false, continue after REPEAT.
EXIT	(---)	Terminate execution of colon definition. May not be used within DO-LOOP.
EXECUTE	(addr ---)	Execute dictionary entry at compilation address on stack; for example, address returned by FIND.

Terminal Input/Output

. (period)	(n ---)	Print number with one trailing blank.
U.	(un ---)	Print top number as unsigned integer with one trailing blank.
CR	(---)	Do a carriage return and line feed.
SPACE	(---)	Type one space.
SPACES	(n ---)	Type n spaces, if n>0.
."	(---)	Print message (terminated by ").
TYPE	(addr n ---)	Type string of n characters, starting at addr.
COUNT	(addr --- addr+1 n)	Change address of string (preceded by length byte at addr) to TYPE form.

−TRAILING	(addr n1 −−− addr n2)	Reduce character count of string at addr to omit trailing blanks.
KEY	(−−− char)	Read key and leave ASCII value on stack.
EMIT	(char −−−)	Type ASCII value from stack.
EXPECT	(addr n −−−)	Read n characters (or until carriage return from terminal to address, with null(s) at end.
QUERY	(−−−)	Read line of up to 80 characters from terminal to input buffer.
WORD	(char −−− addr)	Read next word from input stream using char (usually blank) as delimiter. Leave address of length byte. *In fig-FORTH, WORD does not leave addr.*

Numeric Conversion

BASE	(−−− addr)	System variable containing the current input/output numeric conversion base (2-70).
DECIMAL	(−−−)	Set decimal number base.
CONVERT	(d1 addr1 −−− d2 addr2)	Convert string at addr1+1 to double number. New value is accumulated into d1 (initially zero), and left as d2; addr 2 is address of first nondigit.
<#	(−−−)	Start converting numeric output string.
#	(ud1 −−− ud2)	Convert next digit of unsigned double number and add character to output string.
#S	(ud −−− 0 0)	Convert all significant digits of unsigned double number to output string.
HOLD	(char −−−)	Insert ASCII character into output string.

SIGN	(---)	Add minus sign to output string if n<0. *SIGN differs somewhat in fig-FORTH.*
#>	(d --- addr un)	Terminate output string (ready for TYPE).
Mass Storage Input/Output		
LIST	(n ---)	List screen n and set SCR to contain n.
LOAD	(n ---)	Interpret screen n.
BLOCK	(n --- addr)	Leave memory address of block reading from mass storage if necessary.
BLK	(--- addr)	System variable containing current block number, or 0 if from terminal.
SCR	(--- addr)	System variable containing current screen number.
UPDATE	(---)	Mark last buffer as modified.
BUFFER	(n --- addr)	Obtain the next memory buffer and assign it to block n. If this buffer has been UPDATEd, write its previous contents to mass storage.
SAVE-BUFFERS	(---)	Write all UPDATEd buffers to mass storage. *Called FLUSH in fig-FORTH.*
EMPTY-BUFFERS	(---)	Mark all block buffers as empty.
Defining Words		
: name	(---)	Begin colon definition of name.
;	(---)	End colon definition.
VARIABLE name	(---)	Create a two-byte variable named name. Returns address when executed. *In fig-FORTH, the variable is initialized.*
	name: (--- addr)	

CONSTANT	(n ---)	Create a two-byte constant named name. Returns value when executed.
CREATE . . . DOES>	does: (--- addr)	Used to create a new defining word, with execution-time routine in high-level FORTH. *In fig-FORTH, CREATE is called <BUILDS.*
Vocabularies		
CONTEXT	(--- addr)	System variable pointing to vocabulary where word names are first searched for.
CURRENT	(--- addr)	System variable pointing to vocabulary where new definitions are put.
FORTH	(---)	Main vocabulary. Execution of FORTH sets CONTEXT vocabulary.
DEFINITIONS	(---)	Set CURRENT vocabulary to CONTEXT.
VOCABULARY name	(---)	Create a new vocabulary named name.
' name	(--- addr)	Find address of name in dictionary; if used in definition, compile address.
FIND	(--- addr)	Leave compilation address of next word in input stream. If not found in CONTEXT or FORTH, leave 0.
FORGET name	(---)	Forget all definitions back to and including name.
Compiler		
,	(n ---)	Compile a number into the dictionary.
ALLOT	(n ---)	Add n bytes to the parameter field of the most recently defined word.

IMMEDIATE	(---)	Mark last-defined word to be executed, rather than compiled, when encountered in a definition.
LITERAL	(n ---)	If compiling, save n in dictionary, to be returned to stack when definition is executed.
STATE	(--- addr)	System variable whose value is nonzero when compilation is occurring.
[	(---)	Stop compiling input text and begin executing.
]	(---)	Stop executing input text and begin compiling.
COMPILE	(---)	Compile the address of the next nonIMMEDIATE word into the dictionary.
[COMPILE]	(---)	Compile the following word, even if IMMEDIATE.

Miscellaneous

(	(---)	Begin comment, terminated by) on same line; space is required after (.
ABORT	(---)	Clear data and return stacks, set execution in mode, return control to terminal. *In fig-FORTH, ABORT also prints an error message.*
QUIT	(---)	Like ABORT, except does not clear data stack or print any message.
HERE	(--- addr)	Leave address of next available dictionary location.
PAD	(--- addr)	Leave address of a scratch area of at least 64 bytes. *In fig-FORTH, the pad area is usually 68 bytes beyond HERE.*

>IN	(--- addr)	System variable containing offset into input buffer. Used, for example, by WORD. *Called IN in fig-FORTH.*
79-STANDARD	(---)	Verify that system conforms to FORTH-79 Standard.

fig-FORTH WORDS

Stack Manipulation		
DUP	(n --- n n)	Duplicate top number on stack.
DROP	(n ---)	Discard top number on stack.
SWAP	(n1 n2 --- n2 n1)	Exchange two top numbers.
OVER	(n1 n2 --- n1 n2 n1)	Make copy of second number on top.
ROT	(n1 n2 n3 --- n2 n3 n1)	Rotate third number to top.
-DUP	(n --- n (n))	Duplicate only if nonzero. *Called ?DUP in FORTH-79.*
>R	(n ---)	Move top number to return stack.
R>	(--- n)	Retrieve number from return stack.
R	(--- n)	Copy top of return stack onto data stack. *Called R@ in FORTH-79.*
S0	(--- addr)	User variable containing initial value for the stack pointer.
SP!	(---)	User-supplied procedure to initialize the stack pointer from S0.
SP@	(--- addr)	Leave address of top of the stack.
Comparison		
<	(n1 n2 --- flag)	True if n1 is less than n2.
=	(n1 n2 --- flag)	True if top two numbers are equal.

>	(n1 n2 --- flag)	True if n1 is greater than n2.
0<	(n --- flag)	True if top number is negative.
0=	(n --- flag)	True if top number is zero (reverses truth value).

Arithmetic and Logical

+	(n1 n2 --- sum)	Add numbers.
D+	(d1 d2 --- dsum)	Add double numbers.
−	(n1 n2 --- diff)	Subtract numbers (n1-n2).
1+	(n --- n+1)	Add 1 to top number.
2+	(n --- n+2)	Add 2 to top number.
*	(n1 n2 --- prod)	Multiply.
/	(n1 n2 --- quot)	Divide (n1/n2). Quotient rounded toward zero.
MOD	(n1 n2 --- rem)	Leave remainder from division n1/n2. Remainder has same sign as n1.
/MOD	(n1 n2 --- rem quot)	Divide, leaving remainder and quotient.
*/MOD	(n1 n2 n3 --- rem quot)	Multiply, then divide (n1*n2/n3), with double-precision intermediate product.
*/	(n1 n2 n3 --- quot)	Like */MOD, but leave only quotient, rounded toward zero.
U*	(un1 un2 --- udprod)	Multiply unsigned numbers, leaving unsigned double-number product.
U/	(ud un --- urem uquot)	Divide double number by single, giving single remainder and quotient, all unsigned. *Called U/MOD in FORTH-79.*
M*	(n1 n2 --- dprod)	Same as [U*], but numbers are signed.
M/	(d n --- rem quot)	Same as [U/], but numbers are signed.
M/MOD	(ud un --- rem dquot)	Same as [U/], but leaves single remainder and double quotient.

MAX	(n1 n2 --- max)	Leave greater of two numbers.
MIN	(n1 n2 --- min)	Leave lesser of two numbers.
ABS	(n --- n)	Absolute value.
DABS	(d --- d)	Absolute value of a double number.
MINUS	(n --- −n)	Leave two's complement. *Called NEGATE in FORTH-79.*
DMINUS	(d --- −d)	Leave two's complement of a double number. *Called DNEGATE in FORTH-79.*
AND	(n1 n2 --- AND)	Logical AND of n1 and n2.
OR	(n1 n2 --- OR)	Logical OR of n1 and n2.
XOR	(n1 n2 --- XOR)	Logical exclusive-OR of n1 and n2.
Memory		
@	(addr --- n)	Fetch number at address.
!	(n addr ---)	Store number at address.
C@	(addr --- byte)	Fetch one byte only.
C!	(n addr ---)	Store least-significant byte only.
?	(addr ---)	Display number at addr.
+!	(n addr ---)	Add n to number at addr.
MOVE	(addr1 addr2 n ---)	Copy n numbers starting at addr1 to memory starting at addr2, if n>0.
CMOVE	(addr1 addr2 n ---)	Copy n bytes starting at addr1 to memory starting at addr2, if n>0.
FILL	(addr n byte ---)	Fill n bytes in memory with byte, starting at addr.
ERASE	(addr n ---)	Fill n bytes in memory with zeroes, starting at addr.
BLANKS	(addr n ---)	Fill n bytes in memory with blanks, starting at addr.

Control Structures

DO . . . LOOP	do: (end+1 start ---)	Set up loop, given index range.
I	(--- index)	Place current DO-LOOP index on data stack.
DO . . . +LOOP	do: (limit start ---) +loop: (n ---)	Like DO . . . LOOP, but adds stack value to index, instead of always 1. Loop terminates when index is greater than or equal to limit (n>0), or when index is less than limit (n<0).
LEAVE	(---)	Terminate loop at next LOOP or +LOOP, by setting limit equal to index.
IF . . . (true) . . . ENDIF	if: (flag ---)	If top of stack is true, execute. *ENDIF is called THEN in FORTH-79; THEN is also valid in fig-FORTH.*
IF . . . (true) . . . ELSE . . . (false) . . . ENDIF	if: (flag ---)	Same, but if flag is false, execute ELSE clause.
BEGIN . . . UNTIL	until: (flag ---)	Loop back to BEGIN until flag is true at UNTIL.
BEGIN . . . WHILE . . . REPEAT	while: (flag ---)	Loop while flag is true at WHILE; REPEAT loops unconditionally to BEGIN. When flag is false, continue after REPEAT.
EXECUTE	(addr ---)	Execute dictionary entry at compilation address on stack.

Terminal Input/output

. (period)	(n ---)	Print number with one trailing blank.
.R	(n fieldwidth ---)	Print number, right-justified in field.
D.	(d ---)	Print double number, with one trailing blank.
D.R.	(d fieldwidth ---)	Print double number, right-justified in field.
CR	(---)	Do a carriage return and line feed.

SPACE	(---)	Type one space.
SPACES	(n ---)	Type n spaces.
."	(---)	Print message (terminated by ").
DUMP	(addr n ---)	Dump n words, starting at addr.
TYPE	(addr n ---)	Type string of n characters, starting at addr.
COUNT	(addr --- addr+1 n)	Change address of string (preceded by length byte at addr) to TYPE form.
-TRAILING	(addr n1 --- addr n2)	Reduce character count of string at addr to omit trailing blanks.
$TERMINAL	(--- flag)	True if terminal break key is pressed.
KEY	(--- char)	Read key and leave ASCII value on stack.
EMIT	(char ---)	Type ASCII value from stack.
EXPECT	(addr n ---)	Read n characters (or until carriage return) from terminal to address, with null(s) at end.
WORD	(char ---)	Read next word from input stream, using char (usually blank) as delimiter. *In FORTH-79, WORD leaves the address of the length byte.*

Numeric Conversion

BASE	(--- addr)	System variable containing the current input/output numeric conversion base (2-70).
DECIMAL	(---)	Set decimal number base.
HEX	(---)	Set hexadecimal number base.
NUMBER	(addr --- d)	Convert string at addr to double number.
<#	(---)	Start converting numeric output string.
#	(d1 --- d2)	Convert next digit of double number and add character to output string.
#S	(d --- 0 0)	Convert all significant digits of double number to output string.

HOLD	(char ---)	Insert ASCII character into output string.
SIGN	(n d1 --- d2)	Add minus sign to output string if n<0. *SIGN differs somewhat in FORTH-79.*
#>	(d --- addr un)	Terminate output string (ready for TYPE).
Mass Storage Input/Output		
LIST	(n ---)	List screen n and set SCR to contain n.
LOAD	(n ---)	Interpret screen n.
BLOCK	(n --- addr)	Leave memory address of block, reading from mass storage if necessary.
B/BUF	(--- n)	System constant giving disk block size in bytes.
BLK	(--- addr)	System variable containing current block number, or 0 if from terminal.
SCR	(--- addr)	System variable containing current screen number.
UPDATE	(---)	Mark last buffer as modified.
BUFFER	(*n* --- addr)	Obtain the next memory buffer and assign it to block n. If this buffer has been UPDATEd, write its previous contents to mass storage.
FLUSH	(---)	Write all UPDATEd buffers to mass storage. *Called SAVE-BUFFERS in FORTH-79.*
EMPTY-BUFFERS	(---)	Mark all block buffers as empty.
Defining Words		
: name	(---)	Begin colon definition of name.
;	(---)	End colon definition.
VARIABLE name	(n ---)	Create a two-byte variable named name, with initial value n. Returns address when executed. *In FORTH-79, the variable is not initialized.*
	name: (--- addr)	

CONSTANT name	(n ---) name: (--- n)	Create a two-byte constant named name, with value n. Returns value when executed.
;CODE	(---)	Used to create a new defining word, with execution-time "code routine" for this data assembly language.
<BUILDS . . . DOES>	does: (--- addr)	Used to create a new defining word, with execution-time routine in high-level FORTH. *In FORTH-79, <BUILDS is called CREATE.*

Vocabularies

CONTEXT	(--- addr)	System variable pointing to vocabulary where word names are first searched for.
CURRENT	(--- addr)	System variable pointing to vocabulary where new definitions are put.
FORTH	(---)	Main vocabulary. Execution of FORTH sets CONTEXT vocabulary.
EDITOR	(---)	Editor vocabulary; sets CONTEXT.
ASSEMBLER	(---)	Assembler vocabulary; sets CONTEXT.
DEFINITIONS	(---)	Set CURRENT vocabulary to CONTEXT.
VOCABULARY name	(---)	Create a new vocabulary named name.
' name	(--- addr)	Find address of name in dictionary; if used in definition, compile address.
VLIST	(---)	Print names of all words in CONTEXT vocabulary.
FORGET name	(---)	Forget all definitions back to and including name.

Compiler

, (comma)	(n ---)	Compile a number into the dictionary.

C,	(byte ---)	Compile a byte into the dictionary.
ALLOT	(n ---)	Add n bytes to the parameter field of the most recently defined word.
IMMEDIATE	(---)	Mark last-defined word to be executed, rather than compiled, when encountered in a definition.
LITERAL	(n ---)	If compiling, save n in dictionary, to be returned to stack when definition is executed.
STATE	(--- addr)	System variable whose value is nonzero when compilation is occurring.
COMPILE	(---)	Compile the address of the next nonIMMEDIATE word into the dictionary.

Miscellaneous

(	(---)	Begin comment, terminated by) on same line; space is required after (.
ABORT	(---)	Clear data and return stacks, set execution mode, return control to terminal and print an error message.
QUIT	(---)	Like ABORT, except does not clear data stack or print any message.
HERE	(--- addr)	Leave address of next available dictionary location.
PAD	(--- addr)	Leave the address of a scratch area, which is a fixed offset (usually 68 bytes) above HERE. *In FORTH-79, the location of the pad area is unspecified, but must be at least 64 bytes long.*

IN (--- addr)

System variable containing offset into input buffer. Used, for example, by WORD. *Called >IN in FORTH-79.*

APPENDIX C

Double Number Extension Words

Since most applications need no more precision than can be provided by 16 bits, FORTH words in both the FORTH-79 and fig-FORTH dialects primarily operate on numbers. However, both dialects also include a few words that can operate on 32-bit *double-number* values as a basis from which you can develop words to process larger values. This appendix summarizes the words described in the FORTH-79 Standard's Double Number Extension Word Set, and gives colon-definitions you can use to add these capabilities to your own system.

DOUBLE NUMBER EXTENSION WORD DESCRIPTIONS

2! d addr ---
Store d in four consecutive bytes, beginning at addr.

2@ addr --- d
Leave on the stack the contents of the four consecutive bytes beginning at addr.

2CONSTANT d ---
A defining word used in the form
d **2CONSTANT** name
to create a dictionary entry for *name*, leaving *d* in its parameter field. When name is later executed, d will be left on the stack.

2DROP d ---
Drop the top double number on the stack.

2DUP
Duplicate the top double number on the stack.

2OVER d1 d2 --- d1 d2 d1
Leave a copy of the second double number on the stack.

2ROT d1 d2 d3 --- d2 d3 d1
Ratate the third double number ta the tap af the stack.

2SWAP d1 d2 --- d2 d1
Exchange the tap twa dauble numbers an the stack.

2VARIABLE
A defining ward used in the farm
2VARIABLE name
ta create a dictianary entry far name and assign faur bytes far storage in its parameter field. When name is later executed, it will leave the address af the first byte in its parameter field an the stack.

D+ d1 d2 --- d3
Leave the arithmetic sum af d1 and d2.

D− d1 d2 --- d3
Subtract d1 fram d2 and leave the difference d3.

D. d ---
Display d, converted accarding ta BASE, in a free-field farmat, with ane trailing blank. Display the sign only if negative.

D.R d n ---
Display d, converted accarding ta BASE, right-aligned in an n-character field. Display the sign anly if negative.

DO= d --- flag
Leave true flag if d is zera.

D< d1 d2 --- flag
True if d1 is less than d2.

D= d1 d2 ---flag
True if d1 equals d2.

DABS d --- |d|
Leave the absalute value of the double number d.

DMAX d1 d2 ---d3
Leave the larger af twa dauble numbers.

DMIN d1 d2 --- d3
Leave the smaller af twa dauble numbers.

DNEGATE d --- −d
Leave the twa's camplement af a dauble number. DNEGATE is called DMINUS in fig-FORTH.

DU< ud1 ud2 --- flag
True if ud1 is less than ud2. Bath numbers are unsigned.

COLON-DEFINITIONS FOR DOUBLE NUMBER EXTENSION WORDS

```
: 2!        ( Store dauble number at address )
  SWAP OVER ! 2+ ! ;
```

```
: 2@        ( Fetch double number from address )
  DUP 2+ @ SWAP @ ;

: 2DROP     ( Drop double number )
  DROP DROP ;
: 2DUP      ( Duplicate double number )
  OVER OVER ;

: 2OVER     ( Leave copy of second double number )
  4 PICK 4 PICK ;

: 2ROT      ( Rotate third double number to top )
  6 ROLL 6 ROLL ;

: 2SWAP     ( Exchange top two double numbers )
  4 ROLL 4 ROLL

: D-        ( Subtract double number )
  DNEGATE D+ ;
```

(*Note:* In fig-FORTH, DNEGATE is called DMINUS.)

```
: D+-       ( Apply sign of number to double number )
  0< IF DNEGATE THEN ;
```

(*Note:* D+- is included in fig-FORTH.)

```
: DABS      ( Leave absolute value of double number )
  DUP D+- ;
```

(*Note:* DABS is included in fig-FORTH.)

```
: D.R       ( Print double number right-aligned in n-char field )
  >R SWAP OVER DABS
  <# #S SIGN #>
  R> OVER - SPACES TYPE ;
```

(*Note:* D.R is included in fig-FORTH.)

```
: D.        ( Print double number followed by one blank )
  0 D.R SPACE ;
```

(*Note:* D. is included in fig-FORTH.)

```
: D0=       ( Leave true flag if double number is zero )
  OR 0= ;

: D<        ( True if 2nd double no. less than 1st double no. )
  D- SWAP DROP 0< ;

: D=        ( True if two double numbers are equal )
  D- D0= ;

: DMAX      ( Leave the larger of two double numbers )
  2OVER 2OVER D<
  IF 2SWAP THEN
  2DROP ;
```

```
: DMIN          ( Leave the smaller of two double numbers )
  2OVER 2OVER D< NOT
  IF 2SWAP THEN
  2DROP ;
```

(*Note:* In fig-FORTH, replace NOT with 0=.)

```
: 2VARIABLE         ( Double variable )
  CREATE 4 ALLOT DOES> ;

: 2,          ( Enter stack double number to dictionary )
  HERE 2! 4 ALLOT ;

: 2CONSTANT         ( DOUBLE CONSTANT )
  CREATE 2, DOES>
  2@ ;
```

Index

The Blacksburg Group

According to Business Week magazine (Technology July 6, 1976) large scale integrated circuits or LSI "chips" are creating a second industrial revolution that will quickly involve us all. The speed of the developments in this area is breathtaking and it becomes more and more difficult to keep up with the rapid advances that are being made. It is also becoming difficult for newcomers to "get on board."

It has been our objective, as The Blacksburg Group, to develop timely and effective educational materials that will permit students, engineers, scientists, technicians and others to quickly learn how to use new technologies and electronic techniques. We continue to do this through several means, textbooks, short courses, seminars and through the development of special electronic devices and training aids.

Our group members make their home in Blacksburg, found in the Appalachian Mountains of southwestern Virginia. While we didn't actively start our group collaboration until the Spring of 1974, members of our group have been involved in digital electronics, minicomputers and microcomputers for some time.

Some of our past experiences and on-going efforts include the following:

–The design and development of what is considered to be the first popular hobbyist computer. The Mark-8 was featured in Radio-Electronics magazine in 1974. We have also designed several 8080-based computers, including the MMD-1 system. Our most recent computer is an 8085-based computer for educational use, and for use in small controllers.

—The Blacksburg Continuing Education Series™ covers subjects ranging from basic electronics through microcomputers, operational amplifiers, and active filters. Test experiments and examples have been provided in each book. We are strong believers in the use of detailed experiments and examples to reinforce basic concepts. This series originally started as our Bugbook series and many titles are now being translated into Chinese, Japanese, German and Italian.

–We have pioneered the use of small, self-contained computers in hands-on courses for microcomputer users. Many of our designs have evolved into commercial products that are marketed by E&L Instruments and PACCOM, and are available from Group Technology, Ltd., Check, VA 24072.

–Our short courses and seminar programs have been presented throughout the world. Programs are offered by The Blacksburg Group, and by the Virginia Polytechnic Institute Extension Division. Each series of courses provides hands-on experience with real computers and electronic devices. Courses and seminars are provided on a regular basis, and are also provided for groups, companies and schools at a site of their choosing. We are strong believers in practical laboratory exercises, so much time is spent working with electronic equipment, computers and circuits.

Additional information may be obtained from Dr. Chris Titus, the Blacksburg Group, Inc. (703) 951-9030 or from Dr. Linda Leffel, Virginia Tech Continuing Education Center (703) 961-5241.

Our group members are Mr. David G. Larsen, who is on the faculty of the Department of Chemistry at Virginia Tech, and Drs. Jon Titus and Chris Titus who work full-time with The Blacksburg Group, all of Blacksburg, VA.