microFORTH PRIMER

FORTH, Inc.

microFORTH PRIMER

Second Edition

FORTH, Inc.

815 Manhattan Avenue

Manhattan Beach, CA 90266

(213) 372-8493

August 1978

Second edition
9 8 7 6 5 4 3 2 1

This book was produced by use of the textFORTH System.

FORTH, Inc.

815 Manhattan Avenue

Manhattan Beach, CA 90266

TABLE OF CONTENTS

FIGURES AND TABLES

FIGURES

TABLES

PREFACE

In order to make reading our documentation as easy as possible, we at FORTH, Inc. use the following conventions in manuals:

1. All FORTH words that appear in prose passages as examples of commands are enclosed by at least one extra space on each side. Words defined as occasional examples are also set off.

2. Where there might be confusion about who types what, we underline the computer's output.

3. In all examples that show stack usage, the top item of the stack appears to the right (as it does on your terminal screen when you are entering).

4. We provide brief examples of definitions as often as possible. After the normal, horizontal placement of a definition, we often provide a vertical breakdown with components of the definition in a column to the left and explanations or comments on key words in a column to the right:

```
: DEFINITION    condition  IF this  ELSE that
    THEN continue ;
```

where:

: DEFINITION	
condition	Places a condition (non-zero/zero) on the stack.
IF	Removes and tests the number on the stack.
this	Executes "this" if the number was non-zero (true).
ELSE	
that	Executes "that" if the number was zero (false).
THEN	
continue ;	Continues from both lines.

This expanded version is for illustration only; you will always use the horizontal format.

Additional conventions used in FORTH manuals are those that FORTH programmers use to make source screens readable. At the end of Chapter 2 we provide a full listing of FORTH editing conventions for source text. Here are three that you will observe in the examples of Chapters 1 and 2:

1. Although only one space is absolutely necessary between each word of a definition, spacing three times after a new word that is being defined sets off the major components.

2. Double spacing between phrases (logical clusters) of a definition also helps make source text legible.

3. When a definition takes up more than one line, the following lines begin with an indentation of three or more spaces to save the left margin for words being defined.

While illustrative figures closely follow the portions of text to which they apply, we place tables at the ends of Chapters 1, 2, and 3 for ease of later reference. There are a few fundamental procedures that must be observed in order to write clear FORTH programs; we call these Rules. Each is set off in upper-case and numbered on its first appearance. They are also printed together as a list which makes up Appendix A. Appendix B consists of a microFORTH Glossary that contains words common to all systems.

For microFORTH users we publish three levels of documentation. The microFORTH Primer covers the broadest and most basic aspects since it is intended for the newcomer to programming to work through before commencing study of the microFORTH Technical Manual. The Primer also serves as a prospectus for experienced programmers to read quickly in order to spot philosophical differences between FORTH and other high-level languages and/or operating systems.

Although the Primer is written for novice programmers, we do assume a basic understanding of computer terms. If you lack background, we suggest that you study first either of Adam Osborne's groundwork books, An Introduction to Microcomputers, Volumes 0 or 1 (Berkeley, CA: Osborne and Associates, Inc.) or Granino A. Korn's Minicomputers for Engineers and Scientists (N.Y.: McGraw-Hill).

The second level of documentation is more specific. The microFORTH Technical Manual presents the basic material in a condensed manner before expanding on the subjects of the compiler, the assembler, and the cross-compiler. Since hardware varies from chip to chip, we publish four versions of the Technical Manual, one each for the 8080, 6800, 1802, and Z80. Each version will soon include the appropriate glossaries for the hardware-dependent definitions of assembler and cross-compiler. When the 4th, completely new edition of this manual comes out (in 1979), all microFORTH customers will receive copies; updates and/or errata sheets will be issued thereafter.

Besides the differences in chips, variations exist within each chip category for particular development systems on which microFORTH has been produced. These differences are documented in the listings and CPU-specific instructions that are issued with each microFORTH system. When users report especially useful solutions to problems that arise during initial use of microFORTH systems, we

share the information in these packets. The documentation of Options likewise accompanies delivery of each particular optional program.

Since we want you to make the most of your microFORTH system, we have developed a Hotline service, programming classes, and the FORTH, Inc. Newsletter.

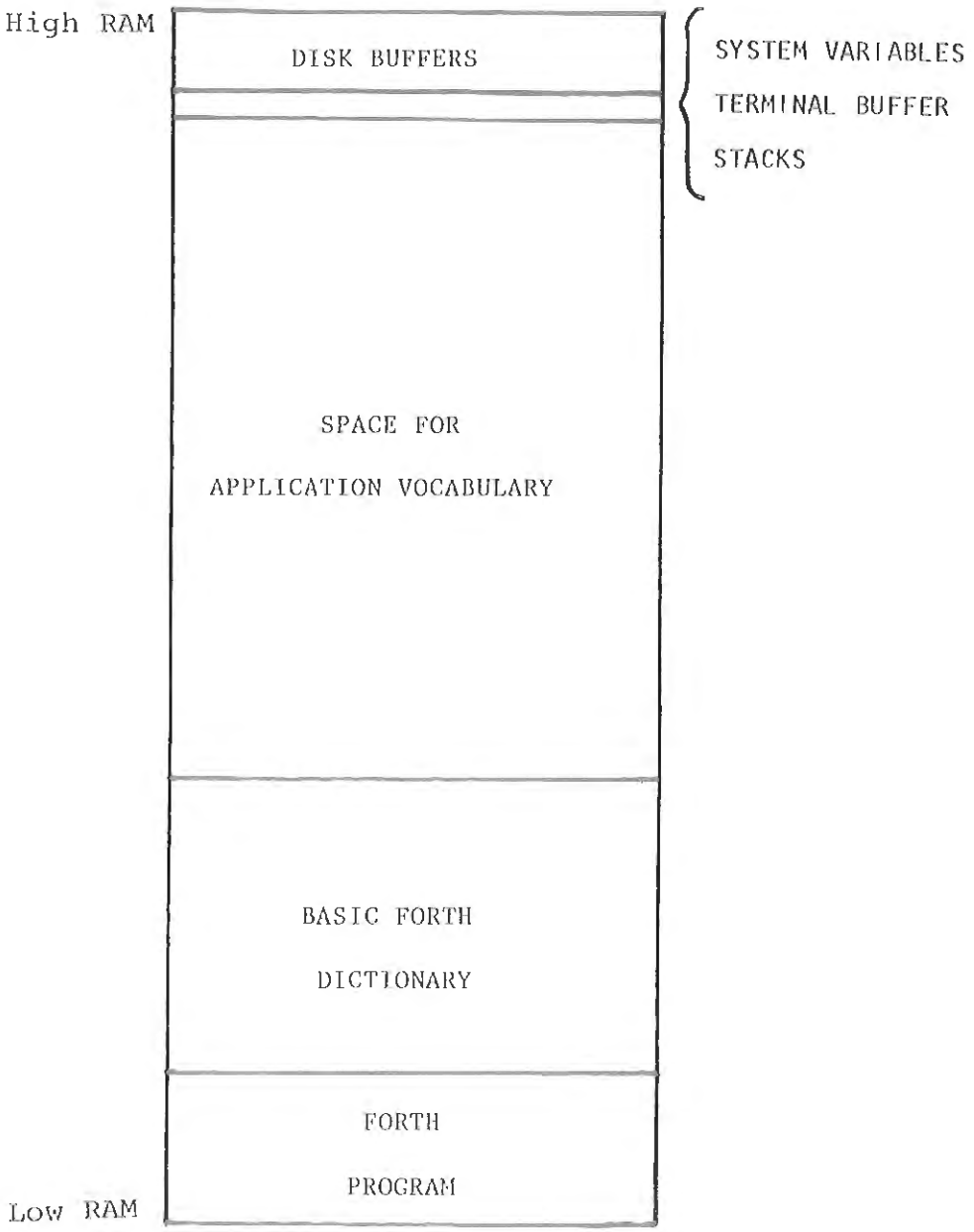
During regular business hours a programmer is available to help you with suggestions about troubleshooting. Since all of our programmers work on-site when necessary, at times there will be a slight delay before someone returns your call. Try, therefore, to place your call as soon as you are sure you have a problem.

Classes are held at FORTH, Inc. whenever the number of potential students warrants one. Each microFORTH class features an overview and then the focus turns to the specific needs of those who attend. If you would like to participate in such a class, use the Hotline to add your name to the request list for the next scheduling.

A new aid, the FORTH, Inc. Newsletter (And So FORTH ...), is being published quarterly. Each customer receives two copies; each issue features articles on programming. Since the content is intended to reflect user concerns, your questions and suggestions will be appreciated.

While you are reading any FORTH, Inc. manual, we hope you will make notes about questions raised but not answered, passages that are not clear, and, especially, any mistakes you may find. We include a "Reader Comment Form" with both the Primer and the Technical Manual to remind you that we need your feedback in order that we may serve all our users well.

Typical microFORTH System



1.0 BASIC OPERATIONS

Before you begin reading this manual, please take the time to read through the Preface so that you understand the editing conventions used in manuals written at FORTH, Inc. and the relationship of the Primer to microFORTH documentation in general.

The easiest way to learn FORTH is to use it. Since FORTH is an interactive language, you can and should experiment with it directly at your terminal. In this introductory manual we will present many examples to illustrate the capabilities of FORTH; we urge you to try these examples yourself at your terminal. There are exercises at the end of most of the chapters to help you learn to use FORTH on your own. Other problems may suggest themselves to you as you progress.

1.1 GETTING STARTED

Physically, your microFORTH system consists of a suitably configured microprocessor development system, a computer terminal, a diskette that contains the microFORTH system, and, in some cases, a microFORTH boot PROM.

Conceptually, any microFORTH system (Figure 1) includes:

the FORTH program, including interpreters, compiler, assembler, and disk operating system;

the basic FORTH dictionary;

variables, buffers, and stacks appropriate for your microprocessor;

and memory available for an application vocabulary to be programmed by the user.

The detailed startup procedure is outlined in the Installation Instructions provided with your microFORTH diskette. Using these instructions, you cause the FORTH system on the diskette to be read into the memory of your development system. You can then use FORTH immediately to solve programming problems without need for any other "monitor" or "operating system."

When your microFORTH system has successfully loaded, it will type out OK and space to a new line. The OK response is returned whenever the FORTH text (or outer) interpreter has successfully completed your last request and is awaiting new input. You type commands at the terminal, concluding them with a carriage return. When the carriage return is typed, FORTH echoes a space to

separate your input from any generated output and then begins interpreting your commands.

Until you have actually typed the carriage return, you may change your commands by using the RUB OUT key to delete any unwanted characters (pressing it once for each character to be deleted) and then retyping the remainder of the line. On CRT terminals the cursor is backspaced once for each RUB OUT. Do not use the key marked BACKSPACE when you want to rub out characters since only the key marked RUB OUT (or DEL for delete) will perform this action.

The simplest command that you can give to FORTH is an empty line. If you type a single carriage return at your terminal, FORTH will respond with a space, inspect the input, see that there is nothing to do, output an OK , space to the next line, and then wait for more input. You should try this to assure yourself that your microFORTH system is alive and listening to you.

1.2 WORDS

The basic command unit of FORTH is called a word. A word consists of a string of characters (letters and/or numbers) that is delimited by spaces. There are no restrictions on the characters that make up a word (except that a word may not have an embedded space, carriage return, or backspace character), nor on the number of characters in a word. This principle is important enough to summarize:

RULE 1: FORTH WORDS ARE MADE UP OF AN ARBITRARY NUMBER OF CHARACTERS, SEPARATED BY SPACES.

After you close a line of text with a carriage return, the FORTH text interpreter scans the input line, breaking it up into words which will be executed in the order of entry. Each word in FORTH has a name (the way you refer to it; its proper spelling) and a definition (the meaning, i.e., the work that is to be accomplished).

To execute a word, the interpreter must determine the word's meaning by searching the dictionary. The interpreter searches for the name of each word in order to locate its definition. If the word is found in the dictionary, then the definition is interpreted. If the word is not found in the dictionary, the interpreter attempts to convert the word to a sixteen-bit, fixed-point integer.

When the interpreter cannot interpret a word (if the word is not found in the dictionary and is not a number modulo the current base), then an error message is given: the unknown word is echoed back to you on the terminal, followed by a question mark. There is no OK or carriage return after an error message.

Words are added to the dictionary by defining a "new" word in terms of currently existing words in the dictionary. This extends the basic FORTH system in the specific direction you want to take. Rather than individual special programs, you will create a growing, more powerful FORTH vocabulary. That means that you will spend more time on your first application than the second. By the time you reach your third application in microFORTH, you'll find that a suitable vocabulary already exists to solve most of your programming problems with very little additional effort.

1.3 NUMBERS

Numbers can be expressed in any base--decimal, octal, and hexadecimal are standard. At any time you can use the commands DECIMAL , OCTAL , or HEX , or you can define another base to establish the appropriate way to treat all succeeding numbers, both for input and output. In general, you should pick one base and stick with it throughout all your definitions to avoid conflicts in interpretation. Initially, FORTH assumes the DECIMAL mode.

Numbers may be typed in as positive (unsigned) or negative (preceded by a minus sign) integers. Positive numbers in the range 0 through 65535 are acceptable because they can be stored in sixteen bits. Signed numbers in the range -32768 to 32767 can be accepted; they are stored in two's complement form in sixteen bits. It is important to note, however, that positive numbers larger than 32767 can be interpreted as two's complement negative numbers, especially if they are involved in arithmetic operations. Numbers larger than sixteen bits will merely be truncated to sixteen bits.

Numbers are entered by typing them at the terminal. As with words, numbers are bounded by spaces. The interpreter will first search the dictionary for the "word" entered. If it is not found in the dictionary, the interpreter tries to convert the "word" to a number. If the number conversion succeeds, it is placed on the parameter stack, which will be described in the next section.

Since all numbers are stored in binary form, you can take advantage of numeric base selection to perform number conversions. To convert a decimal number to hexidecimal, for example, type:

```
DECIMAL 583 HEX .
```

and you will receive the response:

```
247 OK
```

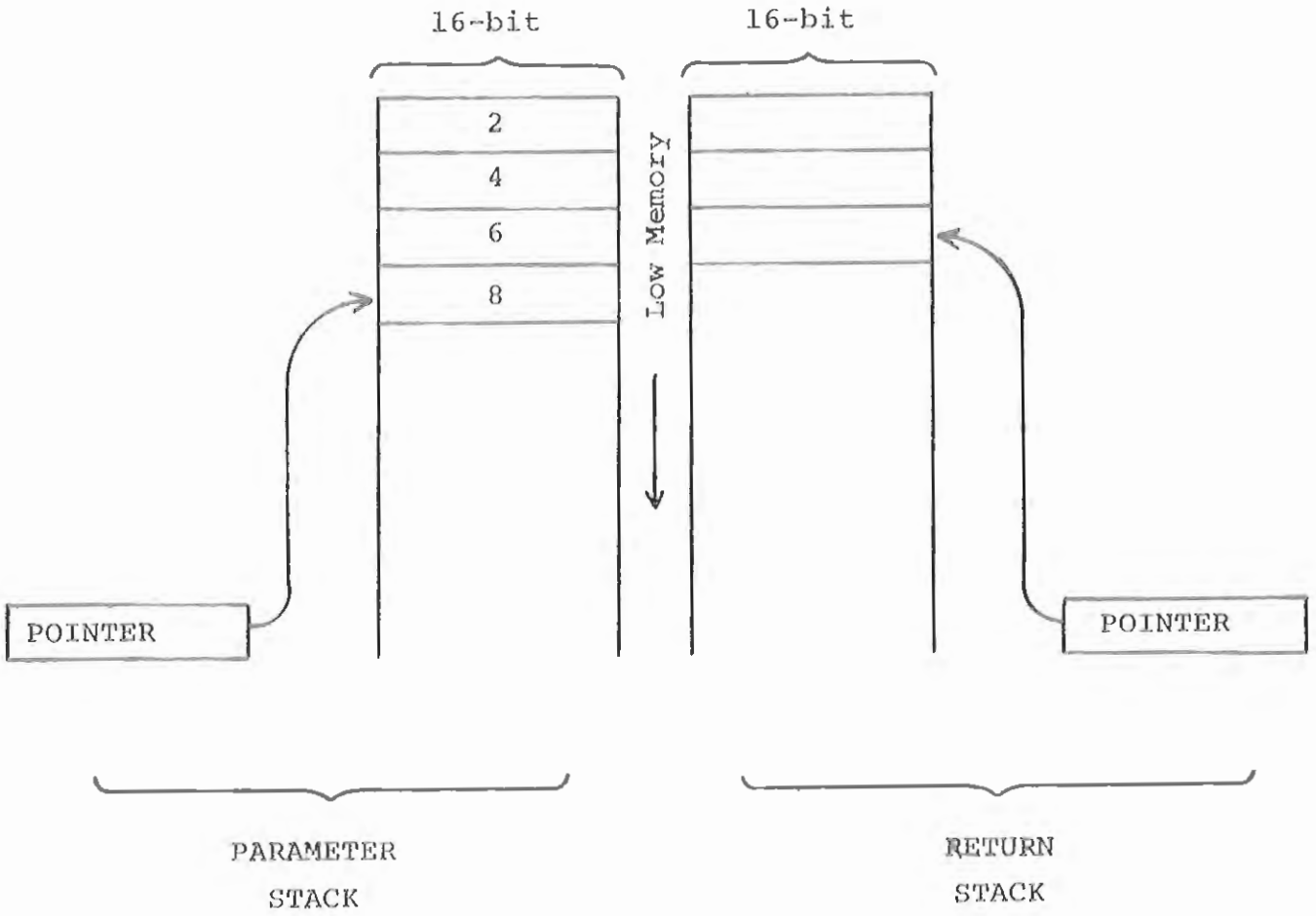
But remember that you stay in HEX mode until you type in the command DECIMAL again.

1.4 THE PARAMETER STACK

All computer programs exist to manipulate data by using an established set of parameters. Most of the parameters that FORTH words use to manipulate data are maintained on a push-down stack, called the parameter stack. This stack, which is sixteen bits wide, is similar to those in pocket calculators that use postfix function keys or Reverse Polish Notation. A push-down stack is a particular arrangement of memory storage; FORTH words that refer to the parameter stack do so by accessing only the topmost items (the ones most recently placed on the stack). A conventional random-access memory (RAM) can be used as a Last-in-First-Out (LIFO) stack, such as those shown in Figure 2.

During the boot procedure a single pointer is initialized to point to a particular location in memory. Once this pointer is initialized, the parameter stack grows toward low memory. When information is to be written onto the stack, the address in the pointer is decremented and the information is then stored at the location being pointed to. When information is to be read from the stack, the information is fetched and then the pointer is incremented. Note that while reading from the stack effectively removes the information forevermore (unlike reading from conventionally organized memory), writing to the stack preserves

FORTH's Two Stacks



all the prior contents of the stack that have not yet been read.

One of the basic rules of FORTH, then, is:

RULE 2: MOST WORDS REQUIRE PARAMETERS ON A PUSH-DOWN STACK.

Actually, FORTH manages two stacks, although we will defer discussion of the other one (always called by its full name, the return stack) until later in the manual.

To place a number on the stack, you can type it as part of your input commands. Now type:

2 4 6 8 (followed by a carriage return)

When you have typed in the example above, you have created a push-down stack that looks like the parameter stack in Figure 2.

One of the simplest kinds of operations FORTH provides for manipulating the stack is one that prints the contents of the topmost item. FORTH's predefined symbol, the period or dot, causes the topmost item of the stack to be removed, converted to a number, and then displayed. If the stack is as shown in Figure 2A, then for each period you type in, the successive topmost item of the stack will be revealed. If you type five periods followed by a carriage return, your CRT screen will look like this:

. . . . . 8 6 4 2 0 . STACK EMPTY!

Each time the dot is encountered, the stack is depleted by one item. Since there were only four items on the stack (put there by the four operations 2 4 6 8), the fifth request for a display from the stack displays garbage (in this case the number 0), and underflows the stack. FORTH won't let you get away with that; it issues a copy of the offending word (in this case, a period), followed by an error message, STACK EMPTY!

When you receive any error message (whether STACK EMPTY! or ? or DICTIONARY FULL!), you must remember that your stack has been emptied. Before you can perform the operation you were attempting, you must reenter the necessary parameters.

RULE 3: ANY ERROR MESSAGE EMPTIES BOTH STACKS.

The dot operator is useful when you are debugging a definition. If you have trouble, you can display the stack, item by item, and compare the contents with what you expected. That will usually isolate the problem. Of course, reading the stack items will cause them to be removed. If you're debugging, after you're satisfied with the displayed values, you may simply type those values back in again:

2 4 6 8

1.5 ARITHMETIC

FORTH has a pre-defined set of arithmetic operators (see Table I). Since FORTH uses a push-down stack and Reverse Polish Notation, parameters must be on the stack before the operation can be performed. Thus, to add two numbers together and display the results, type in:

```
5 27 + . 32 OK
```

Breaking this line down into its constituent parts, you will find that:

5	Pushes the value 5 onto the stack.
27	Pushes the value 27 onto the stack.
+	Removes the top two items from the stack, adds them together, and places the sum back onto the stack. Note that the stack has a net loss of one item.
.	Removes the top item from the stack and displays it: 32 OK

Thus you leave the stack just as it was before you started.

For programmers who have had some experience with algebraic languages (like FORTRAN), FORTH's postfix notation may seem unusual. It will be familiar, however, to most users of pocket calculators and is extremely effective when used properly.

Processing of comparisons may also be unfamiliar. FORTH assumes the conventions of positive logic: one (or non-zero) implies true, zero is false. The FORTH relational words (such as > , < , or =) may be remembered as written between the second stack entry on the left and the top stack on the right. Thus A B < will test for A < B and leave only a truth value on the stack, since both A and B have been removed. The word NOT (or 0=) reverses the truth value of the top item, changing zero to one and all else to zero. Logical branching words (among others) in FORTH depend heavily on these comparison methods and their resulting stack values (see Table II).

1.6 STACK MANIPULATIONS

Other frequently performed operations are classified as stack manipulations, for which FORTH provides a few simple words. These words (described in Table III) are generally used to maintain discipline in the stack when it contains parameters. Experimenting with these words will make them useful to you quickly.

While you are practicing, keep in mind two elementary rules that you must observe with respect to stacks. The most fundamental rule is to maintain "parity" of operations on the stack: everything you put onto the stack must be removed by some operation. If you leave parameters on a stack, it is because you have forgotten some step; this leads to stack overflow. In other words:

RULE 4: ALL PARAMETERS PUT ONTO A STACK MUST BE REMOVED WHEN THEY ARE NO LONGER NEEDED. THE ORDER WILL BE LAST IN, FIRST OUT.

Also remember that you should never remove more items than you have first pushed onto the stack.

After you have become familiar with both the arithmetic operators and the stack manipulators, you will want to create your own combinations. For example, a convenient way to double a number is to add it to itself. This can be accomplished by the sequence:

```
DUP +
```

where the DUP is used to duplicate the number on the stack. Similarly, to square a number you use:

```
DUP *
```

1.7 DEFINITIONS

Part of FORTH's power lies in your ability to define your own new words. Imagine that you frequently want to add two numbers and print the sum; you could always type in + and . for each operation. That, however, could lead to errors. Even in such a simple case it is possible to utilize FORTH's power, since it would be easier to type one word instead of two.

Try defining a new word, named ADD , by entering:

```
: ADD   + . ;
```

Here is what each component in this defining operation does:

:	A colon begins a definition.
ADD	The name of the word to be added to the dictionary follows the colon that starts a definition. For reading ease the name is followed by three spaces.
+ .	The FORTH words define what to do for ADD .
;	A semicolon ends a definition.

After making this definition, you merely type in two numbers and then the word ADD in order to compute and print the sum of the two numbers:

```
1 2 ADD 3 OK
4 5 ADD 9 OK
854 21 ADD 875 OK
```

Since ADD has now been defined to be identical to the executed sequence of + followed by . , you can use either ADD or the sequence to get the desired answer printed out.

What would have happened if you had used ADD before the word was defined? FORTH won't allow that. It prints out the undefined word followed by a question mark. Try this with a different word, such as PLUS ; you will see the error message:

```
1 2 PLUS PLUS ?
```

That points up another fundamental FORTH rule:

RULE 5: ALL WORDS MUST BE DEFINED BEFORE THEY CAN BE USED.

Fortunately, especially for the novice programmer, FORTH has a rich vocabulary of predefined words. Such a word is ? , which prints the contents addressed by the top of the stack; ? has a simple definition:

```
: ?   @ . ;
```

Another simple combination that is predefined in standard microFORTH, 2* , doubles the top stack item. If it were defined in high level, its definition might read:

```
: 2*   DUP + ;
```

Table IV provides easy reference to other fundamental microFORTH operators. The four tables include only the most basic microFORTH words; for a complete glossary (exclusive of a few words specific to particular CPUs), see Appendix B.

By taking advantage of FORTH's ability to define new operations, formulas may be neatly factored, with common components being defined as operator words. Making good use of predefined FORTH words and choosing good names for your new operators can make the resulting definitions both compact and readable.

For instance, the stack manipulation words (DROP , DUP , OVER , SWAP , and ROT) can be used to assemble complex arithmetic calculations. Given the constants A , B , and C , you can define a word named QUADRATIC :

```
: QUADRATIC   DUP A * B + * C + ;
```

to compute the quadratic function $Ax^2 + Bx + C$, where x is the value on top of the stack.

1.8 MODES

The FORTH text interpreter operates in two modes: immediate execution and compilation. In immediate execution mode each word of the input string is looked up in the dictionary and executed. During compilation, on the other hand, most words are not executed; instead a reference to them is compiled into a definition in the dictionary. The word : (colon) places the interpreter in compile mode, whereas ; (semicolon) returns it to immediate execution.

The compiled form of the definition consists of pointers to the addresses of routines that will be executed by the address (or inner) interpreter when the definition is executed. This form of interpretation (address interpretation) is extremely fast.

To distinguish between the modes of immediate execution and compilation, try the following examples:

905 . 905 OK	Executes immediately. Note the interaction.
: SHOW 905 . ; OK	Compiles; nothing happens yet.
SHOW 905 OK	Executes the compiled routine to produce the desired result.

There are vast numbers of variations on these basic themes, as well as whole groups of other words already defined in FORTH. To learn quickly, you MUST practice with the basic FORTH words, the words described in the following chapters, and the words you evolve out of experiments. Develop a kind of notation which will leave you with a sketch of what you have done (to help you avoid making the same mistakes twice).

EXERCISES

1. What is the difference between:

   DUP * DUP * and DUP DUP * *

2. Using only two FORTH words, define a word called 2DUP to duplicate the top pair of stack items. That is, after:

   1 2 3 2DUP

   the stack should contain:

   1 2 3 2 3 (second 3 on top)

3. What is the difference between:

   OVER SWAP and SWAP OVER

Table I. ARITHMETIC OPERATORS

WORD	DESCRIPTION	EXAMPLE OF STACK BEFORE (top ↓)	EXAMPLE OF STACK AFTER (top ↓)
+	Adds.	9 6 2	9 8
-	Subtracts.	9 6 2	9 4
*	Multiplies (unsigned).	9 6 2	9 12
2*	Doubles an entry (unsigned).	9 6 7	9 6 1 4
/	Divides (unsigned).	9 6 2	9 3
ABS	Leaves the absolute value.	9 -6 -2	9 -6 2
MAX	Leaves larger of top two entries.	9 6 2	9 6
MIN	Leaves smaller of top two entries	9 6 2	9 2
MINUS	Performs twos complement (unary minus).	9 6 2	9 6 -2
MOD	Leaves modulus (division remainder).	9 6 2	9 0

Table II. COMPARISON OPERATORS

WORD	DESCRIPTION	EXAMPLE OF STACK BEFORE	EXAMPLE OF STACK AFTER
		top ↓	top ↓
<	Compares; leaves 1 if second entry less than top; otherwise 0.	9 6 2	9 0
>	Compares; leaves 1 if second entry greater than top; otherwise 0.	9 6 2	9 1
NOT or 0=	Tests for zero; leaves 1 if top entry is zero; otherwise 0.	9 6 2	9 6 0
0 <	Tests for negative; leaves 1 if top entry is less than 0; otherwise 0.	9 6 2	9 6 0
=	Tests for number equals; leaves 1 if top entry is zero; otherwise 0.	9 6 2	9 6 0

Table III. STACK MANIPULATION OPERATORS

WORD	DESCRIPTION	EXAMPLE OF STACK BEFORE	EXAMPLE OF STACK AFTER
		top ↓	top ↓
.	Prints the item that is on the top of the stack	1 2 3	1 2
DROP	Discards top entry.	3 2 1	3 2
DUP	Duplicates top entry.	3 2 1	3 2 1 1
-DUP	Duplicates top entry if it is non-zero.	3 2 1	3 2 1 1
		or	
		3 2 0	3 2 0
OVER	Copies second entry over top entry.	3 2 1	3 2 1 2
ROT	Rotates top three entries.	4 3 2 1	4 2 1 3
SWAP	Swaps top two entries.	3 2 1	3 1 2

2.0 EDITING AND PRINTING

When you first brought up your microFORTH system, you inserted a diskette in the disk drive and booted up. Very simply, this boot procedure read a precompiled program from the diskette which in turn read in microFORTH source text from diskette and compiled it into RAM. Compiling, in FORTH, is a process which translates source text into dictionary entries that contain machine code and addresses. Only the machine code and addresses reside in memory; source text remains on diskette.

When you simply type trial definitions at your terminal, they are compiled immediately into your dictionary and your source text is "lost" (i.e., not preserved on diskette). When you reboot your system, any such occasional definition will have been cleared out. (It is this aspect of FORTH that allows you to test definitions in an impromptu manner.) If you should then want to use any of your occasional definitions again, you would have to type them in once more.

It is much more convenient to put your tested source text on the diskette just as the microFORTH system programmers have done. Using the same process that boots the system (see Section 2.1.2), your definitions can be compiled into memory from the diskette rather than from the terminal.

Before discussing how text is entered on diskette, let's consider the structure of the diskette.

2.1 BLOCKS AND SCREENS

Each diskette contains a fixed number of blocks; each block is numbered and contains 128 bytes. Whenever a block is needed, it is requested by its block number. The block number reflects its relative (logical) position on the diskette so that Block 0 is the first block, Block 1 the second, and Block 1999 the last block.

Source text in microFORTH is formatted for the terminal in a screen. A screen is a set of 1024 characters formatted as sixteen lines of sixty-four characters each. Since a character occupies one byte, it takes 1024 bytes to hold an entire screen. A screen of text is therefore held in eight contiguous disk blocks. A diskette may hold 250 screens of source text, numbered from 0 to 249. Screens 0 to 60 contain your basic microFORTH source, including whatever options you may have ordered, plus a small amount of pre-compiled binary information. The rest of the screens are available for your use.

2.1.1 Blocks

To place a specific disk block into computer memory, use the FORTH word BLOCK preceded by the appropriate number. Thus,

```
15 BLOCK
```

places Block 15 in memory. BLOCK also leaves the memory address of the zeroth word of the block on the stack.

If any data is written into the block in memory, the block must be marked so that the revised block contents will replace the old on the diskette. The word UPDATE is used for this purpose. UPDATE marks the most recently used block for writing to diskette, although it may or may not be written to diskette immediately. The word FLUSH is used to force any "updated" blocks out to diskette and should be used before shutting down the system.

These words comprise the basic block I/O routines. To understand them more thoroughly it is necessary to understand the nature of virtual memory as FORTH uses it. The memory address that is returned to you in the routine named BLOCK denotes the location of one block buffer. The block buffers reside at a fixed location in high memory (above the return and parameter stacks) and each contains 128 bytes of data from diskette plus a block ID word (two bytes) with a value between 0 and 1999. When memory permits, your microFORTH system as delivered will have eight block buffers. This number gives a good trade off between memory usage and disk accessing.

The word BLOCK first searches the block buffer IDs to see if the block currently resides in the block buffers. If it does, no diskette access is made. Otherwise, the block is read from diskette. UPDATE sets the high-order bit of the high-order byte of the Block ID to one if the block is to be written to diskette. A block is written to diskette if its buffer is needed by another BLOCK request or if FLUSH is specified.

2.1.2 Screens

When source text is to be interpreted, you will use the word LOAD . Thus,

```
150 LOAD
```

interprets the eight blocks that make up Screen 150 as if you had typed all the text this screen contains at the terminal. The immediate implication is that a screen can contain both definitions and any executable commands. Any definitions will be compiled; any other commands will be executed.

Line 0 of an empty screen on a microFORTH diskette begins with two null characters to prevent the PRINTING utility from listing them. For this reason, Line 0 must always be replaced before attempting to LOAD or SHOW .

To display the contents of a screen, use the word LIST . For example,

```
120 LIST
```

formats Screen 120 into sixteen lines of sixty-four characters each and displays them on the terminal. LIST also remembers the current screen so that once you have listed a screen, you may re-list it by simply typing L . L looks at the contents of the user variable SCR , which contains the screen number of the current screen. LIST is available at all times on the system.

2.2 THE EDITOR

The conventions for the EDITOR are identical to the conventions for the rest of FORTH. On most microFORTH systems the EDITOR is resident. On these, to gain access to the EDITOR, simply type:

```
EDITOR
```

On the RCA COSMAC, however, the EDITOR is not resident and must be loaded by typing the command:

```
EDIT LOAD
```

On all microFORTH systems, you retain access to the EDITOR until you compile a definition. After compiling a definition, you need to invoke the EDITOR vocabulary again if you are to use it. On the COSMAC only, if you have loaded an application that replaced the EDITOR, you must reload it (see Section 2.5, "Overlays").

Within the EDITOR you can call on two types of editing words: those that operate on whole lines and those that work on characters. Screen 14 contains the former and Screen 21 the latter.

2.2.1 Editing by Line

In Line 0 of each screen that contains source text, it is a FORTH convention to place a parenthetical comment that briefly describes the contents of that screen. Comments are written in FORTH in the following form:

```
( COMMENT)
```

The word (starts a comment; one or more spaces must follow the left parenthesis because (is a word. The succeeding characters are ignored by FORTH until the next right parenthesis. The PRINTING utility (described in Section 2.4) can produce a convenient disk index for you by displaying the first line of each screen, preceded by its screen number.

After listing a screen into which source text is to be entered and gaining access to the EDITOR, enter a line of text in the current screen by typing:

```
0 P ( PRACTICE SCREEN)
12 P THIS IS OLD LINE 12
```

Here zero is the first line number to be entered and P (for Put) is an EDITOR command which puts the following line of text (up to the carriage return) in the designated line. In this example, you have put a comment line at the head of a screen and the text, "THIS IS OLD LINE 12," in Line 12. If you enter fewer than sixty-four characters of text for any one line number, the remainder of the line is blanked. If more, your text will be truncated to sixty-four characters.

Note that because P is a FORTH word, you must space once before you type the text. Any additional spaces, however, are included in the text to be inserted. If you had spaced four times after the first P in the example above, Line 0 would have begun with three blank spaces before the left parenthesis.

When using P it is important to enter at least one character of actual text. For instance, to blank a line you must type at least two spaces after the P , one as a delimiter and the other as text. This rule also applies to the EDITOR word A .

The EDITOR provides capabilities to display individual lines as well as to replace, insert, or delete lines. To implement these features, the EDITOR uses a sixty-four-character buffer, called PAD. The P command, for example, puts the succeeding text not only in the line whose number is on the stack, but also in PAD. (Note that the next text written to PAD overwrites, i.e., destroys, this text.)

The EDITOR word T is used to Type out (display) the line whose number is on the stack. Specifically, T does the following things:

1. Transfers the line whose number is on the stack to PAD.

2. At the beginning of the next line on your terminal, indents two spaces and types the line now in PAD. (The OK appears after the sixty-fourth character.)

3. Leaves the line number on the stack.

T is specifically designed to be used in conjunction with P in modifying a previously entered line. Note that the line number is left on the stack, so it need not be re-entered. Also, the typed line is indented two spaces, leaving room for a P and a space on the next line on the terminal so that you can easily copy down the text in parallel, making corrections. For example, try this (remember that the computer's response is underlined):

```
12 T
  THIS IS OLD LINE 12                                    OK
P THIS IS NEW LINE 12 OK
12 T
  THIS IS NEW LINE 12                                    OK
```

The EDITOR word R will Replace the line whose number is on the stack by the contents of PAD. For example, to make Lines 12 and 13 identical, you type:

```
12 T 13 R
```

If you want to blank out several lines, you can use a combination of EDITOR

words P and R . After such a command as 1 P followed by two spaces and a carriage return, you simply declare the other lines that you want to be Replaced:

```
1 P  OK
2 R  12 R  OK
```

converts Lines 1, 2, and 12 to spaces.

The EDITOR word D is used to Delete the line whose number is on the stack. The deleted line is first moved to PAD so that it is not actually lost. The succeeding lines in the screen are moved up one and renumbered. Line 15 remains the same and is not blanked. For example, 13 D will move Line 13 to PAD, move Line 14 to Line 13, and duplicate Line 15 as Line 14. Never use D on Line 15! If you want to blank Line 15, use R or P ; if you want to copy it elsewhere, use T and then I or R .

The line in PAD can be Inserted by the EDITOR word I . The contents of PAD are inserted in the next line after the line whose number is on the stack. Succeeding lines are moved down; Line 15 is lost. Never try to insert after Line 15; to insert text as Line 0, use -1 I .

D and I can be used to move lines of text from one place to another. To exchange Lines 4 and 5, for instance, all you need to do is type:

```
4 D     Puts Line 4 in text buffer and moves succeeding lines up
        one.

4 I     Inserts it after the new Line 4 and moves succeeding lines
        down one.
```

This example was purposely used to illustrate an important point about D and I . A line number refers to the current position of a line within a screen. Because D and I move lines beyond the point of insertion or deletion, they effectively renumber all subsequent lines. While you could get in the habit of always relisting the current screen after each and every use of D or I , this is tedious if your terminal is, say, a teletype. You should acquire the habit of always visualizing the effect of these words as you type them.

The word A (for Add) combines some of the properties of I and P . A expects a line number on the stack, after which a line of text will be inserted. The line of text to be inserted is not yet in PAD, however; instead, you must enter it after the A and a space. As with P there must be at least one character of actual text; as with I , all succeeding lines are moved down and Line 15 is lost. The text is terminated by a carriage return.

To summarize, the vocabulary for editing lines includes the following words:

n P text	Puts succeeding text (until carriage return) in PAD and Line n.
n T	Copies Line n into PAD, types it, and leaves n on the stack.
n R	Replaces Line n by the contents of PAD.
n D	Copies Line n into PAD and deletes it from the current screen. (Avoid 15 D .)
n I	Inserts the contents of PAD after Line n. (Avoid 15 I .)
n A text	Puts succeeding text (until carriage return) in PAD and inserts it after Line n. (Avoid 15 A .)

A little practice with these commands will make them your tools.

While you are editing, some portions of the screen you are working with are in main memory and some of them may be back out on the diskette. To be certain that all your changes get out on the diskette, you need to type:

FLUSH

to force writing the buffers to disk. This is most important before executing a new definition, removing a diskette, or shutting down your system.

2.2.2 Editing by Character

In the EDITOR vocabulary there are also commands for editing characters. The character editing commands operate on strings within a selected line. A character pointer reminds you of your position on a specific line.

The command T positions the character pointer to the beginning of a line. Thus 1 T prints Line 1 of your screen. The command TOP will position the character pointer at the beginning of the screen (e.g., the first character of Line 0). The character position in the screen is controlled by the user variable R# . It is used to compute the line number and to control where searches begin. The following commands are available. (Remember these are FORTH words and therefore they must be separated by a space from any text following them.)

F text	Finds the next instance of "text" after the current pointer position. Prints the line on which it occurs and positions the character pointer at the character after "text."
N	Finds the next occurrence of "text" (used after F); prints the appropriate line; and positions the character pointer at the character that immediately follows the text.
C text	Inserts "text" at the current pointer position in the current line.

X text — Finds and deletes "text" from the current screen. The search for "text" begins at the current pointer position.

TILL text — Deletes all characters through and including "text," beginning at the current pointer position. TILL only operates within the current line.

As stated earlier, most character editing commands operate within the line pointed to by R# . To compute the line number, divide R# by sixty-four (the number of characters in a line).

The commands F , X , and N are not limited to a single line; multiple lines are searched, beginning at the current pointer position. However, the text may not cross a line boundary, i.e., with portions on two different lines. Searches are performed on a line-by-line basis.

C also operates within the line pointed to by R# . Therefore all characters are lost which are moved beyond the line limit of sixty-four characters. Lastly, any error message will reposition the pointer at the top of the screen.

The following is an example of how the character editing words operate. In this example the computer's output will not be underlined. Instead the underscore here represents the one you see on your screen while you are editing; it indicates the pointer to the current character. Another convention adopted only for this example is the use of the symbol cr to show the point at which the carriage return button was pressed.

Given:

```
0 THIS  IS  SOME  TEXT
1 THIS  IS  LINE  10
2  THIS  IS  NOT  LINE  2
```

```
TOP X TEXTcr
THIS IS SOME _                          0

C TEST!cr
THIS  IS  SOME  TEST!_                  0

X 10cr
THIS  IS  LINE  _                       1

F THIS  IScr
THIS  IS_   NOT  LINE  2                2

X   NOTcr
THIS  IS  _LINE  2                      2
```

2.2.3 COPYing

Besides these words which you use to work on individual lines, the EDITOR includes another word that allows you to copy one entire screen to another:

from to COPY

where from and to are screen numbers. COPY is useful when you want to modify a program slightly without destroying the original. Another use of COPY is to delete a screen by COPYing a blank screen on top of it.

A word of caution: it is best to use the command FLUSH both before and after each use of COPY . The word COPY only changes the block ID in the block buffer and marks the block for writing. Thus for systems with eight or more block buffers:

```
120 121 COPY  121 122 COPY
(followed by a carriage return)

FLUSH
(followed by a carriage return)
```

copies Screen 120 to 122 only, leaving Screen 121 unchanged. In other words, Screen 120 was read into the block buffers; its IDs were changed to match Screen 121 (since no FLUSH was specified, the change was not written to the diskette). On the second COPY , Screen 121 already existed in memory so again no writing occurred. Only the block IDs changed to match 122. FLUSH forced only the new Screen 122 out to diskette.

2.3 LOADING MULTIPLE BLOCKS

When you have a good application program stored away on diskette, you can begin your testing and debugging phase. If your source text is in Screen 120, you compile it with:

```
120 LOAD
```

When you have multiple screens in your program, you will want to use one screen to load the whole program at once. For example, you might edit into Screen 102 the commands:

```
120 LOAD  121 LOAD  122 LOAD
```

Then, to load in the entire application, you type 102 LOAD .

2.4 THE PRINTING UTILITY

The PRINTING utility (not to be confused with PRINTER) allows you to display source text and generate an index to your listings. The basic format unit of three screens fits the usual 8 1/2 by 11" printed page (of sixty or so lines). To obtain access to PRINTING, type:

```
PRINTING LOAD
```

Three important words that are loaded by this command are TRIAD , SHOW , and INDEX .

TRIAD generates a list of three screens on a single page. The screen number of the top screen in a triad is always evenly divisible by three. TRIAD types the triad of screens that includes the screen number you put on the stack. Thus typing 3 TRIAD , 4 TRIAD , or 5 TRIAD produces the same output.

The phrase:

n m SHOW

generates all triads necessary to show the range of screens from n through m inclusive.

The phrase:

n m INDEX

produces an index of the screens in the range of n through m-1. The index consists of the first lines of the requested screens, preceded by the screen number. The index is formatted sixty screens to a page.

In the PRINTING utility, Line 14 of Screen 23 is printed at the bottom of each page of output. You may wish to edit appropriate information into this line, such as your company's name and/or the date.

If you use a TTY, can use the microFORTH PRINTER screen on some other printing device, or have written a printer driver, you can direct the listings to the printer by typing:

PRINTER LOAD

before PRINTING LOAD . The PRINTER screens re-define the input and output words to access a hard-copy device.

2.5 OVERLAYS

On your development system, you may define several application vocabularies which you may not wish to be compiled all at the same time (indeed, they may not all fit). An overlay is an application vocabulary which, when loaded, automatically replaces the previous application.

Overlays are implemented as follows. The last definition in the resident FORTH system is the null definition:

: TASK ;

All user-defined words will be loaded after TASK . If a load screen begins with the phrase:

FORGET TASK : TASK ;

then all dictionary entries from TASK on are "forgotten" from the dictionary. The null definition : TASK ; is then put back in the dictionary to mark the beginning of the overlay. Conventionally the phrase:

FORGET TASK : TASK ;

appears in Line 0 of the load screen (after the descriptive comment), so that it will identify this screen as the load screen of an overlay in an index. See the PRINTING utility (Screen 27) for an example of such an overlay.

On any system, the use of overlays insures that the definitions of each application are made in terms of the initial, standard set of microFORTH words rather than in terms of any new meaning that another vocabulary may have given to a particular word. This will happen automatically if you adhere to the convention of using FORGET TASK : TASK ; at the beginning of each application vocabulary.

On RCA COSMAC systems, due to the limited amount of memory available, this is especially important; the EDITOR here is an overlay that needs to be replaced by your application vocabulary.

EXERCISES

1. Enter some text, such as a series of punch lines to jokes or names of friends, into each line of your practice screen.

2. Exercise until you can quickly:

   a. Exchange Lines 13 and 14 (with no duplication of text).

   b. Return your screen to its previous state.

   c. Exchange Lines 15 and 13.

   d. Exchange Lines 15 and 0. Repeat.

   e. Blank Line 15.

   f. Replace Line 5 with brand new text, two different ways.

3. Blank Lines 1 through 15 of your practice screen.

   Add the conventional phrase to the first line that will make this screen a sample overlay.

   Using such FORTH words as the arithmetic operators (Table I), the stack manipulators (Table II), and others from Appendix A, create at least three new words.

   Enter your definitions of 2DUP and the new words into the practice screen.

   Review Table VI, "Editing Conventions," and check the screen.

   Load it.

   Test and debug it.

Table IV. EDITING COMMANDS

n A text	Puts succeeding text (until carriage return) in PAD and inserts it after Line n. (Avoid 15 A .)
C text	Inserts "text" at the current pointer position in the current line.
n D	Copies Line n into PAD and deletes it from the current screen. (Avoid 15 D .)
F text	Finds the next instance of "text" in the screen after the current pointer position. Prints the line on which it occurs and positions the character pointer to the character after "text."
n I	Inserts the contents of PAD after Line n. (Avoid 15 I .)
N	Finds the next occurrence of "text" (used after F); prints the appropriate line; and positions the character pointer to the character that immediately follows the text.
n P text	Puts succeeding text (until carriage return) in PAD and Line n.
n R	Replaces Line n by the contents of PAD.
n T	Copies Line n into PAD, types it, and leaves n on the stack.
TILL text	Deletes all characters through and including "text," beginning at the current pointer position. TILL only operates within the current line.
X text	Finds and deletes "text" from the current screen. The search for "text" begins at the current pointer position.

TABLE V. EDITING CONVENTIONS

At FORTH, Inc. we have evolved conventions for editing screens to make source text more readable. While these conventions are not dictated by the nature of FORTH, we recommend them as good programming practice.

1. Line 0 of each screen begins with a parenthetical comment that describes the contents of the screen. The comment identifies the screen and is conveniently listed by the PRINTING utility's INDEX .

2. A single screen contains source text for words related to some one function or isolatable portion of a function. Do not put unrelated words in the same screen.

3. Do not overpack a screen. Leave several blank lines for expansion. There is no advantage to conserving screens.

4. Do not define more than one word on a line. An exception might be two or three related constants or variables, or a couple of very brief related colon definitions.

5. Leave three spaces after the name being defined in a colon definition, to set it off from the definition.

6. Break colon definitions up into phrases, separated by double spaces, so that each phrase describes a particular operation:

```
: DOUBLE   X @  2*  X ! ;
```

7. If a definition takes more than one line, indent three or more spaces on the second and succeeding lines.

8. Separate instructions in CODE definitions with three spaces. For example:

```
CODE KEY   BEGIN   F7 INP   2 # ANA   0= NOT END
   F6 INP   A L MOV   0 H MVI   HPUSH JMP
```

The definitions and screens in Chapter 5, as well as the listings that accompany your microFORTH system, provide good examples of well-organized FORTH screens.

3.0 DATA DECLARATIONS

Frequently it is useful to set aside cells in memory to reserve constants, variables, and arrays. There are a series of words in FORTH that allow you to allocate these types of data structures. This chapter discusses the appropriate commands.

3.1 CONSTANTS

If a value is used frequently or if a value is associated with a specific function, you might want to name it. Often the name is easier to recall and enter correctly than is the number itself. A named value is a constant and CONSTANT is the FORTH word used to assign dictionary names to constants. For example, if you are converting miles to feet, you can define a CONSTANT named FT/MILE . Thus,

```
5280 CONSTANT FT/MILE
```

creates the new word FT/MILE and assignes it the value 5280.

After FT/MILE has been defined, you can use it just as you would 5280 to place a value on the stack. That is, if you type FT/MILE , the value 5280 will be placed on the stack. The phrase:

```
FT/MILE 3 *
```

computes the number of feet in three miles. Once a value is defined as a CONSTANT, its binary value is independent of the current number base.

3.2 SIXTEEN-BIT VARIABLES

A value which changes frequently is called a variable. The FORTH word VARIABLE names a location whose value is likely to change. For instance, you might want a variable to keep track of the number of customers who have walked into a new store. You can do that with a statement like:

```
0 VARIABLE PATRONS
```

which means "define a variable named PATRONS with the initial value 0."

When you invoke a constant by its name, its value is placed on the stack. Invoking a variable, on the other hand, places its address on the stack. After placing the address of PATRONS on the stack, you will sometimes wish to obtain the contents of PATRONS . The FORTH word @ (called "at" or "fetch") replaces the address on the stack by the contents of the two bytes at that address. To get the current number of customers into the top of the stack

for processing, you write:

```
PATRONS @
```

Sometimes you need to examine the contents of a variable. The FORTH word ? (question mark) puts the current value on the stack and shows it:

```
PATRONS ? 0 OK
```

The word ! (called "store") is used to store a sixteen-bit value into a location. ! uses the value, which is the second item on the stack, and an address to store into, which is on top of the stack, to alter the contents of a VARIABLE . If you write,

```
5 PATRONS !
```

FORTH will store the value 5 into the VARIABLE named PATRONS .

To put whatever value is currently on the top of the stack into a specified location, then, you merely need to specify the address (by name) and invoke the ! operator:

```
(variable-name) !
```

The FORTH word +! (called "plus-store") adds a new value to a variable:

```
101 PATRONS +!
```

In the same manner as ! , +! increments the contents of the item whose address is on top of the stack by the second item on the stack. After the operations above, the new value of PATRONS will be 106.

To copy data from one place in memory to another, say from the variable OLDPATRONS to PATRONS , you can use the sequence:

```
OLDPATRONS @ PATRONS !
```

Set up your own variables and use these operators (@ ? ! and +!) until you understand how each works.

One of the surprising aspects of FORTH programming is how few CONSTANTs or VARIABLEs are needed. Since the parameter stack is used to hold values which need not be named or need not take up dictionary space, the need to define every literal or temporary value as a CONSTANT or VARIABLE is eliminated. Only fundamental parameters in an application will need dictionary space.

The most common failing of inexperienced FORTH programmers is excessive use of constants and variables. To realize the most value from your microFORTH system, try to be alert to this tendency and resist it.

3.3 BYTE VARIABLES

Just as @ and ! transfer data in sixteen-bit units between the stack and memory, the FORTH words C@ (byte fetch) and C! (byte store) transfer data in eight-bit bytes. Eight-bit numbers occupy the low-order half of a stack entry. C@ fetches a single byte from the location specified by the top of the stack and puts a zero into the high-order byte (to fill out the stack entry). C!

takes the low-order byte of the second stack item and stores it into the byte addressed by the top item, deleting both items from the stack.

As one might expect on microprocessor systems, the byte fetch and store operations are both faster and more conservative of memory than their sixteen-bit counterparts. This makes a noteworthy difference between FORTH and other high-level languages. FORTH does not discriminate between data types by context but rather by the operators that are used to manipulate the data. Thus a sixteen-bit named variable could contain either two characters of a word, or two eight-bit binary numbers (such as a byte vector), or a sixteen-bit binary number. Its usage depends upon the operators that you choose to manipulate its data. This method produces more readable definitions, more efficient execution, and more flexible programming.

Byte variables can be declared in a manner similar to their sixteen-bit counterparts by using the FORTH word CVARIABLE . Just as VARIABLE does, CVARIABLE needs an initial value on the stack, followed by the name being defined. The space used, however, is only one byte wide, which limits you to numbers in the range 0 to 255. Often this range is more than you need. If you wanted to keep track of the current channel number to which a particular TV set was tuned, you could use:

```
n CVARIABLE CHANNEL
```

where n represents the initial channel number. (The number of TV channels would never exceed 255.)

After a CVARIABLE is defined the operators C@ and C! may be used on them. Mixing C@ C! @ ! and +! between definitions is perfectly legal. Be sure, however, that you understand exactly what result you intend to achieve.

3.4 ARRAYS

Arrays of data items are important in many applications. For example, instead of handling a set of ten different temperature readings as T0, T1, ..., T9, it would be better to use ten successive data elements named TEMP . Through suitable addressing arithmetic you can compute the requisite element's address. This is more flexible to program as well as more economical of dictionary space.

An array is established by setting aside space in the dictionary. This is done by using the FORTH variable H , which points to the next available byte in the dictionary space in memory. By incrementing H you can skip over a specified number of bytes, thus creating space for the appropriate number of elements. In the case of a set of temperatures, you simply write:

```
0 VARIABLE TEMP  18 H +!
```

where:

0 VARIABLE TEMP	Defines a VARIABLE (two bytes wide) named TEMP and initializes that space to zero.
18	Puts 18 on the stack.
H	Fetches the address of H ; its contents point to the next byte after TEMP .
+!	Increments the contents of H by 18, thus preserving an additional nine byte-pairs for the other nine temperature readings.

By using the commands HERE and . (dot), you place the current dictionary pointer on the stack (HERE) and printed (by the dot). If you use these commands before and after the definition of your array, you can verify that H is actually incremented. That is:

```
0 VARIABLE TEMP  HERE . 18 H +! HERE .
```

After the dictionary entry is made, H is printed; after H is updated, H is printed again.

To access the nth temperature, then, place n on top of the stack and follow it with:

```
2* TEMP + @
```

The word 2* is used to convert the count in byte-pairs to bytes. This offset is then added to the address provided by TEMP ; finally, the value at that address is fetched by @ . Because the elements are numbered from zero, for ten temperatures the count must be in the range zero through nine; other values will give unpredictable numeric results.

To initialize the nth temperature, type:

```
(value) n 2* TEMP + !
```

! places the value entered into the nth entry in the array.

3.5 OTHER MEMORY OPERATIONS

There are two words which can be used to manipulate memory locations. They are MOVE and ERASE .

ERASE is used to zero a region of memory:

```
address length ERASE
```

zeroes the region that begins at the address specified, for the specified length given in bytes.

MOVE is used to transfer a region of memory to another location:

```
source-address destination-address #bytes MOVE
```

moves the number of bytes specified, beginning at the source address, to the

destination address. The contents of the destination region are overwritten; the source region remains the same.

Because they are destructive writes into memory (where the dictionary resides), these words must be used extremely carefully. When a region of memory is specifically reserved, however, as with arrays or block buffers, ERASE and MOVE can be used to initialize arrays or to copy arrays from one place to another.

In the example used above,

```
TEMP   20   ERASE
```

clears the temperature arrays.

Defining a second array:

```
0 VARIABLE 2TEMP   18   II   +!
```

and using MOVE :

```
TEMP   2TEMP   20   MOVE
```

would copy the array in TEMP into 2TEMP.

EXERCISES

1. Define EXCHANGE to exchange the contents of two variables. That is, if A and B are variables, then the result of the command A B EXCHANGE should be to place the value of A in B and the value of B in A .

2. Define TRANSFER to move data between two arrays of the same length. (Define CONSTANT to specify a length.)

3. Using the arrays defined above, clear the first array and TRANSFER the initialized array to the second array.

Table VI. MEMORY OPERATORS

WORD	DESCRIPTION	EXAMPLE OF STACK BEFORE (top ↓)	EXAMPLE OF STACK AFTER (top ↓)
@	Fetches the contents of the item whose address is on the top of the stack.	2000	257
!	Stores the second item on the stack into the location whose address is on top of the stack.	3 2000	empty
?	Fetches, prints the contents of the location whose address is the top stack item.	2000	empty
+!	Increments the location whose address is on top of the stack by the second item on the stack.	101 2000	empty
C@	Fetches a byte whose address is on top of the stack. The byte is right-justified on the stack.	2000	1
C!	Stores a byte into the location whose address is on top of the stack. Only the right-most byte is stored.	254 2000	empty
H	Points to the next available byte in the dictionary. (H is a variable.)	empty	empty
ERASE	Zeroes memory at the address given for the number of bytes specified. Use: adr. count ERASE	2000 6	empty
HERE	Places the address of the next available byte in the dictionary on the stack.	empty	3576
MOVE	Moves bytes from one location to another: source-adr. dest.-adr. MOVE	2000 2006 6	empty

4.0 CONDITIONAL BRANCHES AND LOOPS

Many definitions execute words (made up of other definitions), one right after another. However, it is often necessary to alter the order in which words are executed in a definition so that, for example, words may be replaced without being re-typed. This is accomplished by using the FORTH structures for loops and conditionals. Loops cause a sequence of words to be repeated a specified number of times; conditional structures allow the application to choose a sequence of words based upon a given test (or condition). The following words compile logic which alters the execution of words within a definition:

IF ... ELSE ... THEN

DO ... LOOP or DO ... +LOOP

BEGIN ... END

IMPORTANT: NO loop or conditional structure can be executed directly from the terminal without being included inside a definition. The control words listed above are designed to compile appropriate logic control and thus are meaningless if used outside a definition.

Remember this rule:

RULE 6: COMPILING WORDS MUST NEVER BE USED OUTSIDE A DEFINITION.

This chapter contains discussions of various conditional structures, loops, and related matters; each discussion is capped by appropriate exercises.

4.1 CONDITIONAL BRANCHES

Three compiling words, IF , ELSE , and THEN are used to compile conditional branches in a definition. In FORTH, conditional branches examine the top of the stack to decide which branch will be taken. A conditional branch has the following structure:

: DEFINITION condition IF this ELSE that THEN continue ;

where:

```
: DEFINITION
     condition      Places a condition (non-zero/zero) on the stack.
     IF             Removes and tests the number on the stack.
          this      Executes "this" if the number was non-zero
                       (true).
     ELSE
          that      Executes "that" if the number was zero (false).
     THEN
     continue ;     Continues from both lines.
```

IF marks the place where the top of the stack is popped and examined; if the value is non-zero, everything up to ELSE is executed--at ELSE , execution skips to THEN . On the other hand, if the stack value is zero everything up to ELSE is bypassed and everything after ELSE up to THEN is executed.

RULE 7: EVERY IF MUST BE FOLLOWED BY A THEN .

For example, you could print non-zero numbers if you defined .POSITIVE in this manner:

```
: .POSITIVE
     DUP            Duplicates the number.
     IF             Tests and discards the top number.
          .         Prints it if it is positive.
     ELSE DROP      Otherwise drops it.
     THEN ;
```

It was necessary to DUP the number on the stack prior to the test because IF removes the number it tests.

The ELSE clause is optional. For example, to increment the top of the stack only if it is non-zero, you can define INC :

```
: INC
     DUP            Duplicates the number to save it.
     IF             Tests it.
          1+        Increments it if it is non-zero.
     THEN ;
```

The truth value on the stack is often the result of a comparison that uses one of the FORTH words, < , > , or = . These operators test the top two stack items for the relations "less than," "greater than," or "equal to," respectively. They remove the top two items which they test, leaving one for true and zero for false. (Review Table II.)

< Leaves one if the second item is less than the top item.

= Leaves one if the second item equals the top item.

> Leaves one if the second item is greater than the top item.

The comparison operators observe the same FORTH convention of Reverse Polish Notation that arithmetic operators do. All FORTH operators retain their conventional meaning. Thus,

```
9 5 <     is the same as     9 < 5     i.e., false
4 2 >     is the same as     4 > 2     i.e., true.
```

For example, suppose an input data item (placed on the stack by INPUT) is to be regarded as either a decimal digit (if it is $\leq$ 9) or as a code for an analog function (if it is $\geq$ 10). Then the definition:

```
: DECIDE   INPUT  DUP 9 > IF  PERFORM ELSE
     DIGIT !  THEN ;
```

either PERFORMs the analog function if the INPUT is greater than nine or else saves the value in the variable DIGIT .

Two additional comparison operators are 0< and 0= , which may be defined:

```
: 0<   0 < ;
: 0=   0 = ;
```

to test for negative or equality to zero, respectively. (Actually 0< and 0= are defined in machine code while < , > , and = are defined in terms of them.) These operators also replace a single argument by a truth value.

To negate a condition, use the word NOT , which replaces zero by one, and replaces any non-zero value by zero. Because this is identical to the action of 0= , the definition of NOT is just:

```
: NOT   0= ;
```

In the following example we include a specific test for zero before storing a data item:

```
: ITEMS   DUP IF  DATA !  ELSE DROP  THEN ;
```

where:

```
: ITEMS                  (The upper limit is on the top of the stack
                            at entry.)
     DUP IF              Tests for the non-zero upper limit.
          DATA !         Stores the value into the variable DATA
                            if it is non-zero.
     ELSE DROP           Discards any unwanted zero.
     THEN ;
```

Since it is common to ignore zero values during an operation, microFORTH provides the word -DUP . It duplicates the top of the stack only if it is non-zero. Using -DUP , the definition of ITEMS becomes shorter:

```
: ITEMS   -DUP IF  DATA !  THEN ;
```

In other words, -DUP eliminates the use of the phrase ELSE DROP . The definition of -DUP , by the way, is just:

```
: -DUP   DUP IF  DUP  THEN ;
```

4.2 COMBINING TRUTH CONDITIONS

Sometimes it is useful to combine several truth values. For instance, you may want to execute statement X only if both parameters on the stack are non-zero. Although this is a trivial example, it serves to demonstrate that two conditions can be met in one definition. The logical operators found in Table II are used in combining truth conditions.

The word AND performs a logical "and" of the top two stack items (bit by bit). This can be used to define compound conditions. For example, if FROM and TO are constants, then you can define BETWEEN :

```
: BETWEEN
    FROM OVER <       Compares the number with FROM .
    SWAP              Swaps the truth value with the number to
                        be tested.
    TO <              Compares the number with TO .
    AND ;             Takes the "and" of the two truth values.
```

BETWEEN determines if the top stack item is between FROM and TO , exclusive.

Because the logical "or" function can usually be handled by addition, no special FORTH word is supplied for this, although users, of course, can write their own.

Truth values are really no different from numbers and may be used arithmetically. Consider this example, which computes (the characteristic of) the base-ten logarithm of a number:

```
: LOG
    DUP
    9 >               Leaves one if n > 10.
    OVER 99 >         Leaves one if n > 100.
    +                 Adds the running sum of the truth values.
    OVER 999 >        Leaves one if n > 1000.
    +                 Adds to the running sum.
    SWAP 9999 >       Leaves one if n > 10000.
    + ;               Adds to the running sum.
```

EXERCISE

1. Given the constants FROM and TO , define a word named OUTSIDE that will leave true on the stack if the top item does not fall between FROM and TO .

4.3 INDEFINITE LOOPS

All loops are governed by the values on the stack. Here is the structure for an indefinite loop (words in lower case represent your application's named and tested definitions):

```
: EXAMPLE   BEGIN  process  condition  END  continue ;
```

where:

: EXAMPLE	Creates a dictionary entry for the new word, EXAMPLE .
BEGIN	Marks the beginning of an indefinite loop.
process	Defines the action(s) to be executed one or more times.
condition	Leaves a truth value on the stack, either zero for false or non-zero for true.
END	Pops the value off the stack, returning to BEGIN if the condition is zero.
continue ;	Continues execution after the loop ends.

BEGIN marks the beginning of the loop. The body of the loop (here indicated by the words "process" and "condition") is executed each time through the loop. The body of the loop must leave a numeric value on top of the stack; that value is examined each time the END statement is reached. If the value on top of the stack is zero (false), the loop is repeated; to terminate the loop, any non-zero value (true) is placed on the stack. Thus, loop repetition is directly under program control. When the loop is ended, the word following END will be executed. END removes the number it tests from the stack.

Suppose ARRAY is the starting address of an array of sixteen-bit entries containing at least one non-zero entry. You can find the address of the first non-zero entry by the loop:

```
: SEARCH   ARRAY 2 -  BEGIN 2+  DUP @  END ;
```

where:

: SEARCH	
ARRAY 2 -	Decrements the array address by two.
BEGIN	Begins an indefinite loop.
2+	Increments the address by two.
DUP @	Fetches the contents while saving the address.
END ;	Ends the loop at the first non-zero entry.

In the body of the loop, 2+ increments the address on the stack before the examination of the contents of the address. Consequently, you must decrement the address ARRAY with the phrase 2 - before entering the loop. The loop terminates when a non-zero entry is found. Notice that DUP preserves the address on the stack during the loop and that, once the loop is complete, the last address remains on the stack.

It is possible to create a simple program that will execute forever:

```
: FOREVER   BEGIN  whatever  0 END ;
```

The zero preceding END guarantees re-execution of the loop body.

Often you will wish to execute some phrase a specified number of times (say ten). That can be done by writing this BEGIN ... END sequence:

```
: TEN-TIMES   0 BEGIN  phrase 1+  DUP 10 =
     END DROP continue ;
```

where:

```
: TEN-TIMES
     0                        Places a counter on the stack.
     BEGIN
          phrase              Provides the action(s) to be repeated.
          1+                  Increments the counter.
          DUP 10 =            Tests for equality to ten.
     END
     DROP                     Discards the counter.
     continue ;
```

Initially, the top of the stack is zero; after the body of the loop (the useful work) is performed, the counter at the top of the stack is incremented and compared to ten. If ten has been achieved, the END word will cause control to "drop through" to the next word. Otherwise, control will return to the BEGIN at the start of the loop.

Notice that there are seven words that must be executed (five of them repeatedly) to implement this loop. Also, if the body of the loop needs to add or remove successive items from the stack, the counter at the top of the stack must be accounted for and operated around. In this case a controlled loop would serve your purpose better. Controlled loops are discussed in Section 4.5, after consideration of the return stack, which is frequently needed for the initialization of controlled loops as well as other operations.

4.4 THE RETURN STACK

As explained in Chapter 1, FORTH uses two stacks. The most visible stack is the parameter stack, which is used to manipulate parameter values and memory locations. The second stack, the return stack, is used primarily for program control. Values saved on this stack include return addresses for colon definitions and counters for controlled loops. The two stacks segregate parameters from program control values so that FORTH code is both more readable and debugged easily.

FORTH's use of the two stack architecture came about because of the hazards inherent in conventional single-stack programs in which parameters, program addresses, and other control information are all combined together in the same stack. In such a system operations that should only be concerned with parameters must keep track of other entries in the stack. When two stacks are used, parameters and return addresses need never be confused.

There are occasions, however, when something on one of the stacks would be useful if available on the other or when one component of a definition requires one or more numbers that would otherwise be buried in the parameter stack. The basic FORTH vocabulary therefore includes three important words for transferring data from one stack to another:

<R Pops a number off the parameter stack and pushes it onto the return stack.

R> Pops a number off the return stack and pushes it onto the parameter stack.

I Copies the number that is on the top of the return stack and pushes it onto the parameter stack, without changing the return stack (see Section 4.5 for an example of usage).

<R and R> enable you to use the return stack as an auxiliary stack. For example, you may transfer a number to the return stack prior to a calculation which makes heavy use of the parameter stack. Since the number is on the return stack, you can fetch it back without disturbing the parameter stack. Judicious use of <R and R> can make definitions more readable.

Because the return stack is primarily used to hold control values, there are two important constraints on your use of it, the first of which is given here. Since the second constraint concerns DO ... LOOPs , it is given as Rule 9 (in Section 4.6).

RULE 8: ANYTHING PUSHED ONTO THE RETURN STACK MUST BE REMOVED WITHIN THE SAME DEFINITION.

For example, if you define CRASH this way:

```
: CRASH   0 <R ;
```

and then try executing CRASH , you will crash because the number placed on the return stack by <R will be used by ; as a return address, with fatal results. On the other hand,

```
: HARMLESS   0 <R  BEGIN  1 END  R> DROP ;
```

is indeed harmless.

Sometimes it's a little difficult to remember which of the two, <R or R> , transfers data to the return stack. FORTH attempts to keep frequently used words short to avoid lengthy manuscripts. The pictorial value of these two words is intended to suggest moving data onto (<R) or off of (R>) the return stack.

You should develop the habit of referring to a FORTH Glossary (Appendix B) when you need a reminder of the definition of any FORTH word.

It is worthwhile to pause here long enough to work out what will happen if you try to use a formation like:

```
... <R  phrase  I phrase  R> ...
```

within a definition. Since <R moves the top parameter stack item to the return stack and I copies the item back onto the parameter stack, this construct results in leaving the parameter stack with a duplicate set of what was formerly its top item. This is probably not the effect intended.

EXERCISE

1. Use <R and R> to define 2SWAP , to swap the first two byte pairs on the stack with the third and fourth pairs. That is, after:

   1 2 3 4 5 2SWAP

   the stack should contain:

   1 4 5 2 3 (with 3 on the top).

4.5 CONTROLLED LOOPS

An alternative definition for TEN-TIMES (from Section 4.3) makes use of the DO ... LOOP construct:

```
: TEN-TIMES   10 0 DO   phrase LOOP ;
```

where:

```
: TEN-TIMES
     10 0                 Places the loop parameters on the stack.
     DO                   Transfers the loop parameters to the
                             return stack.
          phrase
     LOOP ;               Repeats the loop ten times.
```

The first two numbers are, as always, placed on the stack (first ten, then zero). The ten becomes the limit of the DO ... LOOP ; the zero becomes the initial value of the loop index. The loop will execute ten times with the index starting at zero. The DO word causes the two top words on the stack to be transferred over to the return stack; that gets the loop parameters off of the parameter stack. After the body of the loop is executed, LOOP will increment the index and compare it with the limit. If the index is less than the limit, the loop is repeated. If the loop limit has been reached or exceeded, the two values are removed from the return stack and the next word (following LOOP) is executed.

Sometimes it is useful to have access to the loop index in a DO ... LOOP . The FORTH word I , which can be thought of as Index in the DO ... LOOP construct, fetches the top of the return stack (where the loop index is stored) and copies it onto the parameter stack without affecting the return stack. To print out ten numbers, from zero through nine, use this sequence:

```
: PRINT   10 0 DO   I .   LOOP ;
```

where:

```
: PRINT
     10 0 DO        Transfers the loop parameters to the return
                      stack.
          I         Copies the loop index (0, 1, 2, ..., 9) to the
                      parameter stack.
          .         Prints out the loop index from the top of the
                      stack.
     LOOP ;         Increments the index (on the return stack),
                      compares, repeats ten times.
```

Notice that DO ... LOOP structures, like those using BEGIN ... END , cannot be executed in the immediate mode; they must appear only in definitions of other words.

The loop index and limit don't have to be specified in the definition. They may be the result of prior computations or other stack manipulations, just as long as they are on the stack when DO is executed. Frequently, a definition that uses a loop requires the upper limit of a loop to be specified. Suppose, for example, you have a word called READ which reads a single data item from a device and stores it in the next sequential location of an array. You could then define ITEMS :

```
: ITEMS   0 DO  READ  LOOP ;
```

To read ten items, then, you would type the desired number of readings before typing ITEMS :

```
10 ITEMS
```

The ten would be on the stack when ITEMS was executed and would serve as the upper limit for the loop.

It is important to remember that any loop will be executed at least one time because the increment-and-test function is at the end of the loop. It is not possible to execute a loop zero times, only one or more.

Another example of a DO ... LOOP you might define at your terminal adds a new capability to your FORTH system. The word LIST , described in Section 2.2, displays a screen from the diskette. Sometimes, however, you'd like to list just a selected range of lines of the sixteen available (especially if your terminal is a teletype). You can now define a new word that will show the selected lines:

```
: SHOW   1+ SWAP DO  CR  I SCR @ LINE
     -TRAILING TYPE  LOOP ;
```

Do not confuse this SHOW with the SHOW in the PRINTING overlay that is defined to perform differently.

This SHOW operates as follows:

```
: SHOW
    1+                  Increments the last line number.
                        This insures that it is included in
                          the loop.
    SWAP DO             Arranges loop control and begins the
                          loop.
        CR              Outputs a carriage return and line
                          feed for a new line.
        I  SCR @        Fetches the line and screen numbers
                          for  LINE .
        LINE            Produces the location and count for
                          the requested line.
        -TRAILING       Reduces the count to omit trailing
                          blanks.
        TYPE            Types the line, removing the location
                          and count.
    LOOP ;              Repeats the loop until done.
```

Then the command,

```
2 5 SHOW
```

will display Lines 2 through 5 of the current screen, inclusive. This example illustrates the useful phrase 1+ SWAP , used to convert an inclusive range of numbers in increasing order to the parameters expected by DO . (The phrase OVER + SWAP can be used in a similar manner to convert a start and count into parameters for DO).

Another word used to conclude loops begun with DO is +LOOP . +LOOP expects a number on the stack, which it adds to the loop index before comparing the index and limit. For example, a word called EVEN may may use +LOOP to print even integers ranging from zero to a specified limit:

```
: EVEN   0 DO  I .  2 +LOOP ;
```

The value 2 placed on the stack before +LOOP is used as the increment to the index, so that the index steps through successive even values. More complicated uses of +LOOP involve computing the increment for +LOOP in the body of the loop.

In use, EVEN produces the following result:

```
10 EVEN 0 2 4 6 8 OK
```

EXERCISES

1. Define SUM to add the contents of an array, given its starting address and length on the stack.

2. Define POWER so that m n POWER computes the n-th power of m, for non-negative n.

4.6 NESTING STRUCTURES

DO ... LOOP and IF ... ELSE ... THEN sequences may contain other such sequences but only if they are properly nested. That is, one entire DO ... LOOP pair may be inside another pair but they may not overlap. The following loops, printed vertically for clarity, print out number pairs in the order (1 1), (1 2), (1 3), ..., (5 3), (5 4), (5 5):

```
: PAIRS
    6 1 DO                  Counts from one to five (major
                              loop).
        I                   Fetches the major count value.
        6 1 DO              Counts from one to five (minor
                              loop).
            DUP .           Duplicates and prints the major loop.
            I   .           Prints the minor loop.
            SPACE           Separates the two pairs with an extra
                              space.
            CR              Types a carriage return at the
                              terminal.
        LOOP
        DROP                Discards the old major count value.
    LOOP ;
```

This definition of PAIRS has one DO ... LOOP construction nested within another. This brings us to the second nesting rule:

RULE 9: WHEN NESTING STRUCTURES IN FORTH, YOU MUST NEST EACH STRUCTURE COMPLETELY WITHIN ANY OUTER STRUCTURE.

For example, you may not use IF to branch into or out of a loop or another conditional. Some examples of nesting are given in Figure 3.

Another example of nesting is provided by a different definition of EVEN (Section 4.5). This EVEN performs as the first did but in addition assures that the limit is not zero:

```
: EVEN  DUP IF  0 DO  I .  2 +LOOP  THEN ;
```

where:

Nesting DO ... LOOPS

RIGHT:

```
100 0 DO  I 10 + I DO  I .  LOOP  CR  10 +LOOP
```

this I refers to this loop

this I refers to this loop

WRONG:

```
X @ 0 DO  I 100 > IF  LOOP  THEN
```

this is not what you had in mind!

RIGHT:

```
[limit on stack] 100 MIN 0 DO  I 50 < IF ... ELSE ... THEN LOOP
```

<50 case ≥50 case

RIGHT:

```
[limit on stack] -DUP IF  0 DO  ...  LOOP  THEN
```

prevents executing loop if limit is 0

NOTE: without the IF, the loop would have been executed once

```
: EVEN
     -DUP IF                    Checks for a non-zero limit.
     0 DO
               I .              Prints the even index value.
               2 +LOOP          Increments the loop counter by two.
     THEN ;
```

The complexity of the actions taken on either branch of an IF ... ELSE ... THEN or within a DO ... LOOP structure is virtually unlimited. For example, a complete IF ... ELSE ... THEN structure may be used within an IF branch as in this definition of NEXT , which stores a number into the next empty one of three locations, given the number and the first address on the stack:

```
: NEXT
     DUP @                      Fetches the contents of the first
                                   location.
     IF                         Tests it for zero.
          1+ DUP @              Fetches the contents of the next
                                   location.
          IF                    Tests for zero.
               1+               Increments to the last location.
          THEN
     THEN
     ! ;                        Stores the number in the first,
                                   second, or third location.
```

Here we have nested one IF ... THEN structure entirely within another. This conforms to Rule 9 given above: when nesting structures in FORTH, you must nest each structure completely within any outer structure.

EXERCISES

1. How would you define MAX , MIN , and ABS ? (All are supplied as part of standard microFORTH.)

2. Define FACTORIAL to compute the factorial of a number.

4.7 RECAPITULATION

The words IF ... ELSE ... THEN, DO ... LOOP , BEGIN ... END , and DO ... +LOOP are all compiling words. That is, they direct the compiler to build branches within a definition, which will later cause the interpreter to re-execute or skip over words in the definition when the defined word is actually invoked. It is the function of these words to place items in a definition; therefore you must conform to Rule 6: compiling words must never be used outside a definition.

The least that can happen by ignoring this rule is that trash will be left at the top of the dictionary or stack. The worst is that trash may be deposited into existing definitions, making them unusable.

5.0 SAMPLE PROGRAM DEVELOPMENT

5.1 PROGRAMMING PHILOSOPHY

Understanding the problem you are trying to solve is essential to writing a successful application. Otherwise, sitting down at a terminal is like driving a great distance without a road map—possible, but not efficient. In order to uncover the essential features of the desired application, long hours of study may be required. Some tentative hardware/software tradeoffs must be made. Initial decisions need to be made about how much circuitry will be installed for interfaces and how much software will be relied upon to control the various discrete input/output signals. Once the hardware configuration is decided upon, application design and implementation begins.

There are two schools of thought in programming methodology. The first holds that you should write your program at your desk, check it carefully, and then use your terminal to implement and test it. The other school would prefer that you write the program at the terminal, while you're on-line to the development system. You may choose either (or both), depending upon your skills and willingness to experiment. Eventually, however, you will need to load in the microFORTH system and start operating.

When the initial application words have been tested, you will probably edit the source text onto the diskette using the EDITOR (discussed in Chapter 2). An application name, such as CONTROL , can be associated with source text by editing a constant into the diskette directory (usually in Screen 19). For example,

```
120 CONSTANT CONTROL
```

associates 120 with CONTROL ; 120 is the load screen number of the application (which may, in turn, load other screens). Typing CONTROL LOAD will then compile your application into memory for testing.

Your microFORTH development system is primarily a tool for interactive program development. When your application is completely debugged, the interactive features, such as the compiler and the dictionary search capability, are no longer needed. At this point you can use the microFORTH "cross-compiler," described in the microFORTH Technical Manual, to generate a compact program to run on your target system.

5.2 TOP-DOWN DESIGN

Frequently you can sketch out an application in reverse. That is, you begin by writing tentative definitions of words to perform the major functions of your application. The effort of drafting these definitions will clarify for you what other words need to be written first in order for the major words to work. In this way you continue backwards until you determine what variables, constants, and basic "building block" definitions you will need. Of course, to be loaded your application will have to be entered, starting with the simplest definitions and building up to the most complex. But the top-down approach will enable you to identify the most convenient elements to define. The example presented in this chapter will follow the top-down approach.

5.3 TESTING AND DEBUGGING

You will seldom write a complete application at once. It is better to write and debug a screen or two of source text at a time. You can compile your new definitions and test them out individually at the terminal, checking that they maintain the stack as you desire and otherwise perform correctly. This facilitates debugging by breaking the application up into easily tested modules.

A technique that is frequently helpful in testing a partially developed application is the use of stubs. Stubs are short FORTH definitions used temporarily to define words that the software under test will invoke. Stubs can be used to simulate the behavior of hardware which isn't currently connected. Another use that arises with the top-down approach is the simulation of a basic word before its final definition is written, in order to debug a word at a higher level.

5.4 CROSS-COMPILATION

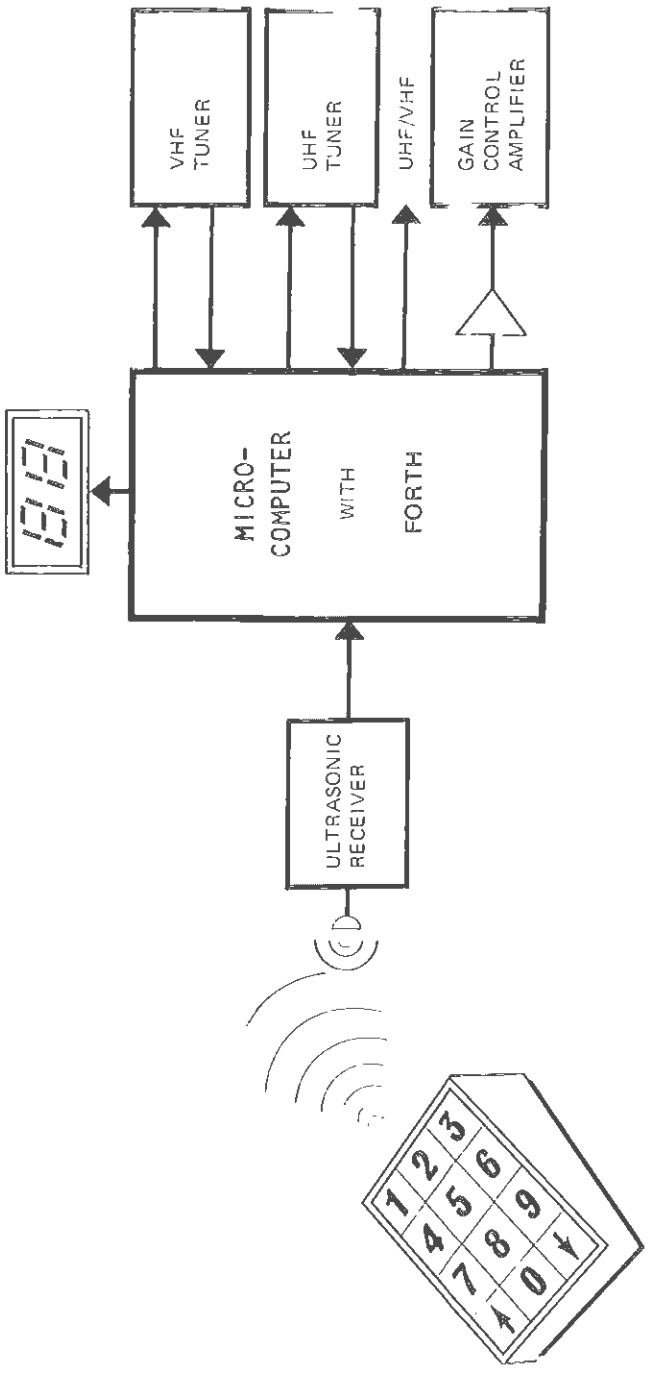
After you have made your entire application operate correctly, you may wish to create an efficient subset of it for production use. The microFORTH cross-compiler is used to generate a final object program that excludes the FORTH compiler and other features that you won't need in the production version of the system. The microFORTH address interpreter, however, is included in the final object program, since it is needed to control the execution of your high-level definitions.

There are some subtle limitations on what is acceptable to the cross-compiler. You should read the microFORTH Technical Manual carefully when you plan to cross-compile. You can safely defer that reading and understanding, however, until after you have your program finished; the modifications needed are unimportant for now.

5.5 THE TV REMOTE CONTROL UNIT

We have chosen the implementation of a remote control unit of a television set as an example (see Figure 4). The purpose is not to elaborate any particular application but rather to illustrate the programming techniques and development cycle discussed above.

In this example a hand-held remote control paddle sends out serial five-bit data codes to an ultrasonic receiver in the television itself. Each five-bit code represents one of the keys in the remote control paddle, although all thirty-two



TV Remote Control Unit

possible keys might not exist in every model of the hand-held paddle. The remote control paddle is so designed as to continue sending five-bit codes at 100 microsecond intervals for as long as a key is depressed. For simplicity we consider only the channel selection and volume control functions. These will allow us to show how microFORTH can be used to solve different kinds of problems. To implement fine-tuning, color and tint control, or an overlayed display of time on the TV screen, the same principles can be duplicated in hardware and software.

Following the top-down approach, we will first lay out the definition of TV , the word which will have overall control of the television. Basically, TV sits in an endless loop, waiting to receive signals from the paddle and then acts on them. The overall structure of TV will take the form:

```
: TV   BEGIN ... 0 END ;
```

where 0 END provides the unconditional return to BEGIN .

In the process indicated by ... in the above definition, the following discrete operations will have to be performed:

INPUT	Receives data from the paddle.
PROCESS	Performs an operation that depends upon the nature of the input.
IDLE	Waits for the next operation, since the process may well be complete before the user removes a finger from the button.

Thus, our full description of the process runs:

```
: TV   BEGIN INPUT  PROCESS  IDLE  0 END ;
```

Now, one of the cardinal rules of testing is:

RULE 10: NEVER PUT AN UNTESTED ROUTINE INTO A LOOP.

This is because a stack underflow, overflow, or other error is easiest to diagnose and handle in a single usage. When an error is repeated many times in a loop, however, it can cause both more serious failures and also a more confusing situation to debug. Therefore, our next task is to define and test INPUT , PROCESS , and IDLE .

The routine INPUT will want to read a signal. Since the operator may not be transmitting a signal, we will have to loop indefinitely until a signal is received. Let us project for ourselves a word READ (which will be defined later), to look for an input value from the interface and return either the value itself or else zero if no value is received. This value may be used directly as a truth condition (since true is defined as non-zero).

To await a signal we can use the phrase:

```
BEGIN  READ  -DUP  END
```

The word -DUP duplicates a non-zero value but leaves only one copy of a zero value. As long as there is no signal, the zero left on the stack will cause the loop to repeat. On the other hand, when a signal <u>is</u> received the END will

remove the copied value from the stack, leaving that value on the stack for further processing.

Thus we have the following definition for INPUT :

```
: INPUT   BEGIN  READ  -DUP  END ;
```

We can simulate the behavior of READ by defining a variable to contain a value from a hypothetical paddle and a dummy READ to fetch it:

```
0  CVARIABLE  PADDLE
: READ   PADDLE  C@ ;
```

Edit these definitions (PADDLE and READ) into a screen; each must be defined before it is used. Although we have designed this program in a "top-down" fashion (overall control first, increasing levels of detail later), it must be loaded and tested in a "bottom-up" order. Use a separate screen for stubs and dummy definitions; it can be replaced by the "real" screen later with minimum impact on the application.

Given that these routines have been loaded, we can now begin testing:

```
1  PADDLE C!
READ  . 1 OK                 READ produces a correct result.
INPUT  . 1 OK                INPUT also works.
```

At this point we must test the value received to decide whether channel selection or volume control is indicated. We will adopt the convention that a number < 14 indicates a channel, whereas other codes will be reserved for various other functions, such as volume. The phrase:

```
DUP              Saves a copy of the input.
13 >             Compares it with thirteen.
IF ANALOG        Performs the analog function if > 13.
ELSE  DIGITAL    Performs the digital function otherwise.
THEN
```

will perform this decision. Here we have projected the words ANALOG and DIGITAL to handle the two cases. Note that we have saved the value on the stack prior to the test, so that it will serve as the parameter to ANALOG or DIGITAL .

Now our definition of PROCESS is:

```
: PROCESS   DUP  13 >  IF ANALOG  ELSE DIGITAL  THEN ;
```

As with READ , we'd like to defer actual coding of ANALOG and DIGITAL , so we replace them with stubs whose primary purpose is to verify which path was taken. We'll do this by having ANALOG identify itself by printing a zero followed by its value, with DIGITAL printing a one followed by the value. We print the value not only to confirm that it is there, but also because ANALOG and DIGITAL will use (destroy) the value. This is an indirect way of demonstrating an important rule about stubs:

RULE 11: A STUB MUST REPRODUCE THE BEHAVIOR OF ITS INTENDED COUNTERPART WITH RESPECT TO STACK USAGE.

Thus we have:

```
: DIGITAL   0 . . 0 PADDLE C! ;
: ANALOG    1 . . 0 PADDLE C! ;
```

The last phrase (0 PADDLE C!) ensures that IDLE (which will use the READ stub in testing) will find a zero value. After editing and loading the above definitions (again, in bottom-up order), we can test them:

```
 1 DIGITAL 0 1 OK
14 ANALOG 1 14 OK
 1 PROCESS 0 1 OK                DIGITAL  was selected.
14 PROCESS 1 14 OK               ANALOG   was selected.
```

The hardware will perform the action faster than the operator can remove a finger from the key. It is necessary to await an idle before repeating the loop. Basically, this can be done with:

```
BEGIN  READ  NOT END
```

which loops until a zero signal indicates that the previous signal is no longer being read.

Thus our definition of IDLE is:

```
: IDLE   BEGIN  READ  NOT  END ;
```

This is really the opposite of INPUT and may be tested by using the same stub definition of READ :

```
0  PADDLE  C!
IDLE OK                          Returns immediately.
1  PADDLE  C!
IDLE                             Pauses indefinitely; use the RESTART
                                 switch to recover.
```

We may now use our definition of TV , except without the BEGIN ... END structure:

```
: TV   INPUT  PROCESS  IDLE ;
```

and test it:

```
1  PADDLE  C!
TV 0 1 OK                        DIGITAL  was executed.

14  PADDLE  C!
TV 1 14 OK                       ANALOG  was executed.
```

When DIGITAL and ANALOG are coded in more detail, these routines may be substituted directly. When hardware is available, its control will replace stubs in ANALOG , DIGITAL , and READ . The final addition of the BEGIN ... END structure in TV should occur after all functions are available and individually tested.

These substitutes for fully-defined words are a good example of the use of stubs, which allow a large program structure to be tested and verified before the details need to be defined. That is especially important when you are designing

Definition of TV in Screens

```
951

  0 ( TV CONTROL PADDLE)        FORGET TASK    : TASK ;
  1 ( TESTING STUBS)  952 LOAD
  2 ( ACTUAL ROUTINES TO BE PUT IN SCREEN 953)
  3
  4 : INPUT    BEGIN  READ -DUP END ;
  5
  6 : PROCESS    DUP 13 . IF  ANALOG
  7         ELSE DIGITAL  THEN ;
  8
  9 : IDLE    BEGIN  READ  NOT END ;
 10
 11 : TV    BEGIN  INPUT PROCESS IDLE  0 END ;
 12
 13
 14
 15



952

  0     ( TESTING STUBS FOR TV PADDLE I/O)
  1
  2 0 CVARIABLE PADDLE
  3
  4 : READ    PADDLE C@ ;
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

programs for microprocessors. Although the actual application (in this case, a TV tuner) may be well understood, the details of how the input/output ports work may not yet be settled. The software writer can have work underway while the hardware designer finishes work because in microFORTH it is easy to replace a stub by a whole definition at a later time.

A set of screens that include the component testing definitions of TV is shown in Figure 5. You now know enough to understand how this program works if you spend a small amount of time studying it.

6.0 ASSEMBLER FEATURES

FORTH, by design, is a programming language that allows you, the programmer, to concentrate on what the job is, not on how it is to be done on a particular computer. This independence from unnecessary detail is one of FORTH's most powerful features. There are times, however, when you must descend into the native language of your microprocessor.

In general, you should count on writing very little assembly language code under microFORTH. There are, however, two kinds of needs that are best fulfilled by using assembly code. First, on many computers, when you have designed a unique hardware/software interface, you may have to write a short code sequence to communicate with your device. Second, there are occasions when speed is more important than the compact programs that FORTH naturally provides. In either of these cases, assembly language is a powerful tool.

6.1 CODE DEFINITIONS

Like the compiler, the assembler adds definitions to the dictionary but with the difference that the assembler uses the defining word CODE . The word CODE sets the inner interpreter to the ASSEMBLER vocabulary. A major difference between the compiler and the assembler is that the assembler remains in execute mode and never changes to compile mode. Therefore, parameters for instructions are placed on the stack during the assembly process.

The CODE definition construct looks like:

CODE name (machine instructions) NEXT JMP or NEXT

where:

CODE	Enters the ASSEMBLER vocabulary and adds the following "name" to the dictionary.
name	Names (labels) the CODE definition.
(machine code)	Provides instructions for the microprocessor.
NEXT JMP or NEXT	Returns to the inner interpreter. (This is a CODE ending only.)

In its most primitive form, microFORTH assembly language consists of nothing more than a sequence of machine instructions in numerical form. First these numbers are placed on the stack. Then two special words (C, and ,) remove a number from the stack and enter it into the dictionary as either a single byte (C,) or a byte pair (,). When the CODE word is invoked by name, such numbers are executed as object code for your processor.

Although at times it may be convenient to write code at the object code level, it is not convenient to write all code at this level.

FORTH's assembler provides a set of macros which resemble the manufacturer's instruction mnemonics. The FORTH words (macros) are executed at assembly time to build the appropriate machine instructions. When the name of the CODE definition is invoked, the machine instructions (those built by the assembler) are executed. It is very important to remember the difference between executing the macros, which create the machine code, and executing the code definitions.

For example, the instruction to add a register to one CPU's accumulator is:

```
CODE ADD    L ADD    NEXT JMP
```

When assembled, CODE creates the dictionary entry named ADD which has one machine instruction that adds the L register to the accumulator. When ADD is invoked, the machine instruction is executed.

Another example helps clarify the difference between assembly-time and run-time modes of operation.

```
HERE  2 - TST
```

While assembling, this sequence places an address (HERE 2 -) on the stack; the macro TST uses that address when it creates the machine instruction. Now, when the word which contains this instruction is invoked, the address computed at assembly time is used. HERE 2 - is not executed at run time.

6.2 NOTATIONAL CONVENTIONS

Because assembly language is so primitive, many of the niceties that FORTH usually takes care of for you are unavailable. For example, operations in FORTH fetch parameters from the stack and put results there. In assembly language, however, you must manipulate the stack yourself. Furthermore, to write in assembly language you must understand not only the machine instruction set for your CPU but also the unique allocation of CPU registers assumed by FORTH.

The FORTH assembler does follow standard FORTH conventions such as Reverse Polish Notation. This requires that parameters (such as accumulator mnemonics, addresses, etc.) precede the instruction mnemonic. The order in which parameters for instruction mnemonics are entered also follows this convention so that the destination addresses/registers precede the source addresses/registers. Examining assembler screens in your microFORTH system will clarify this to you.

In addition, there are standard mnemonics for fundamental FORTH pointers and addressing conventions.

S	The address of the top of the parameter stack.
W	Usually the address of the parameter field of the current definition. (The specific position of the pointer is defined in your microFORTH Technical Manual Appendix.)
R	The address of the top of the return stack.

Wherever they may be, these names may be used in code to refer to the areas. In CODE definitions, parameters are handled explicitly by using S (the parameter stack pointer) and the code-ending returns that push or pop the stack before executing NEXT .

CODE definitions must end with a jump to the inner interpreter. They do so by using a special routine called NEXT , whose primary function is to execute the next FORTH word in a colon definition. On some processors the jump to NEXT is explicit:

```
NEXT JMP
```

On others, a macro called NEXT is used to assemble the appropriate code. Several words are available to modify the stack before returning to NEXT ; these are summarized in the microFORTH Technical Manual. Not all of these are available on all processors; consult the appendix of the microFORTH Technical Manual for your processor.

There are other notational conventions in microFORTH for more advanced assembler operations. These also may be found in the microFORTH Technical Manual.

In assembly code, for instance, there are logical control structures such as IF ... ELSE ... THEN and BEGIN ... END which can alter the flow of execution. They are analogous to the FORTH high-level logical structures in operation. Because the assembler operates in execute mode, however, these words are executed at assembly time, placing addresses on the stack which the assembler uses to build jump instructions.

This chapter's discussion of the basic assembler package is not intended as an in-depth treatise on your assembler. Rather it is intended to provide you with the fundamental elements so that you may, if you wish, practice writing very simple routines before you tackle the more detailed material presented in the microFORTH Technical Manual.

APPENDIX A. SUMMARY OF FORTH RULES

1. FORTH WORDS ARE MADE UP OF AN ARBITRARY NUMBER OF CHARACTERS, SEPARATED BY SPACES.

2. MOST WORDS REQUIRE PARAMETERS ON A PUSH-DOWN STACK.

3. ANY ERROR MESSAGE EMPTIES BOTH STACKS.

4. ALL PARAMETERS PUT ONTO A STACK MUST BE REMOVED WHEN THEY ARE NO LONGER NEEDED. THE ORDER WILL BE LAST IN, FIRST OUT.

5. ALL WORDS MUST BE DEFINED BEFORE THEY CAN BE USED.

6. COMPILING WORDS MUST NEVER BE USED OUTSIDE A DEFINITION.

7. EVERY IF MUST BE FOLLOWED BY A THEN .

8. ANYTHING PUSHED ONTO THE RETURN STACK MUST BE REMOVED WITHIN THE SAME DEFINITION.

9. WHEN NESTING STRUCTURES IN FORTH, YOU MUST NEST EACH STRUCTURE COMPLETELY WITHIN ANY OUTER STRUCTURE.

10. NEVER PUT AN UNTESTED ROUTINE INTO A LOOP.

11. A STUB MUST REPRODUCE THE BEHAVIOR OF ITS INTENDED COUNTERPART WITH RESPECT TO STACK USAGE.

APPENDIX B. microFORTH GLOSSARY

This glossary includes all words, definitions, and screen assignments that are common to all CPUs. Because of the flexibility of the FORTH language, however, you may find a few exceptions on your diskette. These will have been caused by our programmers' making improvements to the microFORTH system you have received.

Within this glossary there are also a few words whose exact behavior varies from chip to chip because the implementation of each is machine dependent. The end behavior of these words, however, is the same on all machines; the most obvious variations of implementation occur in M* and M/MOD . They are used by */ /MOD */MOD and MOD . Do not use M* and M/MOD unless you understand exactly how these words modify the stack pointer on your particular CPU. Use */ /MOD */MOD and MOD to perform the appropriate arithmetic.

Because they are CPU-dependent, no ASSEMBLER, CROSS-COMPILER, or RECONFIGURE words are given in this glossary. The microFORTH Technical Manual gives specifics of these three application vocabularies by chip type, while the Development System Documentation that you received with your system contains information about any additions or other changes.

Short glossaries for the microFORTH vocabularies that pertain only to Options (such as Extended-Precision Math or File Management) are provided with the options when the number of words warrants it.

The order followed here is that of the ASCII character codes.

MICROFORTH GLOSSARY

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
!	FORTH	0	2	0

Stores the second number on the stack into the address which is on the top of the stack. For example, if VALUE is a VARIABLE , then 32767 VALUE ! changes VALUE to 32767.

"	EDITOR	14	0	0

Used to enter a line of text into PAD; the text is terminated by the delimiter " .
Usage: " TEXT" 1 I
This example inserts TEXT in Line 2 of the current screen.

#	FORTH	12	1	1

Converts the least significant digit of a 16-bit binary number to its ASCII equivalent using the current BASE. The ASCII character is placed in the output string.

#>	FORTH	12	1	2

Terminates the pictured numeric output, leaving the byte count of the string on top of the stack and its address beneath for TYPE .

#LEFT	EDITOR	21	0	1

Computes the number of characters remaining in the source text line.

#S	FORTH	12	1	1

Converts any remaining digits of a 16-bit binary number on the stack to their ASCII equivalents, using the current BASE. The ASCII characters are placed in the output string. At least one digit will be converted if the number is zero.

'	FORTH	11	0	1

Places the address of the parameter field of the next word in the current input stream onto the top of the stack. Searches first the CONTEXT vocabulary, then the CURRENT vocabulary, before giving an error message.

'S	FORTH	10	0	1

Places the address of the top of the stack on the stack, i.e., the address of the top of the stack before 'S was invoked.

(	FORTH	3	0	0

Begins a comment, which is terminated by) . Comments are ignored by the system and may appear inside or outside a definition. They may not, however, cross an even line boundary in source text screens.

(.)	FORTH	12	1	2

Converts a sixteen-bit signed number on top of the stack to its ASCII equivalent, leaving the byte count of the string on the top of the stack and its address beneath for TYPE .
Used by . (i.e., dot).

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
(MARK)	FORTH	9	1	0

Compiles a backward jump in a logical structure.

(MATCH)	FORTH	22	4	2

Usage: string-A count string-B count (MATCH)
Counts must be <256. Searches for the 1st occurrence of A in B. Returns the end byte plus 1 of the matched string in B and a truth value: zero if no match and non-zero if match.

(MATCH)	EDITOR	22	4	2

In the EDITOR vocabulary on COSMACs only. Behaves like the FORTH vocabulary (MATCH) .

(MOVE)	FORTH	22	3	0

Only exists on 6800s and COSMACs; in the EDITOR vocabulary on COSMACs. Same as MOVE except there is an intermediate move to HERE .

(NUMBER)	FORTH	10	1	2

Same as NUMBER except that the ASCII string may begin with a minus sign. Also, if the terminating character is not a space, (NUMBER) will exit with an error message. The top of the stack is either the terminator or garbage.

(THEN)	FORTH	9	1	0

Completes a forward jump in a logical structure.

*	FORTH	5	2	1

Performs an unsigned multiply of the low-order byte of the top number on the stack with the sixteen-bit number beneath it, leaving a sixteen-bit product.

*/	FORTH	5	3	1

Multiplies the second and third numbers on the stack, then divides by the top number, leaving the quotient on top of the stack. This is an unsigned operation with a twenty-three-bit intermediate result.

*/MOD	FORTH	5	3	2

Multiplies the second and third numbers on the stack, then divides by the top number, leaving the quotient on top of the stack and the remainder beneath. This is an unsigned operation with a twenty-three-bit intermediate result.

+	FORTH	0	2	1

Replaces the two numbers on the stack by their sum.

+!	FORTH	0	2	0

Increments the sixteen-bit word whose address is on the top of the stack by the amount in the second word of the stack.

+LOOP	FORTH	0	1	0

Terminates the range of a DO ... LOOP. Increments the index by an unsigned sixteen-bit number on top of the stack, removing the number. The loop is terminated if the new index equals or exceeds the limit (unsigned compare).

MICROFORTH GLOSSARY

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
+LOOP	FORTH	9	1	0

Defines the compile-time behavior of +LOOP .

,	FORTH	0	1	0

Places the sixteen-bit value on top of the stack into the next dictionary position (at HERE) and advances H by two.

-	FORTH	0	2	1

Subtracts the top stack item from the second stack item, leaving the difference on the stack.

-'	FORTH	0	0	2

Returns a nonzero value if the next word in the current input stream cannot be found in the dictionary, and 0 if it can be found. If the word is found, the second item on the stack is the address of the word's parameter field.

-DUP	FORTH	3	1	2

Reproduces the top of the stack only if it is non-zero.

-MOVE	FORTH	22	3	0

Same as MOVE except that the count must be less than 256 and the block of memory is moved in reverse order, beginning at its highest address. (8080s and Z80s only.)

-TRAILING	FORTH	13	2	2

Reduces the byte count on the top of the stack by the number of trailing blanks found in the string whose address is the second item on the stack.

.	FORTH	12	1	0

Outputs a signed sixteen-bit number from the top of the stack.

.R	FORTH	13	2	0

Outputs the second number on the stack, right-adjusted in a field whose width is specified on the top of the stack.

/	FORTH	5	2	1

Unsigned division of the second word (full sixteen bits) of the stack by the top (max value 128), leaving the quotient on the top of the stack.

/MOD	FORTH	5	2	2

Performs an unsigned division of the second stack item by the first, leaving a quotient on the top of the stack and a remainder beneath.

0<	FORTH	0	1	1

If the top stack item is less then zero, replaces it with one; leaves zero otherwise.

0=	FORTH	0	1	1

If the top stack item equals zero, replaces it with one; leaves zero otherwise.

MICROFORTH GLOSSARY

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
1+	FORTH	0	1	1

Adds one to the top stack item.

1LINE	EDITOR	21	0	1

Given a string in PAD, searches for the string in the current line. Leaves zero if the string is not found and one if it is. Leaves the cursor positioned at the end of the matched string or at the end of line if not found.

2*	FORTH	0	1	1

Doubles the value of the top item on the stack.

2+	FORTH	0	1	1

Adds two to the top stack item.

8*	FORTH	3	1	1

Multiplies the top value on the stack by eight.

:	FORTH	0	0	0

Creates a dictionary entry for the word following : . Puts the interpreter into compile mode.

;	FORTH	0	0	0

Terminates a : definition. Toggles the user variable STATE .

;CODE	FORTH	4	0	0

Ends the creation portion of a new defining word and begins the code portion (run-time behavior) of it.

;CODE	FORTH	0	0	0

When executed, sets the code address of the new wod to point to the code that follows ;CODE .

;S	FORTH	0	0	0

Ends the loading of any screen in which ;S is executed. Within a definition causes an exit to the next outer definition.

<	FORTH	0	2	1

If the second stack item is less than the top, replaces the top two items on the stack with one, zero otherwise. This is a limited signed compare. Equivalent to - 0 <.

<#	FORTH	12	0	0

Begins pictured numeric output. Sets HLD to PAD. sixteen-bit binary number must be on the stack.

<BUILDS	FORTH	3	0	0

Begins the compile-time behavior of a new "high-level" defining word. Defined as 0 CONSTANT ; used with DOES >.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
<R	FORTH	0	1	0

Removes the top item on the parameter stack and places it on the top of the return stack.

=	FORTH	0	2	1

If the top two stack items are equal, replaces them with one; leaves zero otherwise.

>	FORTH	5	2	1

If the second item on the stack is greater that the top item, replaces both with one; leaves zero otherwise. This is a limited signed compare. Equivalent to SWAP - 0< .

?	FORTH	12	1	0

Outputs the contents of the word address which is on the top of the stack. Equivalent to @ . (dot).

?STACK	FORTH	10	0	0

Checks for stack underflow and overflow and issues an error message if appropriate.

@	FORTH	0	1	1

Replaces the address on the top of the stack by the contents of the two-byte word at that location.

A	EDITOR	14	1	0

In the current screen, adds the line of text that follows A AFTER the line number given. Line 15 is lost. The added line remains in PAD.

ABS	FORTH	5	1	1

Replaces the top stack item with its absolute value.

AND	FORTH	0	2	1

Performs the logical sixteen-bit AND operation on the top two stack items.

ASSEMBLE	FORTH	9	0	1

For COSMACs only, a constant which gives the load screen of the ASSEMBLER vocabulary.

ASSEMBLER	FORTH	0	0	0

Sets CONTEXT to the ASSEMBLER vocabulary.

AT	EDITOR	21	1	1

Calculates the physical address in memory of the current cursor position within the current screen.

B	EDITOR	21	0	0

Positions the cursor in front of the string just found. Used in conjunction with F .

MICROFORTH GLOSSARY

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
BACKUP	DISKING	24	0	0

Copies an entire diskette from Drive 0 to Drive 1.

BASE	FORTH	0	0	1

A user variable that contains the radix for number conversions on input or output. It is one byte long and is used with C@ and C! .

BEGIN	FORTH	9	0	1

Marks the beginning of an indefinite loop which is terminated by END . Leaves its address on the stack.

BLANK	FORTH	22	2	0

Given an address in the second stack position and the byte count (<256) on top, stores blanks into that region of memory. Also in the EDITOR vocabulary.

BLK	FORTH	0	0	1

A user variable that contains the number of the block being interpreted during a LOAD . If BLK contains zero, input is from the terminal. Overlaps the user variable IN .

BLOCK	FORTH	3	1	1

Replaces the block number on the top of the stack by the starting address of its block buffer in memory, adding in OFFSET .

BUFFER	FORTH	0	0	1

Returns the address of the block ID of a free block buffer. The ID resides two bytes before the beginning of the block buffer.

C	EDITOR	21	0	0

Inserts the string that follows C into the current line, beginning at the current cursor position. Extra characters (at the end of the line) will be lost.

C!	FORTH	0	2	0

Stores the eight-bit value in the low-order byte of the second item on the stack into the address on the top of the stack.

C#	EDITOR	21	0	1

Calculates the character position of the cursor in the current line.

C,	FORTH	0	1	0

Places the low-order byte of the top of the stack into the next dictionary position at HERE and advances H by one.

C@	FORTH	0	1	1

Replaces the address on the top of the stack with its contents. The high-order byte is zero filled.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
CODE	FORTH	4	0	0

Begins a dictionary entry for the word following it and enters the ASSEMBLER vocabulary.

COMPILE	FORTH	0	0	0

Changes the user variable STATE ; used by : and ; .
(Changes the name field in the dictionary entry. The byte changed is machine-dependent.)

CONSTANT	FORTH	0	1	0

A defining word which creates a dictionary entry for a sixteen-bit value. When the name is invoked, the value is placed on the top of the stack.

CONTEXT	FORTH	0	0	1

A user variable whose contents point to the vocabulary in which searches begin.

COPY	EDITOR	14	2	0

Copies one screen to another. The source screen is unchanged. Usage: source-screen destination-screen COPY

COUNT	FORTH	15	1	2

Takes the address of a character string whose first byte is a character count and replaces it with a character count on top of the stack and the address of the first character beneath. In Screen 16 on COSMACs.

CR	FORTH	12	0	0

Sends a carriage return and line feed to the terminal.

CREATE	FORTH	0	0	0

When executed, creates a dictionary header for the word that follows it. Used in the definition of all defining words.

CROSS	FORTH	19	0	1

A CONSTANT that places the load screen number of the cross-compiler on the top of the stack.

CURRENT	FORTH	0	0	1

A user variable whose contents point to the vocabulary in which new definitions are added. The CURRENT vocabulary is searched when the search of the CONTEXT vocabulary ends.

CVARIABLE	FORTH	4	1	0

A defining word which creates a dictionary entry for an eight-bit value. When the CVARIABLE name is invoked, the address of the value is placed on the top of the stack.

CZ	FORTH	0	0	1

Places one byte of zero on the stack. Increments the stack pointer by one byte.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
D	EDITOR	14	1	0

In the current screen, deletes the line specified on the top of the stack and places it in PAD. Succeeding lines are moved up; Line 15 is duplicated.

DECIMAL	FORTH	5	0	0

Sets BASE to radix ten for number conversion.

DEFINITIONS	FORTH	11	0	0

Sets CURRENT to CONTEXT. Used to specify the vocabulary in which definitions will be entered.

DELETE	EDITOR	14	1	0

Stores zero into the first two bytes of the specified screen to mark the screen as unused. This screen then will not be listed by INDEX , SHOW , or TRIAD in the PRINTING utility.

DEVICE	PRINTER	17	0	0

Marks the load point for the PRINTER vocabulary. (Not available on COSMACs.)

DISKING	FORTH	19	0	1

A CONSTANT that gives the load screen number of the DISKING utility.

DO	FORTH	9	0	0

Defines the compile-time behavior of DO .

DO	FORTH	0	2	0

Begins a finite loop whose index (the top stack item) and limit (the second stack item) are moved to the return stack when it is invoked.

DOES >	FORTH	0	0	0

A defining word which marks the beginning of the run-time portion of a new defining word. Used with <BUILDS .

DOWN	DISKING	24	2	0

See RIGHT .

DR0	FORTH	19	0	0

Sets the user variable OFFSET to zero for absolute access by BLOCK and LIST .

DR1	FORTH	19	0	0

Sets the user variable OFFSET to 2000 for relative access to Drive 1 by BLOCK and LIST .

DROP	FORTH	0	1	0

Removes the top item from the stack.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
DUMP	FORTH	13	2	0

Outputs the values contained in a specified region of memory.
Usage: start-addr count DUMP

DUP	FORTH	0	1	2

Duplicates the top of the stack.

ECHO	FORTH	15	1	0

Sends the character in the low-order byte of the top stack item to the terminal.

ECHO	PRINTER	17	1	0

Sends the character in the low-order byte of the top stack item to the printer device. (Not available on COSMACs.)

EDIT	FORTH	19	0	1

A constant that is the load screen number of the EDITOR vocabulary. For COSMACs only.

EDITOR	FORTH	14	0	0

Sets CONTEXT to the EDITOR vocabulary. It is IMMEDIATE so that it may be invoked inside a definition.

ELSE	FORTH	0	1	1

Used within the IF ... THEN structure, ELSE begins the "false" part. The words that follow ELSE are executed if the top stack item was zero (false) when IF was invoked.

ELSE	FORTH	9	0	0

Defines the compile-time behavior of ELSE .

END	FORTH	0	1	0

Terminates an indefinite loop started with BEGIN . Returns to the start of the loop if the top stack item is zero (false); terminates the loop if the top stack item is non-zero (true). (Not available on 6800s.)

END	FORTH	9	0	0

Defines the compile-time behavior of END .

ERASE	FORTH	4	2	0

Given the byte count on top of the stack and the address beneath, stores zeros in a region of memory.
Usage: start-adr. count ERASE

ERASE-CORE	FORTH	3	0	0

Stores zeros in all the block buffers. Does not write to disk any block buffers marked for writing.

ERR	EDITOR	21	1	0

Uses the error condition code on top of the stack; if true, moves text from PAD to HERE and invokes 0 QUESTION .

MICROFORTH GLOSSARY

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
ERROR	DISKING	26	0	1

Leaves the value of STATUS masked for error bits.

EXECUTE	FORTH	0	1	0

Executes the word whose parameter field address is on top of the stack.

EXPECT	FORTH	16	2	0

Inputs, from the terminal, the number of characters specified on top of the stack and places them into memory at the address given beneath, followed by 2 nulls. The string is ended when the count is exhausted or by a carriage return.

F	EDITOR	21	0	0

Beginning at the current cursor position in the current screen, searches for the string that follows F and leaves the cursor positioned immediately after that string. Multiple lines are searched.

FILL	DISKING	24	0	0

Sets a non-zero value into the block IDs of the disk block buffers. Used to force the operating system to read disk Block 0 from disk.

FIND	EDITOR	21	0	0

Searches each line of the current screen, beginning at the current cursor position for the string in PAD . Prints an error message if the string is not found.

FLUSH	FORTH	3	0	0

Forces all updated blocks to be written to disk.

FMT	DISKING	24	0	0

Formats the disk on Drive 1 (where appropriate).

FORGET	FORTH	11	0	0

Physically forgets, at execute time, all dictionary entries after and including the word specified in the current input stream.

FORTH	FORTH	11	0	0

The name of the innermost vocabulary. Sets CONTEXT to FORTH . It is IMMEDIATE so that it may be invoked inside a definition.

GAP	EDITOR	14	1	1

In the current screen, pushes all lines that occur AFTER the specified line down one.

H	FORTH	0	0	1

A user variable that contains the address of the top of the dictionary. See HERE .

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
HERE	FORTH	0	0	1

Places on the stack the address of the next available byte at the top of the dictionary. See H .

HEX	FORTH	5	0	0

Sets BASE to radix sixteen for number conversion.

HLD	FORTH	12	0	1

A variable that points at the most recent character of the output string during pictured numeric output.

HOLD	EDITOR	14	1	0

Transfers the line whose number is on the top of the stack to PAD.

HOLD	FORTH	12	1	0

Decrements HLD and places an ASCII character that is on top of the stack into the output string during pictured numeric output. See <# , # and #> .

I	EDITOR	14	1	0

In the current screen, inserts the line that is stored in PAD into the line that follows the one whose number is on top of the stack. Succeeding lines are pushed down; Line 15 is lost.

I	FORTH	0	0	1

Copies the top of the return stack onto the parameter stack; it does not alter the return stack.

IF	FORTH	0	1	0

Begins a conditional structure. Executes the words that immediately follow IF when the top of the stack is true (non-zero); otherwise skips to ELSE (if present) or THEN (if there is no ELSE) or WHILE (instead of THEN).

IF	FORTH	9	0	1

Defines the compile-time behavior of IF .

IMMEDIATE	FORTH	3	0	0

Marks the word most recently defined as a compiling word. The word is executed when encountered inside of a definition.

IN	FORTH	0	0	1

A user variable that points to the relative location in the input stream. IN overlaps the user variable BLK .

IN-LINE	FORTH	11	1	0

Given a number on the top of the stack, compiles it as a sixteen-bit literal.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
IN-LINE	FORTH	0	0	1

Puts a sixteen-bit literal on the stack at run time.

INC	DISKING	24	0	1

A constant that gives the block increment for RIGHT and SWEEP . Must be an odd number.

INDEX	PRINTING	27	2	0

Types the first line of each screen in the range given, sixty lines to a page. The copyright and heading are at the base of each page.
Usage: start-screen# end-screen# INDEX .

INTERPRET	FORTH	0	0	0

Outer interpreter loop; scans and searches for a word (to be compiled or executed, depending on STATE and precedence) in the dictionary. If not found, converts number and compiles literal form if in compile mode.

J	FORTH	4	0	1

Puts the index of the outer of two nested DO ... LOOPS on the stack. Only the indices of the two innermost nested loops are available. In Screen 5 on COSMACs.

KEY	FORTH	16	0	1

Receives and places on the stack a single character from the keyboard. In Screen 15 on COSMACs.

L	FORTH	13	0	0

Lists the screen specified in the user variable SCR.

L#	EDITOR	21	0	1

Calculates the line number of the cursor in the current screen. Implementation is machine-dependent.

LEAVE	FORTH	4	0	0

Sets the limit of a DO ... LOOP equal to zero so that a loop will be terminated. Implementation is machine-dependent.
In Screen 5 on COSMACs.

LEFT	DISKING	24	2	0

See RIGHT .

LF	PRINTING	27	0	0

Sends one line feed.

LINE	EDITOR	14	1	2

Given the number of a line in the current screen on the top of the stack, returns a character count of sixty-four (on top) and the address of the line beneath. The line number is masked by fifteen.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
LINE	FORTH	13	2	2

Given a line number beneath and a screen number on top of the stack, calculates the block address with a count of 64 on the top of the stack. Can be used by TYPE or MOVE .

LIST	FORTH	13	1	0

Lists the screen whose number is found on the top of the stack and places the screen number in SCR .

LOAD	FORTH	3	1	0

Begins interpretation of source text in the screen whose number is on the top of the stack.

LOG	DISKING	26	1	0

Logs a disk error by typing the block number that is on top of the stack, followed by the disk error message and the error status.

LOOP	FORTH	0	0	0

Terminates the range of a DO ... LOOP. Increments the index by one and exits if the index equals or exceeds the limit.

LOOP	FORTH	9	1	0

Defines the compile-time behavior of LOOP .

M	EDITOR	21	1	0

Given a count, moves the cursor forward (positive) or backward (negative). The line that contains the cursor is sent to the terminal.

M*	FORTH	5	2	2

Multiplies the top two values on the stack, leaving a twenty-four-bit product. The output format is chip-dependent. See M/MOD .

M/MOD	FORTH	5	3	2

Divides a twenty-four-bit number by the top stack item, leaving the remainder on top and the dividend beneath. The input format is chip-dependent. See also M* .

MATCH	DISKING	25	2	0

Usage: start-screen# end-screen#-plus-1 MATCH
Compares between DR0 and DR1; does not match screens if both begin with 0. On the first mismatch, types screen# and approximate line# (relative block * 2) of the mismatch.

MAX	FORTH	5	2	1

A limited signed compare between the top two values on the stack that leaves the largest value on the stack.

MESSAGE	FORTH	10	1	0

Types on the terminal a specified line relative to the start of Scr. 23. Omits trailing blanks. Uses Scr. 23 as the logical base, i.e., Message 16 is Line 0 of Scr. 24, Message 32 is Line 0 of Scr. 25, etc.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
MESSAGE	PRINTER	17	1	0

Same as MESSAGE in the FORTH vocabulary. (Not available on COSMACs.

MIN	FORTH	5	2	1

A limited signed compare between the top two values on the stack that leaves the smaller value on the stack.

MINUS	FORTH	0	1	1

Replaces the top of the stack by its two's complement.

MOD	FORTH	5	2	1

Divides the top stack item into the value beneath it, leaving the remainder on the top of the stack.

MOVE	FORTH	0	3	0

Moves a specified region of memory to another region of memory; moves the locations with lower addresses first. The source area remains unchanged.
Usage: source-addr. dest.-addr. byte-count MOVE

MSG	FORTH	15	0	0

Defines a word that will type out the string that follows it in the dictionary. The string is preceded by a character count. In Screen 16 on COSMACs.

MSG	PRINTER	17	0	0

Sets ASCII character codes into a named definition in the dictionary. (Not available on COSMACs.)

N	EDITOR	21	0	0

Finds the next occurrence of a string (found with an F) in the current screen.

NB	DISKING	24	0	1

A constant that gives the number of block buffers.

NEW	DISKING	24	0	1

A constant that gives the first block number on Drive 1.

NOT	FORTH	5	1	1

Reverses the truth value of the top of the stack. Identical to 0= .

NOTIFY	DISKING	26	1	1

Erases the block ID in the buffer whose address is on top of the stack after first fetching the block number contained in the ID. Invokes LOG with the block number and returns the number less the contents of OFFSET to the stack.

NUMBER	FORTH	0	1	2

Given the starting address less 1 of a numeric ASCII string on the stack, converts the string to binary according to the current value of BASE and leaves it in the second stack entry. The top item points to the non-numeric terminator.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
OCTAL	FORTH	5	0	0

Sets BASE to radix eight for number conversion.

OFFSET	FORTH	3	0	1

A user variable whose contents are added to block numbers in BLOCK to determine the physical block number.

OK	FORTH	15	0	0

Types the characters O, K, carriage return, and line feed. In Screen 16 on COSMACs.

OVER	FORTH	0	2	3

Copies the second item on the stack onto the top.

P	EDITOR	14	1	0

Places the line of text that follows P into the specified line. The previous content of the line is lost. The "put" line remains in PAD.

PAD	FORTH	12	0	1

The starting address of a holding buffer, PAD resides sixty-five bytes above HERE and moves as definitions are added to and deleted from the dictionary.

PRINTER	FORTH	19	0	1

A constant that places the load screen number of the PRINTER utility on the stack. (Not available on COSMACs.)

PRINTER	PRINTER	17	0	0

Same as CR . (Not available on COSMACs.)

PRINTING	FORTH	19	0	1

A constant that places the load screen number of the PRINTING utility on the stack.

QUESTION	FORTH	10	1	0

Repeats the last word executed by the text interpreter (found at HERE) and issues an error message as specified by MESSAGE , then empties both stacks and returns control to the operator. No OK is issued.

QUIT	FORTH	16	0	0

Empties the return stack and returns control to the operator. No OK is issued.

R	EDITOR	14	1	0

Replaces the line specified on the top of the stack with the contents of PAD.

R	FORTH	4	0	1

A constant that gives the address of the return stack pointer. For COSMACs, in the ASSEMBLER vocabulary.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
R!	FORTH	16	1	0

Moves the contents of Register U to Register R (i.e., resets the return stack). On COSMACs only.

R#	FORTH	13	0	1

User variable which contains the character position of the cursor in the EDITOR. When file management is in the system R# is the record number of the currently accessed record.

R>	FORTH	0	0	1

Removes the top of the return stack and places it on the parameter stack.

REMOVE	EDITOR	21	1	0

Given the character position of the beginning of the string to be deleted, deletes those characters on the line (up to the current cursor position) and moves all characters up. Trailing blanks are added at the end as needed.

RIGHT	DISKING	24	2	0

Copies the range of screen given from Drive 0 to Drive 1.
Usage: start-screen# end-screen#-plus-1 RIGHT
May be called UP , DOWN , or LEFT .

ROT	FORTH	0	3	3

Rotates the top three stack items, putting the third stack item on the top. On 6800s ROT resides in Screen 5.

S!	FORTH	10	1	0

Sets the address of the current stack pointer to the one given on the stack.

S0	FORTH	0	0	1

A user VARIABLE that contains the address of the bottom of the parameter stack and the start of the input message buffer.

SCR	FORTH	13	0	1

A user variable that holds the current EDITOR screen number.

SHOW	PRINTING	27	2	0

Types TRIADs of screens in the inclusive range given.
Usage: start-screen end-screen SHOW

SIGN	FORTH	12	2	1

Places a minus sign in the pictured numeric output string if the second word on the stack is negative. Deletes this second word on the stack but retains the top word.

SPACE	FORTH	12	0	0

Sends a single space (blank) to the terminal.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
SPACES	FORTH	12	1	0

Sends the number of spaces that is designated by the top stack item. May send zero spaces.

STATE	FORTH	0	0	1

A user variable, one byte wide, that indicates whether the interpreter is in compile or execute mode.

STATUS	DISKING	26	0	1

Returns on the stack the disk status as of the last operator.

STRING	EDITOR	21	0	0

Scans characters in the input stream until the delimiting character (the low-order byte on top of the stack or a carriage return) is encountered. Reads characters from the terminal into PAD with a leading count.

SWAP	FORTH	0	2	2

Exchanges the top two stack items.

SWEEP	DISKING	24	2	0

Reads each screen in the range given to check for disk errors.
Usage: start-screen# end-screen#-plus-1 SWEEP

T	EDITOR	14	1	1

Types the line specified (on the top of the stack) of the current screen and transfers it to PAD. The line number is left on the stack.

TASK	FORTH	3	0	0

Marks the beginning of the application vocabulary.

TEXT	FORTH	13	1	0

Scans characters in the input stream until delimiter (low-order byte as top stack item or carriage return) is encountered. Leading occurrences of the delimiter are skipped over. Input is placed in PAD and is blank filled.

THEN	FORTH	0	0	0

Marks the end of an IF ... THEN structure.

THEN	FORTH	9	0	0

Defines the compile-time behavior of THEN .

TILL	EDITOR	21	0	0

Beginning at the current cursor position on the current line, deletes all characters up to and including the string that follows TILL .

TOP	EDITOR	14	0	0

Positions the cursor at the beginning of the current screen.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
TRIAD	PRINTING	27	1	0

Types a set of three screens, given one screen number. The screen number may be any of the three screens on a page; the top screen is always the screen number modulo three. Copyright and heading appear at page bottom.

TYPE	FORTH	15	2	0

Uses a character count on top of the stack and an address beneath to send characters to the terminal. May TYPE zero characters. In Screen 16 on COSMACs.

TYPE	PRINTER	17	2	0

Uses a character count on top of the stack and an address beneath to send characters to the printer device. (Not available on COSMACs.)

U	FORTH	4	0	1

A constant that gives the address of the pointer to the start of the user area. For COSMACs, in ASSEMBLER vocabulary.

U*	FORTH	0	2	1

Unsigned multiply of the low-order bytes of the top two words on the stack, leaving a sixteen-bit product.

U/	FORTH	0	2	2

Unsigned divide of the second word on the stack by the top word, leaving a quotient on top and a remainder beneath.

UP	DISKING	24	2	0

See RIGHT .

UPDATE	FORTH	0	0	0

Marks the last buffer returned by BLOCK for writing. The block is rewritten on the disk either by the next FLUSH or automatically when the buffer is needed for another block.

USER	FORTH	0	1	0

A defining word, used to name locations at fixed relative addresses within the user area.

VARIABLE	FORTH	4	1	0

A defining word that creates a dictionary entry for a sixteen-bit value. When the VARIABLE name is invoked, the address of the value is placed on top of the stack.

VOCABULARY	FORTH	11	0	0

Defines a word whose parameter field plus two points to the most recent entry of that vocabulary's set of definitions. Executing a vocabulary name points CONTEXT to that vocabulary's parameter field plus two.

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
WHILE	FORTH	0	0	0

Terminates an indefinite loop of the following form:
BEGIN (condition) IF WHILE or BEGIN (condition) IF ELSE WHILE
Allows a test at the beginning of an indefinite loop.
(Not available on 6800s.)

WHILE	FORTH	9	2	0

Defines the compile-time behavior of WHILE .

WORD	FORTH	0	1	0

Reads forward in the current input stream until the delimiter given on the stack. The byte count and text are stored at HERE with the byte count in the first byte.

X	EDITOR	21	0	0

Beginning at the current cursor position, searches for and deletes the string that follows X . Multiple lines are searched.

[.	FORTH	13	0	0

Defines the run-time behavior of [. , which types out text on the CRT. The string resides in the dictionary, preceded by a count. It was laid down at compile time by use of the compiling word [.

[	FORTH	13	0	0

A compiling word which causes the string of characters until the delimiter], following it to be typed when the defined word is invoked.

[']	FORTH	0	0	1

During compilation, pushes onto the stack the sixteen-bit value that follows it.

[']	FORTH	11	0	0

Defines the compile-time behavior of ['] .

[BLOCK]	DISKING	26	1	1

Invokes BLOCK and, in case of read errors, retries up to ten times. Invokes LOG for all but the last retry.

[SWAP]	FORTH	11	1	1

A compiling word which swaps the top two words of the stack during compilation.

\	FORTH	0	0	0

A compiling word that places the address of the word that follows it into a new definition. Used to help define the run-time and compile-time behavior of a compiler word.

eot	FORTH	0	0	0

An ASCII null character that terminates scanning in the current input stream. Null controls the sequencing of the block buffers of a screen.