FORTH

An Application-oriented Language

PROGRAMMER'S GUIDE

BY

E. D. Rather & C. H. Moore

National Radio Astronomy Observatory

&

Kitt Peak National Observatory

Honeywell 316 Version

TABLE OF CONTENTS

Introduction

INTRODUCTION

The only way to learn any programming language or technique is by using it. This is especially true of FORTH -- since it is inherently interactive, if you will spend an hour or two at a FORTH terminal it will instruct you as you use it. If you wish to become a FORTH programmer, or only to become familiar with its advantages and disadvantages, we recommend that you follow these steps:

1. Read this book.

2. Look at the vocabulary lists summarized in Appendix A.

3. Using these lists, the book and your computer manual, study the basic routines in blocks 3 - 10 in Appendix B.

4. If possible, have a look at an example of a FORTH application.

5. Sit down at a FORTH terminal and solve a few simple mathematical problems. Work out a definition or two, type them in and test them.

This book assumes you have had some programming experience. Portions of it assume that you are familiar with the assembly language for your computer. If you are not, you will need to keep your computer manual at hand ... perhaps you will want to study its basic instruction set before attempting to use FORTH.

FORTH PROGRAMMER'S GUIDE

FORTH is a programming technique designed for real-time interactive computer applications, particularly for mini-computers. In such an environment it offers several advantages over more conventional programming languages, such as FORTRAN, BASIC, and assembler language. These advantages are summarized in Fig. 1. The concept of a high-level language which is more core-efficient than assembler language, and very nearly as fast, may be a bit hard for you to get used to -- perhaps you will understand the basis for this claim as you become familiar with FORTH's structure.

The minimum computer configuration for FORTH is shown in Fig. 2. The applications for which FORTH has been used so far have included a radio telescope control system and several data collection systems. Computers for which FORTH systems have been written include Honeywell DDP-116 and 316, NOVA 800, Varian 620i and 620f, and IBM 360-50. Since all but the 360 are 16-bit machines, this manual will assume a 16-bit word length.

FORTH is a language. It is structurally quite different from other languages, however. To begin with, it has no separate compiler or assembler -- the routines that generate and execute machine instructions are always present in the FORTH computer, along with a small, fast interpreter and a sort of executive whose characteristics are determined by the specific application.

LANGUAGE	ADVANTAGES	DISADVANTAGES

FORTRAN	Familiar
	High-level
	Handles mathematical
	  formulae well
	Good output formatting
	  (messages)
		Separate compiler
		Separate loader
		Large program
		Slow program
		Cannot control special
		  equipment

ASSEMBLER	Fast program
	Can control special
	  equipment
		Separate assembler
		Separate loader
		Monolithic
		Inflexible
		Long listing
		Machine dependent
		Addressing problems
		Debugging aid needed
		Hard to evaluate
		  mathematical formulae
		Poor I/O formatting
		Low level

BASIC	Interactive Somewhat machine independent	Large program Very slow program Limited syntax

FORTH	Flexible
	Interactive
	Small program
	Fast program
	High and low level
	Extremely modular
	Somewhat machine
	  independent
	Can control special
	  equipment
	Define functions on-line
	Number conversions easy
	  (octal, sexagesimal, etc.)
	Includes own assembler
	       compiler
	       loader
	       interpreter
		Unfamiliar
		Poor message formatting
		Somewhat hard to evaluate
		  long mathematical
		  expressions

Fig.1   Language Comparisons

8K 16 BIT CORE
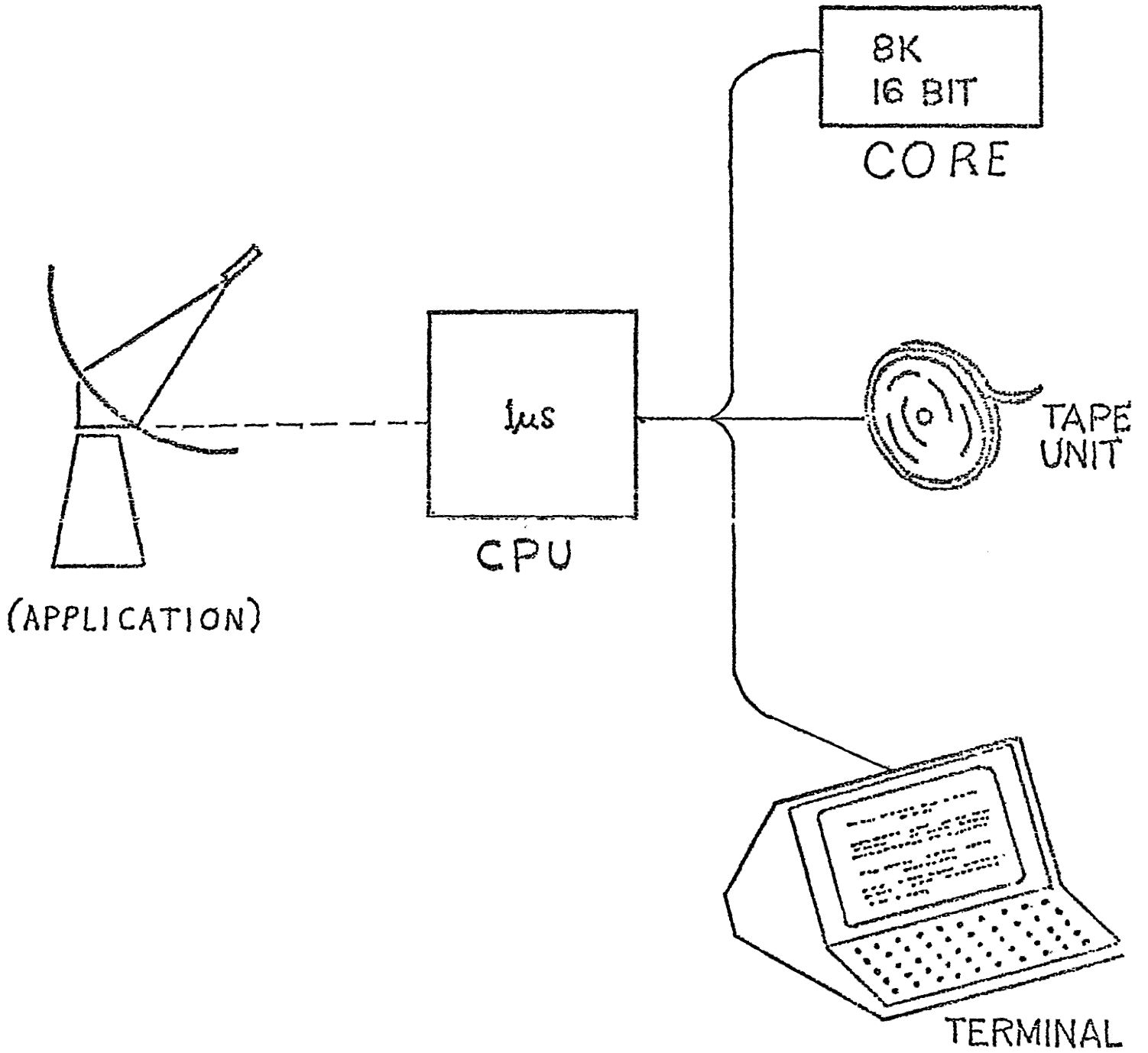
CPU 1μs

(APPLICATION)

TAPE UNIT

TERMINAL

Figure 2

FORTH's minimal computer

It seems to have 5 main elements that together comprise it. 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 (and some remain) a surprise to me.

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 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 mean. 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 load an editing vocabulary to edit text; an observing vocabulary to observe; etc. The vocabulary is just that. It is not a program for it can't stand alone. It depends upon FORTH to do all the work and merely describes what must be done.

Each defined word has an entry in the dictionary. FORTH provides the mechanism for searching the dictionary, executing words, and defining new words. These operations will be described in a moment.

2. Stack

Another important and visible element of FORTH is a push-down stack for parameters. Some words represent operations; operations find their parameters on the stack.

In particular, numbers are placed on the stack; constants are placed on the stack; the addresses of variables are placed on the stack· results are placed on the stack. All this happens in a convenient and natural way, and provides an indispensible tool for describing algorithms.

Another largely invisible stack is also important. It is used to store return information for the interpreter.

3. Code

Some words are defined by code. That means that they cause a sequence of computer instructions to be executed. Such a word is similar to the name of a subroutine. However the analogy is weak, for it isn't a subroutine in the technical sense; nor do you think of it -- or use it -- as you would a subroutine.

4. High level definitions

Some words are defined in terms of other words. Thus they are a sort of abbreviation. However, again the analogy fails, for such definitions are much more powerful than the notion of abbreviation conveys. In fact, perhaps 90% of the words in a vocabulary are definitions. These are computer-independent.

5. Blocks

The final element of FORTH are its blocks -- chunks of secondary memory 1024 bytes (512 words) long. A block may

contain source code -- the text that defines words; or it may contain the data that justifies the computer. Whatever, it has been chopped into fixed-length chunks and assigned a number -- usually between 0 and 511. You may load programs from blocks, or place data in blocks, as if the blocks were core. FORTH provides the I/O required to access them automatically. This form of virtual memory provides a very great service -- at a very small cost.

Keyboard Input

FORTH is a terminal-oriented language. It demands the subtlety of expression that only a keyboard can provide. A FORTH application may well have a vocabulary of several hundred words. Of these, maybe 20 will be of direct interest to the user singly and in combination

The input that FORTH wants is simple:

words separated by spaces.

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

EOT to mark the end of a message.

DEL to erase a letter.

NUL to cancel a message.

This is as simple a way to communicate as I can devise. It does cause some trouble: People may forget to type EOT fail to space between words, or don't know whether to spell the word

GOODBY, GOODBYE, or GOOD-BYE.

But these are just conventions that must be learned.

Internally, FORTH represents characters with 7-bit ASCII code and even parity. This is an extremely important convention, as it permits effortless compatibility between computers. If you have a non-ASCII device, the character-conversion cost is properly assigned to the device. Likewise, parity is much easier to discard than create; a non-parity device (teletype) should bear the cost of providing it.

In the chapters to come we will discuss the basic elements of FORTH in more detail, and describe the process of developing a FORTH vocabulary for your application.

THE FORTH DICTIONARY

The dictionary is a linked list of variable-length entries. It grows toward high core, and each entry points to the one that preceeds it. The beginning of the last entry is pointed to by the variable HEAD. It identifies the head of the chain to be searched. The end of the last entry -- the next available word -- is pointed to by the variable DP (Dictionary Pointer).

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 3 shows, the dictionary can be rather naturally divided into three parts:

The object program contains about 30 words from which all other words can be defined. It is difficult, and normally unnecessary, to change these words.

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 vocabulary contains those words peculiar to your application. You will be changing, rearranging and adding to this vocabulary continuously.

From the point of view of the search algorithm, these vocabularies are indistinguishable. However, you can distinguish them -- and other subvocabularies -- by being able to
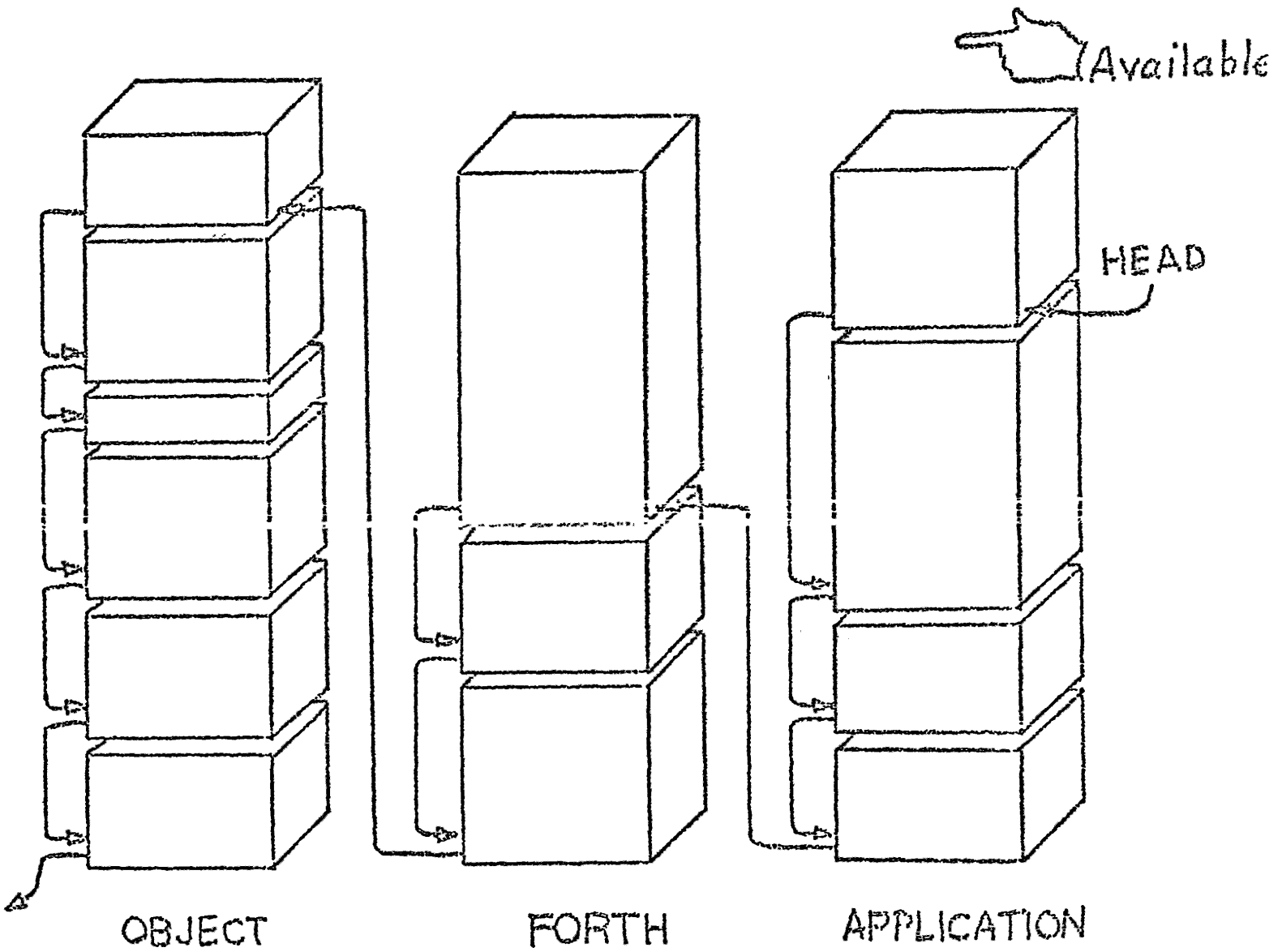
Figure 3

FORTH Dictionary

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

discard them. For example, you might discard one application vocabulary and replace it with another.

The essential structure of all dictionary entries is the same regardless of the type of entry (nouns, verbs, etc.). The first two words contain the count of the number of characters and the first three 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 three 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 third word contains the location of the first word 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. The high order 2 bits of the link word contain a flag called "precedence": It identifies a few special words, such as compiler directives. It is zero for most words.

The forth word contains 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 word 5 of the definition) on the stack.
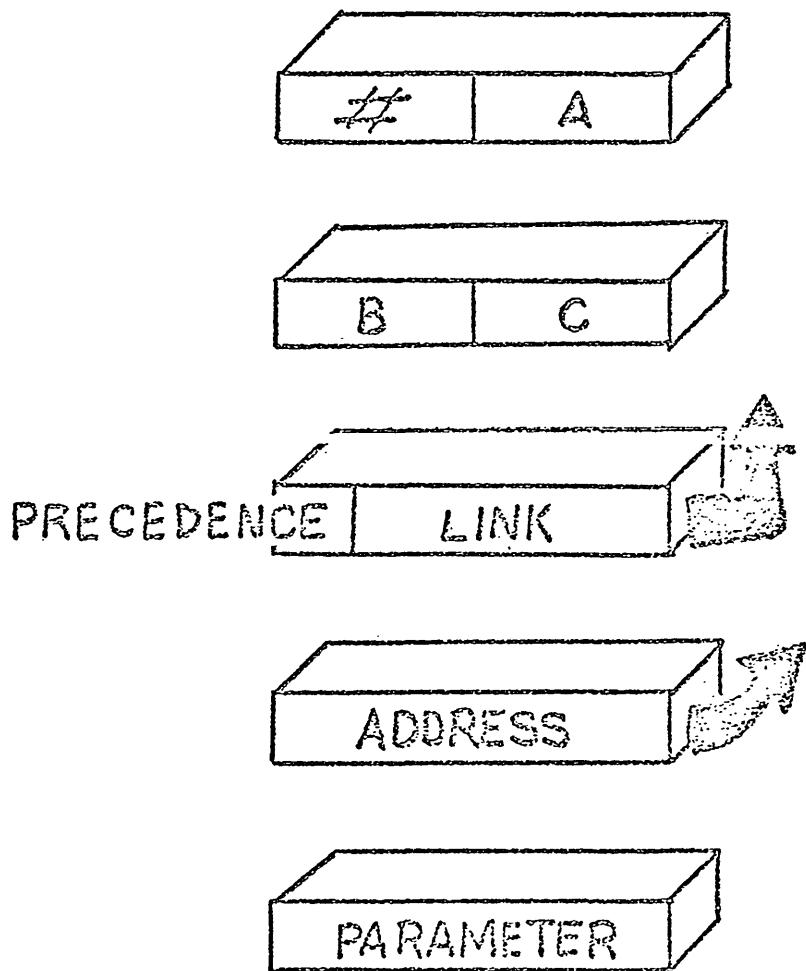
Figure 4

16-Bit Dictionary Entry Format

For an INTEGER, it refers to code that puts the <u>address</u> of the value (again, word 5) on the stack.

For a : definition, it points to a portion of the interpreter, which will begin following a string of addresses beginning in word 5 and continuing until the ; which terminated that definition is found.

For CODE, the pointer is to word 5, 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.

The fifth and subsequent words are sometimes called the "parameter field", which is of variable length. CONSTANTS and INTEGERS keep their values in word 5, as noted above. Other nouns (see "Nouns") may keep several values. In the latter cases, the length of the parameter field is either determined by the type of noun, or is kept in one of the early words in the field.

Figure 4 shows a diagram of the dictionary entry format for 16-bit computers. Similar diagrams of specific types of entries will be found in subsequent chapters.

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 core.
There is a pointer to the word holding the number currently
"on top of the stack" called SP.
To add a word to the stack, the pointer is decremented,
and the number placed in that word. To remove a word from
the stack, the pointer is incremented.

If you type a number on your terminal, the number will
be converted to binary 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 5 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 these words deserve
comment:

    =   is the replacement operator. It expects 2 para-
           meters on the stack: an address on top and a num-
           ber beneath. It stores the number at the address.

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	Address of X (which was previously defined as an INTEGER) pushed on stack.
	=	17 stored in X; both 17 and address (X) removed from stack.
		(Remember 9 is still on the stack from Ex. 1)
(3) X @ *	X	Address (X) pushed on stack.
	@	Address replaced by contents (17).
	*	9 and 17 replaced with 153.
	.	153 typed on terminal.

Figure 5

Thus the FORTRAN statement

   X = 17

in FORTH is

   17 X =

@   is an operator that fetches a value. It expects
      an address on the stack; it replaces it with the
      content of that address. @ is an extremely im-
      portant operator. It distinguishes between loading
      and storing a value into a variable; a function
      FORTRAN accomplishes by context -- in one particular
      way.

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

These operators have been assigned single character mnemonics
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; - subtracts, etc. Refer to the
vocabulary in Appendix A for a fuller 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 the table. A fairly standard set of more com-

Stack Manipulating Words

	Explanation	Before	Operator	After
			Example	

	Explanation	Before	Operator	After
		Top		Top
)RTH words (may be used in definitions or in CODE)				
/AP	reverses order of top 2 entries	1 2	SWAP	2 1
UP	reproduces top entry	1	DUP	1 1
OP	discards top entry	1 2	DROP	1
ER	pushes entry 2 on top of entry 1	1 2	OVER	1 2 1

plicated stack operators is available in some developed appli-
cations, for such things as double precision numbers, fetching
numbers several levels down in the stack, etc.

The order of parameters on the stack is governed by
several fairly 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 will be

   . 3 . 2 . 1

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

   3 SEC =

stores 3 in SEC.

3) Double precision numbers are always placed on the
   stack with the high order part on top and the low
   order part 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 results (if any) on the stack. It will, of course, destroy no more than its own parameters. Thus

1 2 3 + will leave 1 5

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

The stack is located in high memory and extends toward low memory. Unused core 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.

FORTH checks for stack underflow and overflow (since overflow can occur only when memory is full; it normally is not of concern). If either occurs, the error message is

[operator]   ?

where the operator that failed to find or place a needed parameter on the stack is given.

The stack is by far the best place to use for temporary storage, since stack accesses are faster and specific core allocation is not required -- the dictionary entry for a one-word number costs an extra four words of overhead. In particular, the stack is an excellent place for saving the contents of a variable which must be changed temporarily. It will take a while to become comfortable using the stack, but when you do you will be impressed by its convenience and economy.

FORTH COMPILER

The FORTH Compiler is very small and simple -- it is not in any sense comparable to, say, a FORTRAN compiler. But it does make the task of writing inter-related routines extremely convenient and straightforward, and allows as much control over logical flow as one needs even for very complex applications.

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 ; ... words are separated by spaces. Literals may be used anywhere, as long as they are in a recognized form. (See "numbers").

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. This means that forward references (except in case of the IF ... ELSE ... THEN construction, which we'll get to shortly) are not allowed. This may require you to modify your programming style, but it is not a serious inconvenience.

The action of the interpreter in compiling definitions is discussed under "Interpreter". After the basic four word standard dictionary entry heading come a string of addresses of the words that form the content of the definition. These may be addresses of code entries or other : definitions -- it
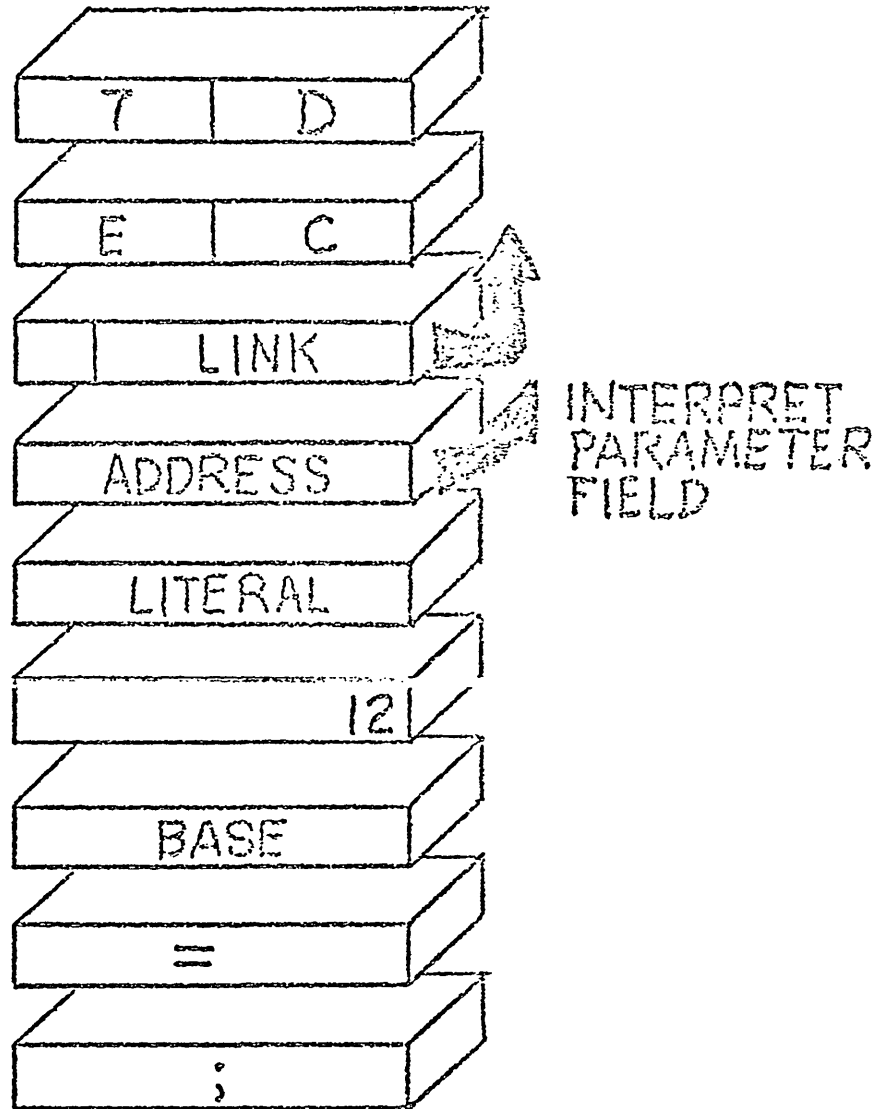
Figure 6

Dictionary entry for : DECIMAL 12 BASE = ;

(Stores 12_8 in the variable BASE)

doesn't matter. Somewhere at the top of the chain, of course, is code which will be executed. Literals require two words: the first being the address of LITERAL, a short routine that pushes the contents of the second on the stack. You should bear this in mind, and define as a CONSTANT any literal that you find yourself using often.

       1   CONSTANT   1

has already been defined for you -- use of "1" in a definition adds only 1 word (the address of the 5-word definition 1) instead of two. Figure 6 shows the dictionary entry for a simple definition.

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 words for forming loops and two-branch conditionals.

LOOPS: Loops are constructed using the FORTRAN-ish word DO (which expects two loop parameters on the stack and two ending words: LOOP (which increments the loop counter by 1) and +LOOP (which increments it by the [signed] amount on the stack). The two DO parameters are the limit (upper or lower) of the loop beneath and the initial value of the counter on top of the stack. Both LOOP and +LOOP compile a conditional return to the location of the DO. When the counter is increasing, the loop terminates when the limit is reached. When the counter is decreasing, the loop terminates when the limit is passed.

The loop parameters are not kept on the stack during the loop. The limit, in fact, is not accessible during the loop. The counter may be accessed using the word I, which places the value of the counter on the stack. Remember that I is a verb! Do not use it as a variable. The counter may not be changed except by the LOOPs. In nested loops, I provides the counter for the innermost loop.

Here are some examples; try to figure out how each works, referring to the vocabulary in Appendix A for unfamiliar words. You may find it helpful at first to keep track of what's on the stack on a piece of scratch paper.

(1)   :   SUM   0   101   1   DO   I   +   LOOP   ;

       adds the numbers from 1 to 100 and leaves the result (5500) on the stack.

(2)   :   SQUARES   0   SWAP   1   +   1   DO   I   DUP   *   +

       LOOP   ;   6 SQUARES

       adds the squares of numbers from 1 to 6.

(3)   :   PRINT   0   DO   I   1   +   .   LOOP   ;

       10 PRINT

       types the integers from 1 to 10.

(4)   :   PRINT   DUP   0   DO   CR   DUP   I   10   +   MIN

       0   DO   I   .   LOOP   10   +LOOP   DROP   ;

       103 PRINT

       prints the numbers from 0 to 102, 10 per line.

(5)   A non-trivial example of a loop is the Euclidean algorithm for the greatest common divisor of two integers:

For a > b, a = qb + r

If d divides a and b it must also divide r

Then GCD (a,b) = GCD (b,r)

The process terminates with r = 0

Suppose a and b on the stack:

```
: GCD 1 0 DO SWAP OVER MOD
    DUP 0= +LOOP DROP ;
```

The loop parameters are kept on the return stack, which also contains return addresses for the interpreter. 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). Within these restrictions, keeping these numbers on the return stack does not interfere with the interpreter's use of it, and the parameter stack is not cluttered by their presence. It is recommended that you study the way DO, LOOP and +LOOP are coded for your computer, because you may wish to code additional related words for your application. Three which are now available on the 316 are:

*LOOP - multiplies the loop counter by a specified
        amount (similar to +LOOP)

J}
K}   push the counters for second and third outer
        loops on the stack (similar to I) - useful
        for matrix operations.

These are not implemented in basic FORTH because of their specialized usefulness.

CONDITIONALS: FORTH offers a convenient one -- or two -- branch IF. Its use is given in this example:

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

This takes an integer S, which is initialized to 0 but whose value may change in use, and compares it to the constant limit of 99. If the limit is exceeded, S will be set to 100; otherwise, it will be incremented by 1. The word IF compiles a conditional forward jump dependent on the top of the stack being zero (false). The destination of the jump is not known at the time the jump is compiled, but the address of the jump destination is left on the stack. When ELSE is compiled, its location is set as the destination of IF's jump, and an unconditional forward jump is compiled -- its destination to be similarly supplied by THEN. THEN terminates the conditional, as it compiles no jump. The ELSE clause may be omitted entirely. Note: Every IF must be followed by a THEN.

Remember that the IF will jump to ELSE or THEN if the top of the stack contains zero (false). Therefore, the IF inherently contains an "equal to zero" test. Remember also that IF will destroy its parameter (the condition), like all FORTH words. You thus may want to include a DUP in your definition:

```
X  @  DUP  IF  ....  ("If X is zero, ....")
```

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

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

0 if zero or positive.

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

> Opposite of <.

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 four words. Thereafter, add one word for every defined word in the definition, including the semi-colon, and two for every literal. Timing is more a function of the execution time of each word in the definition than their number -- only you can estimate that[†]-- but you must add the interpreter cycle overhead for each word, which is 12 μs on the 316. If you are trying to decide whether to define a word 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 two - five times*, and will save (length - 1) words for every subsequent usage. The cost in time will be 1 interpreter cycle per usage -- not very much for what can be an extremely great saving in core.

[†]Typical times for simple operators such as +, -, stack operations, @, =, etc. are around 22 μs.

* Total length:	5	6	7	>8
Uses to break even:	5	4	3	2

FORTH INTERPRETER

Everything FORTH does is controlled by its interpreter, and everything you will do using FORTH is therefore also controlled by the interpreter. The interpreter itself is quite small. But it controls several routines that are invisible to the user that do quite a lot for you, including number conversions, dictionary searches, generating dictionary entries, managing the stack, etc. Some of these functions are very intricate. Many of them do not have to be understood by a user or applications programmer, and I will not attempt to explain these in detail here. If you are interested, however, please inquire. In this chapter I will explain what the interpreter does for you, and how you will use it.

I. Interpreting a string of words typed at the terminal

Your main communication with FORTH is through a terminal. You type one or more words and they are interpreted and obeyed. If you have requested something to be typed out, it will be; if you have asked that something be done which requires no typing, FORTH will reply OK when it has done it. If you send a null message, in fact, FORTH will cheerfully reply OK.

The operations that have been performed are the following:

1) A word is taken from the terminal line buffer. (Remember, a word is a character string bound 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 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 in-
   terpreter 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.
But - if at any time you ask that something be typed back to
you, or if the string is aborted (2c), the line buffer is de-
stroyed, and interpreting ceases.

II. Interpreting source blocks. If you type a load command,
like

    3   LOAD

the interpreter operates in a slightly different mode, since in
the process of executing the word LOAD it must interpret words
in a block read from disk or tape, rather than the terminal
line buffer. It keeps track of what block it is taking its in-
structions from; the terminal, for this purpose, is "block 0".
The only differences in interpreter behavior between operating
on "block 0" and any other block are (1) that only "block 0"
gets an OK on completion; (2) interpreting in other blocks ceases
 when ;S is encountered. Note that, just as "block 0" is
loading a block, that block may load other blocks.

III. Compiling definitions. If the interpreter encounters a
command, it will be executed. Defining words (such as : CODE,
CONSTANT, etc.) are commands whose execution causes the inter-
preter to behave in a special way. The generation of CODE
entries and nouns is discussed elsewhere in this manual ("Assembler"
and "Nouns"). A colon not only generates the beginning of a
dictionary entry for the word immediately following the colon,

it also sets a flag for the interpreter. Thereafter, the words in the definition will not be executed -- their addresses will be placed in the dictionary entry being compiled. This flag will be reset by the ; .

IV. <u>Executing definitions.</u> The code address for : definitions points to a routine which at executing time sets the compile/ execute flag (called STATE) to "execute", sets the instruction counter (IC) to the parameter field of that definition, and jumps to NEXT. NEXT is the most fundamental routine of the interpreter -- the basic loop that goes on to the next word, controlled by IC. When a : definition is being executed, the interpreter is going down the addresses supplied in the defini- tion,and going off to execute whatever is at those addresses. Of course, these addresses might point to other : definitions, and so on, so the code for : saves the current IC on a special push- down stack called the "return stack." The code for ; therefore resets IC from 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 with a jump to NEXT (often after passing through stack manipulating routines).

It is clear, then, that the greatest overhead in running FORTH is in this interpreter loop, and great pains have been taken to code it as tightly as possible.

A most important aspect of FORTH is its ability to define
new words. New definitions and code are devised continously.
Likewise, new constants or variables are created. However,
a more challenging and significant kind of creativity is in-
volved in the definition of new kinds of words. Not defini-
tions or constants or any other kind of common word, these
can share the attributes of both nouns and verbs. But since
they are mostly used as nouns, I shall refer to them as such.

A new kind of word may be defined by writing a definition
that includes the verb CONSTANT. CONSTANT takes a value on
the stack, reads the next word from the input string and con-
structs a dictionary entry for it initialized to the value on
the stack. When used in a definition, that definition must
be terminated by the word ;CODE (instead of ";"). ;CODE
completes the dictionary entry by beginning a code string
which follows immediately, and changes the code address of the
word to point to this code string. In a real sense, ;CODE
combines the functions of the separate words ";" and CODE.

For example, the definition of INTEGER for the 316· is

: INTEGER CONSTANT ;CODE 0 LDA, ONE 3 + ADD, PUSH

Given that definition, the phrase

0 INTEGER M

then acts as follows: The word
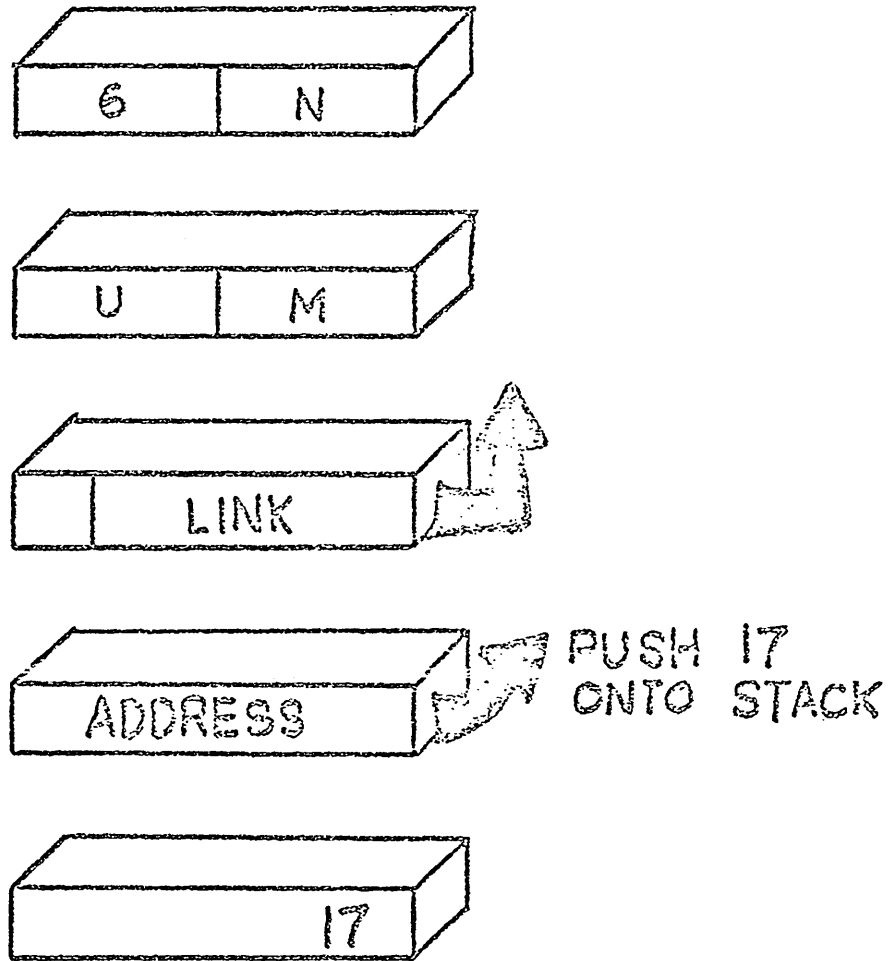
INTEGER - executes the definition, wherein

Figure 7.

Dictionary entry for
17 CONSTANT NUMBER

CONSTANT - reads the word M and constructs a dictionary entry for it, with the exception of the code address for M. The value of the parameter field is initialized to the value on the stack, in this case 0.

;CODE   completes the entry for M by supplying the code address -- the word immediately following ;CODE. When ;CODE is compiled, there is no code there -- nor any guarantee there ever will be. You must put some there.

0 LDA,   generates the first instruction that will be executed for M (or any other integer). When a code definition is entered, the B register contains the address of the first word of the definitions. TBA transfers the address of the first word in the dictionary entry for M from B to A.

ONE 3 + ADD,   increments that address to the location where the value of M is stored.[1]

PUSH   jumps to code that puts that address onto the stack.

Every time an INTEGER is <u>defined</u> the definition is executed.

Every time an INTEGER is <u>referenced</u> the code is executed.

The definition of what INTEGER <u>does</u> is the same on all FORTH computers; the code of course varies with the computer.

1   ONE begins an array of useful integer values.   ONE   3   + is 4

INTEGER is only one kind of noun. Other kinds of nouns can be easily imagined: REAL, COMPLEX, VECTOR. Most nouns share the property that they place an address onto the stack. But I would like to suggest some more elaborate nouns.

Consider first the distinction between INTEGERs and CONSTANTs. Referencing a CONSTANT places its value onto the stack; referencing an INTEGER places its address here. However, auxiliary verbs permit using either in an equivalent way:

Given

         7   CONSTANT   A   7   INTEGER   M

If I say M @   or   A

I get the same result, 7 placed onto the stack. Similarly, if I say

          M   or   '   A

I get the same result -- the address of a location that contains 7 (the verb ' provides the address of the parameter field of a word.   The difference between CONSTANT and INTEGER is one of usage -- CONSTANT automatically provides the verb @.

This suggests that we might provide a noun that automatically provides the verb =, another verb often associated with INTEGER. It can be a very effective construct.

I will discuss in a moment a noun that provides an automatic subscript. Often you want to use a noun as a switch, and an efficient, convenient way of changing it is helpful. So

I defined a kind of noun modifier called SET: Given a switch
called FLAG, defined O INTEGER FLAG

    O FLAG SET AZ   1 FLAG SET EL

SET constructs a dictionary entry which will behave in the
following way: Whenever I reference AZ, the constant O is stored
into FLAG. Similarly, I can name any value of FLAG, or any
other variable, and establish that value by typing the name.
Of course, there are other ways of accomplishing this same effect:

    : AZ O FLAG = ;

is completely equivalent to the above. But it is considerably
less efficient, both in space and time (15 us $\underline{vs}$ 92 us). A
noun modifier such as SET simply extends the basic philosophy
of FORTH: to define those words that are convenient to describe
your problem.

A more elaborate noun is one I call VECTOR. Referencing
a VECTOR places an address onto the stack: the address of the
first work in the vector plus the contents of a noun, (), which
is a general subscript. The definition of VECTOR on the 316
would be very similar to that for INTEGER:

    O INTEGER ()

    : VECTOR CODE ;CODE () LDA, 1 ALS,

    ONE 3 + ADD

The word CODE is used in this instance to set up the dictionary entry
instead of CONSTANT because CONSTANT requires an initial value
and CODE does not. Thus the value of the vector is not initialized
by the definition. It is simpler to initialize the VECTOR
explicitly to the proper length:[1]

    VECTOR X O , O ,

In conjunction with SET this leads to some pleasant notation.

---

[1]. The operator "," enters a number on top of the stack into
the dictionary.

       0 () SET LAT   1 () SET LONG   2 () SET BEARING
       VECTOR SIN 0 , 0 , 0 , VECTOR COS 0 , 0 , 0 ,

and store values in the arrays SIN and COS, I can reference the

respective values by saying

       LAT SIN   BEARING COS   LONG COS

in a natural manner.

       An even more intricate noun is invaluable for evaluating

expressions involving constants. I call it POLY (after polyno-

mial) and will give an example to show its use:

       4 POLY PRIME 2 , 3 , 5 , 7 ,

POLY acts like CONSTANT in that it provides a value rather

than an address. The first time I say PRIME I get 2 on the

stack; the next time 3; then 5; then 7; then 2 again. Each

reference to PRIME gets me the next value in closed circle.

The number 4 above advises of the size of the circle. Another

word is used to keep track of the present position.

       A more practical example might be evaluating a polynomial:

       5 POLY K -- , -- , . . .

If I say

       K X * K + X * K + X * K + X * K +

(or the equivalent in a loop) I have referenced K exactly 5

times, extracted the values I wanted in the proper sequence,

and reset K in anticipation of my next evaluation. All with-

out explicit subscripts or any other notational baggage.

A noun analogous to POLY that provides addresses instead
of values I call ARRAY. It is extremely useful for storing and
retrieving values into an array in sequence. Yet another kind
of noun that is useful is one which provides an array auto-
matically subscripted by I, which fetches a current loop index.

I hope that I've shown that different kinds of words can
be usefully defined. Basic FORTH provides only CONSTANT and
INTEGER, but standard definitions of most of the nouns dis-
cussed here are available. If you encounter more than one
instance of a particular kind of noun, or use such a word fre-
quently, it can pay off in convenience, efficiency and elegance
to name and characterize those properties that make it unique.

ARITHMETIC

It is routine to implement the ordinary arithmetic operations in FORTH. The problem is to decide which will be useful and exactly how they should act. Keep in mind that these operations are much too elementary to be of value in themselves; but they form essential building blocks for later definitions.

A characteristic of elementary operations is that there are usually machine instructions that perform them. To embed such an instruction in code so as to make it accessible adds significantly overhead, easily 5 times the instruction execution time. But keep in mind that our purpose is to make these instructions readily executable on arguments conveniently found on the stack. The value of accessibility versus efficiency is strongly in favor of the former, since we are viewing the computer from the other side of a keyboard. You can always construct code for an inner loop where efficiency is a problem. Furthermore, the use of definitions results in very significant savings in core.

Certain operations are absolutely essential, as can be verified by their frequency of use:

Op.	Example	Description
+	A B +	leaves A+B on the stack.
-	A B -	leaves A-B on the stack.
*	A B *	leaves A*B on the stack.

```
/              A B /                 leaves A/B on the stack.
MINUS          A MINUS              leaves -A on the stack.
```

(Notice that the order of the operands for - and / [non-commutative operations in general] is such that if the operator were moved from a suffix position to an infix position [A B - to A - B] the operands are in their customary positions. Such a mnemonic aid is essential to help recall the order of arguments.)

Op.	Example	Description
MAX	A B MAX	leaves the greater of A and B on the stack.
MIN	A B MIN	leaves the lesser of A and B on the stack.
ABS	A ABS	Replaces a number on the stack by its absolute value.
/MOD	A B /MOD	Returns the quotient on top of the stack and remainder beneath after division.
MOD	A B MOD	Leaves the remainder after division on the stack.
SIGN	A B SIGN	Give A the sign of B; if B is 0, replace A with 0.

Many other operations could be added. Some of these can be defined in terms of those above. If that is awkward or inefficient, code can be written for them. Remember that we are dealing with 16-bit 2's-complement numbers. This provides integers from -32768 to 32767. Another number format that is useful is 14-bit fractions. It can provide numbers from -2.0000 to 1.9999. The above operations work for such numbers, with the exception of * and /.

* *, product of 2 arguments, one of which is a fraction
       (shift left 1).

/, quotient expressed as a fraction (dividend shifted
       right 1).

These are useful operations. You must watch out for the size
of arguments in division (quotient 2). The product works for
fraction*integer or fraction*fraction. The quotient works
for fraction/fraction or integer/fraction or integer/integer.
Fraction/integer should use the integer divide.

The notation is a compromise. A better convention would
be *. and /. where the point symbolizes the decimal in the
fraction. However, that is used by the double-precision opera-
tions in most applications. I compromised by replacing the
point by a comma, recalling European custom, although the
comma is badly over-worked. You are free to redefine them, of
course. If you use them a lot (and don't need the integer
operations) consider:

  : * *, ;   : / /, ;

Two trinary operations are valuable if you do any extensive
arithmetic:

*/   A B C */   leaves (A*B) /C on the stack

+*   A B C +*   leaves A + (B*C) on the stack

+* is completely equivalent to * followed by +, but the com-
bination occurs so frequently that the added efficiency of
combining them is welcome. */ is a natural combination since
the double-precision product in the A and B registers is per-

fectly positioned for the divide. In general, you get better precision by the combination than by separating the operations.

As an example of the value of the */ operator, the following definitions might be used to implement fixed-point decimal arithmetic (4 decimal places) as opposed to the fixed-point binary arithmetic discussed above.

: * 1.0000 */ ; : / 1.0000 SWAP */ ;

The numbers in this case are integers with an assumed decimal point 4 places from the right, and can represent fractions between -3.2768 and 3.2767 with no decimal-binary conversion error. Such arithmetic is perfectly reasonable, the extra cost of the scaling being completely negligible. However, it doesn't generalize to double-precison very well.

Numbers.

As previously noted, numbers typed at the terminal or encountered in a block being loaded are converted to binary (from base BASE, usually 8 or 10) and pushed on the stack. This is done by a single routine called NUMBER. NUMBER follows the following conventions:

1. Positive numbers are unsigned, negative numbers have a leading minus sign.

2. Legal digits are 0 through BASE - 1.

3. Single precision integers may run from -32768 to 32767.

4. A decimal point encountered anywhere in the number causes the number to be converted as an un-normalized double-precision (31-bit) integer, in standard Honeywell double-precision

format, and pushed on the stack with the most significant part on top. The variable D in this case will contain the number of digits found to the right of the decimal point.

5. A colon encountered anywhere in the number will cause the rest of the number to be converted as a sexagesimal number. Thus, 30:00 will be converted as 1800.

Remember that a word will be looked up in the dictionary first, and NUMBER will attempt to convert it only if the dictionary search fails. This enables the user to define commonly used literals (such as 0 and 1) as CONSTANTs, thus saving space in definitions. As a side benefit, you may define any kind of word with a "numeric" name ... for example, a precession routine might define 1950.0 to be a verb which precesses 1950 coordinates to current positions.

THE FORTH ASSEMBLER

FORTH can assemble machine-language definitions of words. Among the many examples in the basic vocabulary are the arithmetic operations.

+ - * / MOD MAX MIN

The assembler is not intended for conventional programs. Additional words would be needed for that -- as in the BOOTSTRAP vocabulary for recompiling FORTH.

Words defined by the assembler are CODE entries. They have a standard dictionary entry with the address field pointing to the next word (the parameter field).

The instruction mnemonics are operations - they compile the instruction and address (if any). As usual in FORTH, operands (addresses) precede operations (instructions). Depending on computer, several kinds of instructions and addressing are possible:

CPU instructions have no address. They assemble a 1-word instruction

MR instructions have a memory address. The address may have to be adjusted before the instruction code is added and the address must be in page o or the current page.

I/O instructions have a device address.

S instructions are shift instructions, and allow insertion of the length of the shift.

In definitions (beginning with :) the placing of parameters
on the stack and retrieving them from it is automatic -- only
such words as SWAP, DUP and OVER are normally used. CODE,
however, requires that parameters be handled explicitly, using
S) and the returns that push or pop the stack (see "Assembler").
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 - LDA,

at assembly time places the current dictionary location on
the stack (HERE) and decrements it by 2. This number is then
the parameter for LDA, which <u>assembles</u> a machine instruction
which is the equivalent of

    LDA   *-2

in conventional assembler notation. Similarly, such words as
SWAP and DUP are <u>executed</u> at assembly time in CODE, and <u>com-</u>
<u>piled</u> into the dictionary in definitions.

These mnemonics are used to indicate various relative
addresses:

    1)   index register
    0)   indirect address

S)   Stack pointer (indirect through SP)

What effect these words have depends on the instruction. On some they set the address; on others they set the indirect or indexed bits.

For example:

CODE + S) LDA, SP IRS, S) ADD, PUT

will add the top two numbers on the stack and leave the sum on the stack replacing the two numbers.   The action of the assembler in constructing a CODE entry is given in Figure 8. The resulting entry is given in Figure 9.

RETURN: Code must end with a jump to the interpreter (NEXT).   Several addresses are available which modify the stack before returning to NEXT: these are summarized in Figure 10. All of these words except NEXT contain code for checking stack underflow or overflow.   Therefore, you should always end CODE definitions which modify the stack with one of these words.   Examples of all of these are in the basic voca- bulary.   A computer with single-word jump (such as the 316) includes the jump code with these words.

STACK: Most parameters come to code entires via the stack. On the 316, the pointer to the top of the stack is in the variable SP.   Since the top is most often referenced, the word S) refers to it directly:

S)   addresses the top of the stack [SP o)]

For more complicated references, you may wish to put SP in loc.0 (for indexing).

WORD	ACTION DURING ASSEMBLY	ACTION DURING EXECUTION
S)	Leaves a word on the stack with bits for SP 0) (indirect through SP)	
LDA,	Takes op. code for LDA, adds number on stack, compiles result in dictionary, advances DP.	Loads A relative to SP (the top of the stack)
SP	Address of SP on stack	
IRS,	Compiles instruction to increment SP	
S)	Same as above: note that SP has been incremented.	Add relative to SP (top number on stack) to contents of A.
ADD,	Takes op. code for ADD, adds top of stack, compiles result.	
PUT	Compiles a jump to PUT, a routine which replaces one number on the stack by the contents of A, then goes to NEXT.	Jumps to PUT

Figure 8

CODE + S) LDA, SP IRS, S) ADD, PUT
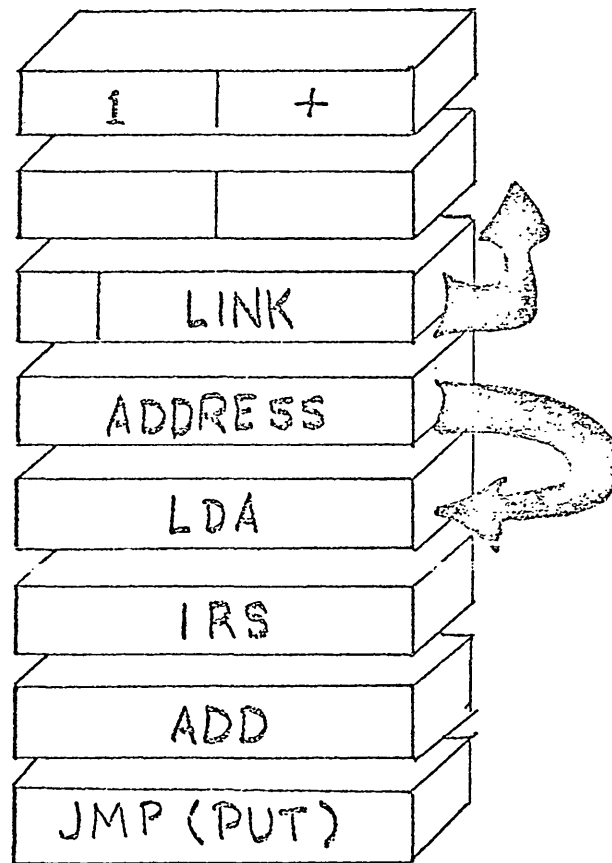
Figure 9

Dictionary entry for

CODE + S) LDA, SP IRS, S) ADD, PUT

Word	Explanation		Before	Operator		After

terpreter returns (terminate CODE entries-contents of A shown as A(n))

Word	Explanation		Before (Top)	Operator		After (Top)
T	replace top of stack with A	A(2)	1	PUT		2
⊔SH	push A onto top of stack	A(2)	1	PUSH	1	2
P	discard top of stack		2 1	POP		2
ⅠNARY	POP followed by PUT	A(3)	2 1	BINARY		3
P.	discard top 2 words from stack		2 1	POP.		
�NARY.	POP. followed by PUT	A(3)	2 1 0	BINARY.		3

Figure 10

Assembler return locations which modify
the stack and return to NEXT

For example:

        S) LDA,

        will load the top of the stack; and

        SP LDX, 2 1) STA,

        will replace the third parameter.

        Parameters may be taken directly from core.

        For example:

        M   LDA,

        will load the value of M.   Often parameters may be picked up

without being named.   So long as their address is on the stack,

it doesn't matter how it got there:

        HERE 55 ,

will assemble the constant 55 and leave its address on the

stack.   The instruction

        LDA,

coming later will encounter its address -- and assemble an

instruction to load it.

INSTRUCTIONS:

        The basic instruction set of each computer is defined

with the manufacturer's mnemonics -- when possible.   Remember

that the count and first three letters must be unique.

        ADD   becomes ADD,

A comma is appended to improve the readability of several

instructions on the same line of text, and to distinguish FORTH

assembler mnemonics from the manufacturer's similar mnemonics.

        The most common instructions are defined and you may add

any others you find useful.   See your FORTH listing or

assembler vocabulary for details.

MACROS:

Many instructions are useful, though not implemented on a particular computer. (Macro) definitions will implement them with several instructions (check what they are; some use registers!):

ADM,   add to memory

  INR,   decrement memory

Other definitions provide forward jumps:

IF,   begin true condition code

ELSE,   begin false condition code

THEN,   end conditional code

IF, and ELSE, leave the address of a jump instruction on the stack; THEN, provides the jump address.

IF, may take a condition code as parameter; or follows a skip instruction depending on computer.

Example:

SZE,   IF, [code for 0]   ELSE, [code for not 0]   THEN,

or

SZE,   IF, [code for 0]   THEN,

Some definitions provide loops:

BEGIN,   begin loop

END,   end loop

BEGIN, puts an address on the stack; END, assembles a
conditional jump back to that address - again either condition
code or skip preceding.

There are no labels in FORTH. You could define them but
their function is performed by CODE names, IF, THEN, BEGIN,
and END,.

All these macros are defined in the basic FORTH vocabulary
for your computer. Examples of their use are there as well. A
bit of study should make clear how they work, and of course you're free
to define any others you wish.

It should be clear by now what the relative trade-offs in time
and core efficiency are between CODE and : definitions of verbs.
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 -
but : 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 a 4-word code string that seems
to perform 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 three words for every usage. Clearly, the saving will be
greater for longer strings, but you should always perform only
a single logical operation in one CODE definition.

BLOCK I/O

Tape and disk I/O is handled by FORTH in standard blocks of 512 words. This fixed block size applies both to FORTH source text and to data taken by FORTH programs. 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. In addition, it eliminates many of the headaches concerned with tape handling: searching, maintaining order, compatability with other computers, etc. The block size chosen is a convenient, modest size. FORTH applications exist with as many as ten data records in a block, and with several blocks forming a data record.

Magnetic tape records have a 513th word at the end, giving a logical block number. The FORTH instruction READ-MAP causes records to be read from tape until a file mark is reached. The logical block numbers read are used to construct a map in core showing the tape record position for the most recent version of each block. This map is 512 words long, expecting one tape file to contain no more than 512 blocks, or 262,144 words. Multiple files may be written. Thus, the tape can be handled as a random access device, in that any block may be fetched directly, with no need for searching. To facilitate this, the current tape record position is kept at all times. Blocks may be in any order on the tape, and updated versions of each block may be written at the end of the tape.

Standard FORTH programs have two 513-word block buffers in core. This number may be increased if the need justifies the amount of core used, for example in an application requiring data records several blocks long. The 513th word contains the logical block number, which will be complemented if the block is updated. If a block is requested (by the instruction

    n BLOCK

where n is the logical block number) the block handler will check the two core buffers to see if the requested block is present. If not, it will fetch the block from disk, if it is there, or tape (according to the tape map). If the block is not available, an empty core buffer will be provided. If a core buffer must be over-written, the block handler will use the one least recently referenced. If the block to be over-written has been updated, the updated block will be automatically written on tape or disk before the requested block is read. Finally, BLOCK will push the address of the first word of the requested block on the stack. A flow chart of this process is given in Figure 11.

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

Here is an example in which data records are smaller

Enter with block no. on stack

Block
in core ?

YES

NO

Determine which
core buffer to use.

Has
block in
target buffer
been up-
dated?

NO

YES

FLUSH block to tape

Check tape map for
requested block

Block on tape?

YES

NO

Read block

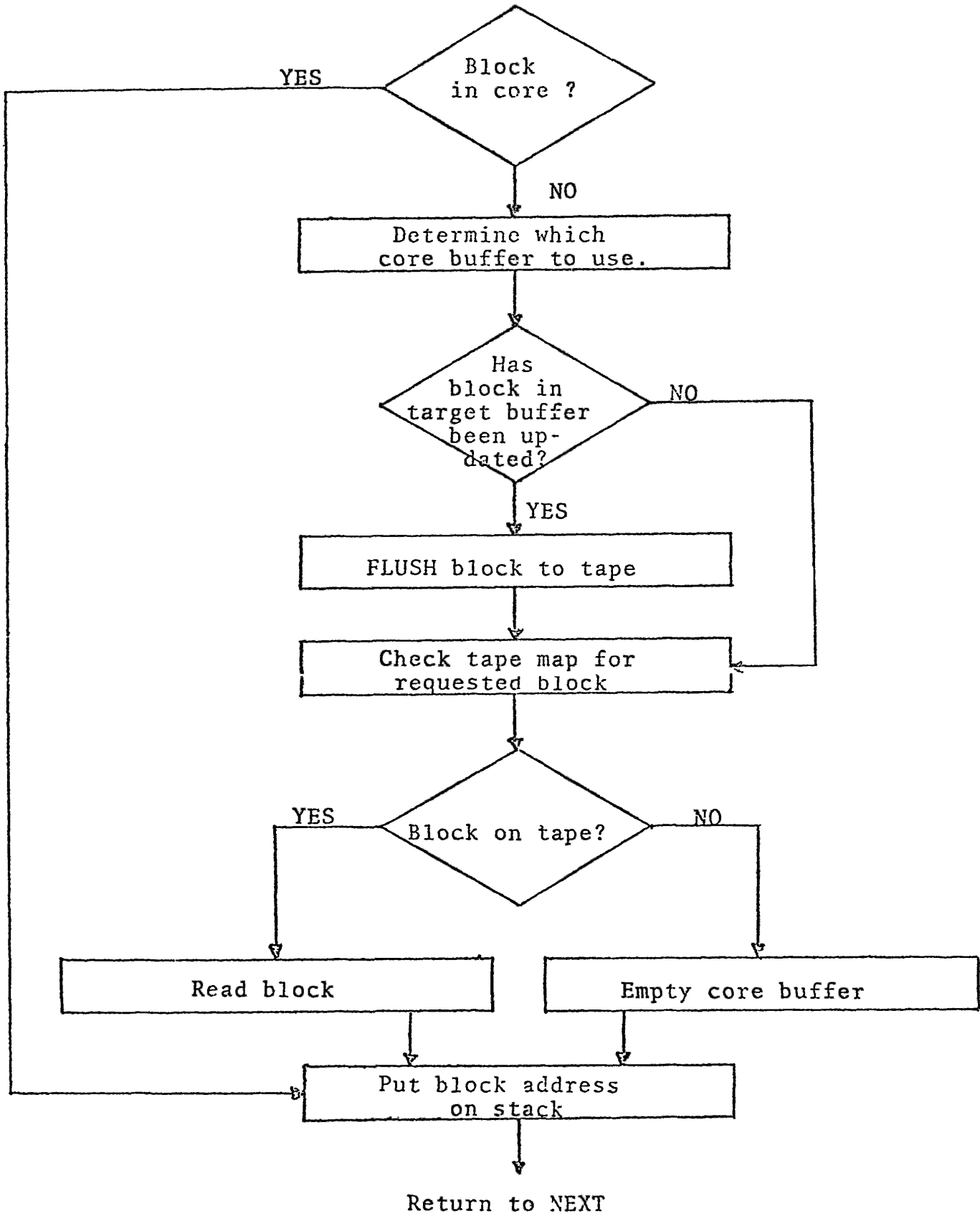Empty core buffer

Put block address
on stack

Return to NEXT

Figure 11

Block handler for tape system

than blocks. Two CONSTANTs have been defined: LR is the data record length in words, and R/B is the number of records per block.

```
: ADDRESS  R/B  /MOD  511  MIN  BLOCK
      SWAP  LR  *  +  ;
```

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

By convention, blocks 1-99 are allocated to program source text. Logical blocks 1 and 2 always contain the only portion of the program which is previously compiled. These 1024 words are read directly by the key-in loader, and contain the basic interpreter and I/O handlers for the terminal and tape or disk, plus character conversion routines and the dictionary building and searching routines. These routines are very rarely changed, and for all practical purposes may be regarded as the "given" or fixed portion of your application.

The remainder is kept in source form on the tape. Source blocks contain 16 lines of 64 characters each (again, 512 words plus block number). These blocks are rarely full. It is to the programmer's advantage to keep logically related routines in the same block or nearby blocks, and to allow for future modifications it is handy to keep a few blank lines in each block. There is essentially no cost for this convenience, since loading is very fast and fairly independent of the number of blocks being loaded (it depends on the amount of program

R/B   (Scan number on the stack.) Pushes on the stack the number of scans per block.

/MOD   Divides the scan number by R/B, leaving the block number on top of the stack and the remainder (which record in the block) beneath.

511   511 pushed on the stack.

MIN   Leaves on the stack the lesser of 511 and the computed block number (if you are going to allow more than 512 blocks using multiple files, a /MOD here will get the file position).

BLOCK   Replaces the block number on the stack by the address of the first word of the block.

SWAP   Exchanges the block address with the record number.

LR   Pushes on the stack the number of words per record.

*   Replaces LR and record number with the offset in words from the beginning of the block.

+   Replaces the block address and the offset with the record address.

Figure 12

```
: ADDRESS  R/B  /MOD  511  MIN  BLOCK  SWAP  LR  *  +  ;
```

being loaded.) This method of keeping the program source con-
veniently available is one of the main features that makes
FORTH so flexible. During program development and testing,
specified source blocks may be edited (using FORTH's built-in
editor) and, if desired, reloaded quickly and easily. When
a FORTH program is running in production, the fact that there
is no complete object program on a mass storage device is no
inconvenience, as the program doesn't need to be reloaded often,
and it is a process taking only a matter of seconds even for
fairly elaborate programs. The basic FORTH package that comes
for your computer includes the compiled object program (blocks
1 and 2) and about a dozen blocks of source including the
assembler, compiler and text editor. A diagram showing this
basic program is shown in Figure 13. In addition, you will
get the source blocks from which the object program was com-
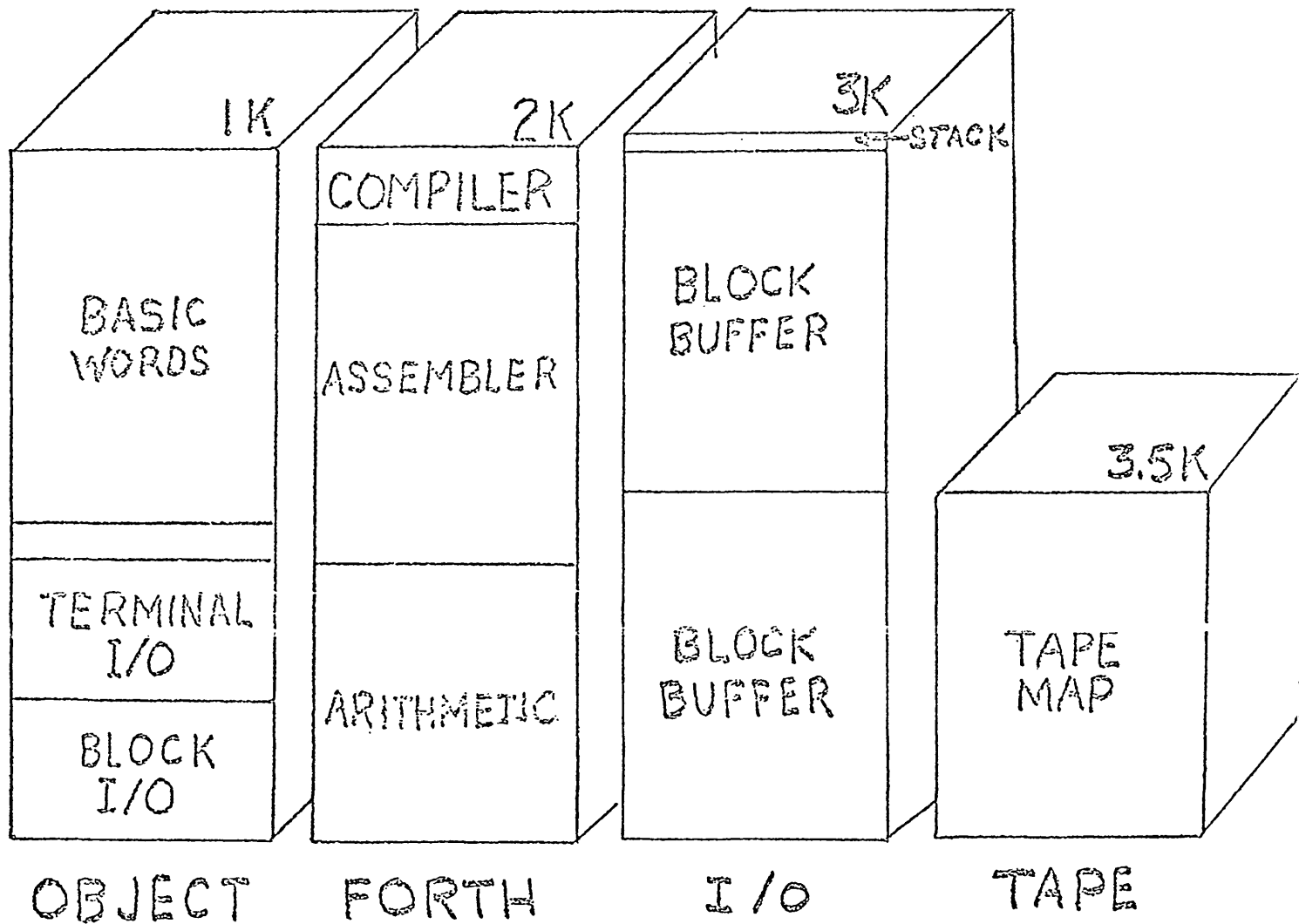piled, and a set of vocabulary lists for all of the above.

Figure 13

Core layout of basic FORTH. The object program and FORTH
occupy roughly the first 2048 words of memory. The tape
map and block buffers are in high memory. The stack extends
down from the bottom of the block buffers toward the dictionary.
The space between the "top" of the stack and the top of the
dictionary is unused core.

FORTH PROGRAMMING

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. The definitions may then be edited into a block or more of source program which will be kept permanently on disk (or tape), load the new blocks, and re-test.

To facilitate testing (and also to allow overlays), the verb REMEMBER is used to mark a place in the dictionary. At some future time, typing the name of the remembered 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

        REMEMBER   TEST

Later, when you are ready to begin editing, or if you feel the dictionary is becoming too cluttered, you may type

        TEST

and everything in the dictionary beyond (and including) TEST's remembered location will go away. Recall that 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.

REMEMBER is also useful for marking sub-vocabularies to be overlaid by other sub-vocabularies. You may, for example, have a telescope control vocabulary, interrupt handler and multiprogrammer that you want to be available all the time, but several observing vocabularies that are mutually exclusive. In the common vocabulary you will want to give a name to the first block of each:

    43 CONSTANT SPECTROMETER 60 CONSTANT COUNTER
    72 CONSTANT IMAGE-TUBE   84 CONSTANT GRATING

and a null definition:

    : OVERLAY ;

Each of these blocks will load the other blocks that are included in the sub-vocabulary. The beginning of each block will contain

    OVERLAY   REMEMBER   OVERLAY

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

    SPECTROMETER   LOAD

or

    IMAGE-TUBE   LOAD

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

If you look at the listing of the text editor (EDIT, block
13) you will see that it starts off,

    DISCARD   REMEMBER   DISCARD

    : DISCARD   FLUSH   DISCARD   ;

By convention, DISCARD is used at the beginning of all major
vocabularies (the portion that is loaded just after basic
FORTH). The editor, therefore, intends to discard this vocabu-
lary. But when the time comes to discard the editor, more
will happen: the word FLUSH will write on tape any blocks in
memory that have been updated and not yet flushed before the
editor vocabulary is discarded. (Remember that when a word
appears to refer to itself it is not recursive -- the second
definition of DISCARD redefines DISCARD, including in the
definition a reference to the previous definition of DISCARD.)

When you type a definition, or use an untested definition,
or load a newly edited block, you may get a diagnostic. Diag-
nostics are very simple:

    word ?

This means either that "word" is undefined, or that it ex-
pects a parameter on the stack and finds none (stack under-
flow). You may determine which easily by typing   word .

If the diagnostic is repeated, the word is undefined. If a number (the location of the word) appears, the word must have been expecting a parameter. If you have typed a string of words, you may normally assume that it has been obeyed as far as the word causing the diagnostic.

When you are testing a new definition it is a good idea to type . after executing it until 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.

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! The text.editor allows you to save definitions permanently on disk or tape.

You may list a block (16 lines of 64 characters each) at any time, by giving its number and LIST:

     13   LIST

Blocks that have names may be requested by name:

     EDIT   LIST

To load the text editor, have basic FORTH running and type

     EDIT   LOAD

This will usually not only load the editor but also discard any part of the application that may be loaded, although on some systems EDIT can co-exist with a portion of the application. The editing commands are:

14 BLK =	Specifies that block 14 is about to be edited. It will be fetched from disk or tape if necessary.
7 T	Type line 7 (place in line buffer).
" text "	Place 'text' in line buffer.*

---

*Quotation marks are the standard delimiting characters for the text editor. For cases in which you need to use quotes in a line of text, two other sets of delimiters have been defined: parentheses and #.

7	R	Replace line 7 with contents of line buffer.
13	I	Insert buffer after line 13 (discard line 16).
13	D	Discard line 13 (place in line buffer). Line 16 will be reproduced and other lines will move up as needed.

If you are about to edit a previously undefined block, type

    n   EMPTY

where n is the logical block number. Thereafter, when you type n BLK = you will have a fresh block to work on. This block is filled with undefined characters. You edit in new lines by replacing lines with text. For example:

    " THIS IS A NEW LINE "   1   R

Begin with "quote space" -- the end of the line is defined by the last quote. The rest of the line will be filled with blanks. When you finish typing in lines of text, you should fill any unused lines with spaces. A blank line is defined by at least two spaces within quotes:

    "   "   13   R   14   R   15   R   16   R

Notice that you may string together R's in this case, becaus the spaces remain 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 BLK =   8 D   24 BLK =   1 R

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

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

The last word of program in a block must be

          ;S

This marks the logical end-of-block, and when the block is loaded, loading will cease for that block when it is encountered. It may appear anywhere in the block, though -- anything in the block after the ;S will be ignored.   You may want to put remarks there.

AFTER WORD

If you have read this book carefully, you already understand how FORTH actually works better (unless you are very exceptional) than you understand the FORTRAN compiler you are accustomed to using. Although you may find FORTH's conventions strange or awkward for a while, you will find very soon that you can accomplish more with your mini-computer, in a shorter time, using less core, than if you use a conventional assembler.

We hope that this manual (and a little practice at a terminal) will be sufficient to get you well under way toward describing your application with a FORTH vocabulary. We will greatly appreciate any suggestions you have for improving this book.

As your application develops, you may make substantial use of vocabularies developed for other applications, or even other computers (FORTH definitions are fairly computer-independent). These include double-precision math, trig functions, interrupt handlers, multiprogrammers, drivers for standard pieces of equipment and many other useful routines. Your inquiries are welcome.

APPENDIX A

Basic FORTH Vocabulary

ABS
: Replaces a signed 16-bit integer on the stack with its absolute value.

BASE
: The base for converting numbers to binary. May be set to 10 by DECIMAL or 8 by OCTAL, or to any other value.

BLOCK
: Takes a block number on the stack and replaces it by the address of the beginning of the block in core, having fetched the block from tape or disk if necessary.

CONSTANT
: Defines a named 16-bit constant. A reference to a CONSTANT causes its value to be pushed on the stack.

CONVERT
: Takes an unsigned 31-bit integer on the stack, whose signed value is in N, and converts it for printing by TYPE. Leaves the character count - 1 on the stack and the character address of the number's character string beneath.

COUNT
: Takes the address of a dictionary entry on the stack and replaces it by the character count in its name (obtained from the left half of the first word of the entry), with the character address below, for TYPE.

CR
: Sends a carriage return and line feed to the terminal.

DECIMAL
: Sets BASE to 10; numbers converted on input and output will be decimal.

DELIM
: The delimiting character that defines the end of a word. Normally a blank, it is sometimes temporarily changed to " of) for special purposes.

DO	Begin a loop. Takes an initial counter value on the stack and a limit beneath.
DP	The address of the pointer to the next available word in the dictionary.
DROP	Discard the top number on the stack.
DUP	Duplicate the number on top of the stack.
ELSE	End the true part of a conditional phrase and begin the false part. Compiles a jump forward to a THEN (for the true part) and sets the address of the forward jump in IF for the false part.
EOT	End of input message. On a TTY, sent by the RETURN key.
ERASE-CORE	Mark both core buffers empty.
F	Width of a numeric field for printing. Numbers will be right-adjusted in the field.
FLUSH	Write any blocks that have been changed from core onto disk or tape.
FORTH	Removes all of the dictionary beyond the end of the object program.
HERE	Push the contents of DP on the stack.
I	Push the value of the current loop counter on the stack. Must be used inside a loop.
IF	Begin a phrase to be executed only if the top of the stack is true (non-zero). Destroys this stack value. Compiles a conditional forward jump to an ELSE, or to THEN if the ELSE clause is omitted.

INTEGER   Define a named 16-bit variable. A reference to an INTEGER causes its address to be pushed on the stack.

LOOP   End a loop. The loop counter will be advanced by 1, and the loop will be terminated if the value is equal to the limit.

+LOOP   End a loop, incrementing the loop counter by the signed amount on the stack. The loop will terminate if the counter value exceeds the limit.

MAX   Compare two numbers on the stack, destroying the lesser.

MIN   Compare two numbers on the stack, destroying the greater.

MINUS   Change the sign of the number on the stack.

MOD   Divide the second number on the stack by the top number, leaving the remainder on the stack.

OVER   Push a copy of the second number on the stack on top of the stack.

READ-MAP Read to the end of file on a tape, constructing in memory a map showing what blocks are in which record positions on the tape.

REMEMBER Mark the beginning of a vocabulary, giving it a name. A subsequent reference to that name will cause the vocabulary to be removed from that point on.

REWIND   Rewind tape.

SWAP   Exchange the top two numbers on the stack.

THEN   End a conditional phrase. At compile time, sets its address as the destination of the jump in ELSE, or IF if ELSE is not used.

TYPE   Move a character string into the line buffer, to be sent to the terminal.

UPDATE   Mark the block most recently referenced as changed. A subsequent FLUSH, or the need to re-use that block buffer, will cause the up-dated block to be written on tape or disk.

WHERE   Print first three characters of the most re-cently defined word.

WORD   Read the next word (until reaching the character specified by DELIM) from the input string, leaving it starting at DP with the character count in the high order half of the first word.

.   Type the number on top of the stack.

:   Begin compiling a definition.

:I   Begin compiling a compiler directive.

;   End a definition.

;CODE   End a definition and begin a code string to be associated with it.

;S   End the part of a source block to be inter-preted.

,   Put the number on the stack into the dictionary and advance DP.

'	Push on the stack the address of the parameter field of the following word.
+	Add two numbers on the stack leaving the sum.
-	Subtract the number on top of the stack from the number beneath, leaving the remainder.
+=	Add the second number on the stack into the memory location whose address is on top of the stack.
@	Replace an address on the stack with the contents of that address.
=	Store the second number on the stack in the memory location whose address is on top of the stack.
?	Fetch and print the contents of the address on the stack.
!	Equivalent to = (defined for consistency with other FORTH systems).
*	Multiply the two numbers on the stack, leaving the product.
/	Divide the second number on the stack by the top number, leaving the quotient.
/MOD	Divide as in / and MOD leaving the quotient on top of the stack and the remainder beneath.
*/	Multiply the second and third numbers on the stack and divide by the number on top. The intermediate product is 32 bits.

0<   Replaces the number on top of the stack by a
                   1 if it is negative, or 0 if it is zero or
                   positive.

0=   Replaces the number on top of the stack by a 1
                   if it is zero or 1 otherwise.

<   Compares the two numbers on the stack. Leaves
                   a 1 if the second is less than the top (both
                   numbers destroyed).

>   Opposite of <.