NATIONAL RADIO ASTRONOMY OBSERVATORY

Tucson, Arizona

COMPUTER DIVISION INTERNAL REPORT NO. 17

Basic Principles of FORTH Language as Applied to a PDP-11 Computer

E. D. Rather
C. H. Moore
Jan M. Hollis

MARCH 1974

TABLE OF CONTENTS

Introduction

1. FORTH Programmer's Guide. General discussion of FORTH.
2. The FORTH Dictionary. How it is organized, and how a dictionary entry is constructed.
3. Stack Manipulation. How the parameter stack is used; stack handling vocabulary and examples.
4. FORTH Compiler. General rules for making : definitions; loops and conditionals.
5. FORTH Interpreter. How the inner and outer interpreters handle words typed at the terminal, load blocks of FORTH source, compile and execute definitions.
6. Nouns. INTEGERs, CONSTANTs, and more elaborate nouns.
7. Arithmetic. Basic single-precision arithmetic; 14-bit fractions; trinary operations; number formats.
8. The FORTH Assembler. Assembler conventions; action of the assembler and execution of assembled code; interpreter returns; stack modification; macros.
9. Block I/O. Tape and disk I/O, with suggestions for organizing data records.
10. FORTH Programming. Suggestions for organizing and testing an application vocabulary.
11. Text Editor. How to generate new blocks of FORTH source on tape or disk, and to modify existing blocks.

After Word.

Appendix A. Basic FORTH vocabulary.

Appendix B. Listing of basic FORTH text.

Appendix C. Comparison of the FORTH language symbolism for register addressing with those of the PDP 11/40 Processor Handbook.

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, Varian, MODCOMP II, PDP 11, PDP 10 and IBM 360-50. Since most of these 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

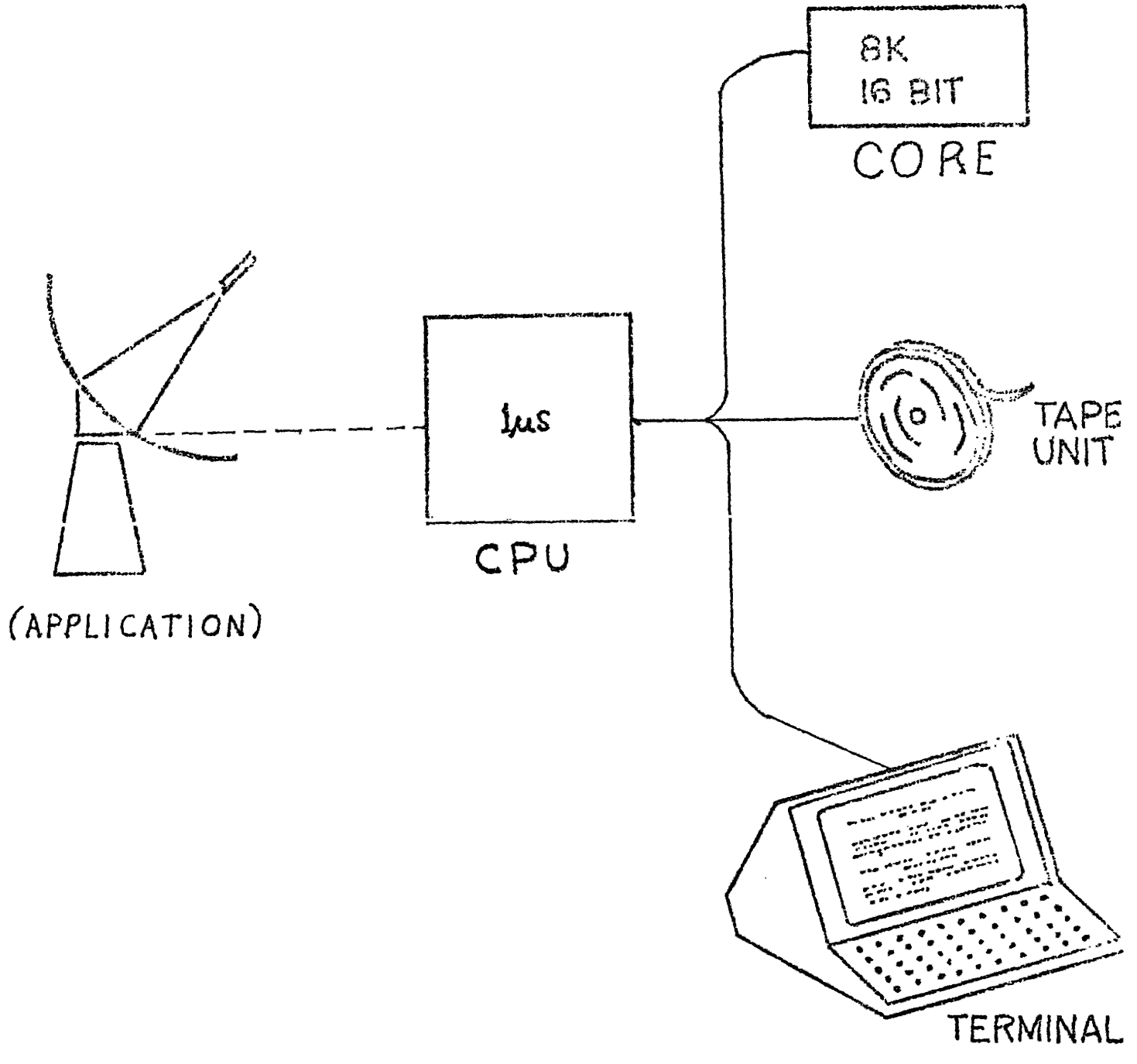

Figure 2

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 us.

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:

RETURN to mark the end of a message.

RUB OUT to erase a letter.

BREAK 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 RETURN, or 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 is pre-compiled (on the same computer or any other FORTH computer), and contains about 30 defined 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 (3 LOAD). 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 distingquish them -- and other subvocabularies -- by being able to

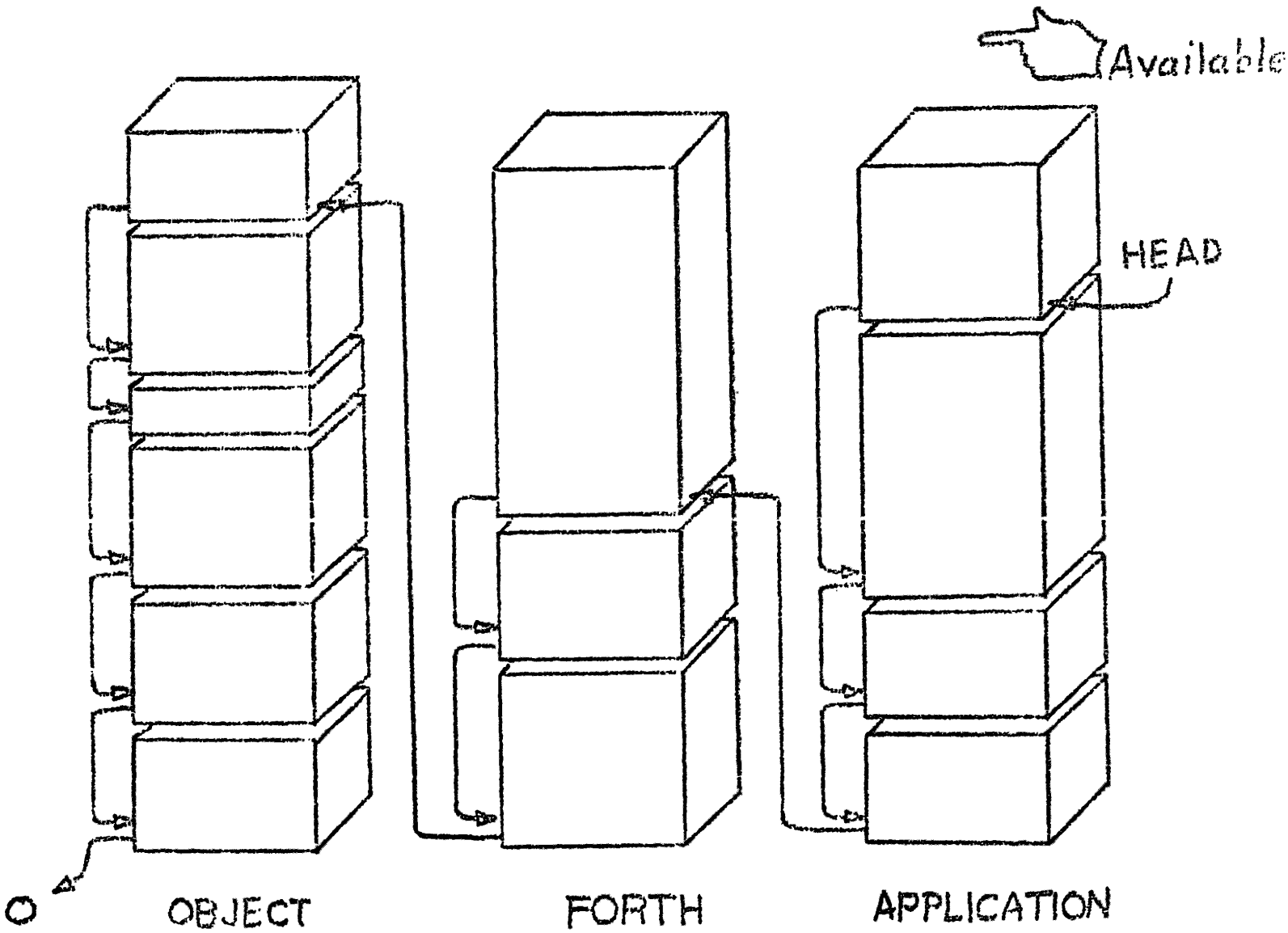

Figure 3

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. It is also possible to define a context in which only certain sub-vocabularies will be searched.

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 fourth 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.

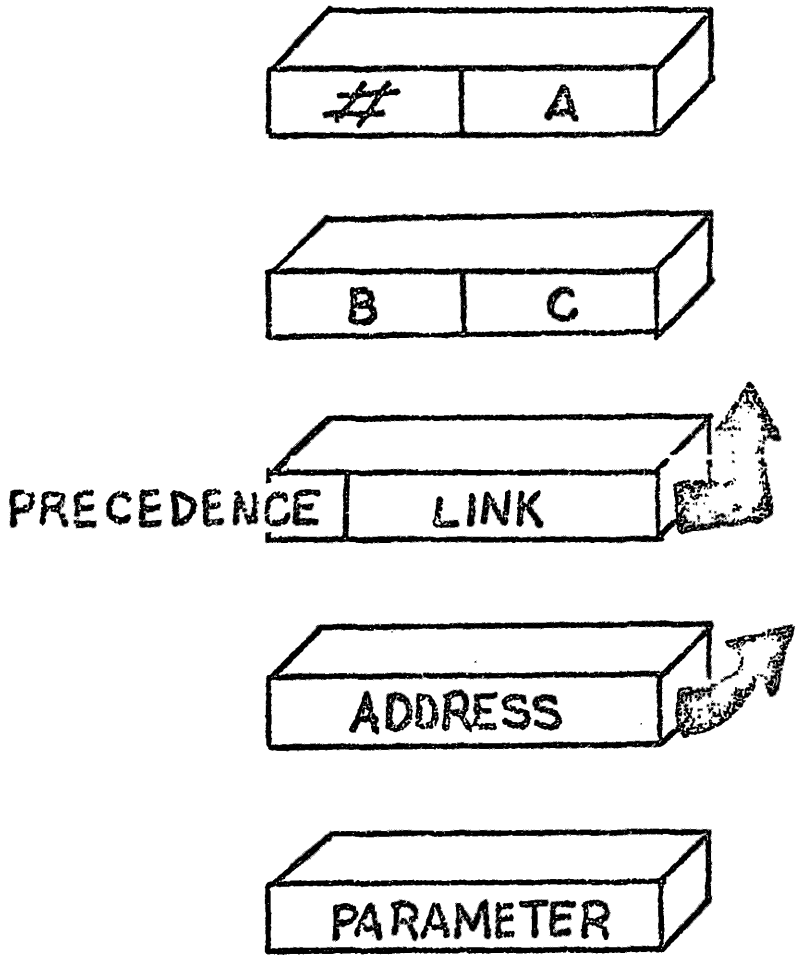

Figure 4

16-Bit Dictionary Entry Format

For an INTEGER, it refers to code that puts the address 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.

STACK MANIPULATION

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" (kept in register 5 on the PDP 11). 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 parameters on the stack: an address on top and a number beneath. It stores the number at the address.

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	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 important 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

Word	Explanation	Example Before (Top)	Example Operator	Example After (Top)
FORTH words (may be used in definitions or in CODE)				
SWAP	reverses order of top 2 entries	1 2	SWAP	2 1
DUP	reproduces top entry	1	DUP	1 1
DROP	discards top entry	1 2	DROP	1
OVER	pushes entry 2 on top of entry 1	1 2	OVER	1 2 1

plicated stack operators is available in some developed applications, 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 (!) 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

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 one of two ending words: LOOP (which increments the loop counter by 1) or +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.

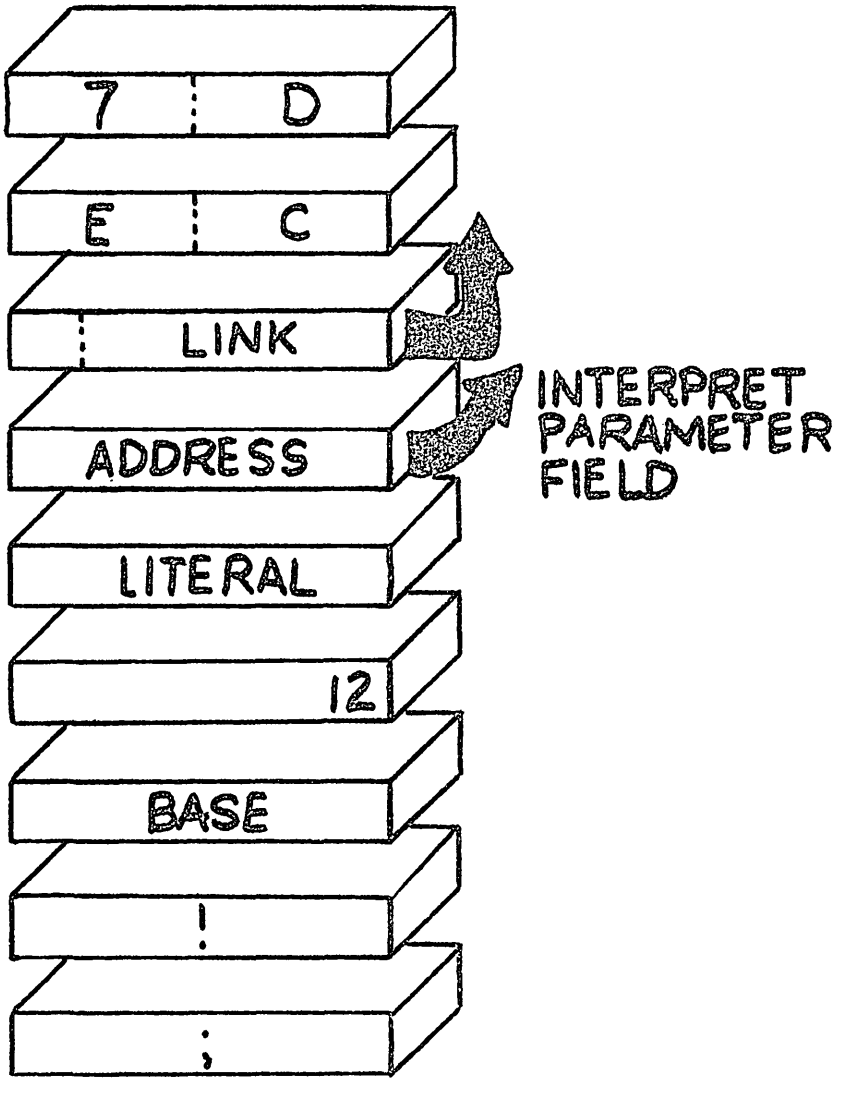

Figure 6

Dictionary entry for

: DECIMAL 12 BASE ! ;

(Stores 12_8 in the variable BASE)

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 (5050) 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 I 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 PDP 11 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 4.6 μs on the PDP 11. 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 6 μs on the PDP 11.

*

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 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 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.

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 instructions 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 interpreter 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. Executing definitions. 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 definition, 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. On the PDP11/40 this step requires only 4.6 μs.

The existence of compiled addresses in definitions is the feature that distinguishes FORTH from other interpretive languages

such as BASIC, which operate only in modes I and II, and are therefore much too slow for many real-time tasks and complicated computations.

NOUNS

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 involved in the definition of new kinds of words. Not definitions 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,we 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 constructs 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 PDP 11 is

```
: INTEGER CONSTANT ;CODE E ASR, S -) E MOV, NEXT
```

Given that definition, the phrase

```
0 INTEGER M
```

then acts as follows: The word

INTEGER - executes the definition, wherein

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.

E ASR, generates the first instruction that will be executed for M (or any other integer). When a code definition is entered, the E register (register 2) contains the byte address of the first word of the definition. E ASR, shifts this address right one bit, thus converting it to a word address. See "ASSEMBLER" for a discussion of addressing on the PDP 11.

S -) E MOV, moves the contents of the word pointed to by E to the top of the stack -- automatically decrementing S to effect a "push" operation.

NEXT goes to the next word to be executed.

Every time an INTEGER is defined the definition is executed.

Every time an INTEGER is referenced the code is executed.

Tbe definition of what INTEGER does is the same on all FORTH computers; the code of course varies with the computer.

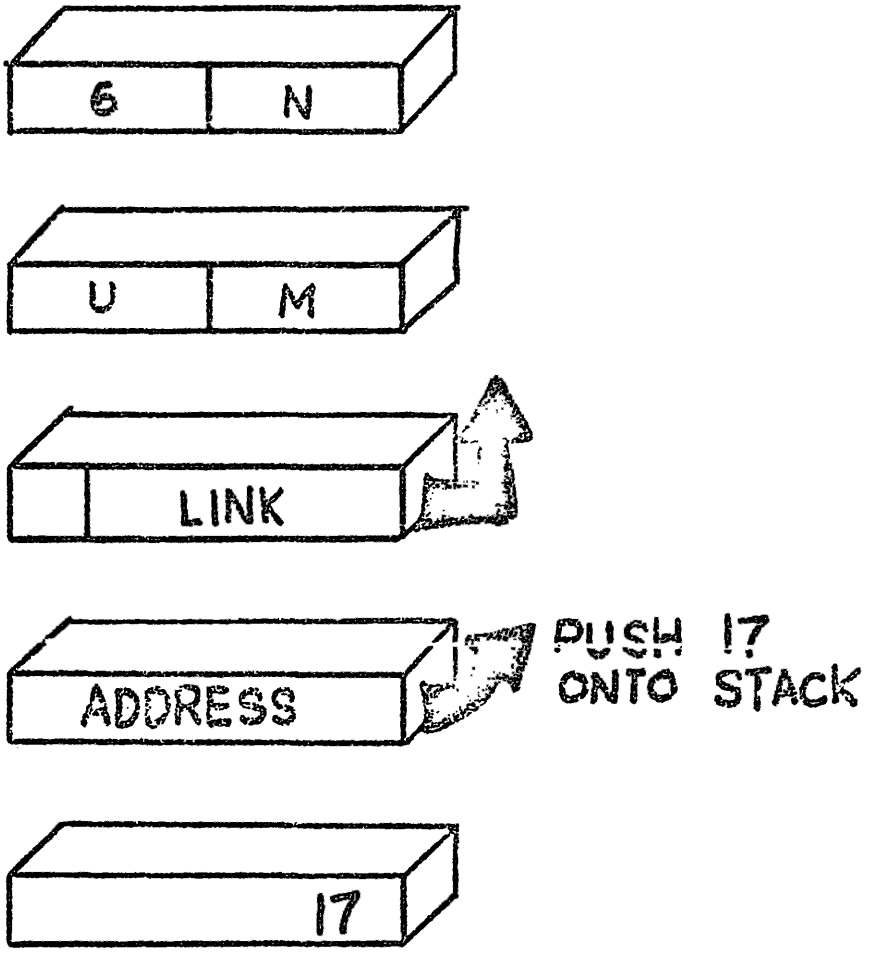

Figure 7.

Dictionary entry for
17 CONSTANT NUMBER

INTEGER is only one kind of noun. Other kinds of nouns can be easily imagined: DOUBLE, 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 0 INTEGER FLAG

```
0 FLAG SET AZ     1 FLAG SET EL
```

Whenever I reference AZ, the constant 0 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    0    FLAG  !  ;
```

is completely equivalent to the above. But it is considerably less efficient, both in space and time (11 μs vs 40 μs). 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 word in the vector plus the contents of a noun, (), which is a general subscript. The definition of VECTOR on the PDP 11 would be very similar to that for INTEGER:

```
0  INTEGER    ()
:  VECTOR  0  CONSTANT  ;CODE  E ASR,  E () ADD,
S  -)  E  MOV,    NEXT
```

Notice that the complete value of the vector is not initialized by the definition. It is simpler to initialize the VECTOR explicitly to the proper length:*

```
VECTOR  X  0   ,  0   ,
```

In conjunction with SET this leads to some pleasant notation.

* The operator "," enters a number on top of the stack into the dictionary.

If I define

```
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 polynomial) 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 without 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 automatically 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 discussed here are available. If you encounter more than one instance of a particular kind of noun, 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.

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, perhaps several 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, leaving the sign in the variable SIGN.
/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.

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, the decimal point suggests printing in most applications. We 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 combination occurs so frequently that the added efficiency of combining them is welcome. */ is a natural combination since the double-precision product in registers 0, 1 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 (to 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 DEC 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 <u>any</u> 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. On the PDP 11 there are two basic formats:

SO instructions have a single operand

DO instructions have two operands: a source and a destination.

Depending on the type of addressing used, an instruction may require 1, 2 or 3 words. Standard instruction mnemonics are used, with a comma appended. Addresses precede operations and destination precedes source. The words are exactly reversed from the DEC format:

destination source MOV, for MOV source, destination

Addresses are 16-bit word addresses (so that they can be incremented by 1 consistent with other versions of FORTH). The assembler multiplies by 2 when it wants byte addresses; how-

ever indexing and immediate addresses are ambiguous. Hence the distinction between W# and W) for word addresses and # and B) for byte addresses.

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 (register 5) and the returns that push or pop the stack. 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. This 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 in CODE, and compiled into the dictionary in definitions.

The most serious design flaw of the PDP11 computer is the way byte addressing is implemented: word instructions use byte addresses. PDP11 FORTH conceals this anomaly in order to stay compatible with versions on other computers. This costs time (900 us/address) and space (∿ 30 instructions) and is regretable. However experience with this problem on the

360 amply justifies the cost/benefit in favor of compatibility.

The rule is simple: addresses on the stack are word addresses. Adding one will address the next 16-bit word. Before such an address can be used by the computer (cn @ and ! for instance) it must be doubled.

This addressing problem cannot be completely concealed in the assembler. Addresses are expected to be word addresses, and are doubled. However if an immediate operand is to be used as a word address, it must be doubled; the operator W# indicates this. Conversely the offset for an indexed byte address must not be doubled; the operator B) indicates this.

Addresses in registers, the CODE field, or compiled into definitions are byte addresses. However, the LINK field has a word address (<16384). This table illustrates where word and byte addresses occur:

registers	byte	
compiled entries	byte	
CODE field	byte	
LINK field		word
stack		word
DP		word
W)		word
B)	byte	
#	byte	
W#		word
@		word
!		word
'		word
INTEGER		word
S)		word
E)		word

WORD	ACTION DURING ASSEMBLY	ACTION DURING EXECUTION
Ø	Ø is placed on stack (will be destination address)	Loads a word address found on top of the stack into register Ø; the stack pointer is advanced to the next word on the stack (that is, the address is discarded from the stack)
S	Puts 5 on the stack (will be source address)	
)+	Adds 2Ø to the 5 on the stack (sets bits for autoincrement, indirect addressing)	
MOV,	Assembles a one word machine MOV instruction; the source address is the top of the stack; the destination address is register zero, which is the second word on the stack. The machine instruction will be 125ØØ. DP is advanced one word.	
Ø	Ø is placed on stack (will be destination address)	Converts the word address in register Ø to a byte address.
Ø	Ø is placed on stack (will be source address)	
ADD,	Assembles a one word machine ADD instruction which looks like 6ØØØØ. DP is advanced one word.	

WORD	ACTION DURING ASSEMBLY	ACTION DURING EXECUTION
Ø	Ø is placed on the stack.	Moves the contents of the top of the stack (a number) to the byte address which is located in register Ø ; The number is discarded from the top of the stack.
)	Adds 1Ø to the Ø on the stack (sets bits for indirect addressing)	
S	5 is placed on the stack.	
)+	Adds 2Ø to the 5 on the stack (sets bits for autoincrement, indirect addressing)	
MOV,	Same as previous MOV except the machine instruction assembled is 1251Ø.	
NEXT	A macro which assembles machine move and jump instructions necessary for the program control loop.	Go to next word.

Figure 8

```
CODE ! Ø S )+ MOV, Ø Ø ADD, Ø ) S )+ MOV, NEXT
```

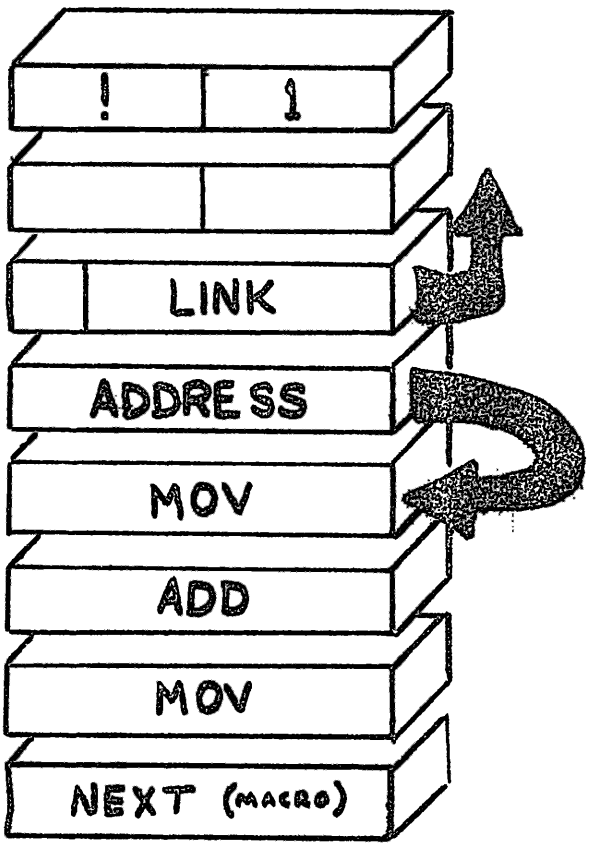

Figure 9

Dictionary entry for store operator (!)

```
CODE !   0 S )+ MOV,  0 0 ADD,  0 ) S )+ MOV,  NEXT
```

Note: PDP11 idiosyncracy in the arrangement of the name field: the bytes are reversed from their order in standard FORTH notation due to PDP11 byte addressing convention.

Several operators specify addressing mode:

```
 )   register indirect
)+   indirect, autoincrement
-)   indirect, autodecrement
W)   word indexing    :  S)    S  W)  ;
B)   byte indexing
I)   memory indirect
 #   immediate
W#   word address immediate
```

For example:

```
CODE  !  0  S  )+ MOV,  0 0 ADD,  0 ) S )+ MOV, NEXT
```

will store the second number on the stack into the address on top of the stack; the 0 0 ADD, is necessary to convert this address to a byte address. 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. Examples of all of these are in the basic vocabulary. A computer with a single-word jump includes the jump code with these words. On the PDP11 these words provide just the address, and an explicit (perhaps conditional) jump must follow.

STACK: Most parameters come to code entries via the stack. On the PDP11, the pointer to the top of the stack is in register 5. Since the top is most often referenced, the word S refers to it directly:

```
S )  addresses the top of the stack
1 S) the next word - etc.
```

Autoincrementing pops a number from the stack, and autodecrementing

pushes a number on the stack. For example:

```
0  S  )+  MOV,
```

removes a number from the stack and puts it in register Ø; and

```
2  S)  0  MOV,
```

will replace the third parameter with the contents of R0.

Parameters may be taken directly from core. The assembler will automatically check to determine whether the address of the argument permits a single word instruction, and if it will not, the extended form is used. For example:

```
B  A  ADD,
```

will add the value of A to B (note this will be a 3 word instruction - the addresses of A and B each requiring a full word. In fact, since the stack is cheaper and faster to use, this form rarely occurs.) 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

```
0  SWAP  MOV,
```

coming later will encounter its address -- and assemble an instruction to move it to R0.

REGISTERS 0 and 1 are always available; they are not disturbed by the interpreter. Register 2 is set by the interpreter but is available as a scratch register. The others may not be used without saving and restoring. The conventional place to save registers is the return stack. The full register allocation scheme is:

register name pointer to:

0		
1		
2	E	parameter field of current word.
3	U	user area of current user (on multi-user systems)
4	I	address of address field of next word to be executed.
5	S	top of parameter stack.
6	R	top of return stack.
7	P	next instruction.

MACROS:

Many instructions are useful, though not implemented on a particular computer. Macro definitions will implement them with several instructions. On the PDP11, NEXT is a macro which assembles two words on the end of a code definition. The following macros are available for branching:

BR, generates a single or double word jump as necessary.

```
BEGIN   =0     END
        <0     IF    ELSE    THEN
        >0
        =0   NOT
        <0   NOT
        >0   NOT
        1V    ( overflow set )
        1C    ( carry set )
```

IF begin true condition code

ELSE begin false condition code

THEN end conditional code

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

IF takes a condition code as parameter; 0 IF is an unconditional branch.

Example:

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

or

```
0= IF  (code for 0)   THEN
```

IF and ELSE leave the address of a branch instruction destination on the stack; THEN provides the address.

IF takes a condition code as parameter; 0 IF is an unconditional branch.

Example:

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

or

```
0= 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 0 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 words 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.

There are two words that function similarly to CODE: ,CODE and ;CODE. ;CODE is used to define classes of words, especially nouns. This use is discussed further under "NOUNS". ;CODE assumes you have included in the word you are defining

a defining word (such as CONSTANT) which at a later time will generate a definition. ;CODE assembles some code at this time, and at the later time will modify the address field of that definition to point to this code.

Example

```
: INTEGER CONSTANT ;CODE E ASR, S -) E MOV, NEXT
```

The definition of INTEGER includes in its parameter field the addresses of CONSTANT and ;CODE, followed by four words of actual code. Later, if FORTH encounters

```
0 INTEGER X
```

a definition will be constructed for X by the action of CONSTANT (which sets up the basic definition with the value 0 in the parameter field, and ;CODE, which put the address of the ASR instruction in X's code address field. Still later, if you type

```
X
```

the code will be executed - its effect will be to push the address of the parameter field of X on the stack. Note the use of E in this example. When the code for a definition is entered, E points to the parameter field of the definition; in this example, when the code for INTEGER is entered, E contains the address of the parameter field of X. It is converted to a word address and pushed on the stack.

,CODE is the same as CODE except it requires a parameter: the number of words in the parameter field to be skipped before the actual code begins, thus allowing a few local constants to be available without their having to be defined separately. It is not used very often.

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.

Word	Explanation		Example Before	Operator	After
Interpreter returns (terminate CODE entries-contents of A shown as A(n))					
PUT	replace top of stack with R0	A(2)	Top 1	PUT JMP,	Top 2
PUSH	push R0 onto top of stack	A(2)	1	PUSH JMP,	1 2
'DROP	discard top of stack		2 1	'DROP JMP,	2

Figure 10

Assembler return locations which modify

the stack and return to NEXT

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

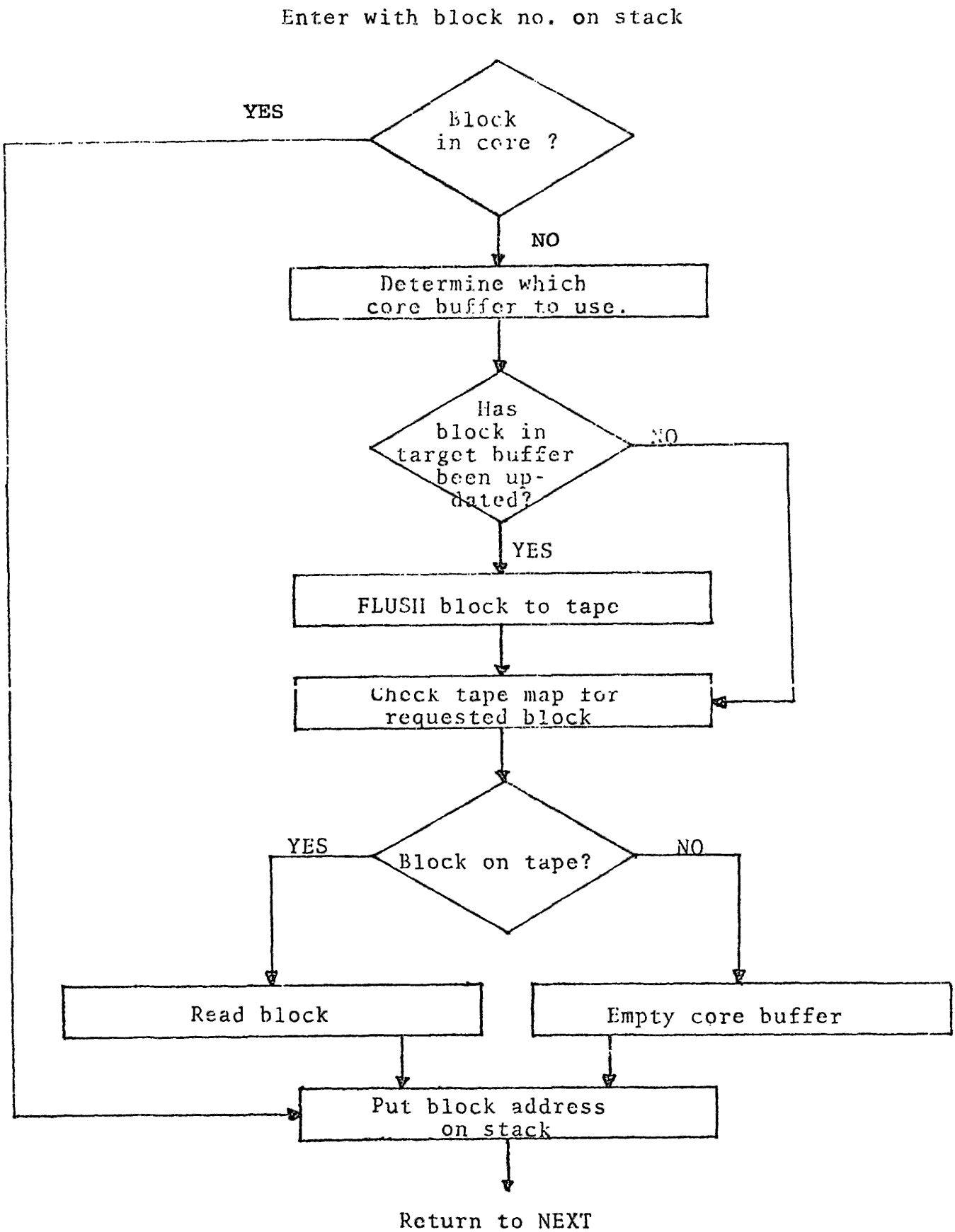

Figure 11

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.

It should be noted that disk systems do not require any sort of map in core, as block numbers are a direct function of disk address (the exact relationship is designed to suit the particular disk involved). Tape utilities are provided to dump blocks onto tape and re-load them as necessary. This saves a substantial amount of core, but has the side effect that all blocks within the capacity of the disk are, by definition, "available". Blocks which are not in use should be flagged, normally by putting Ø in the first word.

By convention, 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 or disk. 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 load time is fairly independent of the number of blocks being loaded (it depends on the amount of program being loaded.) This method of keeping the program source conveniently 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 compiled, and a set of vocabulary lists for all of the above.

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  *  +  ;
```

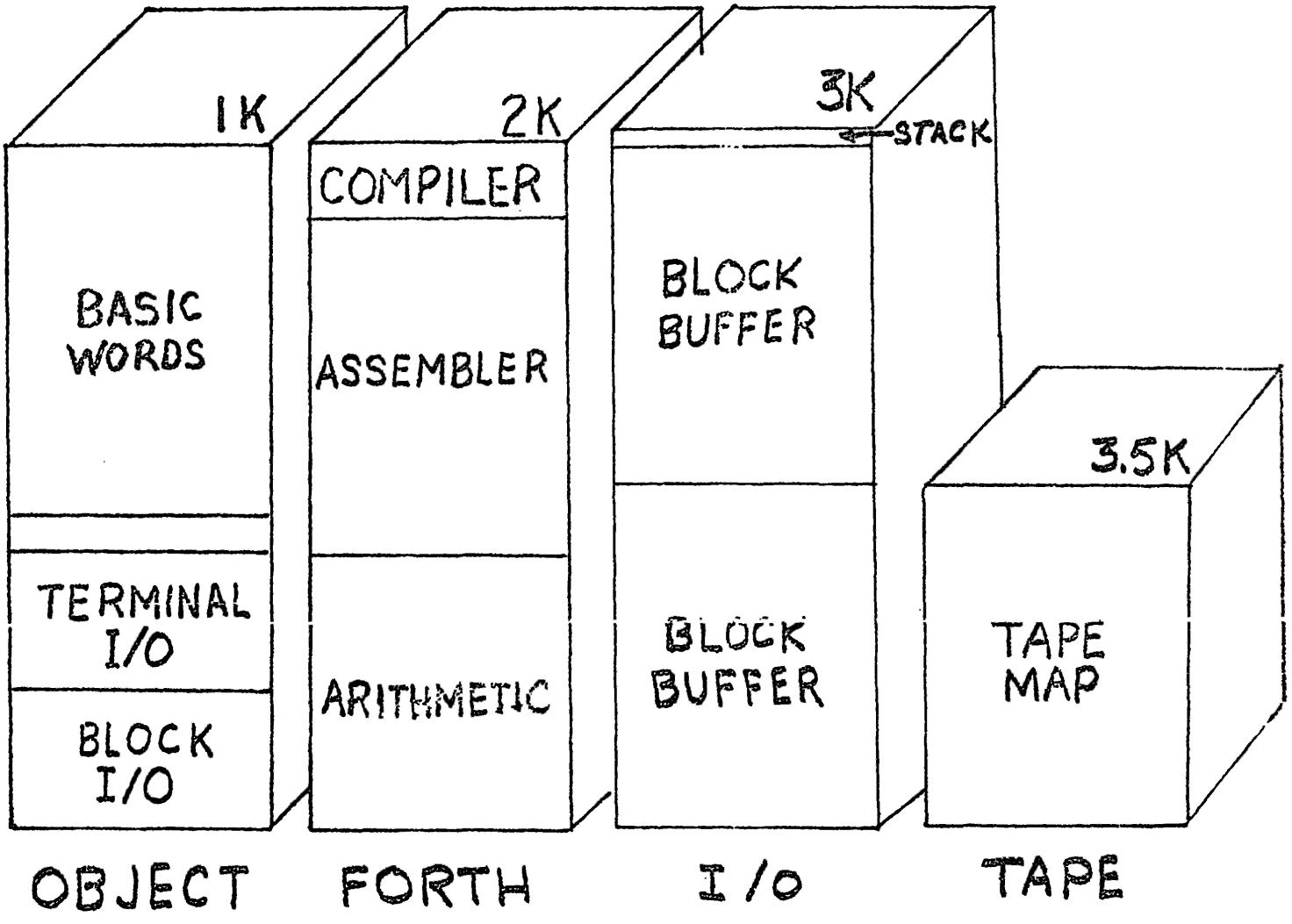

Figure 13

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), 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 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

```
: TEST ;
```

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

```
FORGET 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. Alternatively, you may FORGET any normal dictionary entry, again discarding that entry and everything following it.

This technique 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 at the end of the common vocabulary 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

```
FORGET OVERLAY : OVERLAY ;
```

The FORGET 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 the new definition of 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.

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:

```
word  ?
```

This normally means either that "word" is undefined, or that it expects a parameter on the stack and finds none (stack underflow). 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. On the PDP 11, there are two hardware traps that cause aborts of this type: illegal addresses and "reserved instructions". These may only occur in CODE, thus they may normally be ruled out if you definition is using only tested words. 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.

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:

```
TRACK  LIST
```

To get access to the editing vocabulary have basic FORTH running and type

```
EDITOR
```

Remember that you have access only to the EDITOR vocabulary and basic FORTH - to regain access to your application vocabulary you must type FORTH and the vocabulary name if it has one. 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.

No special action is required to edit a previously unused block. 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, because 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, interpreting 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 minicomputer, 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

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.
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 " or) for special purposes.
DIGITS	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.

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.
FLUSH	Write any blocks that have been changed from core onto disk or tape.
FORTH	Identifies the FORTH vocabulary.
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.
IMMEDIATE	Used in a definition to mark the word being defined as a compiler directive.

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. (Tape systems only).
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 updated block to be written on tape or disk.
WHERE	Print first three characters of the most recently 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.
;	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 interpreted.
,	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.
=	Equivalent to ! (defined for consistency with other FORTH systems).
!	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.
*	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 0 otherwise.
<	Compares the two numbers on the stack. Leaves a 1 if the second is less than the top (both numbers destroyed), or 0 otherwise.
>	Opposite of <.

```
3

 1 4 LOAD   FORTH DEFINITIONS
 2 CODE -U   0 U MOV,   0 ASR,   0 NEG,   PUSH JMP,
 3 : USER   -U + CONSTANT ;CODE
 4    S -) U MOV,   S ) ASR,   S ) E ) ADD,   NEXT
 5 140 USER BASE   BASE 1 + USER DP   DP 6 + USER IN
 6 5 LOAD  7 LOAD   : U)  -U + DUP + 63 ;     6 LOAD
 7 : BR,   0 END ;   10 LOAD  1301 1 - DECIMAL  91 LOAD
 8 237 LOAD  OCTAL  11 LOAD 12 LOAD 13 LOAD
 9 BASE 3 - USER BLK   14 LOAD  16 LOAD
10 ( POWER FAIL)  13 12 W!   137 0 !  ' RESTART 24 + 1 W!
11 DECIMAL 16 CONSTANT 36'   419 CONSTANT BOOTSTRAP
12 : FLUSH   GET BUFFER BUFFER ;
13 : ERR-MSG   32 * 349 BLOCK +  -TRAILING 1 + TYPE ;
14 OCTAL          : BLOCK BLOCK -177 @ DUP 177743 AND IF 13 ERR-MSG
15                      . 0 THEN DROP ;                    DECIMAL
16 ( EDITOR)  13 LOAD   : TASK ;       36' LOAD   ;S
4

 1 ASSEMBLER DEFINITIONS       3 CONSTANT U
 2 6 CONSTANT R   5 CONSTANT S   4 CONSTANT I   2 CONSTANT E
 3 : )   10 + ;   : )+   20 + ;   : -)   40 + ;   : B)   60 + ;
 4 7 )+ CONSTANT #       : W#   # 2* ;       : I)   77 2* ;
 5 : W)   2* B) ;        : S)   S W) ;        : E)   E W) ;
 6 5000 SO CLR,   5100 SO COM,   5200 SO INC,   5300 SO DEC,
 7 5400 SO NEG,   5500 SO ADC,   5600 SO SBC,   5700 SO TST,
 8 6000 SO ROR,   6100 SO ROL,   6200 SO ASR,   6300 SO ASL,
 9  100 SO JMP,    200 SO RTS,    300 SO SWB,    260 SO SET,
10
11 30000 DO BIT,   40000 DO BIC,   50000 DO BIS,
12 10000 DO MOV,   60000 DO ADD,  160000 DO SUB,
13 20000 DO CMP,    4000 DO JSR,
14   2400 CONSTANT <0      1400 CONSTANT =0      3000 CONSTANT >0
15 1 CONSTANT C   103400 CONSTANT 1C   PUT -1 + CONSTANT PUSH
16 : NEXT   E I )+ MOV,   E )+ ) JMP, ;       ;S
5

 1 FORTH DEFINITIONS
 2 CODE -   S ) S )+ SUB,   NEXT
 3 CODE AND   S ) COM,   S ) S )+ BIC,   NEXT
 4
 5 CODE @   0 S ) MOV,   0 0 ADD,   S ) 0 ) MOV,   NEXT
 6 CODE !   0 S )+ MOV,   0 0 ADD,   0 ) S )+ MOV,   NEXT
 7 CODE +!   0 S )+ MOV,   0 0 ADD,   0 ) S )+ ADD,   NEXT
 8
 9 CODE OVER   S -) 1 S) MOV,   NEXT
10 CODE SWAP   0 1 S) MOV,   1 S) S ) MOV,   PUT JMP,
11 CODE DUP   S -) S ) MOV,   NEXT
12 CODE 1+   S ) INC,   NEXT
13 : (   IMMEDIATE   251 DELIM !  WORD ;
14 : HERE   DP @ ;       0 CONSTANT 0   1 CONSTANT 1       ;S
15
16
```

```
6

 1 FORTH DEFINITIONS
 2 : W!   ASSEMBLER 2* ! ;
 3 2 CONSTANT 2   3 CONSTANT 3   4 CONSTANT 4
 4 5 CONSTANT 5   6 CONSTANT 6   7 CONSTANT 7
 5
 6 ASSEMBLER DEFINITIONS
 7 102400 CONSTANT 1V
 8 : RTI,   2 , ;
 9
10 : BEGIN   HERE ;
11
12 : IF   NOT  HERE SWAP  HERE 1+ - , ;
13 : THEN   HERE  SWAP +! ;
14 : ELSE   0 IF  SWAP THEN ;
15 ;S
16
7

 1 ASSEMBLER DEFINITIONS
 2 70000 DO MUL,    71000 DO DIV,     6700 SO SXT,
 3 72000 DO ASH,    73000 DO ASHC,    74000 DO XOR,
 4
 5 FORTH DEFINITIONS
 6 CODE */   1 S )+ MOV,    0 1 S) MOV,    1 S) 1 MOV,    S )+ 0 MUL,
 7    S ) 0 DIV,    PUT JMP,
 8 CODE /MOD   1 1 S) MOV,    0 SXT,    S ) 0 DIV,
 9    1 S) 1 MOV,    PUT JMP,
10
11
12 ASSEMBLER DEFINITIONS
13 CODE NOT   0 400 # MOV,    S ) 0 XOR,    NEXT
14 : END   NOT  SWAP  HERE 1+ -  377 AND + , ;
15 : SOB,    100 1 */  77001 +  SWAP HERE - - , ;
16 ;S
8

 1 FORTH DEFINITIONS
 2 : SET   DUP + SWAP CONSTANT ,   ;CODE   E )+ ) E )+ MOV,    NEXT
 3 : OCTAL 10 BASE ! ;   : DECIMAL 12 BASE ! ;   : HEX 20 BASE ! ;
 4 : INTEGER   CONSTANT  ;CODE   E ASR,    S -) E MOV,    NEXT
 5 : ,CODE   DUP +  CODE  HERE 1 - +! ;
 6
 7 CODE MINUS   S ) NEG,    NEXT
 8 CODE ABS   S ) TST,    <0 IF  S ) NEG,    THEN  NEXT
 9 CODE MAX   S ) S )+ CMP,   >0 IF  S ) -1 S) MOV,    THEN NEXT
10 CODE MIN   S ) S )+ CMP,   <0 IF  S ) -1 S) MOV,    THEN NEXT
11 CODE MOVE   0 S )+ MOV,    1 S )+ MOV,    1 1 ADD,
12    2 S )+ MOV,   2 2 ADD,   BEGIN  1 )+ 2 )+ MOV,   0 SOB,    NEXT
13
14 : *   1 */ ;    : /   /MOD SWAP DROP ;    : MOD   /MOD DROP ;
15 ;S
16
```

```
9

 1 FORTH DEFINITIONS
 2 : COMPILE   1 - DUP + , ;
 3 CODE SKIP   I I ) ADD,   NEXT
 4 CODE IF   S )+ TST,   =0 IF   I I ) ADD,
 5    ELSE   I )+ TST,   THEN  NEXT
 6
 7 : IF   ' IF COMPILE  HERE 0 ,  IMMEDIATE ;
 8 : THEN   IMMEDIATE   HERE OVER -  DUP +  SWAP ! ;
 9 : ELSE   ' SKIP COMPILE  HERE 0 ,  IMMEDIATE  SWAP THEN ;
10
11 CODE 0=   0 CLR,   S ) TST,   =0 IF  0 INC,   THEN  PUT JMP,
12 CODE 0<   0 CLR,   S ) TST,   <0 IF  0 INC,   THEN  PUT JMP,
13 : <   - 0< ;       : >   SWAP < ;       ;S
14
15
16
10

 1 CODE DO   0 S )+ MOV,   R -) S )+ MOV,   R -) 0 MOV,   NEXT
 2 CODE LOOP   R ) INC,   1 R W) R ) CMP,
 3    HERE 100400 ( 0<) IF   I I ) MOV,   NEXT
 4    THEN   I )+ R )+ CMP,   R )+ TST,   NEXT
 5 CODE +LOOP   R ) S ) ADD,   0 R ) MOV,   0 1 R W) SUB,
 6    S ) 77777 # BIC,   0 S )+ ADD,   BR,
 7 CODE I   S -) R ) MOV,   NEXT
 8 CODE I'   S -) 1 R W) MOV,   NEXT
 9 : DO   ' DO COMPILE  HERE  IMMEDIATE ;
10 : LOOP   ' LOOP COMPILE  DUP + ,  IMMEDIATE ;
11 : +LOOP   ' +LOOP COMPILE   DUP + ,  IMMEDIATE ;
12 ;S
13
14
15
16
11

 1 ( NUMBER OUTPUT)   0 INTEGER OP   OCTAL
 2 CODE READY   0 S MOV,   0 -) 0 -) CMP,   0 -) 240 # MOV, B
 3    OP 0 MOV,   NEXT
 4 10 ,CODE DIGIT  HEX B130 , 33B2 , 35B4 , B736 , 39B8 , 4241 ,
 5 44C3 , C6C5 , OCTAL   0 CLR,   1 S )+ MOV,   S ) ROL,   1 ROL,
 6    BASE U) 0 DIV,   S -) 0 MOV,   0 1 MOV,   1 1 S) MOV,
 7    0 ROR, 1 ROR,   BASE U) 0 DIV,   0 ROL, S ) ROR, 0 ROR,
 8    1 S) 0 MOV,
 9    OP DEC,   OP I)  ' DIGIT 1 W) MOV, B   NEXT
10 CODE SIGN   0 OP MOV,   S )+ S )+ CMP,   S ) TST,   <0 IF
11       0 -) 55 # MOV, B   THEN
12    S ) 0 MOV,   0 S SUB,   0 4 W# ADD,   0 NEG,   PUSH JMP,
13
14 : DIGITS  1 0 DO DIGIT OVER ABS OVER + 0= +LOOP  SIGN ;
15 :     DUP ABS 0  READY DIGITS TYPE ;
16 : ?   @ . )       ;S
```

12

```
 1 ( PROGRAMMING VOCABULARY ) OCTAL
 2 DP 7 + USER MSB    : TXTB    MSB @ 2 / 10 + ;
 3 CODE -;    S -) R )+ MOV,    NEXT
 4 : ;:    CONSTANT  -; ,   ;CODE
 5    S -) E )+ MOV,    R -) I MOV,    I E ) MOV,    NEXT
 6 : MSG    ;:    COUNT TYPE ;    HERE 120001 , MSG SPACE
 7 : LINE    1 -   17 MIN   40 *   BLK @   BLOCK + ;
 8 : BLK-TO-MSB LINE TXTB 40 MOVE RELEASE ;
 9 HERE 106405 , 12 , 0 , MSG CR
10 : .0    0 SWAP 0  READY   6 0 DO   DIGIT LOOP   SIGN TYPE ;
11 : DUMP    OVER + SWAP DO   CR I .0   I 10 + I DO   I @ .0
12       LOOP   10 +LOOP ;
13 CODE -TRAILING    S ) ASL,   0 S ) MOV,   0 100 # ADD,   BEGIN
14    0 -) 240 # CMP, B   =0 NOT END   0 S ) SUB,   0 INC,   PUSH JMP,
15 : LIST BLK ! 21 1 DO CR I BLK-TO-MSB I BASE @ < IF SPACE
16    THEN   I .  TXTB -TRAILING TYPE LOOP 0 TXTB ! CR ;    ;S
```

13

```
 1 VOCABULARY EDITOR    EDITOR DEFINITIONS
 2 BASE @ OCTAL        : MOVE    40 MOVE ;
 3 : UPDATE    -200 @ 40 AND IF   CR   2 ERR-MSG   THEN UPDATE ;
 4 0 INTEGER TEXT    37 DP +!
 5 : STRING    120240 HERE !    HERE   DUP 1+ MOVE
 6    DELIM !   -1 IN +!   WORD   HERE 1+ TEXT MOVE ;
 7 : "    42 STRING ;        : (    251 STRING ;
 8 : HOLD    DUP LINE TEXT MOVE ;
 9
10 : T    CR SPACE SPACE   HOLD LINE DUP +   100 TYPE ;
11 : R    LINE   TEXT SWAP MOVE   UPDATE ;
12 : D    HOLD 20 SWAP DO   I 1 +
13    LINE DUP 40 -   MOVE   LOOP   UPDATE ;
14 : I    1+ DUP   17 DO   I LINE   DUP 40 + MOVE   -1 +LOOP   R ;
15 : COPY    SWAP BLOCK 1000 + !   UPDATE ;
16 CHAIN FORTH    FORTH DEFINITIONS    BASE !        ;S
```

14

```
 1 ( VOCABULARY CONTROL)
 2 DP 1 + USER CONTEXT        DP 3 + USER CURRENT
 3 : WHERE    CONTEXT @ @   COUNT 3 MIN TYPE   SPACE ;
 4
 5 : FORGET    IMMEDIATE  '   DUP 2 - @   37777 AND
 6    CONTEXT @   DUP CURRENT !   !   4 - DP ! ;
 7 : VOCABULARY    IMMEDIATE   HERE DUP 3 + CONSTANT ,    IMMEDIATE
 8    ;CODE    E ASR,   CONTEXT U) E MOV,   NEXT
 9
10 : CHAIN    IMMEDIATE  '   2 - CONTEXT @ 1+ ! ;
11 ;S
12
13
14
15
16
```

```
15

 1 ( TAPE I/O)       OCTAL
 2 371 CONSTANT BUF    ( INTERRUPT)  110 @ 112 !
 3 1001 INTEGER LR
 4 BUF  CODE FIX    2 + I) SWB,    NEXT
 5 CODE TAPE    -2527 S )+ MOV,   WAIT I) JMP,
 6 : READ    -2002 -2526 !  BUF @  DUP -2525 !  60103 TAPE
 7    -2525 @  OVER - 2 / LR !  2 / ;
 8 : REWIND    GET  60117 TAPE ;
 9 : END-FILE     60107 TAPE ;
10 : BACKSPACE    -1 -2526 !  60113 TAPE ;
11 : SKIP    GET  0 -2526 !  60111 TAPE ;
12 : ?TAPE    -2530 @ ;      : ?EOF  ?TAPE 40000 AND ;
13 : PARITY    ?TAPE 10000 AND ;
14 : WRITE    -2 * -2526 !  BUF @ -2525 !  60105 TAPE
15    PARITY IF    11 ERR-MSG THEN ;
16 DECIMAL       ;S
16

 1 ( TAPE UTILITY)
 2 0 INTEGER OFFSET
 3 : DUMP    1+ SWAP DO  I BLOCK @ IF  FIX 513 WRITE  FIX
 4    THEN LOOP  ;
 5 : RECORD    BUFFER READ FIX  512 + @ ;
 6 : SPACE    1 -   1 0 DO  DUP RECORD < +LOOP  DROP ;
 7 : SAVE    OFFSET @  BUF @ 2 /  512 + +!  UPDATE ;
 8 : RELOAD    SPACE  1 0 DO  SAVE RECORD  ?EOF +LOOP  FLUSH ;
 9 : BACKUP    1 456 DUMP  0 0 DUMP  END-FILE REWIND ;
10 ;S
11
12
13
14
15
16
17

 1 ;S
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
```

APPENDIX C

GENERAL REGISTER ADDRESSING

MODE	FORTH SYMBOL	FORTH NOMEN.	PDP-11/40 PROCESSOR HANDBOOK SYMBOL		PDP-11/40 PROCESSOR HANDBOOK NOMEN.	REGIS. HAS
0	R	D.A.	R	D.A.	(Register)	A
2	R)+	I.A.	(R)+	D.A.	(Auto-Inc.)	AA
4	R -)	I.A.	-(R)	D.A.	(Auto-Dec.)	AA
6	n R w)	I.A. (OFFSET)	n(R)	D.A.	(Index)	AA
1	R)	I.A.	(R)	I.A.	(Register Def.)	AA
3	R)+)	D.I.A.	@(R)+	I.A.	(Auto-Inc. Def.)	AAA
5	R -))	D.I.A.	@-(R)	I.A.	(Auto-Dec. Def.)	AAA
7	n R w))	D.I.A.	@n(R)	I.A.	(Index Def.)	AAA

P.C. REGISTER ADDRESSING

2	n#	I.A.	#n	D.A.	(Immediate)	*AA
3		D.I.A.	@#A	I.A.	(Absolute)	*AAA
6	(Not Used)	I.A.	A	D.A.	(Relative)	*AA
7	I)	D.I.A.	@A	I.A.	(Rel. Def.)	*AAA

NOTES:
(1) D.A. = Direct Addressing
(2) I.A. = Indirect Addressing
(3) D.I.A. = Double Indirect Addressing
(4) A = Address
AA = Address of Address
AAA = Address of Address of Address
(5) * = Drop off one A if JMP, Instruction Used