microFORTH TECHNICAL MANUAL

FORTH, Inc.
815 Manhattan Avenue
Manhattan Beach, CA 90266

(213) 372-8493

August 1978

Version 3
for RCA COSMAC

Version 3 (revised Appendices)
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

Appendix A. microFORTH IMPLEMENTATION ON THE RCA COSMAC

PREFACE

Because of the importance of the Technical Manual to programmers, we are revising it as quickly as possible and issuing the revised sections as they are completed. When the revision is complete (in 1979), each microFORTH customer will receive a copy; updates and/or errata sheets will be issued thereafter as needed.

The second edition of the Primer and the Appendices in this manual (i.e., the CPU-specific prose and both glossaries) reflect the most recent microFORTH systems. The next item scheduled is an expansion of the CPU-specific glossary to include the cross-compiler; you may request one to be sent at the time of publication by writing to the Editor at FORTH, Inc.

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. 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).

Additional conventions used in FORTH manuals are those that FORTH programmers use to make source screens readable. The most basic are:

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.

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.

The microFORTH Technical Manual forms the second level of documentation. In the four chip-specific versions (8080, 6800, 1802, and Z80), implementation that depends upon hardware is treated in more depth.

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 suggested 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.

1.0 INTRODUCTION

FORTH is a programming technique originally designed for real-time interactive minicomputer applications. In such an environment it offers several advantages: it is interactive, it is compact (uses little memory), and it is very fast.

FORTH has been implemented on many minicomputers and has been used in hundreds of minicomputer applications. This manual documents FORTH as it has been modified for use on 8-bit microprocessors for developing microprocessor applications which might be cross-compiled for production. Such a development system is presumed to have a programmer's terminal and a mass storage device such as a floppy disk.

FORTH is a computer language. It is structurally quite different from other languages, however. FORTH is an interactive system, in that it has no separate compiler or assembler--the routines that generate and execute machine instructions are integral parts of a unified system which also includes a small, fast interpreter and an executive whose characteristics may be modified for a specific application.

1.1 Elements of FORTH

FORTH has 5 main elements. Take away any one and you have something that is not useful. There is a synergistic effect among them that produces a remarkably powerful combination. Many of the characteristics and capabilities of FORTH were a surprise to us and some remain so!

1.1.1 Dictionary

The key element, if one must choose, is the dictionary. A FORTH program is 90% dictionary. This, as implied by its name, is a collection of words (or commands), together with their definitions. We are trying to explain a problem to the computer and do this by explaining what each of a number of words means. Thus it is a Man-to-Computer dictionary.

A collection of words is commonly called a vocabulary. The dictionary defines a vocabulary for the computer, perhaps several distinct vocabularies. Indeed, speaking of a FORTH program is sloppy, for FORTH is the program.

An application coded in FORTH is better called a vocabulary. You use an editing vocabulary to edit text, an observing vocabulary to observe an experiment, etc. The vocabulary is just that: it is not a program since it can't stand alone. It depends upon FORTH to do all the work and merely describes what must be done.

Technically, the FORTH dictionary may be described as a linked list of entries of various kinds. The nature of the linking and the actual content of the entries is described in the chapter "The FORTH Dictionary Structure."

Each defined word has an entry in the dictionary. FORTH provides the mechanism for searching the dictionary, executing words, discarding words and defining new words.

1.1.2 Stack

Another important and visible element of FORTH is its use of "push-down stacks" for parameters. Most FORTH words represent operations; these find their parameters on the stack; the addresses of variables are placed on the stack; results are placed on stack. The use of FORTH's stack will be very familiar to anyone who has used the HP pocket calculators.

In particular, numbers are placed on the stack. Elaborate calling sequences and temporary storage areas are eliminated by its use.

FORTH actually uses two stacks: the parameter stack described above is the most familiar to the casual user; the other is called the return stack. Its primary use is storing return addresses for the interpreter, although it may be used by the programmer for temporary storage.

1.1.3 Code

Some words are defined by code. This means that they contain a sequence of processor instructions to be executed. Such a word is similar to a subroutine.

The use of the FORTH assembler to construct code definitions is covered in Chapter 12.0 (THE ASSEMBLER). It allows the programmer direct access to the processor architecture for coding device drivers, time-critical functions, etc.

1.1.4 High level definitions

Most FORTH words are defined in terms of other previously defined words. Thus they are a sort of abbreviation. Such definitions, however, are much more powerful than the notion of abbreviation conveys. In fact, perhaps 90% of the words in a vocabulary are definitions. These are processor-independent. Such definitions begin with : and end with ; . The : is followed by the name of the new word being defined; this is followed by the words that make up the content of the definition, and finally the ; terminates the definition.

1.1.5 Blocks

The final element of FORTH is its blocks--chunks of secondary memory 128

bytes long. A block may contain ASCII text (such as FORTH source) or it may contain binary data. In either case, this secondary memory has been logically arranged in fixed-length chunks, each of which is assigned a number by which you refer to it. You may load programs from blocks or place data in blocks as if the blocks were memory. FORTH provides the access to them as "virtual memory".

FORTH blocks reside on some form of mass storage: this is most often disk but may be link tape, magnetic tapes, or any similar medium. The particular medium used in a FORTH system is transparent to the user; that is, blocks always appear to be in memory, regardless of where the system fetched them!

Source text is organized in "screens", each of which consists of eight 128-byte blocks (sectors). The size is chosen for convenience in display (as on a CRT terminal screen). The use of FORTH's text editor for managing text screens is covered in Chapter 7.0 (TEXT EDITOR).

1.2 Keyboard Input

FORTH is a terminal-oriented language. It demands the fluency of expression that only a keyboard can provide. The input FORTH wants is simple:

words separated by spaces.

In order to permit correcting errors and changing your mind, it recognizes:

RUB OUT to erase a letter (backspacing on CRT terminals).

When you are satisfied with your input, type:

RETURN to mark the end of a message.

FORTH will respond with a space. Then it will proceed to read and act on each word in the message. When it is done, it responds OK and spaces to a new line. This is as simple a way to communicate as we can devise.

In the chapters to come we will discuss the basic elements of FORTH in more detail and describe the process of using or modifying the FORTH vocabulary for your application.

2.0 USE OF THE STACK

2.1 Parameter Stack

An increasing number of computers and desk and pocket calculators nowadays base their logic on a parameter stack. FORTH has a parameter stack that increases toward low memory. There is a pointer to the word holding the number currently "on top of the stack"; this pointer is kept in a register whenever possible, although it may be in RAM memory. The stack itself is always in RAM. To add a word to the stack, the pointer is decremented.

The stack is 16 bits wide and thus may contain numbers in the range $-32768 \leq n \leq 32767$ or $0 \leq n \leq 65535$. Numbers may be values or addresses; as addresses are then 16-bit addresses, you may reference up to 65536 bytes of memory directly.

If you type a number on your terminal, the number will be converted to a 16-bit binary integer and placed on the stack. Typing . (period or decimal point) will cause the binary number on top of the stack to be converted to numeric characters and printed on the terminal. Most FORTH words expect one or more parameters on the stack (including words in the assembler), so you must make sure that they are there, and that they are in the proper order.

Figure 1 shows the result of typing a sequence of words. Recall that a word is "separated by spaces". There are no special characters in FORTH, so that ? and @#$%& and 4th are all perfectly good words. Several of the most commonly used FORTH words do deserve comment here:

! (pronounced "store") is the replacement operator. It expects 2

FORTH Stack

Example	You type:	Action
(1) 4 5 +	4	Number 4 converted to binary and pushed on the stack.
	5	5 converted and pushed on the stack, over the 4.
	+	4 and 5 replaced by 9.
(2) 17 X !	17	17 on the stack (over the 9).
	X	Location of X (which has been defined as a VARIABLE) pushed on the stack.
	!	17 stored in X; both 17 and location of X removed from stack.
(3) X @ * .		(Remember 9 is still on the stack from example 1).
	X	Location X pushed on stack.
	@	Address replaced by contents (17).
	*	9 and 17 replaced with 153.
	.	153 typed on terminal.
		Stack empty.

Figure 1

parameters on the stack: an address on top and a number beneath. It stores the 16-bit number beginning at the address.

Thus if you wish to place the value 17 in the location whose address is given by X, you might say

```
17 X !
```

@ (pronounced "at") is an operator that fetches a value. It expects an address on the stack; it replaces it with the 16 bit contents of that address. @ is an extremely important operator. It distinguishes between loading and storing a value into a variable, a function FORTRAN-like languages accomplish by context.

. is an operator that types the number on the stack (and discards it).

These operators have been assigned single-character names because they are used so often. Although their mnemonic value is weak, they are worth learning.

Similarly, + adds the two numbers on top of the stack, replacing them by the sum, while - subtracts, etc. Refer to the glossary in Appendix A for a more complete list of the operators available.

Several words have been defined in basic FORTH for manipulating the stack. The operation of these words is summarized in Table I. A fairly standard set of more complicated stack operators is available, in some developed applications, for such things as fetching numbers several levels down in the stack, etc.

The order of parameters on the stack is governed by several well-defined conventions:

1. Numbers are always pushed onto the top of the stack or popped off the top. Thus, if you type 1 2 3, the top is the right-most (or most recent) number. If you print these, the result might look like

```
. . . 3 2 1 OK
```

TABLE 1.

STACK MANIPULATING WORDS*

FORTH Word	Explanation	Stack Before	Stack After
		TOP	TOP
SWAP	Reverses order of top two entries	1 2	2 1
DUP	Reproduces top entry	1	1 1
DROP	Discards top entry	1 2	1
OVER	Pushes entry 2 on top of entry 1	1 2	1 2 1
ROT	Moves entry 3 to top of stack	1 2 3	2 3 1

* Remember that all stack entries are 16 bits (2 bytes).

where the underlined characters indicate generated output.

2. A "store" operation (!) operates from left to right (or entry 2 into entry 1), e.g.,

```
3 SEC !
```

stores 3 in SEC.

3. Multiple precision numbers are always placed on the stack with the high-order part on the top and the low-order parts beneath.

4. Multiple-parameter arithmetic operators use an order such that if the operator were moved from a suffix position to an infix position, the operands would be in their customary position. Thus:

```
A B -        is equivalent to      A - B
A B C */     is equivalent to      (A * B) / C
```

5. An operation will destroy all its input parameters and leave only its result (if any) on the stack. It will, of course, destroy no more than its own parameters. Thus:

```
1 2 3 +      will leave            1 5 on the stack
```

All routines developed for an application should adhere to these conventions.

The stack is located in high (RAM) memory and extends toward low memory. Unused memory is defined as the area between the high end of the dictionary and the top of the stack. The size of the stack is limited only by this amount, sometimes several hundred words. In practice, rarely more than thirty or so stack positions are used, even in highly complex applications. The remainder of this space is used as a scratch buffer area for such things as output formatting.

FORTH checks for stack underflow and overflow after executing each word that is typed at the terminal. Because FORTH does not begin executing these words until after the carriage return is typed, the word that caused the error is typed out on

the terminal, followed by an appropriate error message. If underflow occurs, the error message is

STACK EMPTY!

The overflow message says

DICTIONARY FULL!

Since overflow can occur only when memory is full, it normally is not of concern. Should this occur, however, the only recourse is to discard some portion of your dictionary. See the chapter entitled "The FORTH Dictionary Structure".

The stack is by far the best place to use for temporary storage, since stack accesses are fast and specific memory allocation is not required. In particular, the stack is an excellent place for saving the contents of a variable which will have to be changed temporarily.

2.2 Return Stack

FORTH has a second stack called the "return stack". It is primarily used by the system for keeping return addresses for the inner interpreter. It is available to the programmer on a limited basis, however, and can be a handy place to save numbers temporarily during complicated operations.

The commands which work with the return stack are

<R Pops a number off the parameter stack and pushes it on the return stack;

R> Pops a number off the return stack and pushes it on the parameter stack;

I Pushes the number which is on top of the return stack onto the parameter stack (without changing the return stack).

The return stack is also 16 bits wide.

The main constraint on the use of the return stack is that at the end of a definition the next return address must be on top. Although this is really the system's business, it does impose one important rule:

> ANYTHING YOU PUSH ON THE RETURN STACK MUST BE REMOVED IN THE SAME DEFINITION.

A second important use of the return stack is keeping DO-LOOP indices. This is discussed in further detail in Chapter 5.0 (THE COMPILER). The reason for discussing it here is that this use imposes a second rule:

> ANYTHING YOU PUSH ON THE RETURN STACK MUST BE REMOVED AT THE SAME LEVEL WITH RESPECT TO ANY DO-LOOP.

This rule is rather difficult to state, alas; perhaps it may best be illustrated thus:

> If you put something on the return stack outside of a loop, it must be removed outside the loop.

Some examples will be given in Chapter 5.0 (COMPILER).

3.0 NUMBERS AND VARIABLES

FORTH arithmetic is performed on integers and fixed-point fractions, rather than floating point numbers. The use of integers takes advantage of the speed of integer arithmetic operations in most microprocessors. FORTH's arithmetic operators are designed to make integer operations as convenient as possible, without sacrificing precision and speed. These will be discussed further under "Arithmetic"; the purpose of this chapter is to discuss numbers and number formats as well as named variables and arrays.

3.1 Numbers

All numbers that are typed or used in text blocks are automatically converted to binary and placed on the stack. The rules and conventions governing the use of numbers are as follows:

1. The base used in all number conversions is the current value of the variable BASE. You may set BASE to any value you wish; the commands DECIMAL, OCTAL, and HEX, however, are provided to set the most commonly used bases. BASE controls both input and output; thus typing

```
OCTAL 1000 DECIMAL .
```

will print 512. BASE is the location of an 8-bit variable and care should be taken to refer to it only by C@ and C! (character fetch and character store).

2. During the bootstrap loading, the default base is HEX; after basic FORTH is loaded, it is DECIMAL. Conventionally, you should assume that BASE is DECIMAL in all application blocks; if you wish to use OCTAL or HEX in a block, always return to DECIMAL at the end.

3. A valid number is one which may be converted successfully; that is, it contains only digits less than BASE plus a leading minus sign (-) to identify negative numbers. Note that this does <u>not</u> include + (to indicate positive). No blanks are permitted; conversion stops when a blank is reached.

4. A "-" before the leftmost digit causes the number to be negative (two's complement). Otherwise the number will be positive. For example,

```
-475
```

is a negative number.

5. All numbers are converted to 16-bit integers. Depending on whether you are using unsigned or signed arithmetic, a 16-bit integer gives a range of

$$0 \leq n \leq 65535$$

or $$-32768 \leq n \leq 32767$$

Numbers which exceed these limits will give an incorrect result.

6. Applications may need to handle numbers of more than two bytes in length, such as 24-bit integers, floating point numbers, etc. Routines to handle such number formats are available but are too application-dependent to be offered in the basic system.

3.2 VARIABLEs and CONSTANTs

The basic FORTH system provides two ways to name a location containing a value: VARIABLE and CONSTANT.

```
0 VARIABLE X
```

defines X as a named location 16 bits wide whose initial value is 0, while

```
1000 CONSTANT Y
```

names a location Y whose initial 16-bit value is 1000. The difference between these is that when you execute X (e.g., by typing it), you get its address on the stack, whereas typing Y pushes Y's value on the stack. Each CONSTANT or VARIABLE occupies 10 bytes: two for the value, plus eight for the name and other system information.

Normally a CONSTANT is thought of as a named value which will change seldom or never. The advantages of using CONSTANT are twofold: (a) a standard value which is used in several places should be defined with its value given only in one place, so that it may be changed simply in the future without the risk of a "missing one place"; (b) use of an actual number as a literal in a definition costs three or four bytes, whereas reference to a CONSTANT costs only two.

A VARIABLE, on the other hand, is really a named location. You may fetch its value by using the command @ (at) or store into it by using ! (store):

```
100 VARIABLE A  0 VARIABLE B
     A @ B !
```

moves the value of A to B. The command ? (defined as @ .) is provided for fetching and printing the value of an VARIABLE:

```
X ?
```

Now, this doesn't mean you can't change the value of a CONSTANT. The command ' (tick) fetches the address of the word which follows in the input stream; thus

```
100 ' Y !
```

will store 100 into the address of Y. Of course, this is just a little harder; CONSTANTS are optimized for uses in which a value will be used often but changed seldom.

3.3 Arrays

Most applications contain arrays of some kind. FORTH not only allows you to define kinds of variables other than CONSTANT and VARIABLE, it also allows you to define kinds of arrays that are optimized for a particular usage. Later on we will describe how you may define kinds of variables and arrays yourself (in Chapter 13.0, SPECIAL DEFINING WORDS). The simple use of VARIABLE itself, however, makes a very straightforward kind of array possible.

As your FORTH vocabulary is compiled, a system variable called H (which may be fetched by HERE) always points to the next available byte in your dictionary. Simply adding a number to H will cause that number of bytes to be skipped over in the dictionary, leaving space for the array. Therefore if you say

```
0 VARIABLE DATA 99 H +!
```

you will have defined DATA to produce the address of the first byte of 101 available bytes. The first two bytes will be initialized to 0; the remaining 99 contain undefined values. Here the command +! is used to increment H by 99. Thereafter,

```
20 DATA + @
```

will fetch the 16-bit value beginning at the 21st byte in DATA, etc. If you wish to access the bytes individually, you may use the commands C@ and C!. These fetch and store 8-bit numbers, although those numbers will be 16 bits wide while on the stack.

The words MOVE and ERASE are sometimes convenient to use with arrays of data. The command

```
source destination count MOVE
```

moves 'count' bytes of data from the locations starting at the source address to the locations starting at the destination address. The command

```
address  count  ERASE
```

zeros 'count' bytes of memory starting at the specified address.

In Chapter 5.0 (COMPILER) you will see how to run through arrays in a loop. Chapter 6.0 (BLOCK I/O) will suggest techniques for using FORTH blocks as "virtual memory" for data, a technique which offers great savings in memory if you have access to mass storage in your application.

3.4 USER Variables

There exists a special kind of variable known as a user variable. While normal variables will provide the address of data that is contained within their definitions, user variables point to a separate region of memory known as the user area. User variables are defined by the word USER in terms of their offset from the beginning of the user area. For example,

```
20 USER SCR
```

defines SCR to be the location of the 21st byte in the user area. The base of the user area is controlled by a system variable called U (sometimes in a register).

User variables were initially designed for multi-terminal systems in which each user of the system requires a private copy of the system variables (hence, the name USER). Your development system has only one fixed user area and one terminal. User variables are retained primarily to allow for mutli-programming as an option.

Caution: the user area on your system is of fixed allocation and contains important system variables (e.g., BASE, H). For this reason you may not define your own user variables! However, as the user area is in RAM in cross-compiled applications, it is sometimes a useful place to put a few application variables. Techniques for doing this are discussed in Section 16.2.4.

4.0 ARITHMETIC

FORTH does not attempt to provide a complete set of arithmetic and logical operators. Rather, it offers the operators most commonly useful and encourages the user to add any more he feels will make his particular problem more tractable. Those operators normally available include 16-bit integer addition and subtraction; mixed 8/16 bit integer multiplication and division; and some logical operators.

Remember that all operands for FORTH functions are on the stack, and that all results are left there. Numbers on the stack are all 16 bits wide (2 bytes). In this chapter, when we speak of 7- and 8-bit numbers, we are referring to the significant bits in the number, not to its size on the stack.

Two functions bear special discussion because they are very unusual. The operator */ multiplies a 16-bit number by an 8-bit number and divides by a 7-bit number. The intermediate product is 23 bits, so that the result is fully accurate to 16 bits. Similarly, */MOD multiplies a 16-bit number by an 8-bit ratio with a 23-bit intermediate product, giving a 16-bit quotient and 7-bit remainder. This makes possible accurate integer arithmetic without loss of precision from truncation errors. A detailed listing of the most common operators appears as Table II.

These operations are extremely useful for scaling and unit conversion. For example, suppose you have an application in which the internal unit of length is mils (.001"), but certain lengths need to be entered in millimeters. If you define MM thus:

```
: MM   250 63 */ ;
```

then to input 20mm, you could just type

ARITHMETIC OPERATORS

OPERATOR	DESCRIPTION	COMPUTES
+	Addition	$S1_{16} + S0_{16} = S0_{16}$
-	Subtraction	$S1_{16} - S0_{16} = S0_{16}$
*	Multiplication	$S1_{16} * S0_{8} = S0_{16}$
/	Division	$S1_{16} / S0_{7} = S0_{16}$
*/	Multiply - Divide (23-bit intermediate product)	$(S2_{16} * S1_{8})_{23} / S0_{7} = S0_{16}$ Sum can't exceed 23 bits.
*/MOD	Multiply - Divide (23-bit intermediate product)	$(S2_{16} * S1_{8})_{23} / S0_{7} = S0_{16}, S1_{8}rem$ Can't exceed 23 bits.
U*	Multiply	$S1_{8} * S0_{8} = S0_{16}$
U/	Divide	$S1_{15} / S0_{7} = S0_{8} > \quad S1_{8}rem$
MOD	Modulus	$S1_{16}$ MOD $S0_{7} = S0_{8}$
MINUS	Unary minus	$-(S0_{16})$
MAX	Maximum	larger of (S1, S0), signed
MIN	Minimum	lesser of (S1, S0), signed
<	Less than (truth value)	1 if S1 < S0, signed; 0 otherwise
>	Greater than (truth value)	1 if S1 > S0, signed; 0 otherwise
=	Equality (truth value)	1 if S1 = S0, signed; 0 otherwise
0<	Less than zero (truth value)	1 if S0 < 0; 0 otherwise
0=	Equal to zero (truth value)	1 if S0 = 0; 0 otherwise

The notation in the COMPUTES column above shows the stack position of operands. S0 is the top stack item, S1 is the item below, etc. The subscript gives each number's precession. Remember, all stack entries occupy at least 2 bytes. The high-order byte of an 8-bit number is 0. Except as noted, all multiplication and division are unsigned.

```
20 MM
```

and you'd have the length in mils on the stack. */ is particularly useful for computing percentages. Given

```
: %   100 */ ;
```

then

```
1275 15 %
```

gives you 15% of 1275. Similar definitions might be used to automatically calibrate measured data.

The use of these combined operators makes it rarely necessary to resort to floating point. This is nice since software floating point is slow and cumbersome while hardware floating point is expensive and rarely available on microprocessors. There is another cost, that of space versus accuracy. "Single precision" floating point numbers occupy at least 3 bytes and give only 4-1/2 digit resolution, while in the same 3 bytes you can carry nearly 7 digit resolution as an integer or fixed-point fraction.

Still, there are times when scaling is a problem: for example, when entering input or manipulating data.

Input is most easily handled by determining a reasonable scale and then scaling quantities internally with operators such as */MOD. Therefore you can work in tenths of seconds, millivolts, megahertz, or whatever unit is appropriate to the problem.

Data is best handled using "block floating point". This technique allows you to specify one scale factor which will apply to a block of data. Thus the data can be kept in as 16-bit integers, with a single number carrying the scale. All arithmetic performed on this data can then apply the scale factor as relevant. This technique handles the vast majority of data scaling problems.

FORTH's use of procedural logic (rather than descriptive or algebraic logic) may take a bit of getting used to. But it is extremely effective, even for very elaborate calculations. The important thing is to factor your definitions nicely, identifying like components and then defining these as operators. Choosing good names for these can make the resulting code not only compact but readable.

5.0 COMPILER

FORTH contains a simple and fast 1-pass compiler. It does not rearrange source code as does a FORTRAN or PLM compiler; rather, it generates strings of addresses of previously defined routines. It makes the task of writing efficient high-level routines extremely straightforward and allows complete control over logical flow.

The basic form of a definition is

```
: word   other words ;
```

Many examples are available--just look through the basic program listing. The syntax is as simple as possible. The first word following the colon is the word being defined. There is no punctuation except for the : and ; (remember, words are separated by spaces).

5.1 Literals

Any 8- or 16-bit number may be used directly in a definition. Such literals are compiled in-line as 3 or 4 bytes: the first 2 contain the address of a short routine to push the contents of the remaining byte(s) onto the stack. The longer, 4-byte literals are generated whenever the literal value is negative or larger than 255.

Literals are compiled using the current BASE, which must be set before starting a colon definition. If you include DECIMAL, HEX or OCTAL in a definition, BASE will not be set until the word is executed. Consider, for example,

```
DECIMAL   : A   HEX 20 ;
HEX       : B   DECIMAL 20 ;
```

Then A will put decimal 20 on the stack, and change BASE to 16; B will put decimal 32 on the stack and change BASE to 10.

5.2 Logical Flow

The cardinal rule that must be followed is this:

YOU MAY NOT USE ANY WORD THAT HAS NOT BEEN PREVIOUSLY DEFINED.

Remember, this is a 1-pass compiler. The same rule applies to the assembler (covered later in this text). This means that forward references (except in case of the IF...ELSE...THEN construction, which we'll get to shortly) are not allowed, a rule which has the effect of requiring you to modify your programming style to favor "structured programming".

Since FORTH encourages extreme modularity, your control of logical flow will mainly be through appropriate management of previously defined words in a definition. In addition, FORTH supplies commands for forming loops and two-branch conditionals.

5.3 DO-LOOPs

Most loops are constructed by using the beginning word DO (which expects 2 loop parameters on the stack) and one of two ending words, LOOP (which incrememts the loop counter by 1) or +LOOP (which increments it by the amount on the top of the stack, an amount which is then removed from the stack). The loop parameters are the initial value of the index, taken from the top of the stack, and the upper limit, taken from the second word of the stack. For example, to print out the numbers from 0 to 9, you might define PRINT as

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

These numbers are treated as unsigned (that is, 0 to 65535). The loop terminates

whenever the limit is reached or exceeded. The group of words between DO and LOOP will be executed once every pass through the loop. Because the test for termination is at the end of the loop (after incrementation), the loop will always execute at least once.

The two loop parameters are removed from the stack by DO and pushed on the return stack, which places certain restrictions on their accessibility. The loop counter may be accessed using the word I, which places the value of the counter on the stack. Remember that I is a verb! You may not directly modify it. In fact, the counter may not be changed at all except by the LOOPs. In nested loops, I provides the counter for the innermost loop. J similarly provides the counter for the next outer loop, if any.

Because the loop parameters are kept on the return stack, and because the return stack changes when you begin and end definitions, DO must be in the same definition as its terminating LOOP or +LOOP, as must I or J. Furthermore, you cannot use <R inside a DO...LOOP and still have access to the loop counter using either I or J.

Note: A DO...LOOP structure must only be used inside a definition.

Figure 2 considers a very simple loop in detail. There are also some examples below; try to understand how each works, referring to the Glossary for unfamiliar words. You may find it helpful to keep track of the stack on a piece of scratch paper.

5.3.1 Examples of DO-LOOPs

1. Suppose you want to write a loop to handle a variable number of items in some way. The easiest way is to specify the number of items as a parameter to the command containing the loop. You provide for this simply by omitting the needed parameter from the definition:

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

STEP	COMMAND		ACTION
1	10		10 pushed on stack (upper limit)
2	0		0 pushed on stack (starting index)
3	DO		10 and 0 put on return stack (stack clear).
4	I	Repeated	Pushes current value of loop index on stack (0, then 1, 2, etc. to 9)
5	.		Types out top of stack (0, 1, 2, etc.)
6	LOOP		Increments index, then compares with upper limit; if index < limit, returns to step 4. When index = limit, discards both and continues to step 7.
7	CR		Types carriage return; executed after loop has repeated 10 times for values 0 - 9.

EXAMPLE OF A LOOP IN ACTION

.... 10 0 DO I . LOOP CR ....

Figure 2

Here the limit was omitted. This means you must provide not only the numbers to be printed by . inside the loop but also the number of numbers. A reasonable use of the definition above might be for number conversions:

```
OCTAL  10  100  1000  10000  DECIMAL  4  PRINT
(Prints  4096 512 64 8 )
```

Caution: At run time, no check is made for stack underflow or overflow within a definition. Thus, stack mismanagement in loops can be repeated, and the accumulated errors may be fatal.

2. Sometimes you may want to specify both loop parameters at execution time. Again, you simply leave them out of the definition. As the "natural" order for specifying limits is "lower, upper", which is the opposite from the order DO expects, you might want to SWAP the arguments; moreover, since a loop terminates when the limit is reached after incrementation, you may often want to add 1 to the upper limit.

In the example below 1+ adds 1 to the top of the stack. Assume T will type out a specified line of text. The purpose of this definition is to print a range of lines.

```
: LIST   1+ SWAP DO  I T LOOP ;
```

Thus

```
0 15 LIST
```

will list lines 0 to 15.

3. This example shows how you may integrate a function over a specified range of values. Here we assume FX to have been defined to compute the value of a function of X where X is given as a parameter. The following example computes the sum of values over a given range:

```
: SUM   1+ SWAP DO  I FX + LOOP ;
```

To use SUM you would say,

```
0  100  200  SUM
```

The sum would be left on the stack. The limits are 100-200; the 0 is put on the stack as a starting value for the integration. SUM might be more convenient to use if you didn't have to think about the 0. INTEGRATE will supply it for you:

```
: INTEGRATE   0  ROT  ROT  SUM ;
```

Typing

```
100 200 INTEGRATE
```

will return the same value on the stack that SUM did.

4. Here is an example which shows how you may nest loops to provide two dimensions, in this case several rows of numbers:

```
: DUMP   SWAP DO  CR I 16 + I DO
   I ? 2 +LOOP  16 +LOOP ;
```

This takes a range of memory addresses on the stack and prints out the contents of that region of memory, as 16-bit numbers, 8 numbers per line. A multiple of 8 numbers will always be printed even when the requested range is not a multiple of 8.

Always remember: DO and LOOP must be in the same definition. Likewise, I may only be used in that definition (since entering another definition modifies the return stack).

5.4 BEGIN...END Loops

It is possible to use +LOOP to provide a loop which would run indefinitely, until a condition is met by giving a truth condition (0 or 1) as the increment to +LOOP. A rather more straightforward type of indefinite loop is provided by the commands BEGIN...END.

BEGIN takes no parameters, compiles nothing, and serves merely to mark at compile time the beginning of the phrase to be performed repeatedly. END takes one parameter; if it is zero (false) the phrase will be repeated; if it is non-zero (true) the loop will terminate and the next word after END will be executed. Here is a simple example:

```
: MONITOR   BEGIN READ OVER = END  DROP ;
```

In this example READ is assumed to be a function which reads a value from a device. This loop will monitor the device until it returns a value which is equal to the value which was put on the stack before MONITOR was executed.

NOTE: A BEGIN...END structure must only be used inside a single definition.

5.5 Conditionals

FORTH has a standard 1- or 2-branch conditional statement. The syntax for a 2-way branch is

```
condition  IF true phrase  ELSE false phrase  THEN continuation...
```

Omitting the ELSE and the false phrase produces a 1-way branch:

```
condition  IF true phrase  THEN continuation...
```

These structures also may only be used inside a definition. Use of the conditional statement is illustrated in these examples:

```
: EQUAL  A B =  IF 3  ELSE 9  THEN 1+ . ;
```

will print 4 if A and B are equal and 10 if A and B are not equal. A less naive use is

```
0 VARIABLE S  99 CONSTANT LIMIT
: SLIM  S @ LIMIT > IF  100  ELSE S @ 1+  THEN S ! ;
```

Remember that the IF will jump to ELSE or THEN if the top of the stack contains zero (false). Therefore, the IF inherently contains a "not equal to 0" test.

Note that IF will destroy its parameter. If the value on the stack to be tested by IF is also needed inside the IF...THEN clause, it must be DUPed before the IF. Thus in

```
DENOM @  DUP IF  NUMERATOR @ SWAP /  RATIO !  ELSE DROP THEN
```

the / divides NUMERATOR by DENOM and the DROP drops DENOM in case DENOM was 0. The word -DUP, which DUPs the top of the stack only if it is non-zero, may be used to do exactly the same thing without the need for an ELSE:

```
DENOM @  -DUP IF  NUMERATOR @ SWAP /  RATIO ! THEN
```

Several words have been defined to perform other tests for IF:

0= Replaces a number by 1 if the number was 0, or by 0 if the number was non-zero.

0< Replaces a number by 1 if the number was negative, by 0 if 0 or positive.

= Replaces two numbers by 1 if equal, otherwise by 0. Note that - is sufficient to test for not-equal.

< Replaces two numbers by 1 if the lower is less than the top number, otherwise by 0.

> Replaces two numbers by 1 if the lower is greater than the top number, otherwise by 0.

NOT Reverses the truth of the top of the stack (replaces non-zero by 0, 0 by 1. Same function as 0=).

Figure 3 illustrates some successful and unsuccessful ways of nesting structures.

5.6 Special Loops

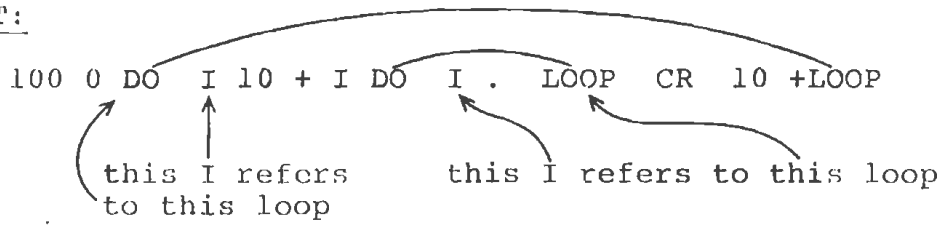
Occasionally a situation arises in which, for any number of reasons, one of the standard loop structures simply does not yield a satisfactory solution to a problem. For example, it is often desirable to terminate a DO loop before its full range has been exhausted, or to terminate a BEGIN...END loop by testing a condition at the middle or beginning of the loop. FORTH provides special help in these situations with the words LEAVE and WHILE.

5.6.1 LEAVE

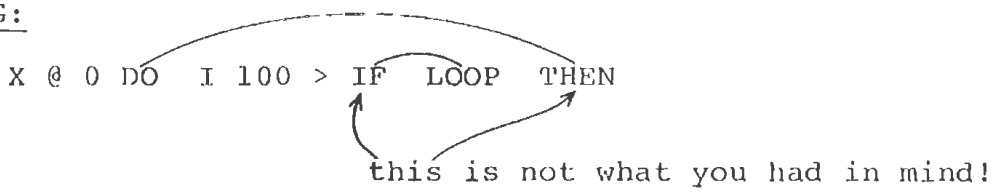
The main difficulty in the use of DO loops is the problem of prematurely terminating the range of the loop (as in search operations). Often this can be achieved by using +LOOP and passing it a very large increment. This presumes some knowledge of the range, however, because if the increment is too large the loop index will overflow and become quite small, achieving the opposite of the desired result (i.e., an infinite loop). Sometimes a BEGIN...END loop will suffice if a Boolean expression can be devised that properly expresses the two conditions for termination and if some substitute for a loop index can be left on the stack. In both of these examples the imposition of additional items on the stack can severely complicate stack management.

The word LEAVE provides an excellent means of ending the range of a DO loop. It does this by setting the loop limit to the current value of the loop index. Then, when the LOOP or +LOOP is reached, any positive or 0 increment will cause the limit to be met or exceeded and the loop ends.

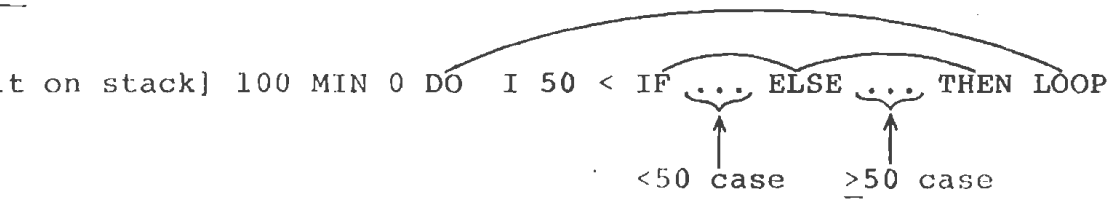
RIGHT:



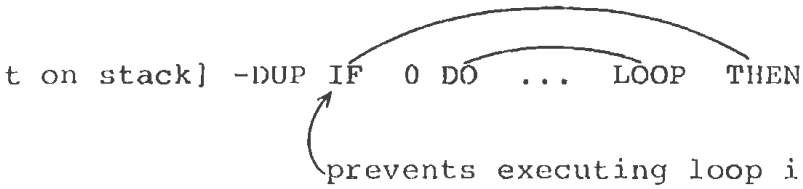
WRONG:



RIGHT:



RIGHT:



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

EXAMPLES OF NESTED STRUCTURES

Figure 3

NOTE: LEAVE must only be used inside a DO loop.

The following example scans a region of memory and converts all ASCII nulls to blanks until either a count is exhausted or an end of text (3) is reached. The count is on top of stack with the memory address beneath:

```
: SCAN   OVER + SWAP DO  I C@  -DUP IF
   3 = IF  LEAVE THEN  ELSE 32 I C!  THEN LOOP ;
```

The phrase OVER + SWAP is commonly used to convert a start address and count into a limit and initial value for DO loops. Note that, since LEAVE works by modifying the loop index, the loop will not actually terminate until LOOP or +LOOP is reached. That is, words between LEAVE and LOOP will be executed normally.

5.6.2 WHILE

WHILE is a word that combines some of the functions of both THEN and END. Syntactically, it is used as follows:

```
BEGIN ... condition IF  true phrase  ELSE false phrase  WHILE
```

or

```
BEGIN ... condition IF  true phrase  WHILE
```

The IF compiles as a conditional branch, as usual. If the phrase immediately before WHILE is executed, then the loop will be repeated. Otherwise the loop is exited. What this means in the first example is that when the IF condition is true, the true phrase is executed and then the loop is exited. As long as the condition is false, however, the false phrase is executed and the loop is repeated. In the second example a false condition at the IF immediately exits the loop, while if a true condition exists, the true phrase is executed and the loop is repeated.

NOTE: WHILE must only be used within a definition.

WHILE can be used to write loops with a pretest rather than a posttest (as in DO-LOOPs and conventional BEGIN-END loops). The pretest is performed by IF, and the body of the loop is between IF and WHILE. Note that the loop may be performed zero times with a pretest, as opposed to a posttest.

For example, suppose you have a queue which accumulates a backlog of actions to perform (the queue might be interrupt-driven). You have defined the words:

ANY	Places true on the stack if there is anything waiting in the queue
PERFORM	Performs the action at the head of the queue
ADVANCE	Advances the queue pointer to the next item

Then the following loop will perform all actions in the queue:

```
: EMPTY   BEGIN  ANY IF  PERFORM ADVANCE  WHILE ;
```

Note that if there are no actions pending, the loop exits immediately.

5.7 Special Literals

5.7.1 Use of [']

You have seen in earlier chapters how ' can be used to provide the address of the word following it. When ' is used within a definition, however, it is compiled and will not be executed until such time as the word in which it is used is itself executed. Thus the ' will get the address of the next word of the input string at execute time, not at compile time. This may be desirable; but if the intent is to get the address of a word compiled as a literal, a different word, ['], must be used.

['] is a word which the compiler executes at compile time. This word uses ' to find the address of the word which follows it and compiles this address as a 16-bit literal.

5.7.2 Use of IN-LINE

Occasionally you will find yourself compiling expressions that are invariant at execution time. That is, they will evaluate to a constant. This is an unfortunate waste of both time and memory. One solution would be to name these expressions as CONSTANTS. This is preferable whenever the value is to be used more than once. Alternatively, the word IN-LINE will remove a number from the stack at compile time and compile it into a definition as a 16-bit literal.This number must be computed and left on the stack before starting the definition and must be on top of the stack at the place where IN-LINE is used. All loops and conditionals also use the stack at compile time. This means that IN-LINE cannot appear inside a loop or conditional phrase and still have access to its intended parameter.

NOTE: ['] and IN-LINE must be used only within a definition.

5.8 Extending the Compiler

The one thing which all of the compiling words (DO, LOOP, +LOOP, BEGIN, END, IF, ELSE, THEN, WHILE, ['], IN-LINE) have in common is that they all execute at compile time to perform some compiler-related activity, rather than being compiled as an address, as FORTH words normally are. The ability to define such words enables the compiler to be arbitrarily extensible. This facility is implemented by means of the word IMMEDIATE (you may see examples in Screen 9).

The word IMMEDIATE is placed immediately following the definition of a new word that is added to the compiler It sets a special flag (called "Precedence") in the new word that prevents that word from ever being compiled into a later definition. Instead, when the compiler comes across this word in a definition, it is executed at

that time and must perform its functions explicitly. Take for example:

```
: [SWAP]   SWAP ;   IMMEDIATE
```

When [SWAP] is encountered in a definition, it swaps the top two items on the compile time stack. This can be useful for making a value destined for IN-LINE available inside of a loop or conditional phrase (since DO, BEGIN, IF and ELSE leave addresses on the stack at compile time).

Another important word that is used in the definition of compiling words is \ . Let us use as an example the definition of DO, defined in screen 9 as:

```
: DO   \ DO  HERE ;   IMMEDIATE
```

The DO which appears after the \ refers to a word that was precompiled in the initial boot load. This previous version of DO (referred to as the "code DO") is the one that, when executed, will move two items from the parameter stack to the return stack. This is the last time that the code DO is ever referred to explicitly because the new definition (called the "compiling DO") will supersede the earlier. The code DO has not been lost, however, because its code address has been compiled into the definition of the compiling DO. Then, whenever the compiling DO is executed, the word \ will compile the address of the code DO into the next free space in the Dictionary. Thus,

```
\ DO      is equivalent to   ['] DO 2 - ,
```

HERE then places on the stack the address of the byte following the newly compiled reference to the code DO. Later, LOOP or +LOOP will use this address in determining where the compiled loop should return to. Finally, IMMEDIATE is used to mark the compiling DO as being a compiler extension.

It should now be clear why compiling words must not be used outside of definitions. The reason is simply that compiling words can only compile. The intended function of DO is not performed by the compiling DO, but by the code DO that is compiled. In order to be executed, the code DO must then be compiled into a definition.

5.9 Memory Usage and Timing

The length of a : definition is very easy to determine. The colon and word generate the dictionary entry, which gives an overhead of 8 bytes. Thereafter, add 2 bytes for every defined word in the definition, including the semi-colon, and 3 for every IF, ELSE, END, WHILE, LOOP and +LOOP. Add 2 bytes for each DO and none for any BEGINs or THENs. Add 3 bytes for short literals and 4 for literals greater than 255 or less than 0. (IN-LINE and ['] literals are always long.) 0 and 1 are defined as constants; they take only 2 bytes. Timing depends on the execution time of the components of the definition (refer to the appendix for your processor). If you are trying to decide whether to define a phrase separately or include its functions in other definitions, you can assume that you break even in memory space if you will use the defined word 2 to 5 times; you will save length - 2 bytes for every subsequent usage.

Length of phrase:	5	6	7-13	≥14
Uses to break even:	5	4	3	2

The cost in time will be 2 interpreter cycles per usage--not very much for what can be extremely great savings in memory. For time-critical portions of the application you may use the FORTH assembler and work at full processor speed.

In cross-compiled applications, the first 6 bytes of every definition are stripped off, since they are needed only for systems with interactive terminals. You break even in space defining an eight byte phrase if it will be used twice.

6.0 BLOCK I/O

Disk I/O is handled by FORTH in standard blocks of 128 bytes. This fixed block size applies both to FORTH source text and to data taken by FORTH programs. (A complete screen, or unit of text for display and editing, consists of 8 contiguous blocks.) This apparent inflexibility may appear strange to programmers accustomed to designing specialized data formats, but in fact causes the entire problem of I/O to disappear behind one standard block handler. The block size chosen is a convenient, modest size. FORTH applications exist with several data records in a block, or with several blocks forming a data record. The FORTH word BLOCK is used to gain access to data blocks. BLOCK takes a block number off the stack and replaces it with the address of a buffer that contains the requested data, performing any reads or writes as necessary.

Most microFORTH systems are configured with eight disk block buffers, 132 bytes long. Each buffer begins with a 2-byte block status word which contains the block number of the data currently occupying the buffer (or 0 if empty). The high-order bit of this status word may be set to indicate that the buffer contents have been updated and must later be saved. The next 128 bytes comprise the data for this block. The FORTH word BLOCK is used to gain access to data blocks. BLOCK takes a block number off the stack and replaces it with the address of a buffer that contains the requested data, performing any reads or writes as necessary. The address returned by BLOCK points to the first byte of this area. The last 2 bytes are always 0. These bytes are very important and care must be exercised to never overwrite them. During the loading of a screen, they serve to stop the scanning of the current block and to pass control to the next block in the screen. The entire buffer area can be cleared to 0 by the word ERASE-CORE. Note that any buffers marked updated will never reach the disk if ERASE-CORE is used. This can be useful if you catch an error in a disk buffer before it is written back on the disk.

The only requirement for fitting data records into this structure is that data record numbers be a fixed function of block number; then a word can be defined that will use BLOCK to fetch the block(s) containing records requested by block number.

Here is an example in which data records are smaller than blocks. Three CONSTANTs have been defined: LR is the data record length in bytes, B/B is the number of active bytes per block, and START is the first block of the file.

```
: ADDRESS   LR B/B */MOD  START + BLOCK  + ;
```

ADDRESS replaces a "record" number on the stack by the address of the first word in the record, having fetched the record as necessary.

It should be noted that disk systems do not require any sort of directory in memory or on disk, as block numbers are a direct function of disk address (the exact relationship is designed to suit the particular disk involved). The fact that a block number is a fixed function of absolute sector address gives you the ability to allocate disk space in a fashion appropriate to your intended use to minimize head motion and thus improve performance. Applications involving management of complicated data file structures sometimes have a disk directory; this is a feature of the application, however, rather than a standard feature of FORTH. Disk blocks which are not in use should be flagged, normally by putting 0 in the first 2 bytes.

You may define named fields in a data record by using a definition such as ADDRESS, above. Suppose you have a variable R# for remembering a record number. Then you could define some fields thus:

```
: FIELD   R#  @  ADDRESS + ;
: OBJECT   2 FIELD ;   : TIME   6 FIELD ;   : DATE   8 FIELD ;
```

After this you may set R# and access these fields as though they were normal VARIABLEs, using @ and ! . You may even define an array:

```
: DATA   10 FIELD + ;
```

so that 1 DATA fetches the address of the second byte, etc.

The one thing you must be careful of in such a scheme is that when you store something in a data block, you must ensure that the block is marked as having been updated so that it will be written out in due course. The command which does this is UPDATE. UPDATE flags the buffer most recently accessed as having been updated. There are many ways of including UPDATE in store operations, of which the simplest is to define a special version of ! for data:

```
: !D    ! UPDATE ;
```

Then you use !D whenever you are storing into a data block.

6.1 Error Checking

BLOCK itself does not perform any error checking. The capability for error checking, however, is included in the disk utility, based on the following defined words:

ERROR — Returns the disk status as of the last operation, masked for error bits.

[BLOCK] — Used like BLOCK, but checks for disk read errors. [BLOCK] will read up to 3 times, issuing error messages, if errors are encountered, and tallying errors. It will also keep whatever it has after the last read. The error message reports the sector number.

You may wish to define a word similar to [BLOCK] which handles errors differently.

7.0 TEXT EDITOR

Although you may type in definitions at any time, they will be lost if you reload the program. Moreover, the source is lost forever--you cannot recall it to refresh your memory! There are two utilities supplied with microFORTH which allow you to maintain the text for your definitions in permament form on the disk. The EDITOR allows you to edit your definitions on the disk and modify the text; the PRINTING utility prints listings and indexes to text on disk.

As the normal 128-byte size is too small for coherent amounts of program source, the EDITOR uses screens (roughly the right amount of text for a CRT screen). A screen consists of 8 contiguous blocks of text, which will be formatted as 16 lines of 64 characters each for display and editing. Lines are numbered 0-15.

You may list a screen at any time by giving its number and LIST. For example,

```
13 LIST
```

lists screen 13. Screens that have names (defined using CONSTANT as described in Section 9.1) may be requested by name:

```
TRACK LIST
```

LIST not only lists the screen, it also sets its number in the variable SCR. The EDITOR uses SCR to "remember" which screen is being edited. Thus, you should LIST a screen before editing it, although you may omit this step by typing

```
screen#  SCR !
```

During editing, you may list your current screen by simply typing L.

7.1 Text Editing Utility

The EDITOR is not resident on the COSMAC system, but may be loaded by:

```
EDIT LOAD
```

On other systems, the editing Vocabulary is accessed by the word EDITOR. You will retain access to the EDITOR until such time as you compile a new word, after which you must retype EDITOR or EDIT LOAD to return to the EDITOR's vocabulary. In particular, if you LOAD a screen, you will leave the EDITOR vocabulary and return to working vocabulary. Vocabularies are discussed in Chapter 14.0.

The editing commands are

7 T	Type line 7 (place in line buffer). The line number is saved on the stack.
" text"	Place text in line buffer.
4 P text CR	Place "text" in the line buffer and then replace line 4 with the line buffer contents. Can be used to edit text that contains the " character. Because the text is terminated only by the carriage return, P must be the last command on a line. Note that the line number for P, if left over from T, need not be repeated.
7 R	Replace line 7 with contents of line buffer. If you are replacing the line just typed (with T), you need not repeat the line number, since T saves it. Normally used after " or D.
13 I	Insert buffer after line 13 (discard last line). Note that -1 I is a valid command, but 15 I is not.

13 D — Delete line 13 (place in line buffer). The last line will be reproduced and other lines will move up as needed. D should not be applied to line 15; blank it out instead.

Definitions edited into screens may span any number of lines (up to 16). Due to the fact that a screen is comprised of 8 separate blocks, however, no single word may be allowed to span between an odd and an even numbered line. For the purposes of this rule, a "word" shall include both bracketed text strings (described under "Output") and parenthetical comments. Subsequent lines of multi-line definitions are typically indented for readability.

No special action is required to edit a previously unused screen. An unused screen is filled with undefined characters. You edit in new lines by replacing lines with text. For example,

```
" THIS IS A NEW LINE " 1 R
```

puts THIS IS A NEW LINE in the line buffer and then into the second line of the screen. This can also be achieved by:

```
1 P THIS IS A NEW LINE
```

Note that P and " are ordinary FORTH words, which must be followed by a single blank before beginning the text. Any line entered by P or " will be padded with blanks at the end. When you finish typing in lines of the text, you should fill any unused lines with spaces. A blank line is defined by <u>at least two spaces</u> within quotes:

```
"  "  12 R  13 R  14 R  15 R
```

fills lines 13-16 with spaces.

Notice that you may string together Rs in this case, because the blank line remains in the line buffer.

To move a line, you may delete it (D places a line in the line buffer) and then use R or I. For example:

```
23 SCR !  8 D   24 SCR !  1 R
```

deletes line 8 of screen 23 and replaces line 1 of screen 24 with it.

Remember that the line numbers are current, ordinal numbers. I and D will renumber the remaining lines in that screen.

An additional word available in the EDITOR is COPY, which is used to copy all 8 blocks of one screen to another.

```
Usage:    source   destination  COPY
Example:  5 105 COPY          (Copies screen 5 to screen 105.)
```

7.2 Program Listing Utility

The PRINTING Utility is used to list source text screens and indices of first lines of a range of screens. It is loaded with the following command:

```
PRINTING LOAD
```

The commands described below are appropriate for use in a printing terminal.

7.2.1 Index listings

PRINTING will produce an index listing, which shows the first line of each screen in a given range of screens.

Generation of an index is specified by the following command:

```
start end INDEX
```

where 'start' is the starting screen number and 'end' is the ending screen number plus one. The index will be formatted 60 lines per page. Should the range of screens be less than 60 or not an even multiple of 60, the last page will be partially filled.

7.2.2 Program screen listings

To list an entire range of screens, use the following command:

start end SHOW

SHOW lists screens three per page, starting each page with a screen number evenly divisible by 3. This means that you may replace an individual page, rather than always having to list an entire application. Only entire pages will be printed, in sufficient quantity to cover the requested screen range. Unused screens will not be listed. On a partially used page, space will be left for unused screens. An unused screen, by definition, contains nulls in its first two bytes. Pages with no used screens will be skipped.

To list a single page, use the following command:

scr# TRIAD

where 'scr#' is the screen number of any screen on the desired page.

8.0 OUTPUT

Most microprocessor applications need only relatively simple numeric output. This chapter describes simple techniques for producing attractive output. Some of these are included in the standard system; others have been found useful in some applications and are offered as suggestions.

8.1 Right-Adjusting Numbers

An important capability for numeric output is the ability to right-adjust a number in a field of fixed width. There is a standard word available for this: .R prints a 16-bit integer right-adjusted in a field of specified width. As an example,

```
30  5  .R          prints    __30
```

Using this word, all you have to do to print a table of numbers in columns is to set up a loop for the number of columns desired across the page. The definition of DUMP, which dumps a region of memory when given the starting address and length, is a good example:

```
: HALF   SPACE SPACE  8 0 DO  DUP C@  3 .R  1+ LOOP ;
: DUMP   0 DO  CR  DUP 5 .R  HALF HALF  10 +LOOP SPACE  DROP ;
```

You should note in this example the use of HALF to print each line in two sections for readability. An example of this output is provided by Figure 4. CR provides a carriage return and line feed. DUMP is available in all systems.

```
0 100 DUMP
   0   E0 90 B1 BF F3 10 A1 41   AF 41 EC 41 AC 41 ED 41
  10   AD 41 EE 41 AE DF 1E 1E   E4  E 73 1E 33 D3 4D B9
  20   4E A9 49 E3 49 A3 30 1D    0 2A 42 BE 42 22 AB 4B
  30   EA 4E 2E 5E 3E 52 9B 22   52 9A 2E 5E DF  0 3F 4D
  40   2E 5E 9F 2E 5E DF  4 45   4C 53 80  0  0 4E ED 3D
  50   F4 AD 3B 53 9D FC  1 BE   DF  2 49 4E 20  0 49  0
  60   61 1E 4E 32 4E 1D DF  2   44 4F 20  0 5C  0 6F 1E
  70   4E AE 1E 4E 22 52 3B 22   52 9F 22 52 DF  4 4C 4F
  80   4F  0 6A  0 35 E2 12 42   FC  1 F7 3B 90 12 1D DF
  90   F4 22 52 22 ED 3D F4 AD   33 9E 9D FF  1 BD DF  3
  A0   45 4E 44  0 80  0 A7 1E   4E 32 94 1D DF  5 2B 4C
  B0   4F  0 A2  0 B5 1E 4E E2   12 F4 12 F7 30 3B  3 43
  C0   4F 4E  B E0  1 40  B 3E    6 61  B BA 49 BE 49 2E
  D0   5E 9B 2E 5E DF  4 55 53   45  0 C1  1 40  B 3E  6
  E0   4D  B BA 3C E9 F4 2E 5E   9C 2E 5E DF  2 3B 3A 20
  F0    0 D3  1 40  B 3E  2 DF    6 61  6 4D  B BA 3D 22 OK
```

Figure 4

8.2 Custom Number Formatting

FORTH provides convenient custom number formatting at high level. The basic procedure assumes that an unsigned 16-bit number to be converted is on the stack. If a sign is to be attached to the number, a signed copy of the number should be in the second stack position. Successive digits are computed as remainders modulo BASE, converted to ASCII characters and placed in an output string. The declining quotient is kept on the stack. Special characters (such as decimal point) are placed in the output string between the conversion of the appropriate digits. Note that the least significant digit is generated first, so the output string is generated in the reverse order from the way it will appear.

Here are the words for number formatting:

<# Initializes the number conversion process by resetting the pointer to the output string.

Converts one digit from an unsigned number on top of the stack and puts it into an output character string, leaving the remainder of the original number on the stack. Always produces a digit whether or not there are remaining significant digits in the number.

#S Converts successive digits until the result is zero. Always produces at least one digit (0 if the value is zero).

SIGN Tests the sign of the number beneath the top stack item. If negative, inserts an ASCII minus sign into the character string. Removes the signed number.

HOLD Inserts, at the current position in the character string being formatted, a character whose ASCII value is on the stack. HOLD must be used between <# and #>.

#> Completes number conversion by dropping the resident number from the stack and leaving there the character count and address

(these are the arguments for TYPE).

These words are all defined in Screen 12 if you wish to study them.

For example, the word .D is often defined to print an integer with a specified number of decimal places:

```
12345 2 .D   prints   123.45
```

The definition of .D might be

```
DECIMAL
: '.'   46 HOLD ;
: .D   <R  DUP ABS  <#  R> 0 DO  # LOOP '.'  #S SIGN #> TYPE ;
```

Here <R saves the decimal places count, while DUP ABS prepares an unsigned number with the sign beneath. The word '.' inserts the decimal point in the output buffer. Note that the digits after the decimal point are generated first, in the DO-LOOP.

All number conversions are made using BASE as the base. Thus, if you had the time in seconds and you wanted a word which would print hh:mm:ss, you might define it this way:

```
OCTAL  : ':'  72 HOLD ;
: :00   #  6 BASE C! #  ':'  DECIMAL ;  DECIMAL
: .SEC   <#  :00  :00   # # #> TYPE SPACE ;
```

Here changing BASE was all that was needed to get the leading digit of seconds and minutes to print and carry properly.

You will note the explicit use of the word TYPE everywhere. The implication of this is that if you want to send your text string to a device other than the terminal, you may simply substitute another output command for TYPE. If you are doing this, you'll want to leave TYPE out of the formatting words themselves and put in a separate definition, as has been done with . itself:

```
: (.)   DUP ABS  <# #S SIGN #>  ;
: .   (.) TYPE SPACE ;
```

Thus, if WRITE were defined to write on some other ASCII device, you could define .W thus:

```
: .W   (.) WRITE ;
```

then

```
SEC @ .W
```

would write the value of SEC on that device in ASCII characters.

8.3 Text Output

Text for titles and remarks is best kept on disk, if disk is available. On disk systems the word MESSAGE is defined to type out a specified line counting from the 0th line of a screen 23. For example, 16 MESSAGE prints out the 0th line of screen 24. Messages are printed till the last nonblank character of the line, plus one trailing blank. Each message is up to 64 characters long, so there are 16 of them in a screen. Messages are put in the screens using the EDITOR.

If you need titles wider than 64 characters, you may write a definition of TITLE similar to that of MESSAGE, with a suitable width. Perhaps you would want to add a CR (which types a carriage return and line feed) before or after the title; since MESSAGEs may need to appear anywhere, there is no CR in the definition of MESSAGE.

For applications for which disk will not be available, text strings must be compiled into memory. This is accomplished by the word [.

The word [is placed inside a definition followed by text terminated with the] character. Remember that [is a word and must be followed by a space that is not part of the text string. The text between the [and the] is inserted into the definition preceded by a reference that will cause it to be typed and skipped over.

NOTE: [may only be used inside a definition.

To see an example of [in use, define

```
: HELLO?   [ HUMBUG!] ;
```

then type HELLO? and observe the response.

Due to the fact that a "screen" of source text occupies 8 blocks (sectors) on disk, an additional rule must be observed when using [in source text: a text string inside a [structure must not extend across sector boundaries. This means it may not extend off the end of an odd-numbered line into an even-numbered line.

The use of bracketed text output does not elegantly lend itself to the generation of strings containing special control characters. The word MSG is used to generate output strings with special characters in a more visible way. Take, for example, the definition of CR, which outputs a carriage return and line feed:

```
HEX
MSG CR  6 C,  0D C,  0A C,  0 ,  0 ,
DECIMAL
```

The 6 gives the length of the string which is explicitly compiled (using C, or ,) after the definition. The four 0 bytes are provided as timing characters as needed by some terminals.

9.0 FORTH PROGRAMMING TECHNIQUES

Since FORTH is interactive, you will spend much more time at your terminal and less at your desk than with non-interactive techniques. You will generally want to write down some notes about the problem you are about to solve, perhaps, and a few lines of program. If it is a big problem, you will want to outline your proposed program in some detail. Then you sit down at a FORTH terminal and type. Your procedure will be to enter a definition or two, test them to your satisfaction, and then combine them to form more powerful definitions, until the problem is satisfactorily described. To keep your definitions permanently, you may edit them into a screen or more of a source text which will be kept permanently on mass storage. You may modify these screens, load them, and re-test.

9.1 Overlays

To facilitate testing (and also to allow mutually exclusive sub-vocabularies to replace one another), you may wish to mark a place in the dictionary with a null definition, so that at some future time typing FORGET and the name of the null entry will cause all of the dictionary generated since that entry to be discarded (or "forgotten"). Thus, when you begin typing provisional definitions, it is advisable to type something like

```
: TEST ;
```

Later, when you are ready to reload your test definitions, or if you feel the dictionary is becoming too cluttered, you may type

```
FORGET TEST
```

and everything entered in your dictionary beyond (and including) TEST's location will go away. A word may be re-defined as often as you like--the most recent entry will be the one used thereafter--but the obsolete entries remain, taking up space. Alternatively, you may FORGET any normal dictionary entry, again discarding that entry and everything following it.

If the compiler should generate an error message before reading the ; at the end of a definition, then you will not be able to forget that definition. This happens because the name of the current definition is incomplete until it is patched up by the ; .

By being able to forget collections of words you can create sub-vocabularies to be overlaid by other sub-vocabularies. The cross-compiler is such a sub-vocabulary, as is the EDITOR on the COSMAC. You may want to have several, each containing an application you're working on. In the common vocabulary one will want to give a name to the first screen of each:

```
43 CONSTANT CALCULATOR        83 CONSTANT IGNITION
120 CONSTANT COUNTER          180 CONSTANT ROBOT
```

and at the end of the common vocabulary a null definition:

```
: TASK ;
```

Each of these screens will load the other screens that are included in the same sub-vocabulary. The beginning of each initial screen will contain:

```
FORGET TASK    : TASK ;
```

The FORGET TASK will discard any of the other sub-vocabularies that might be loaded (the null definition of TASK takes care of the case when none is loaded). Then the new definition of TASK marks the beginning of this sub-vocabulary so that it might be discarded later on. In use, one can change overlays easily by typing:

```
CALCULATOR LOAD
```

or

```
ROBOT LOAD
```

without having to worry about discarding an incompatible set of routines.

9.2 Diagnostics

When you type a definition, or use an untested definition, or load a newly edited block, you may get a diagnostic. Diagnostics are very simple. There are only 3 standard ones. The first is

```
word ?
```

This means that "word" is undefined. That is, it could neither be found in the dictionary nor converted as a number. You may have forgotten to define it, or loaded something which referenced it before it was defined, or simply misspelled it.

The most common diagnostic is

```
STACK EMPTY!
```

This means that either the word you typed or one it used expects a parameter on the stack and finds none (stack underflow).

The third diagnostic is

```
DICTIONARY FULL!
```

which is given whenever the top of the dictionary comes within a certain distance of the top of the stack. When you receive this message, you should consider such expedients as placing arrays in virtual memory (disk), and organizing your application into overlays.

FORTH checks for stack underflow and dictionary overflow after interpreting each source word of the input stream, either from terminal input or while loading a

screen.

Each of these diagnostics uses the standard abort routine, QUESTION. QUESTION repeats the offending word and issues a MESSAGE whose number is a parameter to QUESTION. It then empties both stacks, pushes the current block number onto the parameter stack, and quits (i.e., awaits keyboard input). If the error occurred within keyboard input, the top of the stack will contain a 0. If you were loading screens at the time, you may now type . (period) to find out the block number within the screen being loaded at the time the error occurred.

You may use QUESTION for your own diagnostics (provided you have a disk and terminal in your application) by creating your own error MESSAGE (see Section 8.3) and using its number as a parameter to QUESTION.

9.3 Testing

When you are testing a new definition, it is a good idea to type . after executing it to make sure there are no numbers left on the stack except those you expect to be there. A definition that accidentally leaves numbers on the stack can cause subtle and unpredictable things to happen in entirely unrelated parts of the program! Remember the rule that ALL WORDS SHOULD DESTROY THEIR PARAMETERS AND LEAVE ONLY EXPLICIT RESULTS.

Similarly, whenever you are compiling a new definition or loading a newly edited screen, be sure to start out with an empty stack; then check to see the stack is still empty when you are done. Extra items left on the stack are a sure sign of an IF with no terminating THEN, a BEGIN with no END, or a DO with no LOOP. On the other hand, assuming the stack started out empty, too many ENDs or THENs will terminate compilation with a STACK EMPTY! message. The best way to empty the stack is to type . until you get the STACK EMPTY! message and then type . one more time to remove the item left by the error message.

If your definition doesn't work and the examination of the text doesn't reveal the problem, the standard procedure is to type the words which are used in the definition until something goes wrong. You may monitor the behavior of the stack along the way by typing out the numbers on it until it is empty, then typing those

numbers again to put them back (in the right order!). Of course you must simulate the behavior of DO, BEGIN, or IF structures.

You may wish to define a word to dump the stack contents non-destructively, thus:

```
: ?S   'S  S0 @   OVER -  DUMP ;
```

Because DUMP dumps bytes whereas the stack contains 16-bit numbers, you may wish to define your own DUMP that will output 16-bit values and use it in ?S.

As repeated stack underflows or overflows may be fatal, it is strongly advised that you test loop contents carefully before running the loop.

Do not try to execute too many levels of untested definitions at once--multiple errors can so muddy the waters as to make debugging extremely difficult! Test the lowest levels of new definitions thoroughly before testing words that use them. FORTH's extreme modularity keeps debugging very simple if you always follow this rule.

9.4 Top-Down Design

Although you do test programs--and load them--from the lowest to the highest levels of complexity, you should try to design and write them in top-down fashion. Defining a bunch of low-level words that you think "should be useful" and then trying to integrate them is a sure way to waste time and effort! Suppose that you decide that you want to type

```
5 PHOTOS
```

to make 5 photographs with a processor-controlled camera. Naturally, you will want the definition of PHOTOS to contain a loop in which the key word is PHOTO--which takes one photograph:

```
: PHOTOS   0 DO  PHOTO LOOP ;
```

PHOTO will have certain fairly well defined things to do: open the shutter, time

the exposure, and close the shutter, for example. So PHOTO might look like this:

```
: PHOTO   OPEN SEC @ EXPOSURE  CLOSE ;
```

Here you have introduced a new operational element, the setting of exposure time variable SEC. So now the full operational sequence is

```
10 SEC !  5 PHOTOS
```

Then, depending on the way the hardware is set up, you might define OPEN and CLOSE something like this:

```
: OPEN   0 SHUTTER ;      : CLOSE   1 SHUTTER ;
```

where SHUTTER must be defined to send the specified code to the camera shutter.

The process of generating the definitions necessary to perform an operation tends to be very much the same as illustrated here regardless of the actual application. The important things to note here are:

(a) The top-down method of organizing the definitions (even though they must be tested and loaded in reverse order); and

(b) The extreme simplicity of each level in the process. Each single operation should be defined separately, and all should be kept as simple as possible.

Not only will this make things much easier for you during development, the availability of the lower level definitions will come in handy when some modifications are needed or when there is equipment trouble.

10.0 THE FORTH DICTIONARY STRUCTURE

The dictionary is a linked list of variable-length entries. It grows toward high core and each entry points to the one that precedes it. The beginning of the last entry is pointed to by the variable CONTEXT. It identifies the head of the chain to be searched. The next available word is pointed to by the variable H and may be put on the stack by the word HERE.

The dictionary is searched by following the chain until a match is found or the bottom reached. This organization permits a word to be redefined, since the latest definition will be found first.

As Figure 5 shows, the dictionary can be rather naturally divided into three parts:

> The PROGRAM is pre-compiled (on the same computer or any other FORTH computer), and contains about 60 defined words from which all other words can be defined. It is difficult, and normally unnecessary, to change these words. In some microprocessor systems the program resides in ROM.
>
> The FORTH VOCABULARY is compiled when you load FORTH. It is common to all applications, and though you may change it as you wish, you probably won't.
>
> The APPLICATION VOCABULARIES contain those words peculiar to your applications. You will be changing, rearranging, and adding to these vocabularies continually during the development process.

From the point of view of the search algorithms, these vocabularies are indistinguishable. You can, however, distinguish them--and other

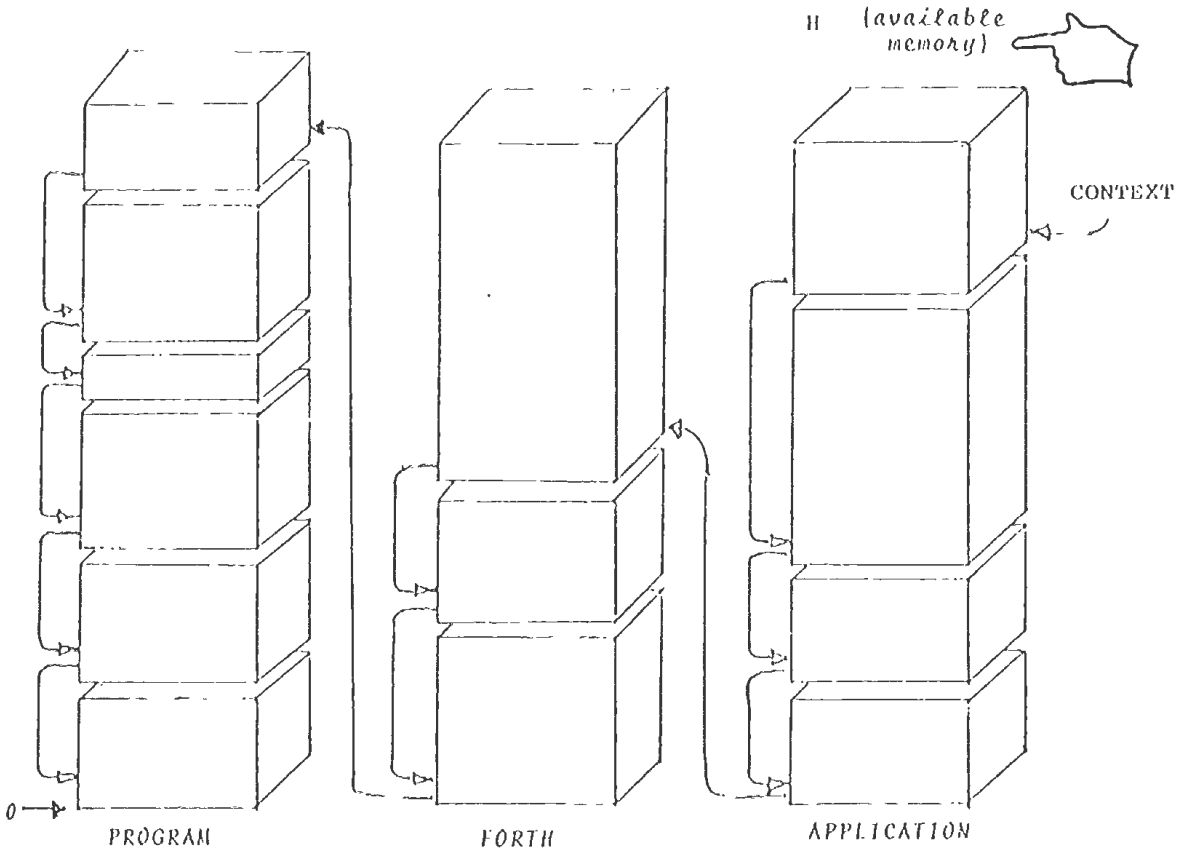


Figure 5

FORTH Dictionary

A compiled dictionary contains segments of logically related definitions, which in turn may be thought of as divided into three major groups.

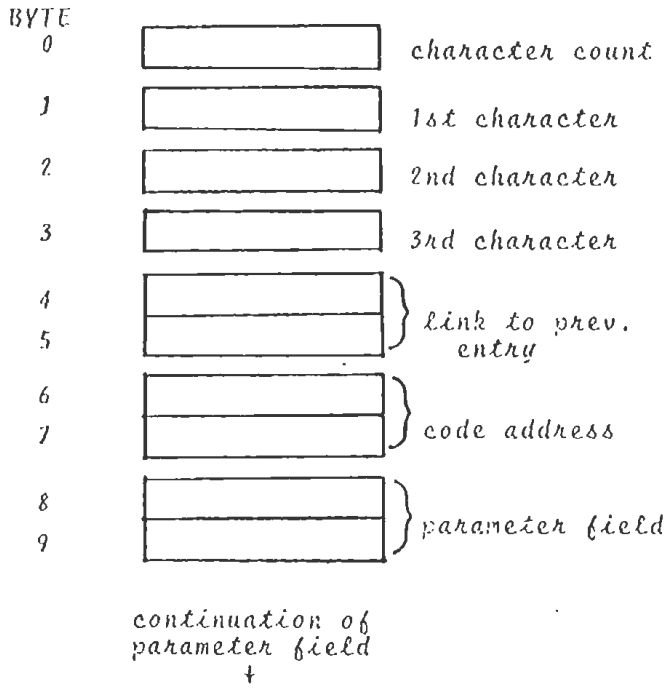


Figure 6
8-Bit Dictionary Entry Format

subvocabularies--by being able to discard them. For example, you might discard one application vocabulary and replace it with another as illustrated in the chapter on programming techniques. All vocabularies are linked to the central FORTH vocabulary, which means that a search will start at CONTEXT and thread back to FORTH, then through it.

The essential structure of all dictionary entries is the same regardless of the type of entry (nouns, verbs, etc.). This structure is diagrammed in Figure 6. The first 4 bytes are called the name field and contain the count of the number of characters and the first 3 characters of the word. Note that although this gives you far more flexibility in naming words than a simple limit on characters, it does require uniqueness in the first 3 characters of words of the same length. Note also that any characters you can type on your terminal are valid for use in words being defined.

The next 2 bytes, called the link field, contain the location of the first byte of the next previous entry. This is to facilitate searches, which start at the "recent" end of the dictionary and work back. This searching order is necessary in order that the most recent definition of a word will be the one used. Also, since in a developed application the user is dealing with the highest level of the program, it optimizes search time. Finally, although this is less relevant in microprocessors than in minicomputers, this searching order permits a "tree" of user Vocabularies, coming together at the trunk FORTH. (See Chapter 14.0, VOCABULARIES).

The next 2 bytes contain a pointer to the code to be executed for the definition. This code address depends on the type of word:

> For a CONSTANT, the pointer refers to code that puts the value of the constant (which is in bytes 8 and 9 of the definition) on the stack (see Figure 7).
>
> For a VARIABLE, it refers to code that puts the address of the value (byte 8) on the stack.
>
> For a : definition, it points to a portion of the interpreter, which will begin following a string of addresses in byte 8 and continuing until the ; which terminated that definition is encountered. A diagram of a :

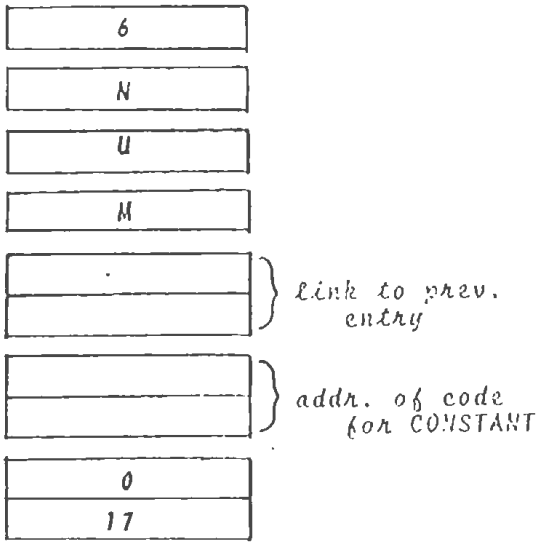


Figure 7

Dictionary entry for
17 CONSTANT NUMBER

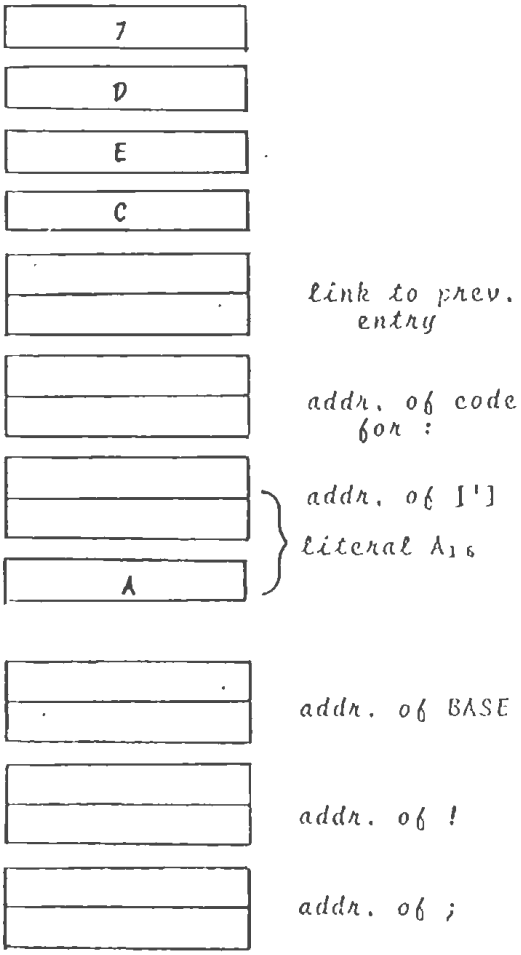


Figure 8

Dictionary entry for

: DECIMAL A BASE ! ;

(Stores 12_8 in the parameter field of BASE).

definition is shown in Figure 8.

For CODE, the pointer is to byte 8 itself, which contains the beginning of the code, which is simply executed directly.

Other kinds of entries have code addresses that point to the appropriate code and will be discussed in Chapter 13.0 (SPECIAL DEFINING WORDS).

The eighth and subsequent bytes are sometimes called the parameter field, which is of variable length. CONSTANTs and VARIABLEs keep their values in bytes 8 and 9 as noted above. Other kinds of words may keep several values. In the latter cases, the length of the parameter field is either determined by the type of word or is kept in one of the early words of the field.

11.0 THE INTERPRETER

Everything FORTH does is controlled by its interpreter. The interpreter itself is quite small. But it controls several important routines, some of which are invisible to the user, including number conversions, dictionary searches, generation of dictionary entries, management of the stack, etc. Some of these functions are very intricate. In this chapter we will explain what the interpreter does and how you will use it.

11.1 Interpreting a String of Words Typed at the Terminal

Your main communication with FORTH is through a terminal. You type one or more words which are interpreted and obeyed. If you have requested something to be typed out, it will be; whatever you have requested, FORTH will cheerfully reply OK. The OK not only notifies you that your request has been satisfied, it also signifies that FORTH is ready for you to type more commands.

The operations that have been performed before the OK reply consist of the following:

1) A word is taken from the terminal line buffer, which is also called the input string. (Remember, a word is a character string that is bounded by spaces).

2) a. If it is a word which can be found in the dictionary, the code for that word is executed.

   b. If it is a number, it is converted to binary, using the value of

the parameter BASE as the base, and pushed on the stack.

c. If neither of the above is possible, the word is sent back to the user with a ? .

3) Following successful completion of 2a or 2b, the interpreter continues on to the next word, if any. At the end of the string of words, it says OK. Therefore, you may string together a number of commands and they will be interpreted and executed, one at a time.

The actual work of taking a word from the input buffer is performed by the routine called WORD. WORD places the word, starting at HERE, with the count of characters in the first byte. This is the same initial structure as the name field of the dictionary definitions. The actual work of searching the dictionary is performed by -FIND.

WORD and -FIND are used by the command ' (tick), which was mentioned in Chapter 3. ' reads the next word in the input string and looks it up in the dictionary, returning on the stack the address of the parameter field of the entry if one is found.

11.2 Interpreting Source Blocks

If you type a load command such as

```
63 LOAD
```

the interpreter operates in a slightly different mode, since in the process of executing the word LOAD it must interpret words in a text screen read from disk or tape, rather than from the terminal line buffer. It keeps track of what screen it is taking its LOAD instruction from; the terminal, for this purpose, is "screen 0". The only difference in interpreter behavior between operating on "screen 0" and any other screen is that only "screen 0" gets an OK on completion. Note that just as "screen 0" may load a screen, that screen may in turn load other screens. The number of the current screen being interpreted is kept in the user variable "BLK".

Normally this method is used to load (compile) vocabularies into memory. But this same technique may be used to perform operations which are infrequently performed and not especially time-critical, and thus not worth allocating memory for.

11.3 Compiling Definitions

If the interpreter encounters a command, it will be executed. The defining word : causes the interpreter to behave in a special way. A colon not only generates the beginning of a dictionary entry for the word immediately following the colon, it also sets a flag called STATE for the interpreter. Thereafter, when the dictionary entry for a subsequent word in the input string has been found, its precedence bit (upper bit in count field) will be compared to STATE. If the precedence is less than STATE, the word will not be executed--its address will be placed in the dictionary entry being compiled. The use of IMMEDIATE for increasing the precedence of a word is discussed in Section 5.8.

11.4 Executing Definitions

The code address for : definitions points to a routine which sets FORTH's interpreter pointer, I, to the parameter field of that definition, and executes NEXT. (Please don't mistake this I for the I that fetches the DO-LOOP counter or the EDITOR's I.) The interpreter's I is only referenced in assembler code. NEXT is the most fundamental routine of the interpreter (sometimes called the inner interpreter). It is the basic loop that goes on to the next word, controlled by I. When a : definition is being executed, the interpreter is going down the addresses supplied in the definition, then going off to execute whatever is at those addresses. Of course, these addresses might point to other : definitions, and so on; therefore the code for : also saves the current I on a special push-down stack called the "Return Stack". Ancillary uses of the Return Stack were discussed earlier. This is the primary function of the Return Stack. The code for ; , therefore, pops I off the top of the Return Stack before returning to NEXT. Of course, at the top of any of these chains of pointers is a CODE definition, and all CODE definitions also end by executing NEXT (often after passing through stack manipulating routines). It is clear, then, that the gretest overhead in running FORTH is in this interpreter loop; great pains have been taken to code it as tightly

as possible. On most microprocessors, NEXT takes 10 or so instructions. It represents an ideal opportunity for a microcode instruction.

The existence of compiled addresses in definitions is the feature that distinguishes FORTH from other interpretive languages such as BASIC, which can only interpret from the terminal or from source, and are therefore much too slow for many real-time tasks and complicated computations.

It should be clear from the foregoing that compiling a : definition into the dictionary entails as many dictionary searches as there are words in the definition; executing a : definition (by typing it at a terminal) involves only one dictionary search. This is the reason high-level FORTH runs so much faster than purely interpretive languages such as BASIC.

11.5 Inner Interpreter Control

FORTH provides access to the inner interpreter through the word EXECUTE. What EXECUTE expects on the stack is the address of the parameter field of some definition. There is no restriction as to the kind of definition it may be--high-level, noun, or code. What EXECUTE does is to remove this address from the stack and direct the inner interpreter to execute/interpret the definition referred to. Note that the end result is exactly the same as though a reference to the "executed" word had been compiled in place of the word EXECUTE.

EXECUTE provides a simple yet powerful means for implementing flexible control structures that are handled with greater effort by FORTRAN-like languages with such techniques as computed GO TOs and externalized function calls.

The following example illustrates the use of EXECUTE to implement a transfer vector:

```
' +  VARIABLE FUNCTION  ' - , ' * , ' / ,
: MATH   DUP + FUNCTION +  @ EXECUTE ;
```

The location of the parameter field of + (given by ' +) occupies the first 2-byte entry in the table FUNCTION. The locations of - , * , and / occupy the second through fourth entries and are compiled explicitly using , . Typing

```
3 9 2 MATH
```

would yield 27, the product of 3 and 9. Within the definition of MATH, the DUP + changes the index to a 2-byte index, FUNCTION + adds this index to the base of the FUNCTION table, and @ EXECUTE fetches and executes the intended word. This kind of technique is especially useful when the function index is a computed value, as might be the case in decoding keyboard inputs.

12.0 THE ASSEMBLER

Forth can assemble machine-language definitions of words. Among the many examples of words defined by machine-language instructions are the arithmetic operations:

```
+  -  SWAP  DROP  U* U/
```

Such words are called code definitions and are constructed using the assembler command CODE. The assembler is not intended for conventional programs. FORTH code routines are distinguished by the fact that all end by returning to the inner interpreter rather than by executing a conventional subroutine return. The assembler for your particular CPU is detailed in an appendix to this manual. This chapter provides a general overview of the assemblers on all FORTH systems.

CODE entries have a standard dictionary entry with the code address pointing to the next byte (the parameter field) where machine instructions are assembled.

To compile a colon definition, the interpreter enters a special compile mode, in which the words of the input string are not executed (unless designated IMMEDIATE) but rather their addresses are placed sequentially in the dictionary. During assembly, on the other hand, the interpreter remains in execute mode. The mnemonics of the processor in question are defined as words, which, when executed, assemble the corresponding operation code at the next location in the dictionary. As elsewhere in FORTH, operands (addresses or registers) precede instruction mnemonics. Depending on the processor, several kinds of instructions and addressing are possible. These are defined in the FORTH assembler for each processor, to assemble instructions in the appropriate format, given the mnemonic operation code and whatever additional parameters are necessary to describe the instruction. The instruction is assembled into the next available location in the dictionary.

For example, the 8080 processor has an ALU reference instruction format for instructions that perform arithmetic computations. The FORTH assembler defines the command ALU, which is used to define mnemonics of the ALU class, which in turn assemble ALU reference instructions. For example, the mnemonic ADD is defined on 8080 systems by

```
80 ALU ADD
```

Then ADD is an operation which assembles an ALU type instruction whose numeric code is:

```
80      (hexadecimal)
```

and whose operand will be on the stack. In use,

```
L ADD
```

assembles an instruction which, when executed, will add the contents of the L register into the accumulator.

It is necessary when you are using CODE to separate in your mind the way you are using the stack at assembly time and at execution time. The words in a code entry are executed at assembly time to create machine instructions which are placed in the dictionary to be executed themselves later. Thus,

```
HERE  2 -  TST
```

at assembly time places the current dictionary location on the stack (HERE) and decrements it by 2. The resulting number is then the parameter for TST which assembles a machine instruction which is the equivalent of

```
TST  *-2
```

in conventional assembler notation. Similarly, such words as SWAP and DUP are executed at assembly time to manipulate the parameters being used by assembler words, although they are compiled into the dictionary in : definitions.

12.1 Code Endings

Code must end with a jump to the inner interpreter. This is a special routine called NEXT, because its 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 Table III. Not all of these are available on all processors; consult the appendix for your processor.

12.2 Notational Conventions

Although the FORTH assembler uses, for the most part, the manufacturer's mnemonics, there are some standard FORTH notational conventions that are shared by all assemblers. Fundamental FORTH pointers have standard names:

S	Address of top of parameter stack.
W	Address of parameter field of the current definition.
I	Interpreter pointer.
R	Address of top of return stack.
U	Beginning of user area.

These are registers if possible, but may be in memory. Refer to the appendix for a discussion of their allocation on your system. Wherever they may be, these names may be used in code to refer to the areas.

In definitions beginning with : the execute-time manipulations of the stack are

TABLE III.

ASSEMBLER RETURN LOCATIONS

Words Used to Terminate CODE entry	Explanation	Contents of Register 0	Stack Before	Stack After
			TOP	TOP
PUT	Replace top of stack with contents of register 0.	12	6	12
PUSH	Push contents Register 0 onto top of stack.	7	4	4 7
POP	Discard top of stack.		2 1	2
BINARY	POP followed by PUT	9	80 3	9

(Assembler return locations modify the stack and return to NEXT.
Please NOTE: some processors use a different register than register 0.)

automatic. CODE, however, requires that parameters be handled explicitly, using S (the parameter stack pointer) and the code-ending returns that push or pop the stack before executing NEXT.

Other standard FORTH assembler notation includes the right parenthesis, which indicates relative addressing when it's by itself, or indexing if combined with an index register designation.

S) Addressing relative to the top of the stack.

S) Indexed by S.

1) Indexed by register 1.

On machines with automatic incrementing or decrementing, the parenthesis may be combined with + or - . On the LSI11, for example,

S)+ Refers to the number on top of the stack, "popping" it off at the same time; that is, incrementing the stack pointer.

S -) Refers to the next available location on the stack, a "push" operation.

Immediate addressing is indicated by # and memory indirect by the right parenthesis again; the assembler can determine from the address whether) means register-relative or memory-relative (indirect). In addition, there are specific items of notation for each processor--these are described in detail in the appendix.

Parameters may be taken directly from memory, if this is permitted by the architecture of the processor. The assembler will automatically check to determine whether the address of the argument permits a short format instruction, and, if it will not, an extended format will be used. Often parameters may be picked up without being named. So long as an address is on the stack, it doesn't matter how it got there:

```
HERE 55 C, . . . LDA
```

will enter the constant 55 in the dictionary and leave its address on the stack at assembly time. (The operation C, puts the low-order byte of the number on the stack into the dictionary at HERE and increments H by 1.) The instruction LDA coming later will encounter the address on the stack and assemble an instruction to move its contents to the A register.

12.3 Macros

Macros may be defined easily in FORTH using : definitions which contain assembler instructions. For example, on the COSMAC one frequently uses the operations DEC and STR successively on the same register. For convenience, the macro:

```
: DST   DUP  DEC  STR ;
```

has been defined. Then S DST will assemble the two instructions:

```
S DEC   S STR
```

Note the way the DUP in the definition of DST has allowed the single parameter S to be used by both mnemonics DEC and STR.

But macros are mainly a convenience; DST assembles 2 instructions, just as if you had written the expression out in full.

12.4 Example

As an example of the action of the assembler, consider the COSMAC code definition for the operator DUP:

```
CODE DUP   S LDA   T PHI   S LDN   S DEC
   S DST   T GHI   PUSH
```

This will reproduce the top number on the stack. The action of the assembler in constructing this CODE entry is given in Figure 9. The resulting entry is shown in Figure 10.

Word	Action during assembly	Action during execution
S LDA	Put 13 (stack pointer register's number) on stack. Assemble a LDA instruction referencing register 13, and put it in the dictionary.	Load accumulator with top byte of stack and increment stack pointer.
T PHI	Put 9 (temporary work register's number) on the stack. Assemble a PHI instruction referencing register 9, and put it in the dictionary.	Store accumulator in top byte of T register.
S LDN	Put 13 on the stack. Assemble a LDN instruction referencing register 13, and put it in the dictionary	Load accumulator with low-order byte on stack.
S DEC	Put 13 on the stack. Assemble a DEC instruction referencing register 13, and put it in the dictionary.	Decrement stack pointer.
S DST	Put 13 on the stack. Assemble a macro: a DEC and a STR, both referencing register 13, putting them in the dictionary.	Decrement stack pointer and store accumulator at byte now addressed by S.
T GHI	Put 9 on the stack. Assemble a GHI instruction referencing register 9, and put it in the dictionary.	Fetch high-order byte from T register.
PUSH	Assemble a macro: S DEC, S STR and a branch to NEXT.	Decrement S, store accumulator and branch to NEXT.

Figure 9
Action of assembler
Assembly of DUP on the COSMAC

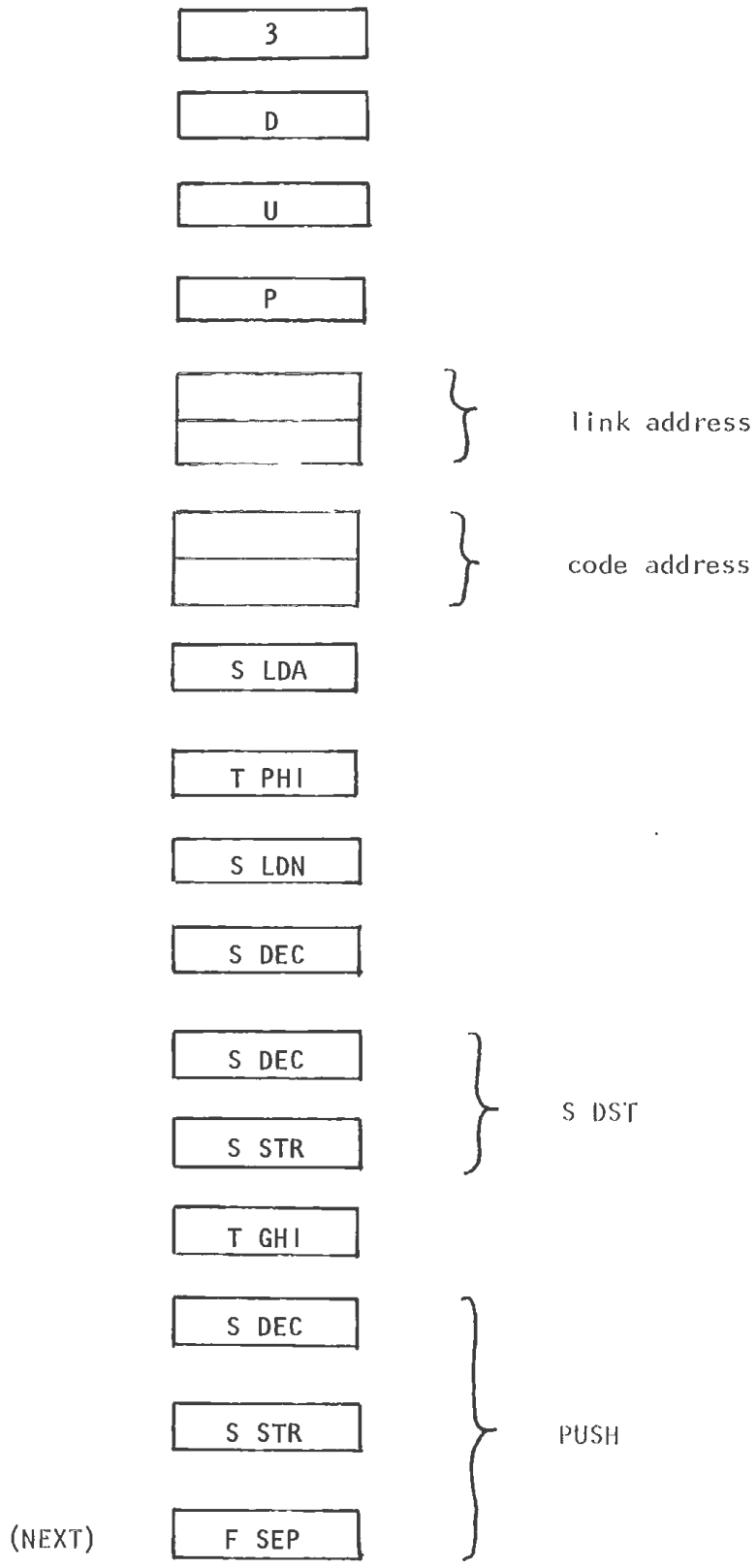

3
D
U
P
link address
code address
S LDA
T PHI
S LDN
S DEC
S DEC
S STR
S DST
T GHI
S DEC
S STR
PUSH
(NEXT)
F SEP

Figure 10

12.5 Logical Structures

Control of logical flow is handled by FORTH's assembler using the same structural approach as high-level FORTH, although the implementation of the commands is necessarily different. The commands even have the same names as their high-level analogues; ambiguity is prevented by use of separate Vocabularies (see Chapter 14). The following are implemented as standard macros:

BEGIN — Puts an address on the stack (HERE).

END — Assembles a conditional jump back to address left by BEGIN. It is preceded by a condition code. The loop is ended if the condition is met. Common condition codes are 0= and 0<, as appropriate to the various CPUs.

NOT — Negates condition code.

IF — Assembles a conditional forward jump, to be taken if the preceding condition is false, leaving the address of this instruction on the stack.

ELSE — Provides the destination of IF's jump (whose address was on the stack) and assembles an unconditional forward jump whose location is left on the stack.

THEN — Provides the destination for a jump instruction whose location is on the stack at assembly time (left by IF or ELSE).

The ELSE clause may be omitted entirely. This construction is functionally analogous to the IF...ELSE...THEN construction provided by FORTH's compiler. For instance,

```
0= IF (code for 0)      ELSE (code for not 0)    THEN...
0= IF (code for 0)      THEN...
```

Since the locations or destinations of branches are left on the stack at assembly time, the structures BEGIN . . . END and IF . . . ELSE . . . THEN may be naturally nested. However, by manipulating the stack during assembly, the programmer can assemble any branching structure. If you wish to branch forward, use 1F to leave the location of the branch on the stack. At the branch's destination, bring the location back to the top of the stack (if it is not there already) and use ELSE or THEN to complete the branch by filling in its destination at the location on top of the stack. If you wish to branch back to an address, leave it on the stack with BEGIN. At the branch's source, bring the address to the top of the stack and use END or a jump mnemonic to assemble a conditional or unconditional branch back. Be sure to manipulate the branch address before the condition mnemonic as the condition codes add one item to the stack.

Suppose, for example, you wish to define a word LOOK, which takes a delimiter on top of the stack, and a starting address under it, and scans successive bytes till it finds either the delimiter or a zero. The number of characters scanned is returned. Here is a definition of LOOK for the 6800:

```
CODE LOOK     B PUL   A PUL   TSX   0 ) LDX
   BEGIN   0 ) TST   0= NOT IF
      0 ) A CMP   0= NOT IF   B 1NC   ROT JMP
   THEN THEN   A CLR   TSX   PUT JMP
```

Here the two phrases 0= NOT IF assemble conditional forward jumps which will be executed if the character scanned is the same as one of the delimiters. If the loop is to be repeated, after incrementing B we must JMP back to the BEGIN. The intervening IFs have left their locations on the stack, so the branch must be assembled by ROT JMP. The ROT, executed at assembly time, pulls the address left by BEGIN to the top of the stack where it is used by JMP as its destination. Finally, the THENs fill in the destinations of the IFs.

There are no labels in FORTH. You could define them, but their function is better performed by the CODE names IF, ELSE, THEN, BEGIN and END. Since CODE definitions are usually extremely short, labels are not particularly desirable; they tend to encourage complicated flow patterns that are not appropriate in FORTH.

Some processors permit you to use literals directly in code; on these, ,CODE is

rarely encountered.

12.6 Device Handlers

Device handlers should be kept extremely short, including only the instructions required to pass a value to or from the stack or to issue a command. Consider, for example, a self-scan character display which is interfaced to a COSMAC as device 2. This is all that is needed to output one character from the top of the stack:

```
CODE DISPLAY   S INC   S SEX   2 OUT   NEXT
```

In this example, S INC increments the stack pointer (to get the low-order byte), S SEX sets S as the output register, and 2 OUT sends the character to the device, incrementing S again to complete a POP.

Given the definition of DISPLAY, you can then define TYPE at high level to display a string of characters whose byte address and length are on the stack.

```
: TYPE   0 DO   DUP C@ DISPLAY   1+ LOOP   DROP ;
```

To convert and display a number on the stack, you might define PRINT:

```
: PRINT   (.) TYPE ;
```

Here (.) performs the conversion, leaving the address of the resulting string and its length for TYPE (see Chapter 8.0, OUTPUT). The point here is that, given the simple code definition DISPLAY, you have easy, full control of the display in high-level FORTH.

12.7 Time and Memory Trade-Offs

It should be clear by now what the relative trade-offs in time and memory efficiency are between CODE and : definitions. CODE will be almost exactly comparable to conventional assembler code, with some advantage due to the handy convention of the stack, which saves the time and complexity involved in parameter passing. On the

other hand, : definitions are very much more compact, being only a string of addresses of previously defined words. The combination of CODE and : definitions means that the overall programs will be extremely compact, as even short code strings will rarely be repeated.

Suppose, for example, you have an 8-byte code string that performs a useful function. It may at first glance seem ridiculous to double its length by making a dictionary entry out of it, but since every subsequent reference to it takes one word, it will take very few uses to recover the cost, and from then on you will save 6 bytes for every usage. Furthermore, in a cross-compiled application, the space cost of the original definition is reduced to only 2 bytes plus the actual code! Clearly, the saving will be greater for longer strings. However, you should only perform a single logical operation in one CODE definition.

13.0 SPECIAL DEFINING WORDS

An important aspect of FORTH is its ability to define new words. New definitions and code are devised frequently. Likewise, new constants or variables may be created. A more challenging and significant creativity, however, is involved in the definition of new kinds of words. Neither : definitions nor CONSTANTs nor any other kind of common word, these can share the attributes of both nouns and verbs. Some special defining words are available to enable the user to develop these new kinds of words.

The identity of a FORTH word is established by the first 6 bytes--the name and link fields. The character of the word, that is, the way it will behave when used, is determined by the next 2 bytes, its code address. So far we have considered the generation of all of these bytes to be a single function. Now we shall consider the management of the code address as a special activity.

A word which constructs a dictionary entry is called a defining word. The most basic defining word is CREATE, which installs the name, link and code address fields in the dictionary, leaving the code address pointing ahead to the first byte of the parameter field. CREATE does not actually reserve any bytes of parameter field, however.

The FORTH program also supplies the defining words : and CONSTANT. A few more are built in from these, of which the most frequently used are CODE and VARIABLE. These themselves are defined so that they first use either CREATE or CONSTANT to construct the dictionary entry for new words and then replace the code address with another that will reflect the special behavior of the new kind of word (e.g., VARIABLE).

13.1 Use of ;CODE

The word ;CODE enables the user to create new classes of words by specifying the custom code for the code address of the words of a class. Let us examine the definition of VARIABLE in detail, in order to understand this process so that you may generalize from it. We will have to keep well in mind that there are 3 moments in time which are of interest: the time when the word VARIABLE is defined, the time when VARIABLE is executed to define a new word, and the time when a word defined by VARIABLE is invoked to push the address of the parameter field of the word onto the stack.

The definition of VARIABLE for the 8080 microprocessor is:

```
: VARIABLE   CONSTANT   ;CODE
   W INX   W PUSH   NEXT JMP
```

When this definition is compiled, it produces the entry shown in Figure 11. Given that definition, the phrase

```
0 VARIABLE M
```

will execute VARIABLE to construct a dictionary definition for M. The code address of M will point to the first instruction after ;CODE in the definition of VARIABLE. The execution of VARIABLE to compile M performs these steps:

CONSTANT	reads the word M from the input message buffer and constructs a dictionary entry for it. The code address points to the code for CONSTANT, and the parameter field is initialized to the value on the stack, in this case 0.
;CODE	completes the entry for M by replacing M's code address with the address of the word immediately following the ;CODE in the definition of VARIABLE, i.e., with the address of the INX instruction.

The key word in the definition above is ;CODE, which immediately precedes assembler

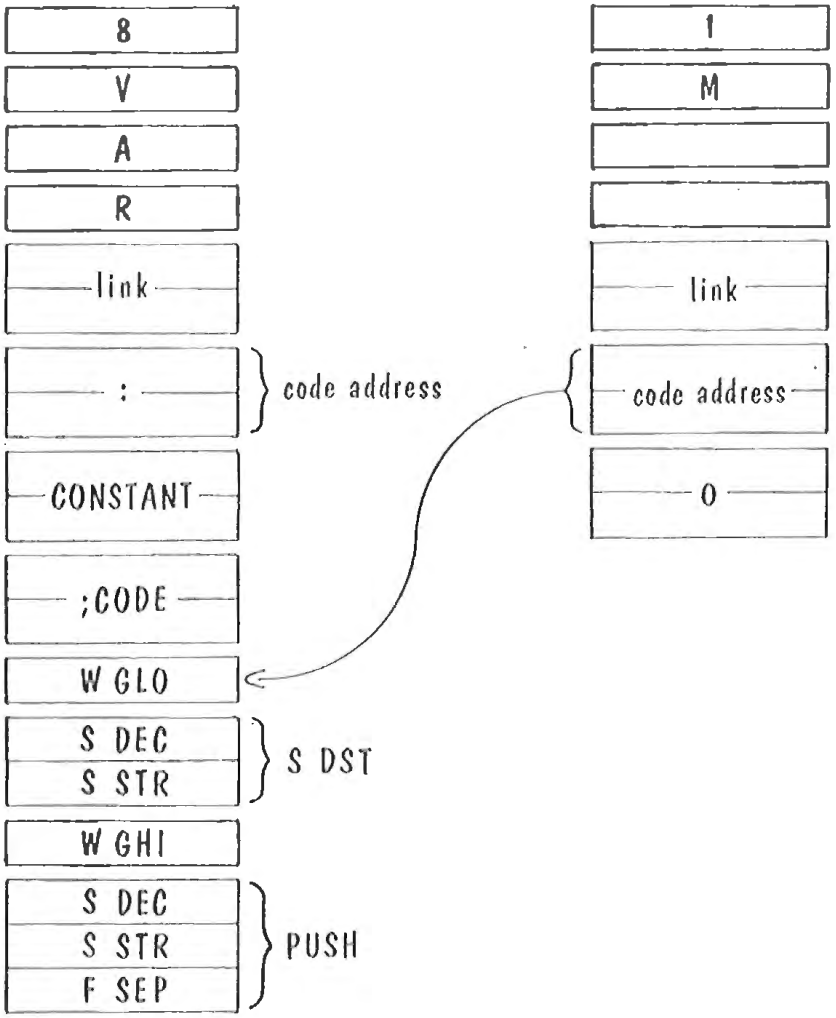


Figure 11

Compilation of VARIABLE and the definition:

0 VARIABLE M

code. When VARIABLE is executed it performs the function of resetting the code address of the word being defined to that of the code which immediately follows the ;CODE.

Note that the code phrase following ;CODE is not executed at the time M is defined. Rather, when M is executed, the inner interpreter will jump through M's code address to the INX instruction. At this time, the system parameter W will contain one less than the address of the parameter field of the word being executed (that is, M). (The exact offset of W from the parameter field is processor-dependent; consult the appendix for your CPU.) The W INX instruction increments this address so that it will point to the location which contains M's value (the parameter field); W PUSH places this address onto the stack; and NEXT JMP returns control to the inner interpreter.

The definition of what VARIABLE does is the same on all FORTH processors.

The key features of this technique to remember in order to define other kinds of words are these:

> USE A PREVIOUSLY DEFINED DEFINING WORD TO GENERATE THE DICTIONARY ENTRY. The most frequently used is CONSTANT, which initializes the parameter field. You may also use CREATE, which takes no parameters, or some other word with similar properties.
>
> USE ;CODE TO END THE DEFINITION AND PLACE IMMEDIATELY AFTER IT THE CODE YOU WISH TO USE TO CHARACTERIZE THE NEW CLASS OF WORDS.
>
> REMEMBER THAT WHEN YOUR CODE WILL BE EXECUTED, W WILL POINT TO THE ADDRESS OF THE PARAMETER FIELD OF THE MEMBER OF THE CLASS. Depending on the processor, W will contain the address of the parameter field minus 0, 1 or 2. Consult the appendix for your CPU.

Here are some examples of words that have been useful in certain applications.

13.1.1 VECTOR

VECTORs are useful in applications that have a lot of 2- or 3-dimensional variables, such as X-Y coordinates, latitude/longitude/bearing in navigational systems, azimuth/elevation, etc. Reference to a vector places one of the values on the stack, selected by the index. That is, if we defined:

```
0 CONSTANT X   1 CONSTANT Y   2 CONSTANT Z
   3 VECTOR SCALE
```

We have a VECTOR named SCALE, of length 3. We can get selected values as follows:

```
X SCALE C@  or   Y SCALE C@   etc.
```

The code for VECTOR expects the index on the stack and automatically adds it to the address of the beginning of the parameter field of any word defined using VECTOR. Here's how the code for VECTOR might look on the 8080:

```
: VECTOR   CREATE   H +!   ;CODE
   W INX   H POP   W DAD   HPUSH JMP
```

13.1.2 ARRAY

ARRAY defines an array in memory. When referenced, the index into the array must be on the stack; it will be automatically applied just as for VECTOR above. Here's an example of its use:

```
100 ARRAY DATA
   : RECORD   100 0 DO   A/D I DATA C! LOOP ;
```

This reads 100 numbers from an A/D converter and stores them in the array DATA.

The code portion of the definition might be the same as for VECTOR. Because this only handles byte indices, however, arrays containing 16-bit data might double the index before adding it to the base. The first portion of the definition might initialize the array to all zeros:

```
: ARRAY   CREATE   HERE OVER ERASE   H +!   ;CODE
   W INX   ... etc.
```

Here CREATE is used to create the basic entry and the phrase HERE OVER ERASE clears as many zero bytes as there are to be entries in the array. H +! encloses them into the dictionary by incrementing H so that the next definition will begin after the requested displacement.

13.2 High-level Defining Words

It is also possible to define defining words entirely in high level. This can be done using <BUILDS and DOES>. The form of such a definition is

```
: [name]   <BUILDS [words to be executed at compile time]
   DOES> [words to be executed as the definition of the defined word] ;
```

An example is

```
: MSG   <BUILDS  DOES> COUNT TYPE ;
HEX   MSG SPACE   1 C,  20 C,   DECIMAL
```

MSG is a defining word, used to define words which will output ASCII strings. SPACE is such a word defined by MSG, to output a single space. The phrase 1 C, 20 C, constructs an output string, consisting of a character count (1) and an ASCII blank (20), in the dictionary. Figure 12 contains a diagram of the dictionary entries for DOES>, MSG and SPACE.

As with ;CODE defining words, there are two executions to distinguish: when MSG is executed to define SPACE, and when SPACE itself is executed. The high level definitions of both are found in the definition of MSG. DOES> marks the end of

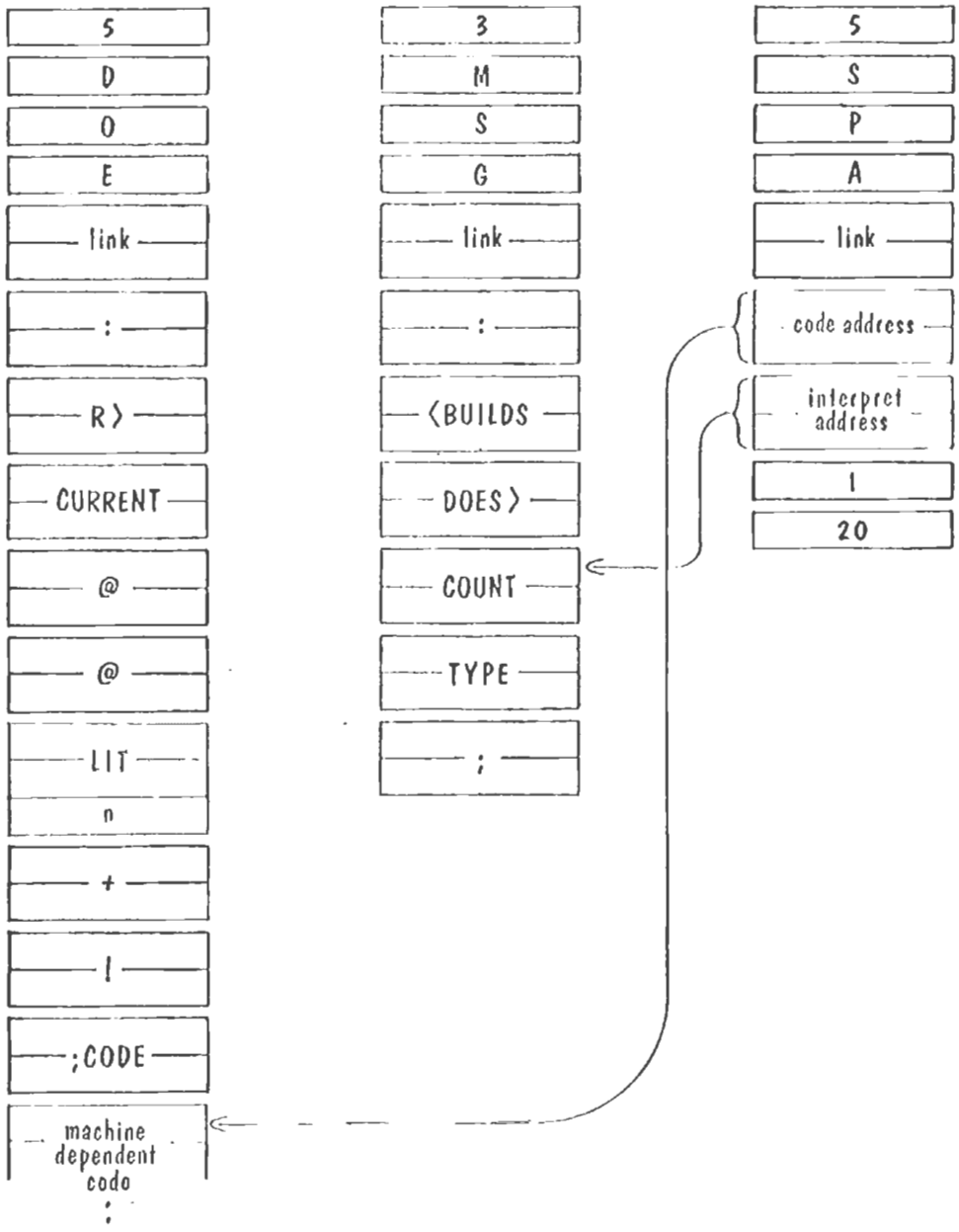


Figure 12
Compilation of DOES>, MSG and :

HEX MSG SPACE 1 C, 20 C,

MSG's execution. The remaining phrase (COUNT TYPE ;) is executed whenever any word defined by MSG (such as SPACE) is invoked.

The definition of BUILDS> is just

```
: <BUILDS   0 CONSTANT ;
```

This creates a dictionary entry and reserves two bytes of parameter field. Later, DOES> will store an address in these two bytes. The output string of SPACE (which is explicitly compiled as 1 C, 20 C,) begins after this address, that is, in the third byte of the parameter field.

DOES> terminates the execution of MSG. The succeeding phrase (COUNT TYPE ;) will be executed by SPACE. However, before this phrase is executed, the address of the third byte of the parameter field will be put on the stack. This is, of course, the starting address of the output string. This address will serve as the argument to COUNT, which proceeds to set up the parameters of TYPE.

The definition of DOES> is

```
: DOES>   R>  CURRENT @ @ n +  !  ;CODE
```

followed by code to be described shortly. In the above definition, n is a processor-dependent literal such that CURRENT @ @ n + is the address of the parameter field of the most recently created entry, in this case, to the 0 compiled into SPACE by <BUILDS. (CURRENT is described in Chapter 14.0, VOCABULARIES) The effect of the phrase

```
R>  CURRENT @ @ n + !
```

is to save the address of the phrase COUNT TYPE ; in the first two bytes of the parameter field of SPACE. Also, this removes the top of the return stack, in effect terminating the execution of MSG.

Later, when SPACE is invoked, the ;CODE phrase of DOES> will be executed. This code phrase does three things:

1. Saves the interpreter pointer on the return stack.

2. Resets the interpreter pointer with the address in the first two bytes of the parameter field of SPACE, i.e., with the address of the phrase COUNT TYPE ; .

3. Pushes the address of the third byte of the parameter field of SPACE on the stack. This becomes the argument of COUNT.

Another useful MSG is CR, defined in HEX by

```
MSG CR   6 C,  D C,  A C,  0 ,  0  ,
```

The character count is 6, D and A are the ASCII codes for carriage return and line feed, respectively, and the 4 remaining null characters sent whenever CR is typed are required for timing in some terminals and printers.

An extension of MSG which reads a string of text and gives it a name which may be used to type it out is STRING, defined (in decimal) by

```
: STRING   MSG   92 WORD   HERE C@ 1+  H +!   ;
```

MSG sets up the initial definition, as above. 92 is the ASCII code for \ , which WORD takes off the stack as its delimiter. WORD puts the characters typed at the terminal following the name, until the occurrence of a \ , into the dictionary at HERE but it doesn't advance H. HERE C@ 1+ gives the length of the string, including the count byte. The H +! increments H by this value, thus enclosing the string in the dictionary. To use STRING, consider the definition of a word ERROR:

```
STRING ERROR BAD!!\
```

Thereafter use of the word ERROR would cause BAD!! to be typed out.

NOTE: DEFINITIONS SUCH AS THESE ARE WASTEFUL OF MEMORY, ESPECIALLY FOR LONG TEXT STRINGS. On disk systems the use of MESSAGE, which keeps its text on disk, is preferred. This is, however, a good way to handle messages in applications that will not have a

disk.

Another example of the use of <BUILDS and DOES> is the defining word FIELD, which is used to define fields in a data block on disk:

```
: FIELD  <BUILDS  C,  DOES>  C@  B# @ BLOCK  + ;
```

Note that the word C, appears between the references to <BUILDS and DOES> in the above definitions. <BUILDS and DOES> are separate words for essentially this purpose, to allow the user to specify the implicit compilation of any size field to follow the definition being created before that definition is completed by DOES>. In this case C, compiles in the BLOCK offset that is later fetched by the C@. Remember that words appearing between <BUILDS and DOES> will be exeeuted when the new word is defined, whereas those words following DOES> are executed when the new word is used.

Given this definition, you might define FIELDS thus:

```
0 FIELD NO.   1 FIELD KIND   2 FIELD VALUE   4 FIELD OFFSET
```

etc. In use it is assumed that B# is a VARIABLE containing the number of some data block. Then VALUE would fetch the address of the third byte of the block (in this case VALUE is assumed to be double length) and OFFSET would fetch the address of the 5th byte. This basic concept can be expanded to some elaborate data file management capabilities.

Consider an alternate definition of VECTOR (with X, Y, and Z defined as above):

```
: VECTOR  <BUILDS  DOES>  + ;
```

and the definition of a VECTOR:

```
VECTOR CORNER  100 C,  40 C,
```

(Typing X CORNER C@ puts 100 on the stack and Y CORNER C@ puts 40 on the stack.)

A comparison of this definition of VECTOR with the first one exhibits the major

differences between DOES> and ;CODE, namely that the use of high-level FORTH following DOES> is often more convenient than supplying code to follow ;CODE. This method is also more machine independent.

Hopefully, it has been shown that different kinds of words may be usefully defined. Basic FORTH provides only CONSTANT and VARIABLE, but standard definitions of most of the words discussed here are available. If you encounter more than one instance of a particular kind of word, or use such a word frequently, it can pay off in convenience, efficiency, and elegance, to name and characterize those properties that make it unique.

14.0 VOCABULARIES

So far we have considered the FORTH dictionary as a single threaded list of definitions of various types. Actually, the dictionary is branched from a central "trunk" and several subsidiary vocabularies may be linked into this trunk.

Multiple vocabularies provide several advantages:

1. In a complex application, dictionary search time is reduced substantially. This is mainly significant in reducing the time required for loading.

2. Security is enhanced by denying access to sensitive commands by isolating them in separate vocabularies. The vocabulary name acts as a key, and may be kept secret.

3. Similar operations in parallel portions of an application may be given the same name, without internal confusion. Judicious use of this capability may make complicated applications much easier for a user to learn.

As you might expect, FORTH is the name of the "trunk" vocabulary. All other vocabularies are chained to FORTH, that is, after a vocabulary search is exhausted, FORTH will be searched starting at its most recent definition.

Two other vocabularies are defined in the basic system: ASSEMBLER contains all assembler instruction mnemonics and other assembler directives, and EDITOR contains commands to the text editor.

The use of the name of a vocabulary sets the variable CONTEXT to the head of the sub-vocabulary that will begin a dictionary search. CURRENT specifies the vocabularies into which new definitions will be linked.

The word used to define a new vocabulary is VOCABULARY.

```
VOCABULARY  TESTING
```

defines a new vocabulary named TESTING. TESTING itself is defined in the vocabulary to which CURRENT was set when it was defined. To enter definitions in this new vocabulary, you need to say,

```
TESTING  DEFINITIONS
```

Here, TESTING set CONTEXT to TESTING, while DEFINITIONS set CURRENT to CONTEXT. Remember, CONTEXT specifies which vocabulary you are searching, and CURRENT specifies which vocabulary new definitions will be put in.

Some defining words affect CONTEXT:

```
CODE        sets CONTEXT to ASSEMBLER
:           sets CONTEXT to CURRENT
```

This has certain consequences which may not be immediately obvious. Suppose you say:

```
TESTING DEFINITIONS   0 VARIABLE S
CODE CHECK   S 0 LDA ...
```

Alas, since CODE set CONTEXT to ASSEMBLER and S was defined in TESTING, you have referenced the ASSEMBLER S (stack pointer) rather than the VARIABLE in TESTING. You have to obtain the address of TESTING's S before entering ASSEMBLER:

```
TESTING DEFINITIONS   0 VARIABLE S   S
CODE CHECK   0 LDA ...
```

Here you fetched the address of S before changing vocabularies. That address is

waiting on the stack for LDA.

As CODE leaves you in ASSEMBLER, you will not be able to find any word in TESTING, (for instance, by typing it) after defining CHECK. You may either reset your previous CONTEXT or else follow the code by a : definition (which will reset it automatically). CONSTANT and other defining words don't change CONTEXT.

There is no requirement that vocabularies be contiguous in memory. In fact, ASSEMBLER and early FORTH are quite interlaced. FORGET (which forgets all dictionary entries subsequent to the one whose name follows FORGET), however, will "forget" in a spatial sense. Thus, if you mix the definitions of several vocabularies, you must forget all of them at once. Consider:

```
VOCABULARY   RED      VOCABULARY BLUE

RED DEFINITIONS
        .
        .
        .
BLUE DEFINITIONS
        .
        .
        .
```

FORGET BLUE is incorrect and will prevent further dictionary searches. RED's definitions have been discarded, but RED's pointer which initiates the search has not been updated. On the other hand, FORGET RED will forget BLUE as well.

Any word (and all following words) may be forgotten, providing the CURRENT Vocabulary is that in which the word was defined. FORGET sets CONTEXT to CURRENT.

You can make a Vocabulary IMMEDIATE when you define it. For example,

```
VOCABULARY VECTOR    IMMMEDIATE
VOCABULARY COMPLEX   IMMEDIATE
```

Then later you may define

```
: OPERATION   VECTOR + COMPLEX  * ;
```

which will compile only two words,

+ from the VECTOR vocabulary

and

* from the COMPLEX vocabulary

The vocabularies that are resident in standard microFORTH are diagrammed in Fig. 13. The organization shown is logical rather than spatial. The circles show the actual definition of the vocabulary, with the dashed lines indicating the linkage from the definition to the head of the vocabulary referenced.

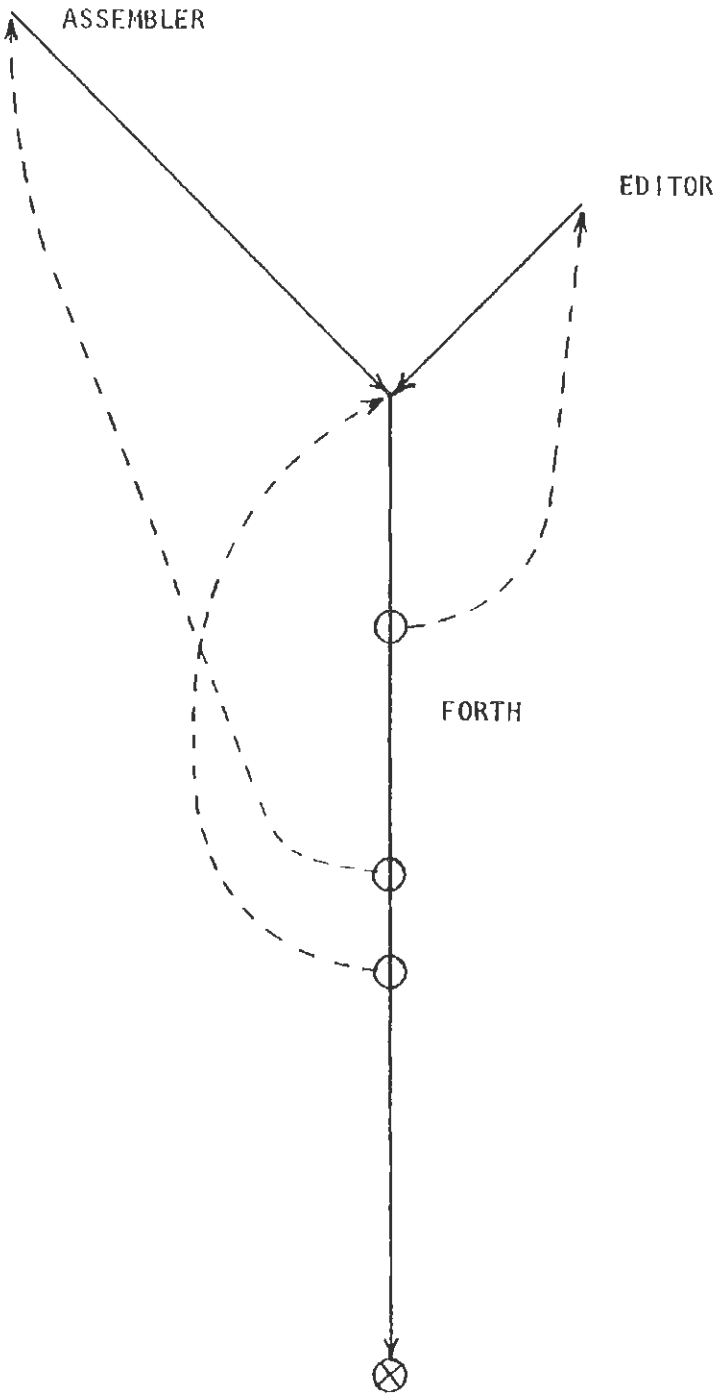


Figure 13
Diagram of Vocabularies in microFORTH

15.0 THE CROSS COMPILER

The cross compiler is best understood in terms of its differences from and similarities to the operation of the normal resident compiler. For this reason it would be well to delay reading this chapter until you have acquired a working knowledge of the normal compilation process by actual experience.

15.1 Explanation of Terms

Before proceeding too deeply we should establish a common understanding of some often used terms.

15.1.1 Cross Compiling

By cross compiling we mean the generation, on one system, of code destined to be executed on another system. The destination CPU may be of a different type than that of the development system, but within the context of this manual it will be assumed to be always the same.

The function of the resident compiler is to add new definitions into the development system, so that they become immediately executable, even before the compilation process has come to rest. In contrast, under the cross compiler, the entire application must be compiled and transferred to its intended system, before any portion of that application can be executed. This corresponds to the meaning of the term "compilation" in the context of conventional programming. On FORTH systems "compilation" has always implied an immediate, interactive process in which each word

compiled becomes an integral element of the development system itself.

There are three principal differences between compiling and cross compiling:

1. The product of the cross compiler is built in virtual memory on disk, allowing it to occupy locations in address space that might be occupied by the development system software. This is the principal reason that the cross compiler's output cannot be executed until it reaches its destination hardware.

2. The output of the cross compiler does not contain the names of the words defined, nor the links that would allow the generated program to search its dictionary at execute time. This is a significant compression of the already compact definitions provided by the resident compiler, on the order of 10 to 30%.

3. The output of the cross compiler does not contain the FORTH text interpreter, resident compiler, disk and terminal support, or other routines concerned with supporting an interactive system. It is instead suplied with a 512 byte subset of the FORTH system composed of the address interpreter and a vocabulary of essential words.

15.1.2 The Target System

The target system (or target computer) is the system on which the software under development will eventually reside. Within the cross compiler, the word "Target" is applied to two different but related entities: the Target Dictionary, and the Target Vocabulary. The Target Dictionary refers to the virtual memory on disk into which the application is compiled, and to the compiled program that resides there. The Target Vocabulary is a formal Vocabulary as described in Chapter 14.0 (VOCABULARIES).

The Target Vocabulary occupies development system dictionary RAM and contains executable definitions. In fact all definitions are executable in that they contain a code address (see the chapter "FORTH Dictionary

Structure"). The purpose of the entries in the Target Vocabulary is to retain the name of each word compiled into the Target Dictionary and its location in target memory. The word TARGET is the name of the Target Vocabulary.

15.1.3 The Host System

The host system is, of course, your development computer. Within the cross compiler the word HOST is the name of the Host Vocabulary. The Host Vocabulary contains the cross compiler itself. It is distinct from the FORTH Vocabulary for an important reason. The majority of the words within the Host Vocabulary are redefinitions of words that occur in the FORTH Vocabulary. The Host Vocabulary redefines such words as:

```
, @ HERE VARIABLE :
```

(and the entire assembler). In general those words which are used in FORTH to describe, build, or modify a definition, must be redefined in HOST to perform an analogous function for the Target Dictionary.

The Host Vocabulary exists separate from FORTH in order to maintain the accessability of normal FORTH words during the compilation of the cross compiler. Visualize for a moment the process of building the cross compiler. As more and more compiling and defining words are given new interpretations (thus burying their previous definitions) it occasionally becomes necessary to slip beneath the new meaning of a word and use the older, resident compiler version in order to properly create some element of the cross compiler. This is the reason that the word FORTH is IMMEDIATE (see Section 5.8). It may be used within a definition to effect an immediate context switch back to the FORTH meaning of a word.

15.1.4 The Nucleus

The heart of a cross compiled program is called the Nucleus. The Nucleus contains the FORTH address interpreter and the definitions of a small but

crucial set of words. The most important words will be those that support literals, loops, and conditional branches. Also included is the code to support the definitions of CONSTANT, USER, VARIABLE, :, ; and DOES>. Several other words are included but the list will vary somewhat between different computers in the interest of limiting the size of the Nucleus to 512 bytes on all machines. The words in the following list, however, can always be assumed to be present:

@	!	C@	C!
+!	EXECUTE	U*	U/
+	-	AND	MOVE
<R	R>	1	0<
0=			

15.1.5 Defining vs Compiling

Many times in the following chapters reference will be made to the terms "defining words" and "compiling words". These terms are not interchangable. They are meant to refer to two distinct, and probably mutually exclusive, dictionary construction activities that a word might be engaged in. The terms are distinguished by the portion of definition construction for which they are responsible.

The domain of a defining word includes the name field, link field, and code address of a definition (collectively referred to as the "head"). In many cases, a defining word will have responsibility for initializing the parameter field, as in the case of CONSTANT or VOCABULARY. The lowest level defining words are CREATE, ;CODE and DOES>. Any definition which refers to a defining word is itself a defining word. Defining words are not IMMEDIATE. A fairly complete list of words would include (aside from the ASSEMBLER mnemonic defining words):

CREATE	CONSTANT	VARIABLE	CVARIABLE	
USER	:	<BUILDS	DOES>	
MSG	VOCABULARY	CODE	;CODE	IMMEDIATE

The domain of a compiling word is anywhere within the parameter field of a colon definition. In order to effect such a definition a compiling word must be IMMEDIATE. The majority of compiling words make reference to the word \ (see the "Compiler" chapter) but that is not a requirement. The primary distinguishing feature of a compiling word is that it will always be IMMEDIATE. The converse is also true; if a word is IMMEDIATE it must also be a compiling word. (It will certainly never be compiled.) Some words that are technically compiling words are also usable outside of a definition. FORTH, BEGIN and (are such words.

It was stated earlier that compiling and defining words were mutually exclusive. An apparent exception is ;CODE. ;CODE was previously declared to be a defining word. It is also obvious from its definition (in screen 4) that it is a compiling word. On closer inspection however, it can be seen that there are actually two ;CODE words. The older, defining word is compiled into a definition by the latter, compiling version.

In a typical FORTH application, the user will not define any compiling words of his own, and only a few defining words, if any. The cross compiler has been constructed, as much as possible, to make the cross compilation of all other words transparent to the user. Techniques for handling user written compiling words and defining words will be discussed in the chapter, "Extending the Cross Compiler."

15.2 Organizing an Application to be Cross Compiled

In any FORTH application it is important to organize the loading of your definitions in a single screen. A "load screen" is a screen whose primary function is loading each application text screen in the desired compiling sequence. It should have few additional responsibilities. Screen 3 on your system disk is such a screen. (Screen 3, however, does include more definitions than a cross compiler load screen would.)

The use of a load screen is good programming practice. It organizes into one visible spot all of the blocks to be loaded for an application. This concentrates and segregates the load function into a small area, away from the actual

definitions.

When the application has been tested and is ready to be cross compiled, you must provide a separate load screen to load it under the cross compiler. This is required because the environment of the cross compiler is different from that of the resident compiler. It is the task of this separate load screen to isolate the actual program text from these differences.

The first responsibility of this load screen is to configure the cross compiler for the kind of output to be generated. Refer to the screen titled "CROSS TEST." Line one will contain the following sequence:

```
NUCLEUS LOAD   PROM LOAD  COLON LOAD
```

The words NUCLEUS PROM and COLON name portions of the cross compiler that you must explicitly select in the load screen because they are optional. The words SYMBOLS and RAM name screens that provide other modes of cross compiling.

The two screens NUCLEUS and SYMBOLS are mutually exclusive; only one of them is loaded in any compile. You must load one of them and it must be the first thing done in the load screen. Their purpose is to empty the TARGET Vocabulary and Dictionary in preparation for a new compile, as follows:

1. Re-initialize the Target Vocabulary to an empty state and FORGET all definitions after the overlay point named COMPUTER.

2. Erase the Target Dictionary to zeros using the word CLEAR.

3. Reset the Target Dictionary pointer to zero.

4. Define the locations of the words contained in the Nucleus.

In addition, the NUCLEUS screen will transfer a copy of the precompiled Nucleus into the Target Dictionary beginning at location zero. You will more typically load NUCLEUS rather than SYMBOLS. SYMBOLS might be used to save compile time whenever the Nucleus already exists in PROM and need not be included as part of each new application.

The words PROM and RAM also name two screens which are mutually exclusive. One of them must be selected and loaded immediately after the loading of either NUCLEUS or SYMBOLS. Their purpose is to create the cross compiler versions of the defining words such as CONSTANT and VARIABLE. If the application is to be executed within PROM memory, VARIABLE cannot be defined in the standard manner. Specifically, its value cannot be contained in the parameter field because that would not allow it to be changed. Instead it is compiled as a constant, whose value is the location in RAM memory where space is allocated for the value. This mode of defining VARIABLE (and CVARIABLE) is selected by PROM LOAD. If RAM LOAD is specified instead, VARIABLE and CVARIABLE will be defined as in the resident compiler.

COLON LOAD completes the cross compiler by redefining : and ; along with several other compiling words. From this point on all words defined by the new : will be compiled into the Target Dictionary. The old meaning of : is available by using the word H: (i.e., Host's :).

16.0 THE CROSS COMPILER ENVIRONMENT

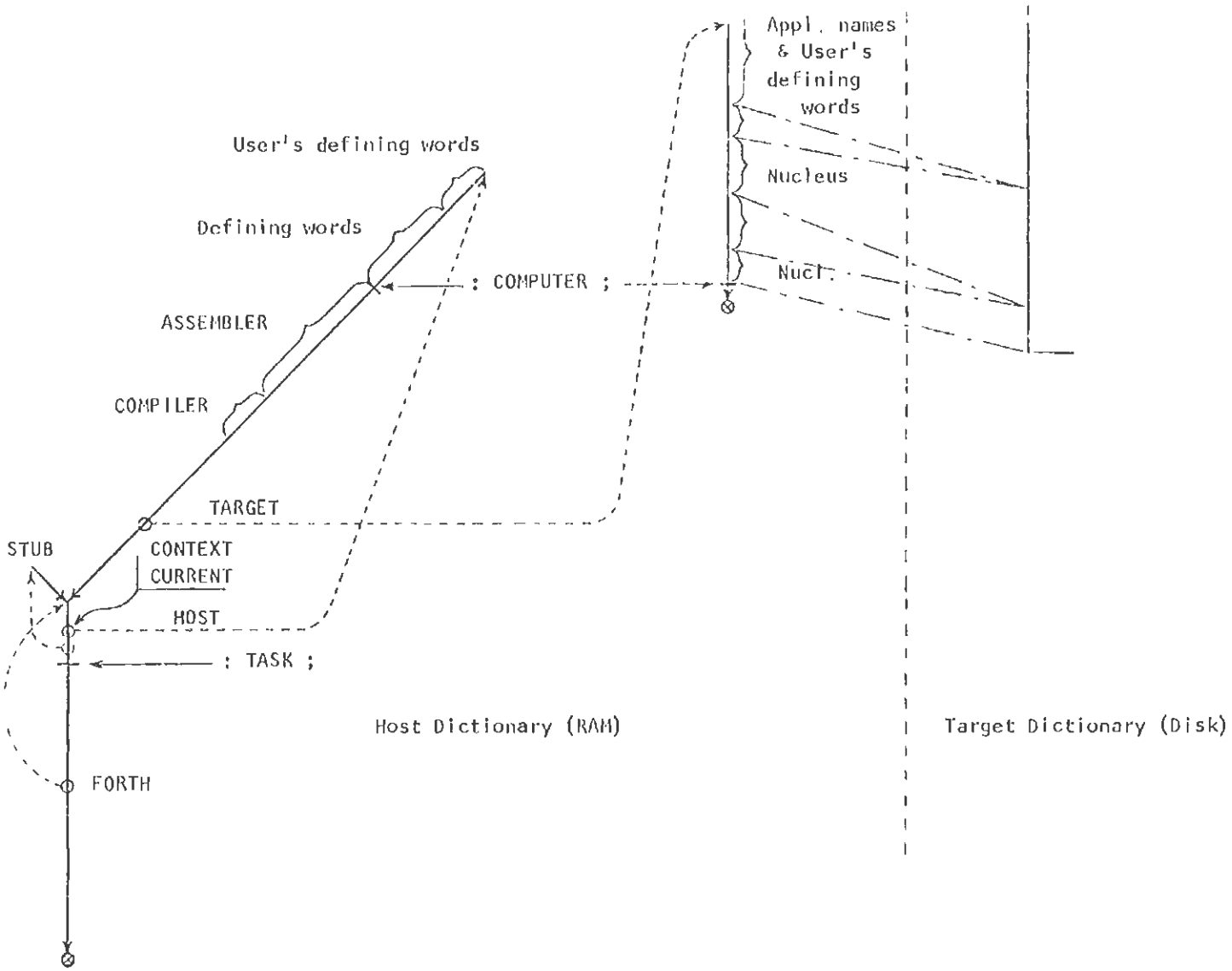
A diagram of the dictionary structure of the cross compiler is shown in Fig. 14. It illustrates the organization of the Vocabularies within the cross compiler and their general contents. The organization is shown in a logical rather than spatial manner. The Vocabulary names are shown on dashed arrows that connect the names to the tops of the Vocabularies they control. The Vocabularies EDITOR and ASSEMBLER are not shown. The short Vocabulary STUB is used solely in the generation of the cross assembler and does not come into play later.

The contents of TARGET are shown to map onto the contents of the Target Dictionary. As stated earlier, the Target Vocabulary contains the names and locations of words whose definitions are built in the Target Dictionary. TARGET is shown separated from the other Vocabularies. This is because TARGET is a "sealed" Vocabulary. Any search begun in TARGET will terminate there and will not continue into any other Vocabularies. This enables the cross compiler to generate an error message upon any attempt to reference an application word that has not yet been defined, even though a word by the same name might occur in either HOST or FORTH.

There are two overlay points shown in the illustration. One is marked by the definition : TASK ; in FORTH, and the other by : COMPUTER ; in HOST. TASK, as always, is used to mark the top of the FORTH Vocabulary. Because TASK is defined in FORTH, and because the cross compiler has normally specified HOST DEFINITIONS, the required sequence for discarding the cross compiler is :

```
FORTH DEFINITIONS   FORGET TASK   : TASK ;
```

The overlay point marked by COMPUTER represents the top of the configuration-independent portion of the cross compiler. Cross compiling for ROM



CROSS COMPILER DICTIONARY STRUCTURE
(not to scale)
Normally CONTEXT = CURRENT→HOST

Figure 14

or RAM for instance, is controlled by words defined after COMPUTER. There is also a utility that can be loaded at this point (OUTPUT LOAD, described in the chapter, "The Cross Compiling Process").

The illustration shows that the TARGET Vocabulary is split (physically) across COMPUTER. This imposes a special constraint on the ability to FORGET COMPUTER. TARGET must be emptied of all entires defined past COMPUTER before COMPUTER is discarded from HOST. The phrase to achieve this appears at the beginning of each of screens called NUCLEUS, SYMBOLS and OUTPUT. It is based on the knowledge that there is only one entry in TARGET that is beneath COMPUTER, and that this definition is located immediately beyond the definition of TARGET itself.

The following sections will discuss each of the major classes of definitions supported by the cross compiler. Refer to Figure 15 as necessary.

16.1 Colon Definitions

The Target Vocabulary exists primarily to support the cross compiler in the generation of colon definitions. Recall that the parameter field of a colon definition is a list of addresses, each of which points to a predefined word. For a resident compilation this list is built by the interpreter whenever the compile flag STATE is set. For the cross compiler this function is performed by the word COMPILE. COMPILE takes over input scanning from the interpreter starting from the first word in the definition until the ; is reached at the end. Unlike the interpreter, however, COMPILE does not examine each word's precedence bit to determine if it should be compiled. Instead, all words are executed. The only requirement is that each word occur in TARGET.

The compiling words IF, ELSE, THEN, DO, LOOP, +LOOP, BEGIN and END have their ordinary definitions in TARGET. Most other words are placed there by the defining word EMPLACE. For example, during NUCLEUS LOAD, the word C! is entered in TARGET by

```
0098 EMPLACE C!
```

(The number 0098 will vary for different computers because it is the address of the parameter field of C! in the Nucleus.) Now when C! is executed by COMPILE, the

address 0098 will be decremented by 2 (giving the address of the code address of C!) and placed at the top of the Target Dictionary. Thus the effect of executing a word defined by EMPLACE is to cross compile the corresponding word on the target system.

The defining words of the cross compiler (CONSTANT, VARIABLE, CVARIABLE, CODE, TABLE and :) also invoke EMPLACE. For example, if you cross compile:

```
10 CVARIABLE BASE
```

then BASE has been EMPLACEd in TARGET. If you then cross compile

```
: HEX   16 BASE C! ;
```

then COMPILE executes BASE and C! by compiling the corrseponding addresses in the Target Dictionary. By the way, HEX is iteslf EMPLACEd in TARGET by this definition, so that HEX may be cross compiled in subsequent colon definitions.

The number 16 in the definition of HEX is not found in TARGET, but instead is recognized as a valid number, and is compiled as an 8-bit literal. COMPILE can also compile 16-bit literals as needed.

The word ; is in TARGET, and its execution compiles a reference to ;S' (the code routine for a subroutine exit) and then exits from COMPILE.

16.2 Defining Words

Each of the important defining words is provided in the cross compiler. Some are not, however, and others are different in small ways that require clarification. In addition there are some defining words in the cross compiler that have no counterparts in the resident compiler. Such words may need to be provided by the user for the resident compiler during the testing phase. The following sections will give a brief description of each of the words that are provided. Of course, : is a defining word, but it has already been covered.

16.2.1 VARIABLE and CVARIABLE

These two words are identical in all respects except for the amount of RAM memory which they reserve for containing their values. They are of special interest because it is around them that the major difference between PROM and RAM compiles revolve. When used in a resident compilation they create a new dictionary entry and install an initial value from the stack into the variable's parameter field. When the resident definition is executed it pushes the address of its parameter field onto the stack. To store data into such a variable you must be able to modify the parameter field.

When cross compiling for RAM the technique is the same as for a resident compilation. The parameter field is initialized with the given value, and the parameter field address is pushed on the stack when the variable is executed on the target system. This method of defining variables is the most economical in terms of memory in that no pointer is required to locate the data. The data location is a function of the location of the definition itself.

For an application to execute in PROM, variables cannot be defined in this manner. The parameter field is not a writable location. Each variable must be defined as a constant whose value is a RAM address.

As each PROM variable is defined it must be allocated sufficient space in RAM for its intended use. This allocation is provided through a separate dictionary pointer named N. N is a HOST variable that points to the next available RAM location. This location is returned by the word THERE. Space is allocated by the word RES. 4 RES will reserve four bytes of RAM by incrementing N by four. This is analogous to incrementing H in the resident compiler.

It is important to keep in mind the difference between PROM and RAM variables whenever writing code that is to be compiled either way. This turns out to be most of the time. You compile into resident RAM for testing, and then cross compile for PROM. The differences between the

PROM and RAM environments can be characterized as follows:

1. The words N, THERE, and RES are only defined for a PROM compile. In your resident compile load screen you might define:

```
: N   H ;     : THERE   HERE ;     : RES   N +! ;
```

2. A PROM variable is not initialized. The initial value given in its definition is discarded.

3. PROM variables occupy contiguous RAM addresses, unlike RAM variables, whose values are separated by the code addresses of their own definitions.

A cross compiled variable is not executable in the normal sense. To find the address of a variable compiled for RAM named LAST you would use ' LAST. If the variable was compiled for PROM you would have to use ' LAST @. Because these two forms are not compatible, special effort has been taken with cross compiled variables. Each variable name is defined twice: once by EMPLACE for colon definitions, and once in HOST as a resident CONSTANT. By simply naming the variable outside a definition you execute the HOST CONSTANT. The value of the CONSTANT is the target address of the variable.

16.2.2 TABLE

TABLE is a defining word that has been devised primarily for PROM cross compiles, although it has also been defined for RAM cross compilation. Its purpose is to name the starting address of a table of constants. For example,

```
TABLE TENS  1 , 10 , 100 , 1000 , 10000 ,
```

will define a table of powers of ten. Invoking TENS in the target system will place the starting address of the table on the stack. Unlike VARIABLE, TABLE does not reserve any space for its parameter field. Instead, the values in the table must be explicitly set by , or C,. Note

that the tabulated items are placed in the dictionary, which is in read-only memory in a PROM cross compilation. (To allocate an array of bytes in RAM, use RES as discussed under "VARIABLE and CVARIABLE" above.) For a resident compile define TABLE as follows:

```
: TABLE    0 VARIABLE   -2 H +! ;
```

16.2.3 CONSTANT

Constants exist and function in the normal manner when executed on the target system. However, their names are not executable in the normal sense during cross compilation to provide their values. To determine the value of a CONSTANT named NAME in assembler code, for instance, use ' NAME @. This phase will work for either a resident or a cross compile.

16.2.4 USER

User variables were discussed briefly in the Chapter 3.0. You were cautioned there against defining any of your own so as not to interfere with the systems assignments. You can provide your own definition for USER that will build words that refer to an area separate from that used by the system as follows:

```
0 VARIABLE AREA 254 H +!    AREA VARIABLE U
: USER   <BUILDS C, DOES> C@ U @ + ;
```

Alternatively USER could be defined as

```
: USER   CVARIABLE ;CODE
```

followed by code that adds the byte addressed by W to the contents of U and pushes the result onto the stack. Either of these forms could be provided in the resident compiler load screen to permit testing.

There are several limitations on user variables:

1. The size of a user area cannot exceed 256 bytes.

2. The user area can cross a modulo 256 byte page only on computers with 16-bit ADD instructions. At the moment this is true only for the 8080A and derivatives.

3. The target user pointer (U) can only be changed by assembler code. This is because U is not defined as a VARIABLE (when it is in RAM) but rather by an EQU (refer to the next section "EQU and LABEL"). Likewise if U is a register (as on the COSMAC) it must be loaded by machine code.

Although we do not presume to tell you how user variables ought to be used, a study of the following list of properties should suggest several possibilities.

1. For PROM compiles, a USER variable occupies only 3 bytes as compared to 4 for VARIABLE or CVARIABLE.

2. The location of the user area need not be known at compile time.

3. The location of the user area need not remain fixed at execute time.

4. USER provides an explicit way to fix the relative positions of a set of contiguous variables.

16.2.5 Code Definitions

The defining word CODE is used, as always, to begin a code definition. It places a code field in the Target Dictionary that points one word ahead to the parameter field, and EMPLACE an entry into the Target Vocabulary that also points to this parameter field. This CODE differs in character from the resident compiler word in that it performs no context switching. It assumes that CONTEXT already points to HOST and does not change it.

This is because the cross assembler is defined within HOST. It does not occupy a Vocabulary of its own.

This is an important point to stress. With all of the assembler redefined in HOST, any word in the assembler having the same name as a FORTH word will cause the FORTH meaning to be buried (unless you explicitly switch back to FORTH). Although the actual words buried will vary between systems having different assembler mnemonics, the words IF, ELSE, THEN, BEGIN and END will always be among them. Later, as you will see in the chapter on extending the Cross Compiler, it will be important to remember this. The primary implication of this separate Host assembler is that you may cross-compile for any target processor.

Within a CODE definition the process of assembly procedes in much the same manner as for the normal assembler: number and register values are placed on the stack, mode words set whatever flags their function requires, and mnemonics generate instruction code. Each word describing this process is found and executed in turn. Arithmetic and stack management words will be found in FORTH as usual. The main difference is that the generated instructions are placed in the Target Dictionary on disk.

Occasionally it is necessary to obtain the location of a variable or the value of a constant to use it within a CODE definition. In the resident assembler you would need only to refer to its name which would be executed to provide the desired value. Within cross assembled code, however, such names would not even be found and even so their execution would not provide the desired result because they are defined by EMPLACE. For this reason the word ' is redefined to search TARGET for the word following, and to return on the stack the location of its parameter field in the Target Dictionary. After searching TARGET, ' returns both CONTEXT and CURRENT to HOST.

To illustrate how ' would be used to provide the intended value of a name, assume the following words to have been defined thus:

```
12 CONSTANT CON      0 VARIABLE VAR      12 USER USE
CODE INST ...        : DEF ;             TABLE TBL
```

The evaluation of each of these words within either the resident or cross compiler/assembler is given below:

Resident	Cross
CON	' CON @
VAR	VAR
TBL	' TBL
USE	' USE C@ origin +
' INST	' INST
' DEF	' DEF

(The word "origin" is used above to represent the address of the base of the user area.) The important thing to note in the above table is that all of the forms used in the "Cross" column will in fact produce results identical to the corresponding forms in the "Resident" column even if those forms were interpreted by the resident interpreter. Clearly the cross compile form should be used from the onset of coding in order to avoid rewrites for cross compilation.

Note also that to evaluate the location of a cross compiler VARIABLE (or CVARIABLE) the word ' is not required. The name of the variable is sufficient. Special effort is taken in defining such words. At the same time that an EMPLACE entry is placed in TARGET, a normal CONSTANT definition for the same name is entered into HOST. The value of the constant is the location assigned to the variable. Thus each VARIABLE or CVARIABLE definition requires twice the host memory of any other cross compiled word.

16.2.6 EQU and LABEL

The defining words EQU and LABEL are unique to the cross compiler environment. For many of their uses no equivalent form exists for the resident compiler. They do not make entries in either the Target Dictionary or the Target Vocabulary. What they do is to enter CONSTANT definitions into the Host Vocabulary. Such definitions then are executable but not compilable, hence they can be referred to only in

assembler code or HOST interpreter.

EQU defines a named constant. It "equates" a value to a symbol. The definition

```
6 EQU PRINT
```

might be used to declare a printer port address to be 6 and to name that port PRINT.

Such a definition costs nothing in Target Dictionary space but can increase readability of code as well as facilitate configuration changes. For the resident compiler you can define EQU to be CONSTANT. Remember that a resident EQU definition will occupy real dictionary space at whatever point it is defined, (i.e. don't define an EQU in the middle of a CODE definition).

LABEL is defined as HERE EQU, where the HERE refers to current top of the Target Dictionary. LABEL has no possible counterpart for the resident compiler because the act of producing the definitions will disturb the value of HERE (as well as that portion of dictionary being labeled). You can either overlook this incompatibility, working around it for the resident compiler, or else restrict your use of LABEL to entirely cross assembled applications, as in a boot PROM.

17.0 THE CROSS COMPILING PROCESS

17.1 Procedure

When your application has been coded and tested to your satisfaction using normal resident compilation you will want to begin cross compiling it. First generate a cross compile load screen as outlined in the Cross Compiler chapter. With that done you are ready to begin a cross compilation.

Before actually beginning any cross compile you must first insert a spare disk into the second drive. This disk will receive the generated core image object program (i.e. the Target Dictionary) in screens 240 thru 249 (490 through 499 relative to drive 0). No other part of either disk will be modified by the cross compiler. These 10 screens represent 10K bytes of dictionary. This should be more than enough for all but the most enormous of applications.

To load the cross compiler type CROSS LOAD and then load your application's load screen. As your application is loaded, the cross compiler will generate a free-form load map of each word as it is compiled. Each individual load command will begin a new line with the screen number (in hex) that is being loaded.

If the compile should stop on an unknown word it will probably be due to one of two errors. Either you have used a word known to the development system that is not an element of this Nucleus, or your cross compile load screen differs from the resident compile load screen in providing for that word to be defined before it is needed. In either case you must return to the EDITOR and correct the omission.

If your EDITOR is resident simply type EDITOR, list the screen in error, and make

FORTH

your corrections. To return to the cross compiler type FORTH HOST and then load your load screen once more. Reloading CROSS is not required. If on the other hand your EDITOR is non-resident (as on COSMAC system) you must type EDIT LOAD and then make your corrections as above. Loading the EDITOR will cause the cross compiler to be discarded. You must therefore type CROSS LOAD again before beginning another compile.

When your compilation is complete you will be left in HOST DEFINITIONS. What this means is that you are in the wrong Vocabulary to FORGET TASK. The correct sequence for deleting the cross compiler is:

```
FORTH DEFINITIONS   FORGET TASK   : TASK ;
```

You may want to include the phrase FORTH DEFINITIONS at the end of your load screen.

17.2 The Cross Compiler Map

Figure 15 shows an example of the free form map produced by a cross compile. The map is a result of redefining the words LOAD and CR, and the action of the word LOG that is nested within all cross compile defining words. LOAD is redefined to issue a CR and then type the screen number before loading it. (Note that the word . is redefined to output always in hex.) CR is then redefined to perform a CR and a SPACE. CR should be edited at an appropriate point (not inside a definition) within any screen whose output exceeds one line, causing subsequent lines to be indented.

The majority of the output is produced by LOG. LOG will output the parameter field location (or symbol value for EQU definitions) and then the name of each word as its definition it begins. For PROM compiled variables the RAM address is output after the name, followed by an extra space to separate it from the next address. If you are not working from a hard copy terminal you may wish to produce a compiler map on the hard copy unit. To force all cross compiler output to hard copy on systems with separate printing terminals, simply type PRINTER LOAD before typing CROSS LOAD. (Note: PRINTER LOAD is not supplied with all systems.)

```
78
45
48 3C00 P 3C02 U 4C WHPUSH 41 HPUSH 42 NEXT 48 WNEXT
9
 E2 'VARIABLE' E7 'CØNSTANT' FC 'USER' FC 'DØES>' 10F ':'
49
43
44 23C C€ 245 < 24D /MØD 255 MAX
7A 3C12 @PØRT 3C1F !PØRT 27F RAM
7B 291 +BIT 2A4 +DIGIT 2B7 BIT 2C4 DIGIT 2D1 NUMBER
7C 2FB -BIT 316 -DIGIT 333 !BIT 348 !DIGIT 35D BCD 36B REV 376 DISPLAY
7D 39A 1MS 3A5 MS 3B3 .1SEC 3C4 PØRTS
20 3E9 E*
7E 40E SIDE 3C02  412 !SIDE 421 LEFT 42A RIGHT 43F DUPLICATED 451 RESULT
80 466 BØTH 476 PASS 47F FAIL 488 CLEAR 49B PRESSURIZE 4A1 ENABLE
 4A7 NEWPØRT 4AD BLACK 4B3 ADVANCE 4B9 CØNVERT 4BF LABELS
 4C5 STØP 4CB MCLR 4D1 CLAMSHELL
7F 4E4 AUTØ 4EA BLØØD 4F0 DIALYSATE 4F6 START 4FC RETEST 502 DØNE
 508 INHIBIT 50E NØRM 514 READY 51A ATM 51E A/D 529 PRESSURE
 546 SENSITIVITY 54C TEST-TIME 552 DELAY 558 SERIAL 55C LØT
82 560 CHAR 57E SPACE 587 SPACES 595 CR 5A8 CR
 5B1 DIG 5BC PRINT 5CC REJECT 5EE S/N 3C03  5F2 LABEL
81 62F C# 3C05  633 0C0L 63E CHAR 66C SPACE 66F SPACES
 67D CR 698 LØG 6A8 T/N 3C06  6AC SERIAL# 6B9 TEST#
83 6D1 FAILED 6E3 TEST 742 LEAK 7C9
```

Figure 15

17.3 Core Image Output

The core image area on disk contains the output of your finished cross compile. What remains is to transfer this program in some form to your target system memory. If the program must load into RAM it is the user's responsibility to determine formatting requirements based on the boot media and boot routine to be used, and to provide appropriate custom output routines. Outputting the program for PROM or ROM memory is easier to consider in the general case.

There are in general two separate approaches to PROM programming. The most direct is to output the generated program into the PROMs through a PROM programming interface attached to the development system. Software to drive such interfaces is available for systems that have them. The operation of such software may vary between systems and is therefore documented separately when delivered.

Another way to program PROMs involves outputting the program to some media in a format appropriate to some external programming device. This is most often paper tape. A paper tape formatting routine is provided as standard and can be operated by the following procedure.

1. Before loading the cross compiler you must first provide access to a punch device. If your terminal is a TTY then skip this step; you already have such access. If you have a TTY as a hard copy device then load the PRINTER screen as described earlier (Section 17.2). If you have a high speed punch drive then load its software just as you would PRINTER, before doing a CROSS LOAD. The important thing is to have ECHO so defined as to output a single character to a punch device.

2. Type CROSS LOAD and then load your application. If the application has already been compiled and is in the Target Dictionary you need not recompile it. If you have just finished a compile and have not yet deleted the cross compiler you need not reload it.

3. Type OUTPUT LOAD. This loads the core-image output word PROMS and leaves you in hexadecimal.

4. If the size of your PROMs is not 512 bytes you must change the value of the variable SIZE. For 1K PROMS type:

400 SIZE H!

5. If the punch device is not the console device then bring it up and online now. If the punch device is a console TTY then do not enable the punch unit until after executing step 6.

6. Type:

start-addr #proms PROMS

where "start-addr" is the first location in the Target Dictionary to begin punching, and "#proms" is the number of PROMs that will be needed to contain the entire program. If your punch unit is a console TTY then enable the punch unit just after the carriage return for the above command, while the leader is being output.

The word PROMS will output 200 characters of blank leader followed by "#proms" records of "SIZE" bytes each and then 200 characters of blank trailer. Each record consists of a sentinel character of all ones followed by "SIZE" bytes of program in binary format and then 100 null characters for record separation. Most PROM programmers are capable of accepting such a format.

17.4 Program Dumps

At some point in the cross compilation cycle you may wish to take a hex dump of the Target Dictionary, either to verify the correct compilation of some application word, or to provide a final hard copy record of the contents of the PROMs. The word DUMP is defined in the OUTPUT screen along with PROMS. You must load it exactly as you would to punch a PROM tape although providing for hard copy is optional depending on your needs. You then use it just as you would use the resident DUMP, that is,

start-addr length DUMP.

17.5 Relocating and Expanding the Target Dictionary

The disk address of the start of the Target Dictionary is controlled by the value of the constant NEW of screen CROSS. Its value is a hexadecimal block address (not a screen number). If you have the PROM programming software, it will contain a definition of NEW, which should be the same.

The size of the Target Dictionary is assumed in three places: the MIN functions in ADRS on of screen CROSS, the loop limit for CLEAR of the screen titled "TARGET VOCABULARY," and the ORG of screen OUTPUT. Relocating the Target Dictionary should not be undertaken lightly. It should be moved only in case of a drive failure or an extremely large application.

18.0 EXTENDING THE CROSS COMPILER

Just as for a resident compilation, you can extend the capabilities of the cross compiler by adding your own compiling or defining words. For the most part such additions can be defined just as they would be for the resident compiler (see Chapter 13.0 and Section 5.8), even to the point where the same definition could be used by either environment. Using the same definition in both environments is very desirable from the standpoint of reliability.

The meanings of many words used in compiling and defining words however are just a little different under the cross compiler. Further confusion is added by being able to use words from either of two Vocabularies: FORTH or HOST. Finally there are some functions which probably cannot be defined in a manner that is transparent to the type of compiler.

18.1 Defining Words

As an illustration lets consider a definition and use of ARRAY as it might occur in a resident compilation.

```
: ARRAY   0 CVARIABLE  1 -  DUP H +!  HERE SWAP ERASE ;
20 ARRAY INPUTS
```

The purpose of ARRAY is to define a named region of memory of some specified byte length and to initialize that region to zeros. Let's explore how to define such a word for a cross compiled application. Assume, for the sake of argument, that the cross compilation is destined for RAM.

The first problem encountered in trying to use this same definition for the cross compiler is that the : used is the wrong one. The cross compiler : will try to define ARRAY as an application word and enter it into the Target Dictionary. In fact the intent here is to define ARRAY within HOST as a normal resident definition so that it may be used during the compilation to make new target words. To do this you must define ARRAY by means of the old : that has been redefined as H: for safekeeping. Then, for compatibility, define H: in your resident load screen as

```
: H:   : ;
```

H: may now be used to add definitions to the Host Vocabulary.

The second problem is unique to the 8080 cross compiler. The word H is the name of an 8080 register defined by that name in both the assembler and cross assembler. The resident assembler is in a separate Vocabulary so no conflict occurs there. The cross compiler assembler is defined in HOST however, and precludes the use of H for a dictionary pointer. The name DP is therefore used instead of H within the 8080 cross compiler. There are two possible solutions. Either define:

```
: DP   H ;
```

in the resident load screen and use DP +! in ARRAY, or define:

```
: ORG   H ! ;
```

for the resident compile and use the phrase HERE + ORG in ARRAY. The use of ORG in the preferred form because it is machine independent. (ORG is defined for the cross compiler by CROSS LOAD.)

The last problem is that ERASE will clear resident memory, not Target Dictionary. One solution is to make a definition for ERASE that can be added to the cross compile load screen.

```
H: ERASE   OVER + SWAP DO  0 I C!  LOOP;
```

This is certainly not the only way to define ERASE, nor is it the best. It is given here solely because it introduces a new complication: its 1 is the wrong 1. If left

as is the I used would be that of the cross assembler. The IMMEDIATE word FORTH must be added in front of the I to ensure finding the correct version. If left in FORTH however, the C! used would be FORTH's, not the one that would store into the Target Dictionary. The word HOST is made immediate for just such a case. After inserting HOST between the I and C! the correct definitions would look like this

```
H: ERASE   OVER + SWAP DO  0 FORTH I  HOST C!  LOOP ;
```

Should a reference to HOST be required within a screen common to either compiler, you could define HOST for the resident compiler by

```
: HOST   ASSEMBLER ;  IMMEDIATE
```

Let's re-examine the final definition for ARRAY.

```
H: ARRAY   0 CVARIABLE  1 -  DUP HERE + ORG  HERE SWAP ERASE ;
```

If the phrases HERE + ORG and HERE SWAP ERASE were merely exchanged, the definitions would not only not work, it would probably crash the system the first time it was used. This would come about because HOST versions of C! used in ERASE would store into resident memory. Recall that the HOST words C@, C!, @ and ! were all redefined to select between target memory and real memory based upon examination of their addresses. The assumption is that it is not reasonable in a FORTH program to refer to locations outside the active dictionary. The rule employed in the cross compiler for these four words can be stated thus:

> All addresses less than or equal to the target HERE and greater than or equal to the value of W0 (the address assigned to the bottom of the core image area) will refer to Target Dictionary, all other addresses will refer to Host memory.

By not including the new region into the active Target Dictionary with the ORG until after the ERASE, the C! in ERASE must assume that the Host memory is being referenced.

If the amount of accommodation and careful coding that was required for the single word ARRAY seems excessively severe, remember that it was chosen as an example in

order to illustrate as many problems as possible. If you intend to create your own defining or compiling words for the cross compiler, you must learn to recognize such problem areas and work around them. Don't ignore the possibility that many problems might go away if the word were differently structured. The following example, for instance, would work equally well in either environment, given only the resident definition for H:

```
H: ARRAY   0 CVARIABLE  1 DO  0 C,  LOOP ;
```

In summary, the user wishing to implement his own defining words in a cross compilation should keep the following points in mind:

1. Define your defining words with H: rather than : .

2. Remember that HOST C@, C!, @, and ! refer to Target Dictionary only in the range from W0 to HERE.

3. Be sure that the words used in the definition of the defining word are coming from the correct Vocabulary (FORTH or HOST). FORTH is searched after HOST, so this is not a problem unless you wish to use a FORTH word that is defined differently in HOST. Note especially that the entire assembler is in HOST. In addition to ! (and, on 8080 systems, H), this buries the FORTH compiling words IF, ELSE, THEN, BEGIN and END.

18.2 Compiling Words

Unlike defining words, compiling words must be in the TARGET Vocabulary, so that they can be found by COMPILE during cross compilation of colon definitions. However, because compiling words must manipulate the stack during cross compilation, they must be defined in terms of FORTH words such as SWAP and HOST Words such as C,.

To achieve this, place an H: definition of the compiling word in the cross compile load screen, and follow the definition by IMMEDIATE. In this way the compiling word will be defined initially in HOST, and then be transferred to TARGET by rearranging the dictionary links (this is the function of the HOST redefinition of IMMEDIATE).

For example, the defining word [discussed in Chapter 8.0 (OUTPUT) might be defined

```
: COUNT   DUP 1+  SWAP C@ ;
: [   I COUNT  DUP 1+ R> + <R  TYPE ;
HEX H: [   \ [  5D WORD  HERE C@ 1+  HERE +  ORG ;      IMMEDIATE
DECIMAL
```

The first [defined must become part of the Target Dictionary, and consequently has a : definition. Note that the user will have to supply his own definition of TYPE, for whatever device he intends his output.

The second [, on the other hand, is not intended to be executed on the Target System, but rather during cross compilation, so it is defined by H: and made IMMEDIATE.

(The word \ was described in Section 5.8.)

19.0 A TYPICAL DEVELOPMENT CYCLE

Let us examine the cross compiler from the standpoint of how it would be used in a typical development cycle. Although a development cycle using FORTH remains quite flexible, it will probably contain each of the following four major phases:

19.1 Research and Design

During this phase preliminary hardware-software tradeoffs are determined, the I/O environment is outlined, and the major elements of the software are blocked out. Often the most productive way to outline the organization of the functional modules is by actually coding a few of the highest level definitions in FORTH. Having done this you will have divided up the functional responsibility into conceptual modules, given descriptive names to those modules, and illuminated the flow of control between them. Since FORTH is interactive, you will want to use your terminal from the earliest stages, editing definitions into blocks as you create them. This is the technique known as "top-down design". You will be well on your way toward the coding of your application.

19.2 Coding and Testing

The dividing line between design and coding is not clear cut, but can be thought of as centering around that period where the description of some portion of the program has reached a sufficient level of detail that it becomes practical to try loading it. That is when the fun begins, for then you begin interacting directly with your application. This could also be a period of great frustration if you have no way to communicate with special hardware which is not available on the development system.

If your application performs little I/O and much computation such a situation is tolerable. But if yours is a control-oriented problem you will be short circuting perhaps the most powerful feature of FORTH should you be unable to communicate with your target devices: that of truly interactive program development.

This interactive nature of FORTH is a function of the development system's resident compiler and word interpreter. It is not available to a cross compiled program. In this respect the cross compiler more closely parallels the conventional concept of a compiler. Its output does not expect the support of the development system, nor can it be executed until it is sufficiently complete that it can stand on its own (and is transfered to the target system). When it does not work you must spend many hours and much inductive logic trying to track the trouble to its source (if indeed it has only a single source). This do-or-die methodology has led to the proliferation of debugging tools such as debug monitors, hardware traces, hardware breakpoints, in-circuit emulators and logic analyzers.

The modular nature of FORTH programs and the accessibility of all levels of a definition facilitates the testing of each word in a vocabulary before it is allowed to wreak havoc upon half a dozen other words. Small, throw-away test cases become unnecessary and large, untested programs cease to exist. The FORTH method assumes, however, that when testing I/O code you have access to ports that are connected to real devices. The easiest means to accomplish this is to assure bus compatability between the development and target systems. Then all that is necessary is to plug the interface cards into the development computer and invoke your control words from the terminal, using the resident compilation of your application. By this means you may test the majority of your application before ever cross compiling it.

If this is not possible you may need a bus extender module to connect the development system's address, data and control lines to the target system's bus. This is usually referred to as In-Circuit Emulation, although the word is occasionally used to mean much more, such as externally controlled breakpoints or traces. Lacking any such inter-system communication, your task becomes more difficult and will almost certainly require several passes through the cross compilation phase.

19.3 Cross Compiling

During the cross compile phase the tested application is passed through the cross compiler to produce an object program on disk. The object program is then transferred to a ROM or PROM. Care must be taken during this phase to avoid introducing any new errors. One potential source is in the initialization code. Because the application must stand alone once cross compiled, the burden of system initialization is placed on the user. This is not an especially difficult task but it should be carefully desk checked to avoid any careless errors.

The following four functions must be included in the initialization code:

1. Load the stack pointer with address of the bottom of the stack. Remember that the stack will grow from high to low memory and is decremented before each push. It should therefore be set to the highest address plus one of the region you wish to allocate for parameter stack. As it is often difficult to determine the amount of stack required, it is simplest to place the parameter stack immediately beneath the return stack in high memory. Then it is free to use all available memory.

2. Load the return stack pointer with the address of the bottom of the return stack, usually the last RAM address plus one (the return stack also grows from high to low memory.) Be sure to allow sufficient return stack for your needs. Remember that active DO loops use 4 bytes of return stack and each active call to a colon definition uses 2 bytes. The procedure for determining return stack requirements is tedious but straightforward. An alternate method is to make a generous guess and then add a fudge factor. You should count up all uses at least once to gain confidence in your ability to estimate them. A return stack overflow is often fatal and always damaging. 64 bytes is usually adequate.

3. Load the interpreter pointer with the address of the parameter field of the outermost definition in your application. This will be the starting point for the interpreter. For this address to be known you must arrange for the initialization code to be the last thing

compiled.

4. Transfer control to NEXT just as you would at the end of a CODE definition.

Other items to be initialized might include the User area base pointer (if you have made use of USER defined variables) or certain regions of RAM that may require known initial states. In addition interupts may need to be enabled and programmable I/O ports will need to be programmed.

19.4 Installation and Checkout

The last phase in developing a cross compiled application is to install the PROMs into the target system and verify that the system functions to specification. Trouble at this stage could be due to one of the following reasons:

1. Failure of an untested definition. If it was not possible to test each definition from the host computer then extra effort is required to desk check such definitions more thoroughly. If a CODE definition may be at fault it could be traced in the target system with the use of the logic analyzer (if available). If a high level definition might be at fault it should be broken up and tested in pieces, substituting "stubs" in place of any untested words it may use. (A stub is a word with a temporary definition, designed to approximate or simulate the definition which will eventually replace it.)

2. Failure due to cross compiling. Check the initialization code thoroughly. Check the operation of any compiling or defining words you may have added by inspecting dumps of the object they produce. In general, suspect all dissimilarities in cross compiled and resident compiled source screens. Reread the section "The Cross Compiler Environment" and be sure you understand the operational differences between the resident and cross compiler.

3. Hardware failure. Hardware problems are best solved with an engineer. You might also find an oscilloscope, logic analyzer, or in-circuit emulation useful.

APPENDIX A
microFORTH IMPLEMENTATION on the RCA COSMAC (1802)

In this appendix we assume that you understand and can use the hardware functions of the COSMAC (described in manuals supplied by RCA). In addition, you will need to have worked through all previous chapters of this Technical Manual in order to be able to use the information given below.

ALL numbers are given in hexadecimal unless otherwise specified.

1.1 DATA FORMAT

The main memory of the COSMAC is accessed in eight-bit bytes. microFORTH provides several words for accessing bytes (CVARIABLE CZ C@ C! C). For the most part, however, microFORTH handles data in two-byte pairs, called words. Both the parameter stack and the return stack can be regarded as a stack of words, each of which is sixteen bits wide. The high-order part of a word is in the byte with the lower address (and this byte provides the word's address).

1.2 REGISTER ALLOCATION

Certain registers have been reserved for system functions and are customarily referred to by their letter names, rather than by numbers:

Name	Register	Assignment
S	D	Data stack pointer
I	C	Next word to be executed
U	4	Address of user's memory
T	9	Temporary register
A	A	Auxilliary register
W	B	Current word being executed
P	3	Program counter
R	E	Return stack pointer

Registers 0, 1, and 2 are used by the hardware for DMA and interrupt handling.

Registers 5 to 8 are not used by FORTH and are available for the user to assign. You may also use registers W, T, and A (9 - B) for scratch registers when no code beginning word is used, or registers P, T, and A (3, A, 9) when a code beginning word is used. (Code beginning words are described in Section 1.5.)

1.3 ASSEMBLER MNEMONICS

The mnemonics of the various COSMAC operation codes have been defined as words which, when executed, assemble the corresponding operation code at the next location in the dictionary. As with other FORTH words, the operands of a mnemonic (i.e., register numbers or names, immediate data, and modifiers) must precede the mnemonic. For example, to select Register S as the index register, you use:

```
S SEX
```

This rule holds for all instructions that specify a register.

1.3.1 Modifiers of Mnemonics

The word +C is used to direct the mnemonic which follows it to assemble an instruction that utilizes the carry flag (DF). This word may be used before the following mnemonics:

ADD SM (subtract with borrow) SD SHR SHL

The word # is used to direct the mnemonic which follows it to assemble an immediate instruction. This may be used with the following mnemonics:

OR AND XOR ADD SM SD LD

For readability, the mnemonic LD has been defined as identical to LDX , the load from index register, for use in assembling "load immediate" instructions.

To use both modifiers together, you could declare:

```
0 # +C ADD
```

which assembles an instruction to add the carry to the accumulator. (This is can be used to propagate the carry from a low-order byte to a high-order byte.)

1.4 TRANSFERS

Program control is effected by the words:

```
BEGIN ... END
IF ... ELSE ... THEN
BR
NEXT
```

NEXT is defined as F SEP . Register F (decimal 15) contains the address of the inner interpreter. All CODE words should return control to the inner interpreter by ending with NEXT or a macro that assembles NEXT , such as PUSH (discussed below).

The words IF , ELSE , and END assemble short (two-byte) branch instructions (instructions 30 through 3F). IF and END should each be preceded by one of the condition codes enumerated in the following table. The branch is taken if the condition is not met.

Mnemonic	Condition
Q	Test for Q set
0=	Test for accumulator zero
DFL	Test for accumulator carry flag (DF) set
<	Test for accumulator borrow (DF reset)
0<	Test for sign bit of accumulator on (destroys accumulator)
n EFL	Test for external flag number n set (n = 1 - 4)

If any of these conditions is followed by NOT , the condition is reversed. For example, < is defined as DFL NOT .

Since the program control words all assemble short branches, the source and destination of a branch must be on the same page. This can be guaranteed by the word PAGE , which advances the dictionary to the start of the next page if there are not as many bytes left in the current page as the number on top of the stack. To determine how many bytes are needed, count eight for the dictionary entry and add the number of bytes up to and including the last BR , END or the first byte after THEN . Each immediate instruction requires two bytes, as does each IF , ELSE , END , and BR .

For example, the definition of ERASE is:

```
13 PAGE
CODE ERASE   2POP   BEGIN   BEGIN   0#LD   A STR   A INC
     T DEC   T GLO   0= END   T GHI   0= END   NEXT
```

In this case the count extends to the second byte of the second 0= END .

The COSMAC's long skip instructions are occasionally useful for conditionally skipping over the following byte. The mnemonic LS is used to generate a long skip and is preceeded by the condition code for the test. The generated instruction will skip whenever the condition is met.

The code P INC may be used to assemble an unconditional skip.

1.5 CODE BEGINNING WORDS

At the start of a code definition it is frequently useful to take parameters from the stack, place them into the A and/or T registers, and increment the stack pointer. Since this is a rather tedious exercise on the COSMAC, the microFORTH assembler includes code beginnings (i.e., sharing of the entry pointer) which will automatically take items off the stack and place them into the registers. This can greatly reduce instruction overhead in a code definition.

To implement these code beginnings, however, it was necessary to reverse the use of the W and P registers (the currently executing word pointer and the program counter, respectively). When a code definition uses a code beginning, the program counter is placed in W and the currently executing word pointer is placed in P . P becomes a scratch register (instead of the usual W). A no-op instruction, then, is no longer P SEP but W SEP in any code definition that uses a code beginning. These two registers were switched so that W can be declared as the program counter (which means that the code

beginnings will return you to your code definition upon completion.

The code beginnings are:

```
BINARY
2POP
1POP
PUT
```

Each will be discussed in terms of the following two items: 1) stack pointer result and 2) registers used.

The BINARY code beginning increments the stack pointer by three bytes (a pop operation) and leaves it pointing to the low-order byte of what was the second stack item. The code may then conveniently place an item (a PUT operation) on the stack. The stack pointer is placed in the index register. In BINARY, the high-order part of the top stack item is placed in the high-order byte of the T register. The low-order byte of the top stack item is placed in the low-order byte of the accumulator.

The PUT code beginning increments the stack pointer by one byte so that a value may replace the current top stack item. In PUT , the top stack item (sixteen bits) is placed in the T register. The low-order byte of T is also in the accumulator.

The 2POP code beginning increments the stack pointer by four bytes. The top stack item is placed in the T register and the second stack item in the A register. The low-order byte of A is in the accumulator.

The 1POP code beginning increments the stack pointer by two bytes. The top stack item is placed in the A register. The low-order byte of A is in the accumulator.

1.6 MACROS - EXTENDING THE ASSEMBLER

Since the mnemonics are defined as executable instructions, they can be compiled into colon definitions that will function as macros. The following examples are part of standard microFORTH; you may wish to write others. The macro DST is defined as:

```
: DST   DUP DEC  STR ;
```

Another useful macro is:

```
: 0#LD   F GHI ;
```

which makes use of the fact that the high byte of Register F is always zero. The macro:

```
: PUSH   S DST  NEXT ;
```

may be used to push the accumulator onto the stack and return to the inner interpreter.

1.7 USE OF THE ALLOCATED REGISTERS

The Registers S and R are used to contain, respectively, the address of the top byte of the parameter stack and the return stack (top byte equals lowest address). Any CODE words which manipulate these stacks must be careful to readjust the pointers before returning to NEXT .

Register U contains the starting address of the user area. The defining word USER is used to name locations relative to the start of the user area. When a USER variable is invoked, this relative address is added to the low-order byte of U and the result (including the high-order byte of U) is placed on the stack. There is no carry from low-order to high-order byte, so the user area cannot cross a page boundary.

Registers I and W are used by the inner interpreter. The heart of the interpreter is:

```
I LDA    W PHI    I LDA    W PLO
W LDA    P PHI    W LDA    P PLO
```

followed by P SEP . Here it is assumed that I is pointing at an address that is compiled into a colon definition. The first phrase transfers this address to W while advancing I . W now contains the address of the code address of the word to be executed. The second phrase moves the code address to P .

The user will not normally want to alter I , since this will alter program flow after returning to NEXT . This is not true, however, for W , since W can usefully transmit the starting address of the parameter field of the word being executed. Consider, for example, the definition of CVARIABLE :

```
: CVARIABLE   USER ;CODE   W GLO   S DST   W GHI   PUSH
```

Using CVARIABLE , we define CVAR :

```
0 CVARIABLE CVAR
```

Now when CVAR is invoked the code following the ;CODE in the definition of CVARIABLE will be executed. This code must put the address of the parameter field of CVAR on the stack. This is done easily by transfering W to the top of the stack.

1.8 INTERRUPT HANDLING

The COSMAC provides only one level of interrupt. When an interrupt occurs, the P and X registers are saved in a special register (T), then P is set to one and X is set to two. Consequently, Register 1 must contain the address of the interrupt handler. Note that setting X to two means that it contains the register designated by microFORTH to point to the top of the return stack.

Since certain FORTH words (I , LOOP , +LOOP) temporarily point R away from the top of the return stack, you can not reliably save T there. Instead, we suggest that some other free register (i.e., 5 - 8) be initialized to point to a save area. In the interrupt handler, use a SEX instruction to designate this register as X prior to a SAV (78) to save T . The interrupt handler should also save the accumulator and carry bit, if these will be altered, plus any registers needed by the handler.

After processing the interrupt, execution must be resumed where it left off at the time of the interrupt. Also, the starting address of the interrupt handler must be put back into Register 1. This is conveniently done by branching to a RET instruction (70) in the byte just prior to the entry point of the interrupt handler. Of course any other restoration, such as of registers, accumulator, or carry, must be done first.

1.9 TIMING CHART

The next page features a list of operators and the number of instructions it takes to implement them, including the count for NEXT . The execution time of a CODE word can be found by adding the number of its instructions, including the branch to NEXT , plus ten for NEXT's execution time.

The execution time of a word defined by a ;CODE defining word is computed similarly, as the time of the code phrase in the definition of the defining word.

To obtain the execution time of a colon definition, add the times of its components, plus twenty-one and fifteen, for entry and exit. For example, the execution time of:

```
: ROT   <R SWAP   R> SWAP ;
```

is:

```
21 + 19 + 28 + 19 + 28 + 15 = 130
```

The execution time of a word defined by a DOES> defining word is the sum of the times of the phrase DOES> through ; inclusive in the definition of the defining word.

TABLE A-1. TIMING CHART

WORD	NUMBER OF INSTRUCTIONS	
NEXT	10	
EXECUTE	17	
8-bit literal	19	
16-bit literal	20	
DO	29	
LOOP	36	if loop terminates, 43 if loop is repeated
+LOOP	37	if loop terminates, 44 if loop is repeated
IF and END	18	if condition is true, 25 if condition is false
ELSE and WHILE	19	
RAM VARIABLEs, CVARIABLEs, and TABLEs	17	
CONSTANTs, PROM VARIABLEs, and CVARIABLEs	19	
USERs	19	
DOES>	27	
:	21	
;	15	
AND	22	
+	22	
-	21	
U*	106	to 114
U/	119	to 127
MOVE	26	+ (6 * count) + (2 * ⌈count/256⌉)
DUP	19	
DROP	13	
SWAP	21	
OVER	26	
@	23	
!	21	
+!	22	
C@	21	
C!	19	
0=	20	if true, 21 if false
0<	18	if true, 20 if false
<R	19	
R>	19	
I	20	
2POP	9	
1POP	5	
BINARY	6	
PUT	5	

1.10 USER AREA MAP

Offset	Name
0 - 4	Reserved for multiprogrammer
5	S0
7	BASE
8	H
A	CONTEXT
C	CURRENT
E	STATE
F	BLK
10	IN
12	OFFSET
14	SCR
16	R#
18	MODE

APPENDIX B

GLOSSARY

microFORTH on the RCA COSMAC

COSMAC GLOSSARY

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
#	ASSEMBLER	6	0	0

Sets the immediate bit in the user variable MODE , for use by the instruction mnemonic.

+C	ASSEMBLER	6	0	0

Sets the carry bit in the user variable MODE , for use by the instruction mnemonic.

0#LD	ASSEMBLER	7	0	0

Sets the accumulator to the value 0. Loaded from high-order Register 15.

0<	ASSEMBLER	8	1	0

A macro that tests for the sign bit in the accumulator. When executed, the contents of the accumulator are destroyed.

0=	ASSEMBLER	8	0	0

A constant that will set the 0= condition code.

1POP	ASSEMBLER	20	1	0

Pops the top stack item into the A register. The stack pointer is incremented by two bytes. The low-order A register remains in the accumulator. A code beginning.

1RG	ASSEMBLER	6	1	0

Creates a "one-argument class" instruction mnemonic.

2POP	ASSEMBLER	20	2	0

Pops the top stack item into the T register and pops the second item on the stack into the A register. The stack pointer is incremented by four bytes. The low-order A register remains in the accumulator. A code beginning.

<	ASSEMBLER	8	0	0

Sets the condition code for DF reset (i.e., a borrow).

A	ASSEMBLER	6	0	0

A constant that declares A to be Register 10, a scratch register.

ADD	ASSEMBLER	6	1	0

Instruction mnemonic.

ALU	ASSEMBLER	6	1	0

Creates an "arithmetic/logical class" instruction mnemonic.

AND	ASSEMBLER	6	1	0

Instruction mnemonic.

BINARY	ASSEMBLER	20	1	0

Pops the high-order byte of the top stack item into the high-order byte of Register T and pops the low-order byte of the top stack item into the accumulator. The stack pointer is incremented by three. A code beginning.

COSMAC GLOSSARY

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
BR	ASSEMBLER	8	1	0

Given an address, assembles a short unconditional branch to that address.

DEC	ASSEMBLER	6	1	0

Instruction mnemonic.

DFL	ASSEMBLER	8	0	0

A constant that will set the DF (carry flag set) condition code.

DST	ASSEMBLER	7	1	0

A macro that decrements the specified register, then stores the contents of the accumulator at the address given by the register.

EFL	ASSEMBLER	8	1	0

Will set a test for the "external flag set" in the instruction. The value on the stack is the flag number (1-4).

ELSE	ASSEMBLER	8	1	1

Assembles a short unconditional forward branch. Completes the branch whose address is on the stack and leaves its own address.

END	ASSEMBLER	8	2	0

Given a condition code on top of the stack and an address beneath, assembles a short conditional branch to the address.

ENTRY	ASSEMBLER	7	1	0

Declares a word such that, when invoked, its value is placed in HERE 2 - . The new word can only be executed under the ASSEMBLER vocabulary and at the beginning of a CODE definition.

GHI	ASSEMBLER	6	1	0

Instruction mnemonic.

GLO	ASSEMBLER	6	1	0

Instruction mnemonic.

I	ASSEMBLER	6	0	0

A constant that declares I , the inner interpreter pointer, to be Register 12.

IF	ASSEMBLER	8	1	1

Given a condition code on the stack, assembles a short conditional forward branch. Leaves the location of the branch on the stack; ELSE and THEN complete the branch on the stack.

COSMAC GLOSSARY

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
INC	ASSEMBLER	6	1	0
	Instruction mnemonic.			
INP	ASSEMBLER	6	1	0
	Instruction mnemonic.			
LD	ASSEMBLER	6	1	0
	Instruction mnemonic. Same as LDX .			
LDA	ASSEMBLER	6	1	0
	Instruction mnemonic.			
LDN	ASSEMBLER	6	1	0
	Instruction mnemonic.			
LDX	ASSEMBLER	6	1	0
	Instruction mnemonic.			
LDXA	ASSEMBLER	6	1	0
	Instruction mnemonic.			
LS	ASSEMBLER	8	1	0
	Given a condition code on the stack, assembles a conditional long skip. Skips on condition true.			
NEXT	ASSEMBLER	7	0	0
	Sets Register 15 to be the program counter. Register 15 contains the address of the inner interpreter. A code ending.			
NOT	ASSEMBLER	20	1	0
	Reverses the value of the condition code on the stack.			
OR	ASSEMBLER	6	1	0
	Instruction mnemonic.			
OUT	ASSEMBLER	6	1	0
	Instruction mnemonic.			
P	ASSEMBLER	6	0	0
	A constant that declares P , the program counter, to be Register 3.			
PAGE	FORTH	4	1	0
	Verifies that there are enough bytes on the current page in memory to contain the number of bytes specified by the top of the stack; otherwise advances H to the next page. Usage: #bytes PAGE (new definition)			
PHI	ASSEMBLER	6	1	0
	Instruction mnemonic.			

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
PLO	ASSEMBLER	6	1	0
	Instruction mnemonic.			
PUSH	ASSEMBLER	7	0	0
	A macro that decrements the stack pointer and pushes the contents of the accumulator onto the stack before performing NEXT . A code ending.			
PUT	ASSEMBLER	20	1	1
	Pops the top stack item into the T register and leaves the stack pointer positioned to the low-order byte of the top of the stack. The low-order T remains in the accumulator. A code beginning.			
Q	ASSEMBLER	8	0	0
	A constant that will set the Q condition code.			
R	ASSEMBLER	6	0	0
	A constant that declares R , the return stack pointer, to be Register 14.			
REQ	ASSEMBLER	6	1	0
	Instruction mnemonic.			
RET	ASSEMBLER	6	1	0
	Instruction mnemonic.			
S	ASSEMBLER	6	0	0
	A constant that declares S , the parameter stack pointer, to be Register 13.			
SAV	ASSEMBLER	6	1	0
	Instruction mnemonic.			
SD	ASSEMBLER	6	1	0
	Instruction mnemonic.			
SEP	ASSEMBLER	6	1	0
	Instruction mnemonic.			
SEQ	ASSEMBLER	6	1	0
	Instruction mnemonic.			
SEX	ASSEMBLER	6	1	0
	Instruction mnemonic.			
SHL	ASSEMBLER	6	1	0
	Instruction mnemonic.			
SHR	ASSEMBLER	6	1	0
	Instruction mnemonic.			

WORD	VOCABULARY	SCREEN	STACK: IN	OUT
SM	ASSEMBLER	6	1	0
	Instruction mnemonic.			
STR	ASSEMBLER	6	1	0
	Instruction mnemonic.			
STXD	ASSEMBLER	6	1	0
	Instruction mnemonic.			
T	ASSEMBLER	6	0	0
	A constant that declares T to be Register 9, a scratch register.			
THEN	ASSEMBLER	8	1	0
	Sets the address of a short conditional forward jump in the dictionary.			
U	ASSEMBLER	6	0	0
	A constant that declares U , the user pointer, to be Register 4.			
VANISH	FORTH	4	0	0
	Removes the entire ASSEMBLER vocabulary from memory. ASSEMBLE LOAD will restore the ASSEMBLER vocabulary in memory.			
W	ASSEMBLER	6	0	0
	A constant that declares W , the current word pointer, to be Register 11.			
XOR	ASSEMBLER	6	1	0
	Instruction mnemonic.			

APPENDIX C

microFORTH GLOSSARIES

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.

Please check your microFORTH screens, therefore, when you want to verify that words (especially in the lower-numbered screens) exist in the same screens as listed here. This would also be an excellent time for you to begin building an application glossary for use by your staff.

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.

Appendix B contains words associated with assembly language and thus dependent on your type of development hardware. For details concerning the mnemonics, see Appendix A.

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).

MICROFORTH GLOSSARY

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).

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 >.

MICROFORTH GLOSSARY

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.

MICROFORTH GLOSSARY

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.

MICROFORTH GLOSSARY

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.