CALTECH FORTH

by MARTIN S. EWING

ewing@alum.mit.edu

Copyright (C) 1983 Martin S. Ewing

Copyright (C) 2006 Martin S. Ewing

C O N T E N T S

PREFACE

Forth is a computer language and programming style that produces efficient programs and allows programmers to work very productively. At the same time, it is unorthodox, resistant to standardization, and difficult to describe. In a profession (and hobby) filled with individualists, Forth almost encourages personal and non-standard computing.

With all its peculiarities and despite its uneven acceptance, Forth has grown from a specialized laboratory minicomputer system in the early 1970s to a widely popular language for both mini and microcomputers. Contemporary implementations range from the smallest 8-bit machines to 32-bit superminicomputers and even some mainframe systems. Forth systems have found significant application in commercial markets, but their peak acceptance (in numbers, at least) has been in the personal computer world.

This book addresses two needs. First, it provides a motivation and description for the basic Forth vocabulary, as embodied in the Forth-79 standard. The treatment is concise and directed toward readers with some familiarity with computing, but it should be accessible to newcomers with general mathematical background.

The second purpose of this work is to satisfy the curiosity and needs of programmers who have developed experience in Forth but who seek a more complete understanding of the internal structure of Forth. With this material, the Forth programmer should be able to adapt his system to new requirements, or to recode it

for different processors.

This description of Forth is based on ten years of experience at the Owens Valley Radio Observatory (OVRO) and the Jet Propulsion Laboratory (JPL) of the California Institute of Technology. Based on a presentation of Charles Moore at the U.S. National Radio Astronomy Observatory in 1972 and Moore's assembler-coded IBM 360 version, we undertook a series of implementations that began with the DEC PDP-10 and the SDS 920. The PDP-10 was a convenient timesharing development system, while SDS 920 Forth successfully controlled the OVRO 40-meter radio telescope for many years.

H. W. Hammond continued with our first DEC PDP-11 system, which still provides control and data collection for the dual 27-meter radio interferometer at OVRO. Yet another generation of Forth, running under the RT-11 operating system, supports an OVRO/JPL processor for Very Long Baseline Interferometry (VLBI), accepting 20 million samples per second of digital data originating at observatories around the world.

Further developments at OVRO have led to a distributed network of more than 7 DEC LSI-11 computers running self-contained Forth systems under the direction of a central PDP-11 whose Forth system runs under the RSX-11/M operating system. Currently, JPL and OVRO are readying a DEC VAX-11/VMS Forth program to control a new generation VLBI processor handling more than 800 million samples per second.

A version of Forth for the PDP-11 running under the RT11 operating system is available from the Digital Equipment Users Society (Maynard, Massachusetts). This release, numbered 11-232,

incorporates some, but not all of the features described in this book.

While developments were occuring at Caltech, Forth was evolving in various directions in other user communities. Vocabularies naturally tended to diverge as the larger Forth community shifted to 8- and 16-bit microcomputers, especially "personal" computers, while Caltech worked with Forth under operating systems and running on larger processors, such as the 32-bit VAX-11. With the publication of the Forth-79 standard, however, there has been a convergence of vocabularies and syntax, even though the language retains the flexibility that allows it to expand and adapt to new problems and computing environments.

I would like to thank H. W. Hammond, D. H. Rogstad, and J. L. Vavrus who have been responsible for many of the developments in PDP-11 and VAX-11 versions of Forth.

Martin S. Ewing

Altadena, California

May, 1983

CHAPTER 1
INTRODUCTION

1.1 BEGINNINGS.

In the early 1970s, Charles Moore revealed a new and iconoclastic approach to programming computers. The environment in which Moore worked was that of a national scientific laboratory (the U.S. National Radio Astronomy Observatory) that was beginning to apply early 16-bit minicomputers for data collection and instrument control.

Programming the new minicomputers was an arduous process, styled after earlier experience with second- and third-generation mainframes (IBM 7040 and 360 series). Since the new small systems could not usefully support Fortran or Algol compilers (which at that time were largely unsuitable for real-time operations), they had to be programmed with machine language assemblers. Input would be on punched cards or paper tape; the operating system, if any, would reside on magnetic tape. Programs would often be assembled on mainframe computers and transported on tape or cards to the minicomputer. This process was so laborious, and debugging tools were so limited, that minicomputer simulator programs running on mainframes were sometimes the most efficient way to check out new programs.

Moore understood that there should be a better way to program small machines. He developed a unique set of tools to permit efficient programming, but, more important, a new style of working, which brought the programmer into intimate contact with his object code and the machine on which it ran.

He named his new system Forth. The name stands for "fourth

generation" software; one letter had to be removed to fit a 5-character field in the IBM 1130 computer that was used for early development.

Forth has been refined and transported to nearly all types of small computers. The language has been adopted in many environments besides the scientific laboratory: microprocessor development systems for industry, and personal computers are two major examples.

1.2 GENERAL CHARACTERISTICS.

Forth exists in numerous varying implementations, but a number of distinctive features are common to most Forth systems.

1. Interaction. Forth is at heart an interactive system. You prepare programs through an on-line editor and are able to compile them rapidly. This approach was almost revolutionary in Forth's early days. Even now, the Forth programmer spends much less time going through the mechanics of editing, compiling, linking, and testing new code than does a Fortran programmer.

2. Incremental Compilation and Assembly. Forth's basic units of code, "words," tend to have short definitions - a few lines each. As you enter a new word definition, it is natural to make an immediate test of its operation. Program modularity is encouraged because words tend to be simple and logically well-defined; they can be tested exhaustively before being used in higher-level constructs.

3. Reverse Polish Notation. The normal location of program input and output is a push-down parameter stack. For keyboard

input, you must type parameters before a command word that operates on them. To users accustomed to other high-level languages, this makes Forth programs somewhat difficult to read. In practice, however, the transition to Forth's parameter ordering is no more difficult thatn switching from a Texas Instruments (algebraic) pocket calculator to a Hewlett-Packard (RPN) unit. The push-down stack simplifies communication and facilitates program re-entrancy, allowing code to be shared between multiple tasks in a real-time or timesharing environment.

4. Simple Logical Structure. Only the most basic program branches and loops are provided in Forth. It is difficult and unnatural to write Forth that is not well-structured. (Unfortunately, this does not mean that all Forth programs are clear!)

5. Extensibility. Forth has built-in capabilies for extension to new data and operation types. List processing and data base management are examples of extensions that are possible with this technique.

6. Mixed High- and Low-Level Programming. Forth words can be defined either as combinations of basic Forth vocabulary words (the "high-level" approach) or directly as machine-language instructions ("low-level"). In a natural way, you can shift between compact (and readable) high-level code and very fast machine code.

7. Machine Independence. High-level Forth programs adhering to the Forth-79 standard can easily be transported between different computer types.

1.3 DEFINITION AND STANDARDS.

Forth is a very personal and malleable language that has historically resisted being formalized with the precision of some high-level languages, such as Pascal or Ada. In the hands of an expert, Forth can easily be recast into forms which emphasize one or another desireable attribute, but which lose compatibility with "mainstream" Forth.

Despite its susceptability to rapid evolution, Forth has been regularized with some success through the work of the "Forth Standards Team." This private group published the Forth-79 document which forms the basis of the notation and vocabulary used in this book.

1.4 ORGANIZATION OF THE BOOK

Chapter 2 is an introduction for the new user of Forth. That chapter and the vocabulary lists of Chapter 4 should provide you with enough information to begin programming at a Forth terminal.

Chapter 3 provides more detailed descriptions of the internal mechanisms of Forth; the presentation assumes some practical knowledge of Forth. This chapter should help you if you develop or maintain Forth systems.

Chapter 4 contains the standard Forth vocabulary, Forth-79, and additions that have proven useful at Caltech-OVRO and JPL.

Chapter 5 treats the special problems of implementing Forth in large-memory systems, including the new 16-bit microcomputers, such as the Motorola 68000, as well as the DEC VAX-11 series.

Appendix A provides details of a PDP-11 implementation, including a PDP-11 Forth assembler. Appendix B is an annotated bibliography of the Forth literature.

CHAPTER 2

FORTH OVERVIEW

2.1 WORDS AND THE DICTIONARY.

The central element of the Forth system is the "word". A Forth word is like a subroutine or procedure in other languages; executing, or calling, a word causes a definite sequence of actions to be performed. The reason for calling a Forth routine a "word" is that it nearly always has a name that is known to the keyboard interpreter: it can be executed simply by typing its name. Thus Forth words are equivalent to words of text that you can type on the keyboard.

NOTE: You must be careful to distinguish a "Forth word", which is to be executed like a subroutine, from a "memory word", which is a unit of storage (e.g., 16 bits).

Words are defined in the Forth "dictionary", which, like ordinary dictionaries, is a table of word-names and their definitions. Two types of definitions occur in the Forth dictionary. Words may be defined in terms of other words that are defined earlier, or words may be defined by a sequence of machine language instructions. Ultimately, of course, all Forth words must resolve into machine instructions.

As a Forth user, you type in words or, more precisely, text strings or "tokens" to your terminal. Forth permits a very general and free-form input. With few exceptions, any combina-

tion of letters, numbers, or other characters can be used to name a word. One character, normally "blank" or "space", is reserved to separate tokens. A few other characters are reserved to let you correct errors in typing. Under DEC operating systems "del" or "rubout" lets you retract the last character you typed, and "CTRL-U" or "^U" cancels the entire current line you are typing.

One rule for recognizing Forth word names may be unfamiliar. Words are distinguished on the basis of their first N characters and their total length. In many Forth variations, N=3, while in the Caltech-OVRO systems discussed here, N=4. The number of characters to recognize is a tradeoff between memory savings and freedom in choosing names. Examples of recognizable and distinguishable Forth word names are presented in Fig. 2.1.

```
1A?@XX.:        SOME-ARE-LONG
X               #                   (recognizable words)
FOURTEEN        SUM

ABCDEFG         (equivalent -- not distinguishable)
ABCDXXX

ABCDEFG         (not equivalent -- distinguishable)
ABCDEFGH
```

Fig. 2.1 Recognition and Distinction of Forth words.

If you type in a token (sequence of characters) that can't be found in the dictionary, Forth sees if it makes sense as a number. If so, the token is converted from ASCII to binary and pushed on the stack (discussed below). If you type a string that is not in the dictionary and is not a number, Forth issues its

standard error message -- a question mark.

2.2 THE STACK.

Numbers and other data are normally handled through the Forth "stack". This is a so-called "push-down" stack. Such a stack is a way to store data such that the most recently stored items are immediately accessible. New data is pushed down on top of older items. When an item is no longer required, it is "popped" from the top of the stack, making older items available again. In other words, the push-down stack is a last-in first-out queue.

The purpose of the stack is to provide you with an efficient means of handling data and intermediate results in the course of a calculation. Labelled variables to hold intermediate data are not required in most cases. Since the space used by the stack is shared by nearly all Forth words, there is a considerable saving in memory.

Most Forth words operate on input data you supply on the stack, pop the input data, and push the results onto the stack. For simplicity, the Forth convention is that you must type the arguments of a function (Forth word) before you type the word itself; i.e., you must give commands in "reverse Polish notation". As an example, the algebraic expression

(1 + 2) * (3 + 4)

may be written

1 2 + 3 4 + *.

2.3 BLOCK STORAGE.

In most practical applications Forth requires an auxiliary mass-storage device. Various devices such as floppy disks, hard disks, or magnetic tape, may be used, but some random-access technique is required.

The storage device is divided into fixed-length "blocks", normally 512 words = 1024 bytes long. These blocks may be used as a sort of "virtual memory", i.e., you may store data in blocks when you don't have enough room in main memory. Blocks are suitable for holding large amounts of business or experimental data, for example. They are also used for the Forth system itself: the Forth binary object program and the Forth source (text) for loading the standard system and for users' applications. When a Forth block is used to hold text, it is called a "screen."

Forth handles its transactions with the block storage device in a simple and device-independent way. Blocks are simply numbered sequentially from 0 to some high number. Two buffers in main memory hold the last two blocks you have used. In order to retrieve a new block, you type BLOCK*, which takes the number you have put on the top of the stack as a block number, reads the block into a buffer, and returns the address of that buffer on top of the stack. If there are multiple disk drives in a Forth system, they are normally treated together as a single unit. Floppy disk drive "0", for example, might be accessed as blocks 0 - 300, while drive "1" might be 1000 - 1300.

---

* Forth words written in this text will be written in capitals and underscored.

If you want to change the data in a block, you type UPDATE after BLOCK. Then, before the buffer holding your block is released for a new BLOCK command, it will be rewritten to block storage. You can type FLUSH to rewrite updated blocks explicitly.

2.4 DEFINING NEW WORDS.

The "standard" Forth system has around 200 words defined in its dictionary. These provide the functions most commonly needed in useful application programs. "Writing" a Forth program actually consists of defining new Forth words, which draw on the old vocabulary, and which in turn may be used to define even more complex applications.

Forth provides a number of ways of defining new words. The language even gives you ways of defining words that define words. (It is an extensible language.)

The word CODE permits you to define words whose actions are expressed directly in machine- or assembly-language (terms used synonymously). CODE words are clearly machine-dependent, but they give you the means to get maximum execution speed. If the tightest loops of your program are in CODE words, you may find that your Forth program is as fast as a pure assembler program.

Figure 2.1 shows a typical screen from a PDP-11 Forth system that contains CODE definitions. A very simple example is the one

```
   .........1.........2.........3.........4.........5.........6....
   12345678901234567890123456789012345678901234567890123456789012345678901234

 1 ( SOME PDP-11 CODE DEFINITIONS)    ASSEMBLER
 2 CODE + S ) S )+ ADD, NEXT,
 3 CODE - S ) S )+ SUB, NEXT,
 4 CODE @ S -) S @)+ MOV, NEXT,
 5 CODE C@ S ) \ 0 S @I) MOV, S ) 177400 # BIC, NEXT,
 6 CODE ! T S )+ MOV, T ) S )+ MOV, NEXT,
 7 CODE C! T S )+ MOV, T ) \ S ) MOV, S )+ TST, NEXT,
 8 CODE OR S ) S )+ BIS, NEXT,
 9 CODE AND S ) COM, S ) S )+ BIC, NEXT,
10 CODE MINUS S ) NEG, NEXT,
11 CODE OVER S -) 2 S I) MOV, NEXT,
12 CODE HERE S -) DP P MOV, NEXT,
13 CODE SOB T S )+ MOV, 6 # T ASH, T 77000 # ADD,
14   S ) 2 S I) SUB, S ) 177700 # BIC, T S )+ ADD,
15   S ) T MOV, NEXT,
16                                        END-CODE  ;S
```

Figure 2.1. Typical CODE Definitions in a screen.

shown in line 2, for the word +. This definition consists of only one machine instruction (ADD) with source and destination parameters that tell the PDP-11 to add the top two stack words and leave the result on the stack. The notation for machine instructions and arguments is peculiar to your particular computer. (In fact, there is little standardization of assembler syntax even among implementations of Forth on the PDP-11.) The basic ("kernel") definitions of most Forth systems will be defined in CODE words.

With the word : (colon) you can define Forth words in terms of other Forth words. Colon definitions are much better standardized among Forth implementations and are relatively machine independent. They do not have the full speed of a CODE word, but they are much easier to write. Colon words often use less memory than CODE words.

Most words that are referenced (functions that are invoked)

in a : definition take one memory word. This memory word holds a pointer to (address of) the Forth word that is to be invoked. The computer operates in an interpretive mode while a : word is being executed: a sequence of pointers controls the computer. The interpreter overhead is quite tolerable in most cases -- ranging from 2 to 8 microseconds for the PDP-11/40 version. These figures are comparable to and often somewhat better than equivalent subroutine calls in assembler language.

Figure 2.2 gives an example of the use of colon definitions. In fact, this one screen is a complete text editor for Forth screens, showing how succinctly it is possible to write useful applications programs in Forth. The standard text editor is described more fully in Section 2.9.

```
   .........1.........2.........3.........4.........5.........6....
   12345678901234567890123456789012345678901234567890123456789012345678901234
 1 ( FORTH STANDARD TEXT EDITOR)
 2 VOCABULARY EDIT   EDIT DEFINITIONS
 3 : EDITOR ;  : FORGET SAVE-BUFFERS FORGET ;   BASE @ OCTAL
 4 VARIABLE TEXT 76 ALLOT
 5 : BLANKIT SPACES @ OVER ! DUP 2+ 37 MOVE ;
 6 : STRING TEXT BLANKIT DELIM ! WORD HERE COUNT TEXT SWAP CMOVE ;
 7 : "  42 STRING ;
 8 : (( 51 STRING ;
 9 : HOLD DUP LINE TEXT 40 MOVE ;
10 : T HOLD LINE 100 -TRAILING 2 SPACES TYPE ;
11 : R LINE TEXT SWAP 40 MOVE UPDATE ;
12 : D HOLD DUP 20 < IF 20 SWAP DO I 1+ LINE DUP 100 - 40 MOVE
13   LOOP ELSE DROP THEN 20 LINE BLANKIT UPDATE ;
14 : I DUP 17 DO I LINE DUP 100 + 40 MOVE -1 +LOOP 1+ R ;
15 : BT SCR @ DUP . CR LIST ;
16                                              BASE ! ;S
```

Figure 2.2. Standard Forth Text Editor.

Another useful Forth colon definition is .:

```
: . CONVERT COUNT TYPE ;
```

Here the word . (period) is defined as the sequence CONVERT, COUNT, TYPE, where these words are assumed present in the dictionary when you type in the example. Semicolon (;) is a word with the special meaning: "end : definition".

There are other, more specialized, ways to define Forth words. Numeric constants can be defined with the word CONSTANT. For example,

```
31415 CONSTANT PI-TIMES-10000
```

defines the Forth word PI-TIMES-10000. Whenever you type this word, the constant value 31415 will be pushed on the stack.

Often you find that it is awkward to have all your data on the stack at once. You can store data in single named memory words. The Forth word VARIABLE lets you reserve and name such locations. Type

```
VARIABLE Q
```

to define the Forth word Q. When you type Q, the address of the storage location corresponding to Q is pushed on the stack. This storage area is two bytes long.

If you need to reserve a multiword block of memory for data, you can use ALLOT:

```
VARIABLE DATA 48 ALLOT
```

This example reserves 50 bytes (including two from the definition of VARIABLE) named "DATA". When you type "DATA", you get back the address of the first memory word. You can add an index to the first address if you want the address of a later word.

2.5 STORING AND RETRIEVING DATA IN MEMORY.

The word @ (called "fetch") is provided so you can "read out" data from any address. You type

<address> @

where <address> is any valid memory address to retrieve the data stored there. (The data replaces <address> on the stack.) Thus type

@ @

to get the integer in variable @.

To "store" data from the stack into a location in memory you type

<value> <address> !

Here <value> is stored in location <address>. More concretely,

148 @ !

stores a new value (148) in variable @. (Note that both "148" and @ push numbers on the stack. The "store" word [!] stores the data away and then pops both input quantitites from the stack.)

Another little program might run

```
VARIABLE ABC 1 ABC !
ABC @ MINUS ABC !
```

In the first line ABC is defined (VARIABLE ABC) and set to the value 1. In the second, the address of the integer (ABC) is placed on the stack, the value at that address is fetched (@), the value is negated (MINUS), the address is again placed on the stack, (ABC), and the negated value is stored back in the integer location (!). This is a slow but feasible way to negate an integer.

2.6 CONTROLLING FORTH -- THE TEXT INTERPRETER.

You normally control a Forth computer from your terminal. The system is idle and listening for anything from the keyboard until you type in a complete line. When Forth gets a full line (ended with "return"), it attempts to execute the words (or convert the numbers) you have typed.

Many times you will want to avoid typing long, standard, or repetitive sequences of words. For example, once you have debugged a new word, you don't want to have to type it in again. The Forth text editor (described below) lets you store away the program in source text form in a block (screen). To define the word, or collection of words, in the future all you need to do is type

<screen #> LOAD

LOAD is a word that temporarily redirects Forth's text interpreter away from your terminal to the screen number you specify. Almost any user commands (Forth words) you could type directly can be executed from a block via LOAD.

Each screen to be loaded may end with the special word ;S, which restores the text interpreter to the source previously in effect. If ;S is not found, interpretation of the screen ends after the last line. Note that LOADs may be nested; a block to be loaded may contain LOADs itself.

A screen might contain the following text:

```
2 2 + .
13 LOAD
```

If you were to load this screen, Forth's response would be to convert and push "2" on the stack (twice), add those numbers, and

type the result (4) on the typewriter. After this, screen number 13 is loaded (with whatever commands are contained there).

2.7 TERMINAL OUTPUT.

Output from Forth normally comes to your terminal. A few basic words will suffice for many applications. You can type a number from the stack with the word . (period). Question mark ? uses an address on the stack and types the number that lies at that address.

The base used for numeric input and output is determined by the variable BASE. BASE may have any value from 2 through 10. Some implementations allow base 16 as well. The special words OCTAL, DECIMAL, and HEX let you set BASE automatically to 8, 10, or 16, respectively. The default number base is normally decimal, but you should check this on your system.

For typing arbitrary strings of characters you may use TYPE. TYPE takes two numbers on the stack:

<pointer> <character count> TYPE

In most Forth systems, a pointer for a character string is simply the byte address of the beginning of the string. Beginning with the specified character, TYPE puts out sequential characters until the count is satisfied.

In some "traditional" Forth systems, terminal input and output save space by using the same buffer in main memory. To avoid problems in these systems you should use only one output word on a command line; you should place an output word at the end of the command. For example

123 . 456 .

typed in as one line will give you only "123" on your terminal. This is because the part of the command line containing "456 ." is obliterated when Forth writes "123" into the buffer for typing.

2.8 CONDITIONAL BRANCHES.

Forth gives you several means to direct the flow of execution. The methods described here work only within : definitions; other similar words are available in the Forth assemblers.

The simplest conditional branch is specified by the words BEGIN and UNTIL. Consider the following example:

: EXAMPLE 1 BEGIN 1 - DUP UNTIL DROP ;

BEGIN signals the beginning of a loop. When the program gets to the UNTIL (during execution of EXAMPLE), control will return to the BEGIN if and only if the current stack value is zero. The value is popped after testing just as most Forth words pop their input arguments.

This is what happens when you execute EXAMPLE: The value 1 is pushed on the stack and the program enters the loop. Again, 1 is pushed; then subtracted from 1 to leave 0. The 0 value is duplicated (DUP) and tested by UNTIL; then the duplicated value is popped from the stack. Since UNTIL found a 0, control returns to BEGIN; 1 is again subtracted, leaving -1. UNTIL finds -1 and control passes through to DROP where the remaining -1 value is popped. Control returns to the calling word, e.g., to the interpreter if you were typing.

The BEGIN - UNTIL construction is useful for program loops where the loop termination condition can conveniently be expressed by leaving a zero or non-zero value on the stack.

A variant of BEGIN - UNTIL is useful for situations in which the termination condition is generated in the body of the loop. You may program the following:

. . . BEGIN . . . <condition> WHILE . . . REPEAT . . .

If <condition> produces a non-zero result on the stack, execution continues with the code between WHILE and REPEAT, and the loop is repeated from BEGIN. If <condition> is zero, the remaining loop code is skipped, and excution continues following REPEAT.

A looping facility more like the Fortran "do-loop" is provided through the words DO, LOOP, and +LOOP. Another example:

: EX2 5 0 DO I . LOOP ;

When you execute EX2, the constants 5 and 0 are pushed on the stack. DO takes these numbers to be the limit and initial index for the loop, respectively. The limit and index disappear from the stack and are placed on a hidden internal return stack stack. Control passes into the loop. The word I retrieves the current loop index value and pushes it on the stack. The value is typed (and popped) by . . LOOP increments the index value by 1, then tests it against the limit. If the new index value is still less than the limit, control returns to the DO (i.e., to the point just after DO). Otherwise the limit and index are popped from the internal stack and control passes out of the loop.

Thus when you execute EX2, you get

0 1 2 3 4

typed on your terminal.

NOTE: The index of a DO stops one short of the limit. The limit gives the number of times the loop is executed if the initial index is 0. The range of a loop is always executed at least once.

Words J and K are defined like I to let you retrieve indices in nested DO loops. In the word EX3, defined as

: EX3 5 3 DO 3 1 DO 1 -1 DO I . J . K . CR LOOP LOOP LOOP ;

I retrieves the innermost index, J the next outer, and K the outermost; CR causes a carriage return. EX3 should give you the following output. (Again, each index stops one short of its limit.)

```
-1 1 3
 0 1 3
-1 2 3
 0 2 3
-1 1 4
 0 1 4
-1 2 4
 0 2 4
```

If you need an increment other than +1 in your loop, you can use +LOOP. Here is an example:

: EX4 0 5 DO I . -1 +LOOP ;

Here again 0 is the limit and 5 the initial index for the loop. EX4 proceeds like EX2, except that +LOOP takes the current stack value to be the loop increment.

+LOOP tests the index in a way that depends on the sign of the increment; this is a historical peculiarity likely to change in future language revisions. For a positive increment the test

is the same as for LOOP; when the increment is negative, the loop will run once with the index equal to the limit. Thus the output of EX4 is

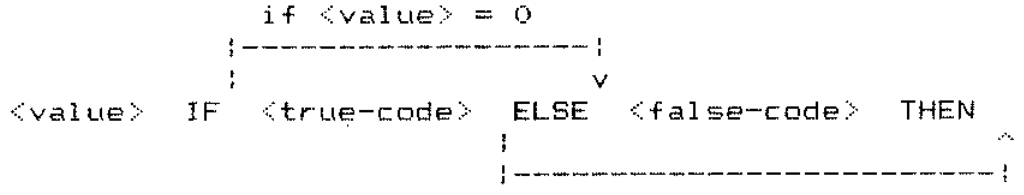
5 4 3 2 1 0 .

Variable increments are also possible with +LOOP: whatever word is left on the stack when +LOOP is executed will be used for the increment.

The general conditional branch in Forth will be familiar to users of Algol or PL/1: an IF - THEN - ELSE construction. Assume that TRUE-CLAUSE and FALSE-CLAUSE are words that have previously been defined; then define EX5 as follows:

: EX5 IF TRUE-CLAUSE ELSE FALSE-CLAUSE THEN ;

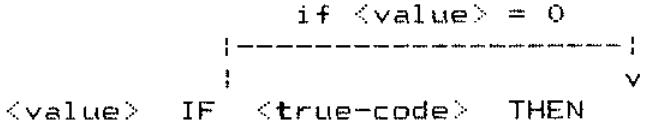
When you run EX5, IF tests (and pops) the current stack value; if it is non-zero, TRUE-CLAUSE runs, otherwise FALSE-CLAUSE runs. In general, control flows as shown in the following line -



In some cases you only need to test for a "true" condition, e.g.,

: EX6 IF TRUE-CLAUSE THEN ;

Here TRUE-CLAUSE is run if and only if the current stack value is non-zero ("true"). The logical diagram is



A more realistic example of a program using conditional branches might look like this:

: FUNCTION DUP 0 < IF MINUS ELSE DROP 0 THEN DUP DUP * * ;

FUNCTION takes the current stack value (say x) as input and returns

$$\begin{cases} 0 & \text{if } x \geq 0, \text{ and} \\ -x^3 & \text{if } x < 0. \end{cases}$$ (Fortran notation)

Let us briefly explain what happens in FUNCTION. The word < is a binary function that returns 1 if the next-to-current stack value is less than the current value; otherwise it returns 0. MINUS replaces the current stack value with its negative, and * returns the product of the top two values.

When you executed FUNCTION, the input value (x) is duplicated (DUP) and tested against 0 (0 <). If $x < 0$, < returns 1, and IF will transfer control to the true-clause (MINUS). The current stack value at this time will be x, since both < and IF will have popped the stack. MINUS then negates x, and control bypasses the ELSE clause (the false-clause) and resumes following THEN. The current stack value ($-x$) is then cubed (DUP DUP * *), and FUNCTION is done.

On the other hand, if x were >= 0, IF would transfer to the false-clause (DROP 0). Here x is popped and replaced with 0. Control then passes over THEN, 0 is cubed, leaving 0 on the stack. Like Fortran and other common languages, Forth lets you nest BEGIN - UNTILs, DO - LOOPs, IF - THENs, etc., provided that the range of a nested loop or branch lies strictly within the

range of all the branches and loops that contain it. For example,

```
... DO ... IF ... IF ... THEN ... ELSE ... THEN ... LOOP
N.L.=1      2      3       3        2       2        1
```

is a valid ordering. (Note the indication of nesting levels.) The following is invalid:

... DO ... IF ... LOOP ... THEN ...

In this case the range if the IF-THEN does not lie within the range of the DO-LOOP.

Unlike Fortran, Forth does not let you "GO TO" an arbitrary location with a statement label (number). In general, IF is the only way you have to make a forward jump. The loss is not serious if you take care to "structure" your programs -- it turns out that most "GO TOs" are unnecessary.

2.9 THE EDITOR.

In preceding sections, the Forth block storage scheme was introduced. A major use for block storage is to hold text data, called screens, of which Forth source code is an example. The way you can enter and modify text in Forth screens is with a Forth text editor.

Many different Forth editors have been written. The basic Forth editor (EDIT), shown in Figure 2.2, is common to most Forth systems; it is very compact but gives you everything you need to modify text a line at a time. The extended editor used at Caltech (XED) includes flexible string manipulations and lets you search for, insert, or delete text strings anywhere in a block.

Most computer systems now support either fast (9,600 baud) serial terminals, or memory-mapped display terminals. Such displays enable you to use "screen editors" that show an entire screen at a time and immediately update the full screen whenever you change any part. A flashing cursor indicates where you may enter new text. By typing control keys you can reposition the cursor at any place on the screen. Screen editors have been written in Forth, but a detailed description is beyond the scope of this book.

The standard block length for Forth systems is 512 16-bit words = 1024 8-bit characters. For use as a text screen a block is conventionally divided into 16 lines of 64 characters. Division of text into lines is only for convenient display; as far as the Forth interpreter is concerned, the 64th character of a line is immediately adjacent to the first character of the next line.

The variable SCR is used to hold the Forth block to be edited, thus to edit block 35, we type

35 SCR !.

If you want to list the entire block 35, you type

35 LIST.

As a side effect LIST sets SCR to equal the specified block. To list blocks 35 through 40 at once, you type

35 40 SHOW.

To list just one line (say the 5th) of the current block, you type

5 I.

You can delete the second line by typing

2 D.

D deletes the line by moving up all the lines following the one you delete. The last line (16) should be filled with blanks.

To enter new text into a block you first need the special words " or (to put a line of text into an internal buffer. Quote (") enters all text up to the next quote into the buffer. Left parenthesis (() does the same except that the text line must be terminated with a right parenthesis ()). Thus

" THIS IS A TEXT STRING"

and

(THIS IS A TEXT STRING)

both place "THIS IS A TEXT STRING" (without quotation marks) into the buffer. If needed, blanks are added to the right to make 64 characters. Note that, like any words, " and (must have a blank following in the input. The text string to go into the buffer begins after this necessary blank. The " or) that terminates the text is just a "delimiter"; it needs no preceding blank.

Once you have got the new text entered in the buffer with " or (, you may use it to replace (R) an existing line or to insert (I) following an existing line. To replace line 3 of block 10 with "FOO BAR", you could type

10 SCR ! " FOO BAR" 3 R.

To insert 'THIS IS A QUOTE: "' after line 12 of block 10 you can type

10 SCR ! (THIS IS A QUOTE: ") 12 I.

(Here you must use the (-) construction to enter a string containing a quote.) I inserts the line following line 12 by first moving lines 13 through 15 down one. The old line 16 is lost.

After a T or D operation the line that was typed or deleted is automatically copied into the internal buffer, ready for a possible R or I. For example

14 D 2 I

has the effect of moving line 14 to line 3, with lines 4 - 13 moving down one.

After an editing session you should be careful that the updated blocks are actually written back into block storage. Forth usually takes care of this correctly, but you still may want to type SAVE-BUFFERS to make certain. You get rid of the editor by typing FORGET EDITOR, i.e., the editor's dictionary space is reclaimed.

CHAPTER 3

THE STRUCTURE OF FORTH.

This chapter more thoroughly describes the Forth system. The reader should be familiar with the preceding chapters and should have had a significant amount of "hands-on" experience with a Forth computer. The presentation is intended for implementers and systems programmers, but it should be useful to more casual programmers who want to know how to make the most efficient use of Forth.

3.1 GENERAL REMARKS.

It is important to stress that Forth is a complete programming system, not merely a language. In some versions, Forth provides all the software functions of the computer on which it is run. This includes preparation of programs (text editing), compilation (or assembly) of programs, debugging and input/output operations through direct-access or typewriter devices. In other versions of Forth, including several Caltech-OVRO systems, Forth runs as a process or task under a standard operating system. The operating system provides standard interfaces for I/O, scheduling, and memory management.

Forth has been designed around certain basic concepts which serve to distinguish it from other systems. These include the dictionary, the address interpreter, and the technique of compilation. Less crucial but still distinctive features are block I/O, the parameter stack, the text interpreter, and the assembly technique.

Such features do not really define a language. There is a Forth language, however. In this language concrete words are defined, such as +, BLOCK, and DO. In this light, Forth may be compared with other programming languages like Fortran, Basic, or Algol. The Forth language could in principle be implemented with a compiler like a Fortran compiler, and run like Fortran in a batch processor. But Forth's distinctive incremental compile/debug approach is much more productive and is well suited to the way real minicomputers are used.

3.2 THE STACKS.

Modern minicomputers generally have very flexible addressing methods; these are heavily used in Forth systems. An important example is the use of push-down stacks. Most Forth systems use two stacks extensively: a parameter stack and a return stack.

The parameter stack, often simply called "the stack", is the one most visible to the applications programmer. It is used as the primary vehicle for input and output data for Forth words. Usually data types such as integer, double precision integer, and floating point are intermixed freely on the stack. Context usually suffices to distinguish types.

The push-down stack accounts for the "unnatural" reverse Polish notation of Forth. That is, all parameters must be placed on the stack before they are operated upon. Thus the algebraic expression

B2 - 4*A*C

could be written in Forth as

B B * 4 A * C * -.

The advantages derived from the stack technique include simplicity in the compiler, easy addressing at execution time, economy of main storage, and ease of providing reentrant code for real-time systems. Against such advantages must be counted the inconvenience, especially for new Forth programmers, of placing all the arguments before the operators.

The parameter stack is commonly implemented beginning near the high end of main memory and growing downward toward the dictionary, which grows upward (see Fig. 3.1).

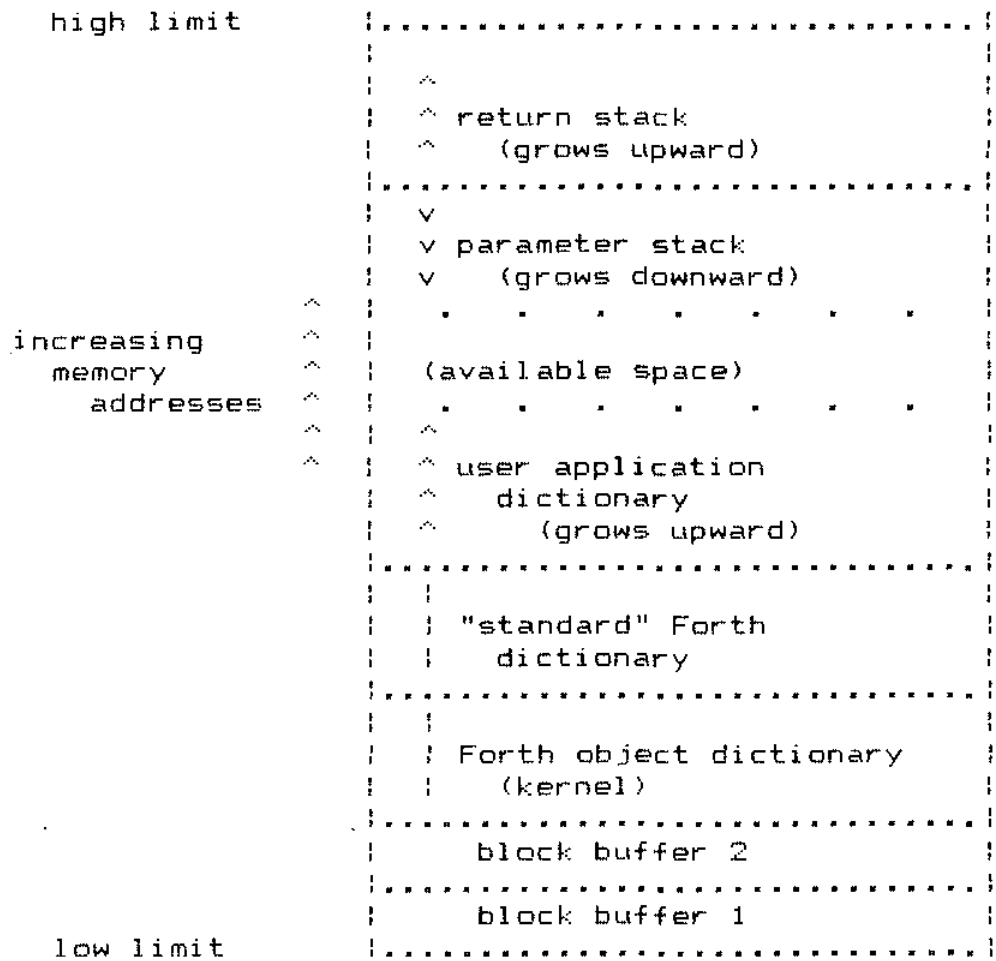


Figure 3.1. Memory layout of a typical Forth system.

The "return stack" is separate from the parameter stack; it is used primarily for the execution of :-words; this application is described later in this chapter.

Various other information may be placed on the return stack. This stack is normally used to hold indices and limits for DO loops. Using the return stack for this purpose, the implementer avoids having the loop information on the parameter stack where it might lie in the way of data for other calculations.

In the same vein, the word >R is defined to take one word from the parameter stack and save it on the return stack. R> has

the reverse effect.

3.3 THE DICTIONARY.

The Forth dictionary is the heart of the system. All programs written in Forth appear as words or collections of words in the dictionary. The organization of the dictionary and the details of dictionary entries differ between various Forth implementations. In this Section we will principally describe the Caltech-OVRO Forth for the Digital Equipment Corporation PDP-11.

3.3.1 Branch Structure.

Forth dictionaries are organized as threaded lists each of whose elements is the definition of a word. The simplest list structure would have a single linear thread connecting the Forth words in the order they have been defined. Few Forth systems use this simple method, since efficiency in search time and memory space can be gained rather easily.

The dictionary list structure developed for the Caltech-OVRO PDP-11 systems is sketched in Fig. 3.2.

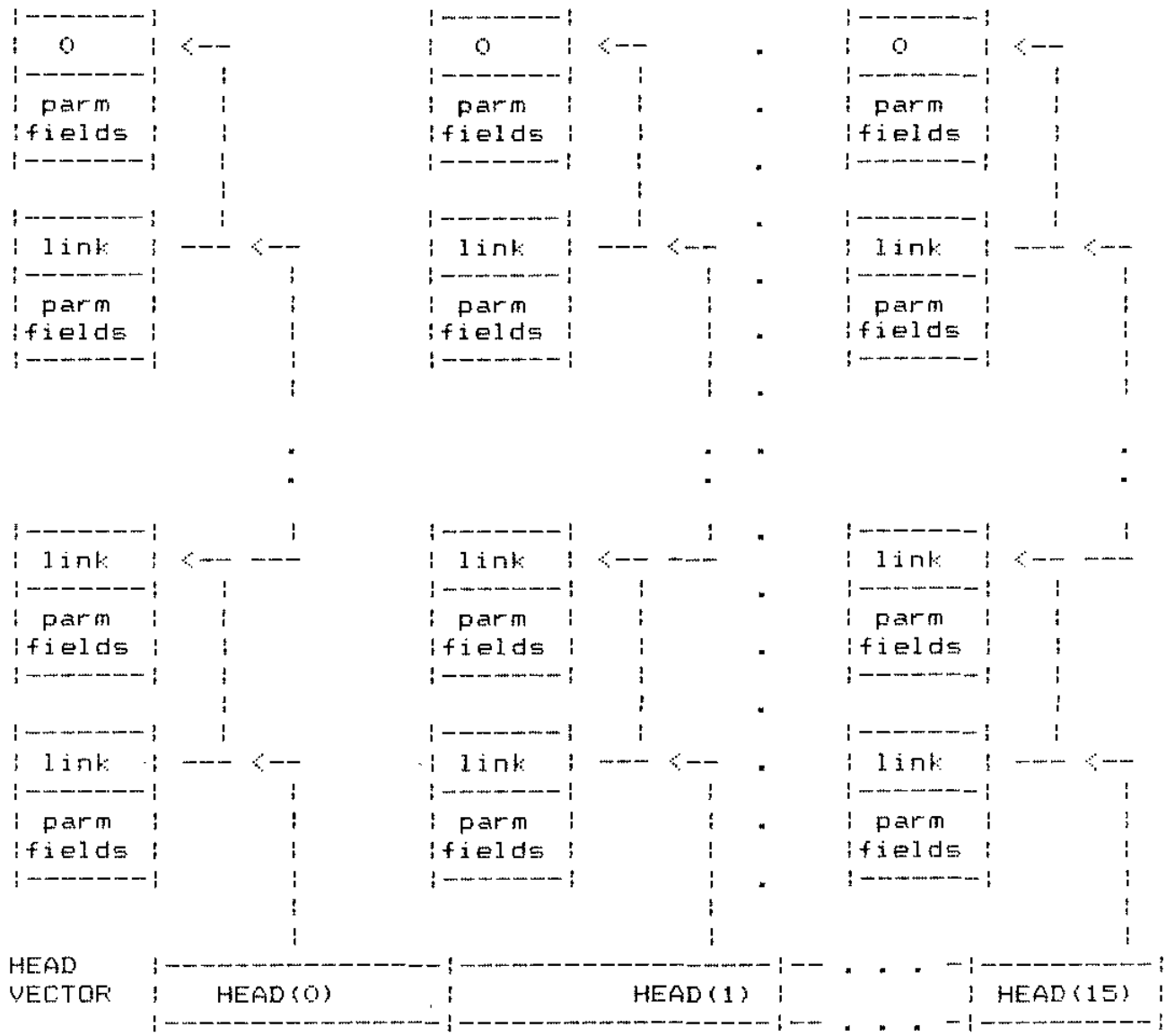


Fig. 3.2 Dictionary Organization.

The dictionary is split into 16 threads or branches. The branch in which a word appears is a function of its name. Thus to find a particular word by name, it is only necessary to search one branch. (The scheme amounts to a "hash code" for accessing words by name.)

The head, or growing end, of the list is defined by a 16-element pointer vector. These pointers aim at the most recently defined word in each branch. A link field in each word

definition is a pointer to the next previous word in the same branch. (The exact target of the link may not be the link of the previous word; some versions have the link pointing to the previous link plus one, for instance.) Each branch terminates with a word having zero link field. Definitions in different branches may be interleaved arbitrarily in memory.

A different dictionary organization has been adopted by most Forth users. The principle is to divide the dictionary into branches similar to those discussed above. In this scheme the branch in which a given word appears is under control of the user. The programmer segregates words according to the context of their application; such groupings are known as "vocabularies". The words VOCABULARY and DEFINITIONS control the branching. Figure 3.3 illustrates the VOCABULARY technique.

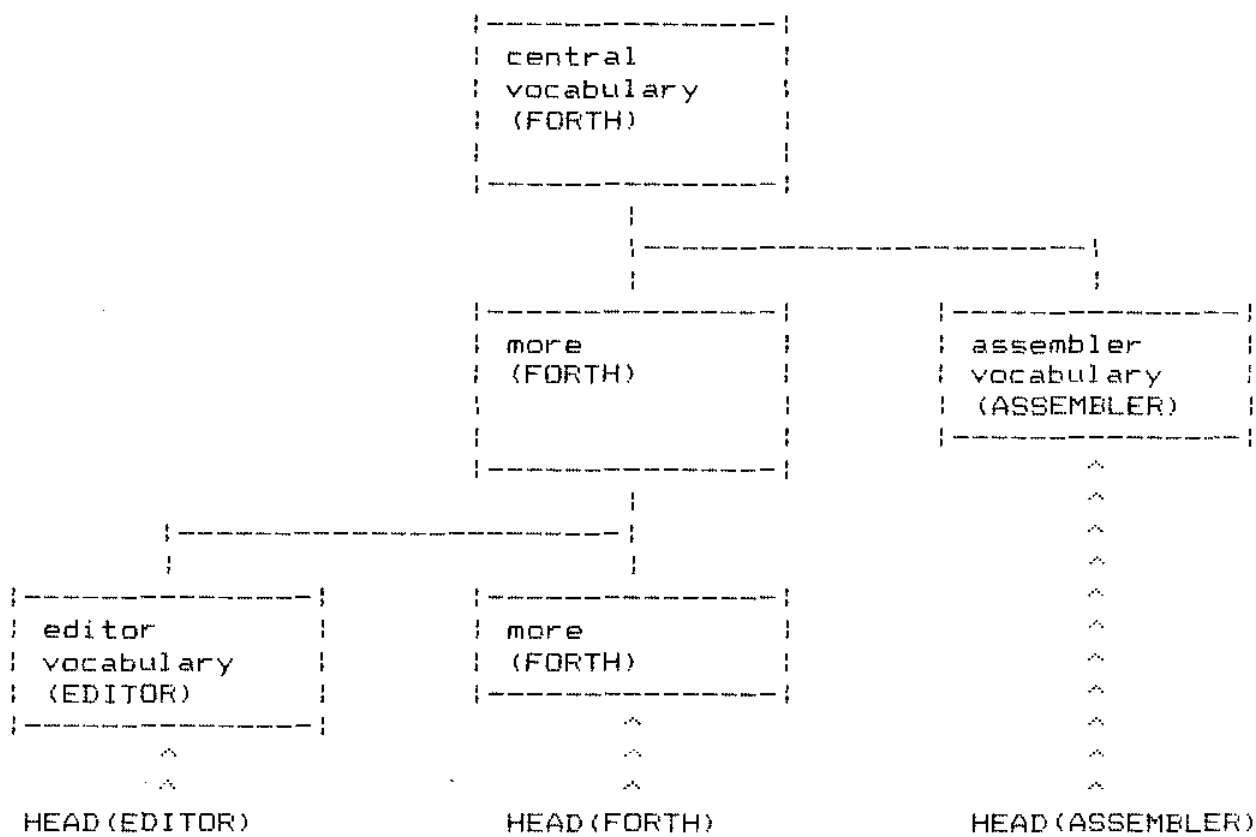


Fig. 3.3 VOCABULARY branching.

The number of HEAD pointers is unlimited; each one points to the last word defined in a dictionary branch. Branches merge as you trace back in memory until finally all searches end at the first Forth word in the root (FORTH) segment. A Forth word in one branch cannot execute (or interfere with) a word in another parallel branch except by explicit arrangement. Thus the VOCABULARY arrangement gives you some program security and can eliminate problems with unintentional multiple word definitions.

There are just two circumstances in which you have to specify what branch you are using. Most obviously, you need to say what branch the interpeter will search when you type a Forth word. Only one branch and its HEAD are active at a time. Thus if EDITOR is the current branch for searching, you cannot type a

word defined only in the ASSEMBLER branch. The other circumstance is when you are defininig new words: what branch should they be compiled into?

The branches in effect for word look-ups and for compiling do not have to be the same. For example, you may wish to use the ASSEMBLER vocabulary when you are compiling a CODE word in some other branch.

We briefly describe the action of VOCABULARY and DEFINITIONS. If you type

VOCABULARY FOO

a new branch of the dictionary is formed. The branch leaves the current dictionary branch (FORTH or the last one specified by DEFINITIONS) at its current head. A new Forth word FOO is created. When you type FOO, the dictionary branch to be used for further dictionary searches is switched to the FOO branch, i.e., the one you've just created. Similarly, any time you type FORTH, ASSEMBLER, etc., you switch to the corresponding branch.

If you type DEFINITIONS, the dictionary branch to be used for compiling is switched to the current branch used for searching.

3.3.2 Header Section.

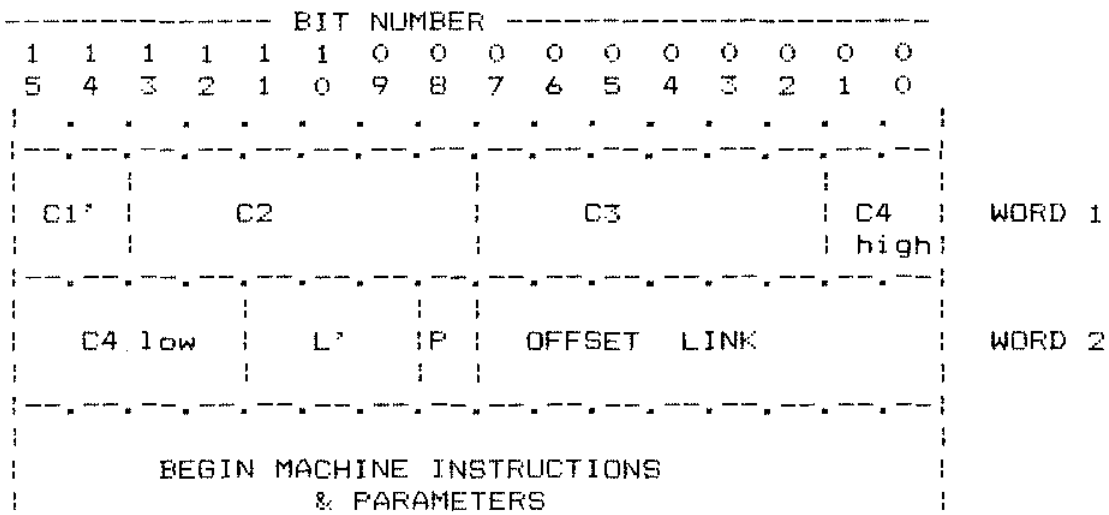
The detailed format of a word in the dictionary varies between Forth implementations. This section describes the format used in the Caltech-OVRO PDP-11 Forth. This format is notable in its very efficient use of memory. Only two memory words of

header are required in most cases, even when we use 4 characters plus count for a word name.*

---

*Previous Forth implementations for 16-bit computers have generally required 3 - 5 words for header information and typically recognized only the first 3 characters plus count. The core savings for the Caltech-OVRO PDP-11 system may exceed 1,000 memory words in a large Forth application.

Each word definition in the 16-way PDP-11 dictionary contains a "header" which defines the word name (first 4 characters and count), precedence, and the link to the previous word in the same dictionary branch. These data are efficiently encoded into two 16-bit memory words as shown in Fig. 3.4.



First four characters of word name:

C1 = C1' * 16 + THREAD#
C2, C3, C4

THREAD# (0 - 15) is the thread in which the word is found.
Characters are 6-bit ASCII codes.

Length of word name:

```
L = L' + 4      if L' <> 0
  = 4           if L' = 0, C4 <> blank
  = 3           if L' = 0, C4 = blank,
                   C3 <> blank
  = 2           if L' = 0, C4 = C3 = blank,
                   C2 <> blank
  = 1           if L' = 0, C4 = C3 =
                   C2 = blank
```

Range of L is 1 - 11 characters. Names with identical first 4 characters and lengths greater than or equal to 11 are indistinguishable.

Fig 3.4 Dictionary Header for PDP-11 (part 1)

("<>" means "not equal to")

Precedence bit:

P = 1 immediate execution (compiler directive)
= 0 normal word, may be compiled.

Link to previous entry:

Previous address = current address - 2 * (offset link)
(if offset link <> 0)

Previous address = long link field
(if offset link = 0)

Long link field is absent if the link span is less than 512 bytes.

Fig. 3.4 Dictionary Header for PDP-11 (part 2)

Some restrictions on the generality of Forth names have allowed the preservation of 4 characters plus count. The character set is limited to the 6-bit ASCII subset, which includes nearly all of the ASCII characters except the lower case alphabet. The 3-bit length field (L') allows lengths of 1 to 10 characters to be distinguished uniquely. Names of 11 or more characters are allowed, but these will be equivalent to Forth if the first 4 characters are the same. The limitation is slight, as most practical Forth code has few names as long as 10 characters.

The following are examples of distinguishable names:

A B ABCD ABCE ABCE1.

However, the following pairs of names are indistinguishable:

ABCD1 ABCD2

C1234567890 C12345678901

ABCD1234567 ABCD09876543210WERTY.

Even with the 6-bit coding and the restricted length field, a further savings in bits is required to fit all the header data

into two words. This is accomplished easily since a natural "key" or hash code for choosing a dictionary branch for a Forth word is one of the characters of the name. In particular the 4 low-order bits of the first character are distributed fairly randomly and are suited for the purpose. We define the following function:

THREAD# = HASH(NAME)

where the hashing function "HASH" is just equal to the number expressed by the 4 low-order bits of the first character of the "NAME" string.

If the HASH function is used to select a branch for the word entry, the Forth word header does not need to contain those bits selected by HASH; they would be redundant. Thus the field C1' in Fig. 3.4 contains only the two highest order bits of the first character; the low-order bits are implied from context, that is, from the thread number.

One bit of the Forth word header is reserved for "precedence". Normally this bit is zero, but for "immediate" words the bit is one. This bit has special importance for compilation; it is discussed below in Section 3.9.

The final header field consists of 8 bits reserved for the offset link. The link points to the last previous word in the same dictionary thread. In most cases the memory spanned by the link is less than 256 words (512 bytes), so that the offset link has enough bits. In cases where the link must cover more than 256 words, the offset link is set to zero and an additional 16-bit "long link field" is allocated. The long link field is a

complete byte address that may direct the dictionary search anywhere in memory. In the special case of the first word (foot) of a dictionary thread, both the offset and the long link field are zero.

3.3.3 Code And Parameter Sections.

A complete dictionary entry contains one or two sections in addition to the header discussed above. These are shown schematically in Fig. 3.5.

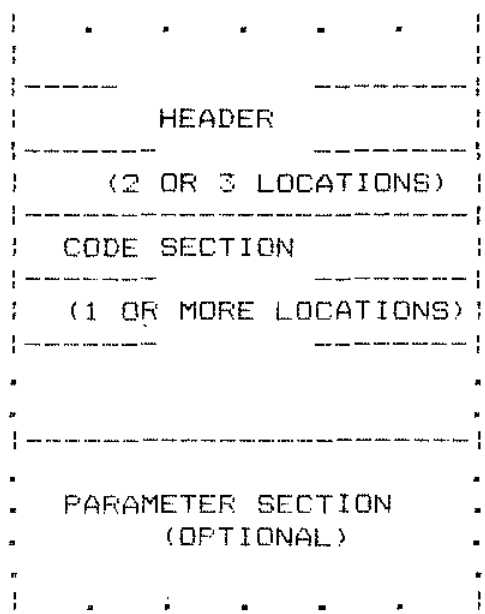


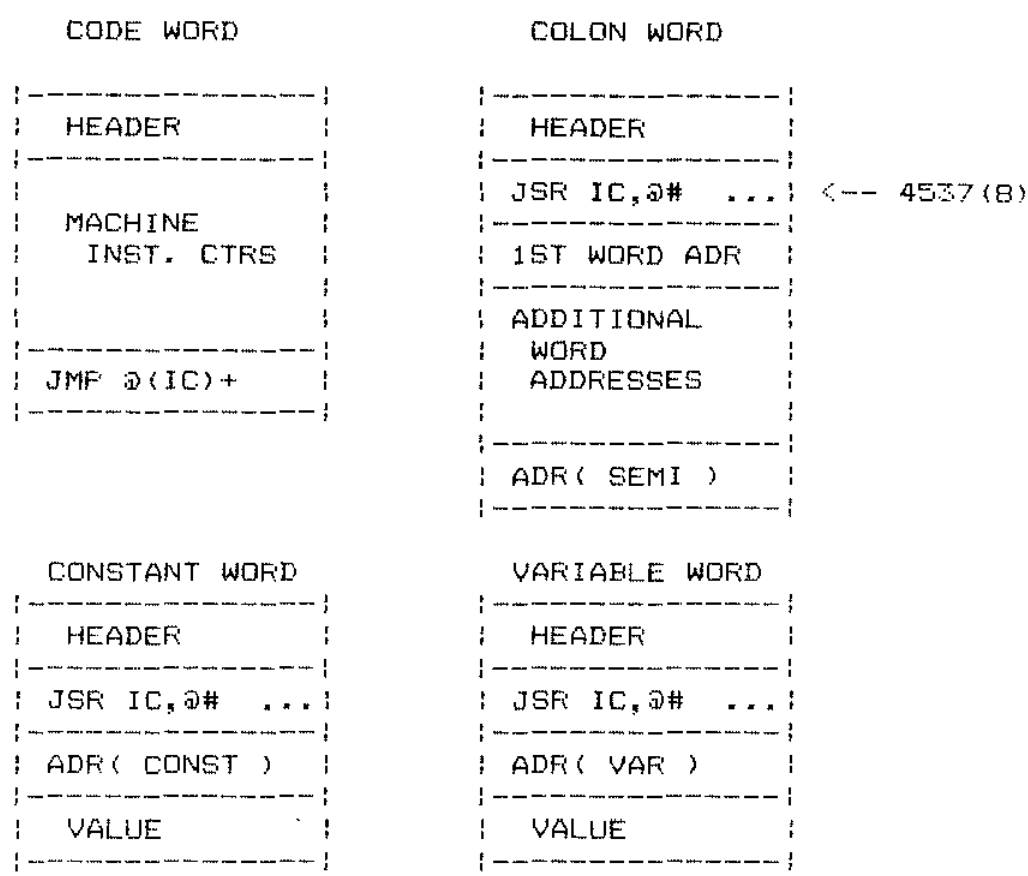
Fig. 3.5 General Forth Dictionary Entry.

Every word must contain a code section; this is one or more machine instructions that are executed when the Forth word is invoked. The address of the first location of the code section is the one compiled into address sequences in : efinitions (see Section 3.9). For CODE words, i.e., those defined by assembly instructions, the code section is normally the final part of the dictionary entry. It will finish by "calling" the address interpreter through executing the instruction NEXT, (JMP @(IC)+,

see Section 3.4).

Other kinds of words, in particular : words, require an additional parameter section in their dictionary entries. In : words the parameter section contains compiled addresses which direct the execution of the address interpreter. Words defined by VARIABLE or CONSTANT use locations in the parameter section to hold data.

Some more concrete examples of dictionary entries for various types of words are presented in Fig. 3.6.



(CODE SECTIONS ABOVE REFER TO FOLLOWING CODE)

```
SEMI:   MOV (R)+,IC      ; POP INST. CTR FROM RETURN STACK
        JMP @(IC)+       ; "NEXT" = ADDRESS INTERPETER

CONST:  MOV @IC,-(SP)    ; MOVE VALUE TO PARAMETER STACK
        MOV (R)+,IC      ; RESTORE IC FROM RETURN STACK
        JMP @(IC)+       ; "NEXT"

VAR:    MOV IC,-(SP)     ; MOVE ADR. OF VALUE TO PARM. STACK
        MOV (R)+,IC      ; RESTORE IC FROM RETURN STACK
        JMP @(IC)+       ; "NEXT"
```

Fig. 3.6 Common Forth Word Formats
(Caltech-OVRO PDP-11).

Note a little trick in the : word: the code section instruction (JSR IC,@#address) is a double-word instruction, but the second location is really just the first location of the parameter field -- as far as the Forth compiler is concerned. This address and those following comprise the sequence that directs the address

interpreter. It turns out that the PDP-11 instruction JSR IC,@#address has precisely the right action to start the address interpreter; it saves the instruction counter on the return stack and directs execution to the code located by the first address of the address sequence.

3.3.4 Expanding And Contracting The Dictionary.

The Forth dictionary is initially set up when the program is first loaded. This dictionary and its associated code are called the "object program" or "kernel". For Caltech-OVRO systems the kernel is defined in assembly language. Other systems sometimes use so-called "Metaforth", which is a Forth program that cross-compiles code from one Forth computer to generate a new kernel for another (or possibly the same) computer.

You extend the dictionary by executing "defining words" -- words that define new dictionary entries. You can do this directly from a terminal (typing :, CODE, etc.) or indirectly by LOADing blocks that contain defining words. The defining words have the logic required to compute the proper thread number and to enter a new element in the corresponding dictionary branch.

At times you need to truncate the dictionary and free up memory areas. You do this with FORGET. Type

FORGET BAR

to look up BAR in the dictionary and truncate all branches at the highest possible memory addresses lower than the beginning of BAR.

Thus BAR and all words defined after BAR (in time sequence)

are deleted. Judicious use of FORGET gives you a simple overlay capability in Forth.

3.4 PROGRAM CONTROL -- THE ADDRESS INTERPRETER.

Another central element of the Forth system is the function of the address interpreter (AI). This code directs the execution of Forth words from address sequences in memory. The normal termination of every CODE word is an invocation of the address interpreter.

The interpreter operates on a sequence of memory addresses which lie in consecutive words of main memory. Such an address sequence is the parameter field of a : word. Each address points to the code section of an earlier dictionary entry. (See Fig. 3.7.)

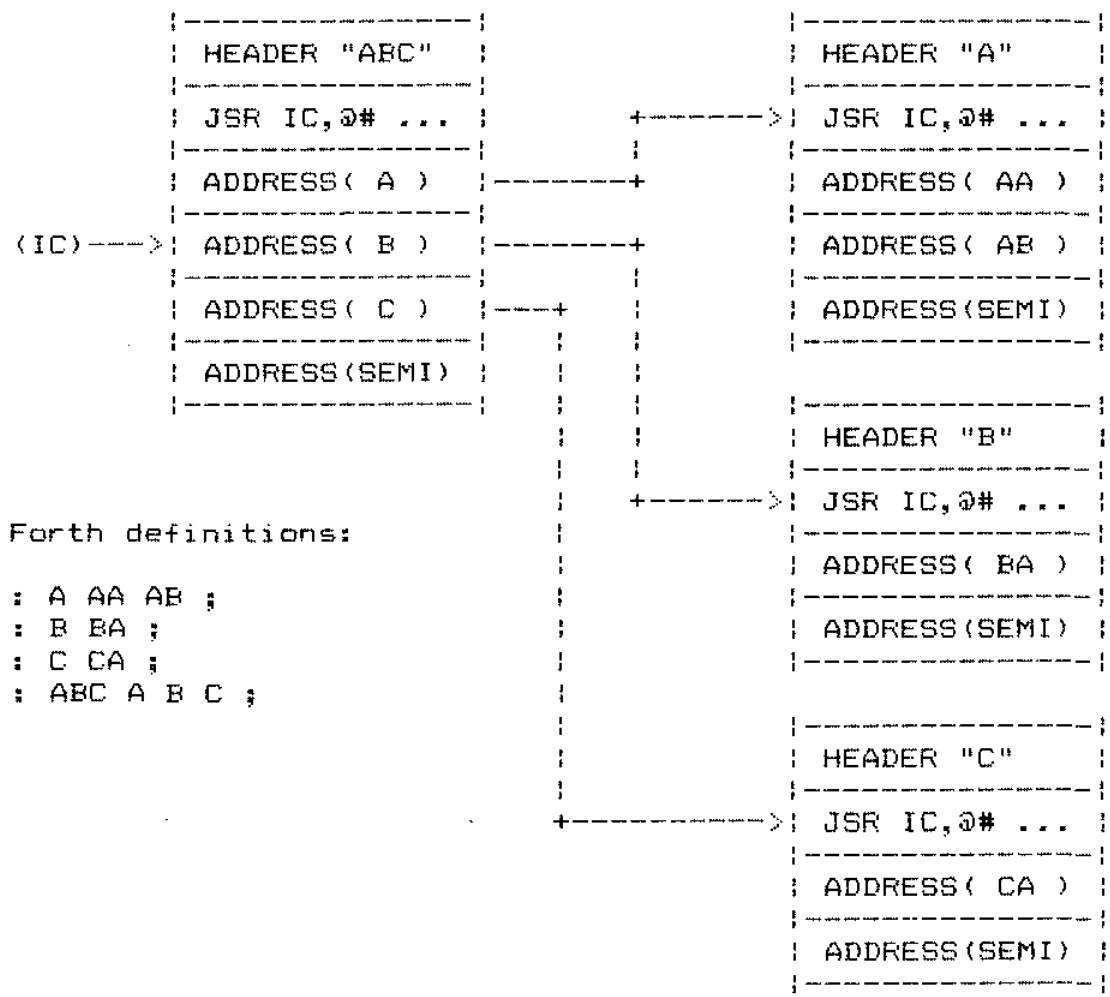


Fig. 3.7 Compiled address sequences.

In each : definition an address sequence specifies the Forth words to be run when the : word itself is executed. I.e., if ABC is defined : ABC A B C ;, the addresses of words A, B, C, and ; are found in the parameter field of ABC. These addresses define what actions occur when ABC is executed.

We can describe the effect of the AI in the following general terms. A register (or memory location) is reserved as the Forth "instruction counter" (IC). Like hardware instruction counters, IC points to the next (Forth) instruction to be

executed. "Instructions" to the AI are just the addresses of Forth words.

The Forth interpreter must pick up the address that IC points to, increment IC to point to the next address in sequence, and finally jump to the code specified by the first address. In terms of Fig. 3.7, the next invocation of the interpreter will pick up the address of the word B, IC will be incremented to point to the next address (address of C), and control passes to the JSR instruction in the code section of B.*

---

*Most Forth implementations use a slightly different algorithm for the AI. In these systems, the first word of the code section is always an address instead of an instruction. The address in turn points to the actual code to be executed. Thus the AI jump instruction must be a double indirect jump. In implementing the Caltech-OVRO system for the PDP-11, we found that core and speed savings could be achieved through adopting the technique described here.

Several computers are so appropriately designed that the entire AI function can be achieved in a single instruction. The DEC PDP-11 and PDP-10 are examples. Fig. 3.8 displays the AIs (NEXT instructions) for 3 types of computer.

```
(PDP-11)        NEXT:   JMP @(IC)+        ; IC is a  register

(PDP-10)        NEXT:   AOJA IC,@0(IC)    ; ditto

(8080)          NEXT:   LHLD IC           ; IC is a 16-bit
                        MOV  E,M          ; double-word
                        INX  H
                        MOV  D,M
                        INX  H
                        SHLD IC
                        XCHG
                        PCHL
```

Fig. 3.8 Address Interpreters for 3 Computers

The discussion to this point tells how the Forth AI progresses through an address sequence a step at a time. The linear flow of execution may be modified in several ways. The simplest would be to alter IC directly in a CODE-defined word, and then to invoke the interpreter.

A more subtle, but more useful redirection of instruction flow is performed every time a ; word is executed from a : word. This is the situation presented above in Fig. 3.7.

A good way to divert the AI is to store away the contents of IC on a stack (the return stack), and to set IC so that it points to the first word of the parameter section of the new word to be interpreted. (In this way, the AI algorithm is recursive.)

In general, what is the appropriate instruction to put in the code section so that the AI is redirected? We need an instruction that lets us push a register on a stack and somehow "remembers" where it is when executed. Usually some kind of subroutine call instruction is appropriate.

As we suggested already, the PDP-11 has an instruction which performs all the right operations by itself. With most other computers you need to write a 2 or 3 word subroutine

(conventionally called COLON) to redirect the AI. The techniques for 3 computers are illustrated in Fig. 3.9.

(PDP-11)

```
Appearance of code section:     JSR IC,@#       !! really one
                                address1        !! instruction
                                address2
                                . . .
```

No subroutine required.

(PDP-10)

```
Appearance of code section:     PUSHJ RP,COLON
                                address1
                                address2
                                . . .

Required subroutine:    COLON:  EXCH IC,0(RP)
                                ADJA IC,@0(IC)  ; (NEXT)
```

(8080)

```
Appearance of code section:     CALL COLON*
                                address1        ; two bytes
                                address2        ; two bytes
                                . . .

Required subroutine:    COLON:  LHLD IC
                                XCHG
                                CALL RPUSH      ; (DE)-->RSTK
                                POP  H          ; FROM CALL INST.
                                SHLD IC
                                JMP  NEXT*
```

*The CALL COLON and JMP NEXT instructions can be replaced by hardware reset (RST) instructions, with a savings of 2 bytes per use. You must have appropriate code at the corresponding low-memory locations.

Fig. 3.9 The COLON Function for 3 Computers.

You end a normal : definition with ;. The semicolon (;) compiles an address called "SEMI" into the dictionary as the last entry in the parameter section of the word you're currently defining. (; also resets the compile state.) SEMI is the

address of a machine code routine that undoes the effect of the COLON function. It must restore the old contents of IC from the return stack. The SEMI routines for the same 3 computers are given in Fig. 3.10.

```
(PDP-11)        SEMI:   MOV   (RP)+,IC
                        JMP   @(IC)+      ; (NEXT)

(PDP-10)        SEMI:   POP   RP,IC
                        AOJA  IC,@0(IC)   ; (NEXT)

(8080)          SEMI:   CALL  RPOP
                        XCHG
                        SHLD  IC
                        JMP   NEXT
```

Fig. 3.10 The SEMI Function for three Computers.

The discussion and figures above indicate that the address interpreter may be nested very deeply, limited only by stack space. In other words, Forth : words can refer to earlier : words, which can refer to yet earlier words, etc. The time overhead for the AI recursion (or the "calling" of one : word by another) is seen to be very nominal -- about equivalent to a conventional subroutine call.

In summary we can say that the address interpreter is the engine that makes : words go. The technique is not new; it is also used in DEC's "threaded code" in PDP-11 Fortran. But in combination with the text interpreter (see below) it is responsible for the unique power of the Forth system.

3.5 THE TEXT INTERPRETER.

In the preceding Section we discussed the address interpreter and how Forth executes : words containing compiled address sequences. There is one fundamental Forth : word (GO*)

whose job it is to interpret what you type in to your terminal. This is called the "text interpreter" (TI). It is distinguished from the address interpreter because its input is text from a terminal (or block) rather than addresses.

---

*Actually GO is an "anonymous" word (without a header) and can not directly be accessed from your terminal.

The TI is really a Forth program in its own right. In fact it is the basic program that executes in normal Forth systems. When you type in a word ("command") to Forth, it is the TI that interprets your command and actually begins execution.

A structured program (in pseudo-code) for a typical TI follows in Fig. 3.11.

```
GO: IF( Input is from typewriter )
        THEN IF( Text buffer is empty )
                THEN Wait for next full input line
                        from typewriter;

    IF( Input is from typewriter )
        THEN Prepare to read typewriter buffer
        ELSE Prepare to read selected block buffer;

    Collect a text string (word) from buffer;

    IF( Word exists in dictionary )
        THEN IF( In compile state )
                THEN Compile a pointer to dictionary
                        word;
                ELSE Execute the dictionary word

        ELSE IF( Input string converts to a number
                in current radix )
                THEN IF( In compile state)
                        THEN Compile a pointer to "LITERAL"
                                followed by number value
                        ELSE Push number value on stack
                ELSE Abort;
    GO TO GO;
```

Fig. 3.11 A Structured Pseudo-code Text Interpreter.

We can elaborate a bit on this program. The input to the TI can be either from the terminal ("typewriter") or from block storage. Nothing happens with typewriter input until you enter a complete line, ended with "return". If a screen is the input source, TI runs straight through without a pause until ;S or the end of the screen is encountered.

"Collecting a text string" means scanning the input source until a complete word-name-candidate (token) is found. That is, scanning begins from the current position of an input text pointer until the first non-blank character is found. Then all the non-blank characters up to the next blank (or other specified delimiter) are moved to a special place*.

---

*Actually to the next several available dictionary locations in case this word is to be entered in the dictionary.

Using the appropriate rules for identifying word names with dictionary entries (e.g., first 4 characters plus length), the TI attempts to find a match with an existing entry in the dictionary. If a match exists, the TI will normally simply execute that word. There is one case where, if you type a word, you don't want it executed: this is when you are defining a : word. If you are defining a : word, the TI will store a pointer to the word in the next available dictionary location.

If there is no matching entry, the TI will try to see if its token will convert properly as a number. If the string does make sense as a number, that number is normally just pushed on the stack. If you happen to be compiling a : word, the TI compiles

a call to a special word "LITERAL" followed by the value, so that the number you've typed will be pushed on the stack when you execute your new word.

If the "word" you've typed can't be found in the dictionary or converted as a legal number, the TI gives up and ABORTs. All the stacks are reset, the compile state is reset, the word itself is typed again followed by a question mark, and Forth starts the TI all over again.

3.6 ERROR MESSAGES -- ABORT.

The only "standard" error routine in Forth is called ABORT. ABORT simply resets nearly everything in the Forth system: the parameter and return stacks, the compile/execute state (to execute), the terminal buffer, etc. Only the dictionary and the current state (block contents and update flags) of the block I/O system are not affected.

In addition to the reset function, ABORT types a very simple error message on the terminal: the name of the last word processed by the text interpreter followed by a question mark.

The action of ABORT in a real time Forth system is not standardized. In most situations with Caltech-OVRO Forth, an ABORT caused by an error in a background (user-terminal) task will not affect a foreground, real-time task. This is simply because the background task only runs when the foreground task is finished, i.e., when the foreground task has nothing to keep on the stacks.

3.7 BLOCK INPUT/OUTPUT.

Forth normally maintains a single direct-access file on secondary storage (such as disk). This storage is not logically required to run Forth; micro-computers, for example, may use a Forth system permanently written in read-only memory. But in general purpose minicomputer systems, much of Forth's versatility depends on adequate block storage.

The conventional record size for block storage is 1024 8-bit bytes, or 512 16-bit words. Blocks are simply numbered sequentially from 0; thousands are typically available.

Typical systems have two block buffers in main memory. When you type

nnn BLOCK

Forth chooses the less recently used buffer, writes its contents back to disk if necessary (i.e., if that block has been UPDATEd), and then finally reads in block nnn from disk. The buffer address is returned on the stack.

Once in main memory, a block may be read or altered in any way. If you want to change a block's contents on disk, you must be sure to type UPDATE following BLOCK. UPDATE sets a flag that insures that the buffer last returned by BLOCK will be rewritten to disk before the buffer is reused for some other block. You can type SAVE-BUFFERS at any time to force rewriting of any UPDATEd blocks to disk.

If you want to be sure that you are dealing with "fresh" copies of disk blocks, you can type EMPTY-BUFFERS before BLOCK. EMPTY-BUFFERS simply sets a flag that marks all block buffers empty; thus any BLOCK following will force a read disk

operation.

Forth blocks are perfectly general in the types of data that they may hold. However one important use for blocks is to hold Forth text, i.e., input for the text interpreter. In this mode a block known as a "screen", and is considered to be a single string of 1024 characters. That is, the text interpreter may scan the entire block without any division into smaller records such as lines.

For text entry, editing, and listing, however, it is convenient to divide the 1024 character block into 16 lines of 64 characters. The lines have fixed length and there is no separation (carriage return or line feed) between the last character of one line and the beginning of the next.

When you type

nnn LOAD,

Forth fetches block nnn, stores the text interpeters input pointers on the return stack, and sets the input pointers to the beginning of the block. The interpreter will then scan the block executing words as they are encountered, until told to do otherwise. The end of the block or a semicolon-S (;S) will terminate the scan on each block.

3.8 FORTH ASSEMBLERS.

Section 2.4 described generally how input text can be converted into machine-language instructions. This process is called assembly. Forth assemblers for different computers will naturally differ according to their

instruction sets. The full assemblers for some representative Forth systems are presented in the Appendices. This section deals with aspects of assembly that are common to most Caltech-OVRO Forth systems.

You can assemble code any time the system is in the execution state, i.e., when it is not compiling : words. Usually you assemble code in the course of a CODE word definition.

The assembler vocabulary consists mainly of op-code words whose names are normally chosen to reflect the conventional assembler codes in a macro assembler. In fact the op-code names are usually just the conventional mnemonic with an appended comma. Thus the PDP-11 move instruction, MOV, becomes MOV, in Forth.

To assemble a machine instruction into the dictionary, you type the address fields and modifiers you need followed by an op-code word. (Remember reverse Polish notation?) There is normally a set of special words to help you set up the correct addressing modes, branch conditions, etc.

A sample CODE definition for the PDP-11 might look like:

```
CODE ADD3 0 S )+ MOV, 0 S )+ ADD, S ) 0 ADD, NEXT,
```

This word will add up the top 3 numbers on the stack, leaving the sum.

The first part of the definition (CODE ADD3) sets up a new dictionary entry (header only) with the name ADD3. The code section of ADD3 is filled in with 4 machine instructions: a MOV, two ADDs, and a JMP (expansion of NEXT,). The first instruction

moves the contents of the top stack location to register 0 and adds 2 bytes to the stack pointer register. The next instruction adds the contents of the next stack location to register 0, incrementing the stack pointer again. The second ADD adds register 0 to the contents of the next (originally the third) stack location without changing the stack pointer. NEXT, expands into the instruction JMP @(IC)+, the address interpreter.

An equivalent MACRO-11 program would look like this:

```
.WORD HEADER1
.WORD HEADER2
MOV    (S)+,R0          ;MOVE STACK TO REG.  0
ADD    (S)+,R0          ;ADD NEXT STACK VAL.  TO R0
ADD    R0,(S)           ;ADD TO NEXT STACK VAL.
JMP    @(IC)+           ;GO TO NEXT FORTH INSTR.
```

Forth assemblers provide forward conditional branches similar to the compiler directives IF, ELSE, and THEN. These are the macro instructions IF,, ELSE,, and THEN, (with ,s). In the case of the PDP-11, these macros set up appropriate conditional branch instructions that test a register. An example:

<load R1> 1 TST, NE IF, <true code> ELSE, <false code> THEN,

This expands into the equivalent of the following MACRO code:

```
        <load reg. 1>   ; set up data in register 1
        TST     R1      ; test register 1
        BEQ     1$      ; branch if equal zero
        <true code>     ; do if R1 .NE. 0
        BR      2$      ; branch around false routine
1$:     <false code>    ; do if R1 .EQ. 0
2$:     . . .           ; end
```

The "else clause" is optional, thus you can write

<load reg. 2> 2 TST, GT IF, <true code> THEN,

which expands to

```
        <load reg. 2>
        TST     R2
        BLE     1$
        <true code>
1$:     . . .           ; end
```

3.9 COMPILATION OF : WORDS.

The use of : words has been discussed above and the dictionary format was presented in Fig. 3.6. The process of producing a dictionary entry from the input text is called compilation for : definitions. Compilation is distinct from assembly, a term which applies only to CODE words.

Forth has two "states": execution and compilation. In execution state the text interpreter operates normally, executing words as they are found in the input text. The word : in the text stream changes the state to compilation; it also invokes WORD to collect the next properly delimited token from the text stream. The token becomes the name of the new word; it is placed in the next available dictionary locations in the correct dictionary format. The link field is set to point to the last-defined word in the same dictionary branch, and the HEAD pointer is set to point to the new entry. A call to the COLON function is placed in the code section. (This is the "half-instruction" JSR IC,@#... in the PDP-11 system.)

(At this point in compilation the dictionary formally contains the new entry, which is not yet fully defined. To prevent false, premature references to the entry, : also alters ("smudges") the name field slightly so that the name becomes unrecognizable. At the conclusion of the definition, ; or ;CODE restores the correct name.)

It now remains to create the parameter field of the new : word. In the compile state, the text interpreter (Fig. 3.11) is modified so that when an input word is found in the dictionary it

is not executed; rather, its address is stored in the next available dictionary location. Similarly, numbers are not immediately pushed on the stack, but the address LITERAL is compiled followed by the literal value of the number. (LITERAL points to a simple code routine that picks up the number following LITERAL's invocation point, pushes the number on the stack, and increments IC in order to skip to the next compiled address.) Thus the number is not pushed on the stack until the new word is executed.

The interpreter will proceed to compile the input text stream into the dictionary until a "compiler directive" is encoun-tered. A compiler directive is a word with a precedence bit set to 1. Such words are executed immediately, even when Forth is compiling.

The most common compiler directive is ;, which compiles SEMICOLON into the dictionary and also resets the compile state. Other compiler directives are IF, THEN, ELSE, ;CODE, etc.

If you want to make a word you've just defined into a compiler directive, simply type IMMEDIATE. (Since IMMEDIATE is itself immediate, you can make a word immediate either by typing "IMMEDIATE" inside or outside the definition. For example,

```
: X IMMEDIATE A B C ;   and
: X A B C ;  IMMEDIATE
```

are equivalent.)

3.10 DEFINING WORDS -- DOES>.

A special technique is available in Forth to define words whose function will be to define words. Some of these "defining

words" are built into the kernel: CODE, :, CONSTANT, etc. A new defining word is appropriate whenever a new class of word functions is required. The availability of defining words makes Forth an unusually extensible language system.

As an example take VARIABLE, which is defined in the standard system. The new class of words provided by VARIABLE consists of words that push the address of their parameter field on the stack. N may be defined a VARIABLE by typing

VARIABLE N.

The dictionary entry created for N is shown in Fig. 3.12.

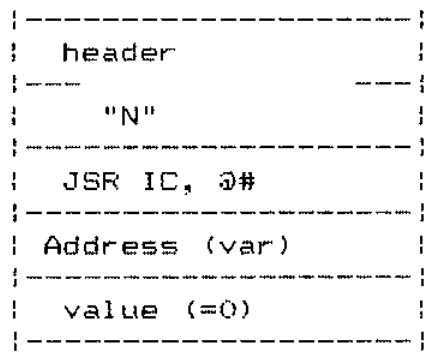


Fig. 3.12 Dictionary Entry for VARIABLE N.

The entry differs from an entry produced by CONSTANT only in the address that appears in the second word of the code section. All VARIABLE words will have the address "var" in this location. This code must pick up the address of the parameter field of the variable word being executed and then push it on the stack.

VARIABLE may be defined in terms of the more fundamental Forth words CREATE and DOES>:

: VARIABLE CREATE DOES> ;.

The definition has two parts; the first is like a normal :

definition. Word names appearing here are compiled into the dicitionary. The : part of VARIABLE contains only CREATE. CREATE makes a new entry in the dictionary (when VARIABLE is executed). The name of the new entry is taken from the token in the input stream that follows VARIABLE, for example, "N" in the case above.

The second part of the example begins with DOES>. DOES> is a compiler directive that compiles an address (called does>), but keeps the system in compile state. Following DOES> are more words to be compile. These instructions define the address sequence ("variable") which will be associated with all VARIABLE words. When this address sequence is interpreted (when "N" is executed, for example), there will be a single parameter passed: the address of the parameter field of the VARIABLE word. In the case of VARIABLE, that parameter is exactly the desired result of the VARIABLE word; therefore, only the terminating definition, ; is required.

The dictionary entry for VARIABLE is shown in Fig. 3.13.

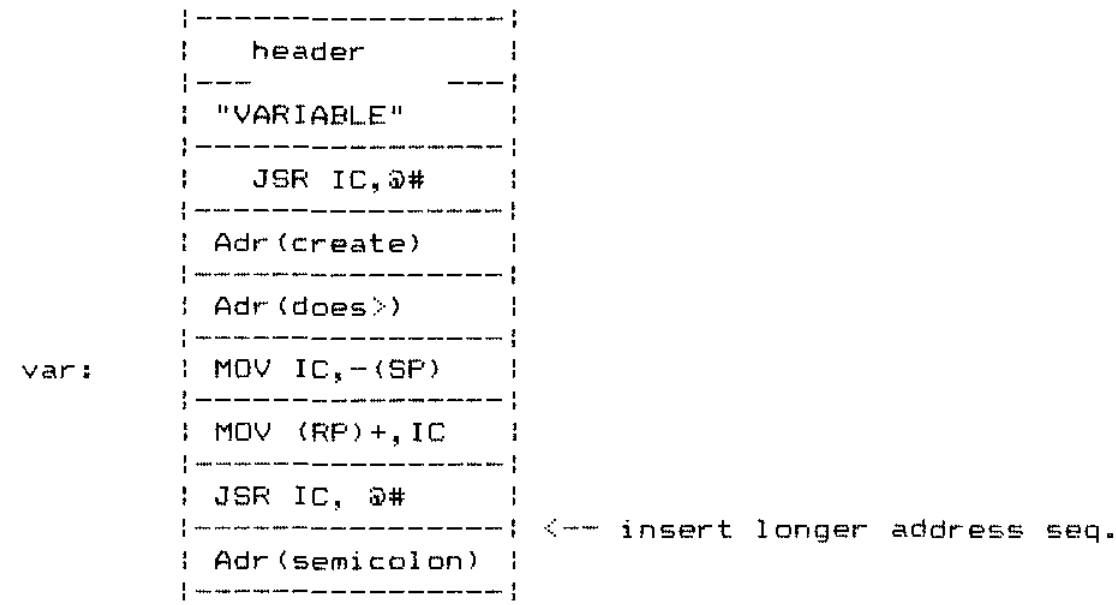


Fig. 3.13 Dictionary Entry for VARIABLE.

What happens when we execute VARIABLE? First, CREATE makes a dictionary entry using the next token in the input stream as its name ("N", for example, in Fig 3.12). At this point, the new dictionary entry has an undefined code section. The code addressed by "does>" causes the code section to be filled in with a "JSR IC,@#var" instruction. When the new word (N) is executed, the "does_start" code will collect the parameter field address of N, which the JSR instruction has placed on the return stack, and push it on the parameter stack. This code furthermore starts the Address Interpreter running at location "var" with the correct return stack contents so that after the terminal ; is interpreted, at execution-time (of N), control returns correctly through the Address Interpreter.

The code routine "var" for any VARIABLE word works in the following way. When N is executed (for example), "var" pushes the contents of register IC on the stack. (It turns out that the

"JSR IC,@#var" instruction puts the address of the first word of the parameter field in that register.) The code must now restore the last generation of the IC from the return stack. In general there will be further Forth words compiled in the

"DOES> ... ;"

section, so the Address Interpreter is invoked through the usual JSR mechanism. In the case of a VARIABLE word, however, there is nothing further to do, and the address of "semicolon" terminates the address sequence.

To summarize, CREATE and DOES> are used to create new code routines which are associated with a defining word. All words defined with that defining word will employ the new code routine. Thus a new Forth word class is defined.

A word closely related to DOES> is ;CODE. You may use ;CODE to make a defining word for a class of words whose action is specified by an assembly language routine. The parameter field address is passed in the same way as for DOES>. Thus an alternative definition of VARIABLE would be

: VARIABLE CREATE ;CODE NEXT,

The associated code routine is null in this case. Figure 3.14 illustrates the appearance of the ;CODE form of VARIABLE.

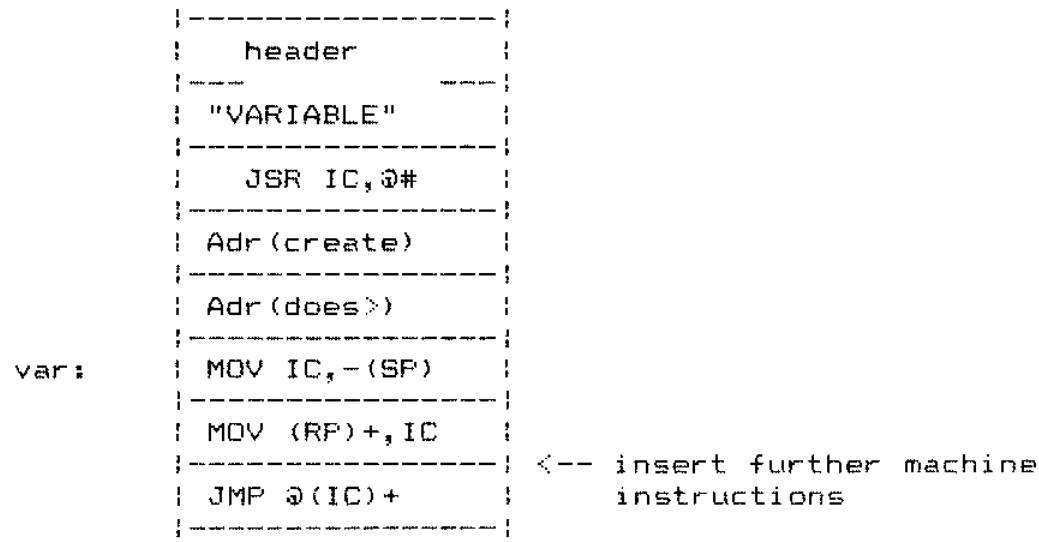


Fig. 3.14 Alternate Dictionary Entry for VARIABLE.

Defining words may be established to define any data type or operation class; examples include VARIABLE, ARRAY, SET, etc. If a class of fixed repetitive operations can be identified it may be most economical of storage and execution time to create an appropriate defining word. An example with CONSTANT: the line

1 CONSTANT ONE

defines ONE as a constant word that will push the value 1 on the stack. This will always be more efficient that using the number 1 literally. (In the text interpreter the number conversion is avoided, and in a compiled definition the call to LITERAL is not needed.)

In practice we use the name "1" instead of ONE. Thus the dubious definition

1 CONSTANT 1.

Of course, you could also define 1 with the following line

: 1 1 ;,

but this way two extra storage locations are used -- for LITERAL

and for SEMICOLON. Because of the return stack operation and the extra interpreter cycles, execution of the : defined 1 would be much slower than the CONSTANT word.

3.11 BRANCHES IN : WORDS.

3.11.1 An Unconditional Branch.

An unconditional branch to any Forth word is provided by the EXEC function. You type

<address value> EXEC

to jump to the address specified. If the address is that of a Forth word, you could type

% <word name> EXEC.

(% returns the code section address of the word whose name follows. Note that in non-Caltech-OVRO systems, the word ' gives the right address. In the Caltech-OVRO system ' returns the address of the parameter field.)

3.11.2 Conditional Branches.

Use of the branches IF, BEGIN, etc. was described in Chapter 2. The discussion here concerns the dictionary entries produced by these words and the state of the stack during compilation.

Consider 0=, which might be defined

: 0= IF 0 ELSE 1 THEN ;

This word tests the value passed to it on the stack; if the value is non-zero, zero is returned. Zero input produces one. The compiled dictionary entry for 0= is presented in Fig. 3.15.

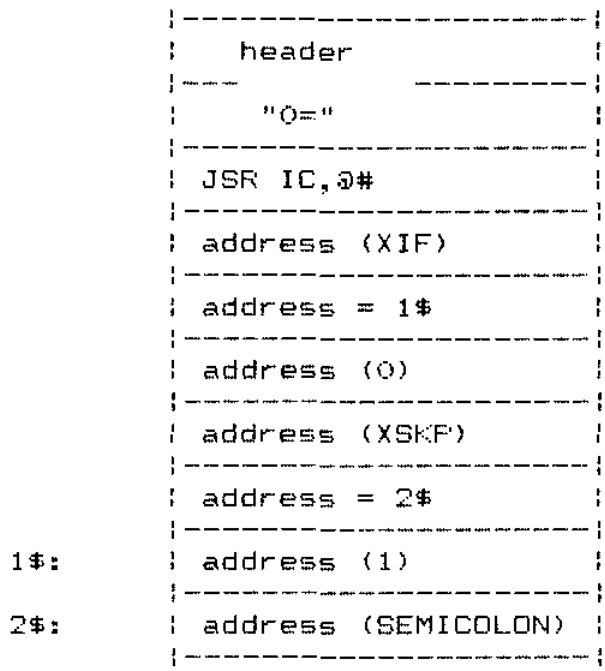


Fig. 3.15 Dictionary Entry Illustrating IF.

The words IF, ELSE, and THEN are compiler directives; they are not compiled in the 0= definition, they are executed. Their execution does compile word addresses and address constants, however. The word addresses are shown in the figure as XIF and XSKP, which actually control branching at execution time.

The example illustrates the operation of IF - ELSE - THEN sequences. The address interpreter begins with the address XIF. XIF tests and pops the stack. A false outcome (zero) will require a branch to the "false clause", i.e. the words compiled between ELSE and THEN. The branch is carried out by loading IC with the contents of the location following the address XIF ("1$"). The interpreter continues at that location, pushing 1 on the stack.

The "true clause", between IF and ELSE, will be executed if the stack tests true (non-zero). In this case XIF simply incre-

ments IC so that the interpreter skips over the address 1$. Zero is pushed on the stack. The interpreter then encounters the address XSKP which unconditionally loads IC with the contents of the following location (2$). Finally SEMICOLON terminates execution of either case.

Other forms of compiled branches work like IF, THEN, etc. Fig. 3.16 is the dictionary entry of a typical DO - LOOP construction:

: LP 4 0 DO RANGE LOOP AFTER ;.

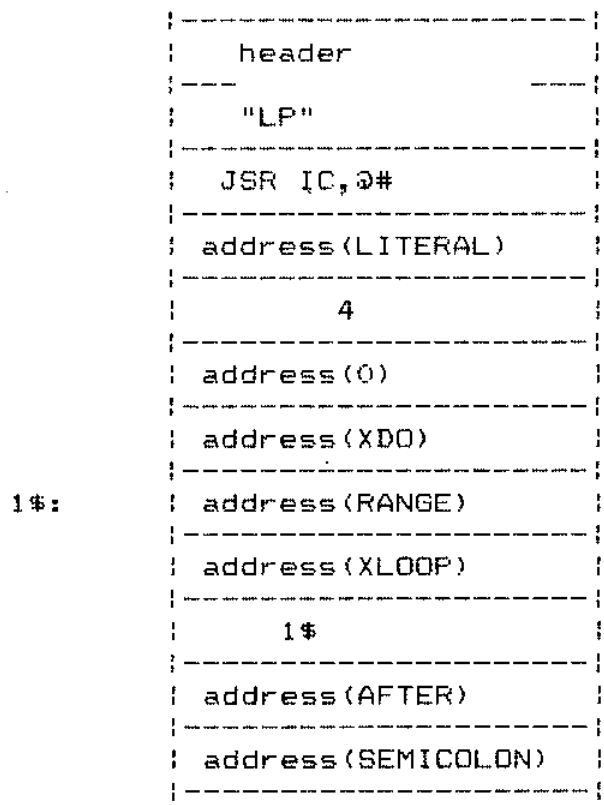


Fig. 3.16 Illustration of DO - LOOP.

A few peculiarities should be explained. We assume that 0 is defined by

0 CONSTANT 0

as discussed above. However 4 is not so defined in this example;

it is treated the way arbitrary numbers are. Thus LITERAL must be executed with argument 4 to get 4 on the stack. (IC increments after LITERAL picks up its argument so that the interpreter resumes with the 0 word. RANGE and AFTER are just random words predefined in the dictionary.

XDO takes the top two stack variables (0 and 4) and pushes them on the return stack as discussed in Chapter 2. Execution proceeds with RANGE. XLOOP increments the loop index, checks the index against the limit, and either branches back to RANGE (by loading IC with 1$) or skips to AFTER.

3.12 INTERFACING WITH AN OPERATING SYSTEM.

A controversial topic among Forth users is the role of general purpose operating systems. The computer vendors supply operating systems with varying levels of function and complexity. Generally their purpose is to allocate, schedule, and promote sharing of computer resources for a single task or for several concurrent tasks. The question is whether the function, standardization, and economy of the operating systems are worth the overhead in speed and memory for particular Forth applications.

Caltech-OVRO systems have been developed both with and without OS support. In this section we consider some criteria for these choices. These topics will be taken up again in Chapter 5 when we consider large-memory Forth systems.

3.12.1 To Stand Alone Or Not To Stand Alone.

We can attack the problem either economically or technically. In economic terms, the price of computer memory (particular-

ly semiconductor memory) is falling rapidly. Low cost peripherals such as floppy disks are widely available. These technological forces tend to reduce the economic penalty for relatively large, general purpose operating systems.

In contrast, the cost of software development steadily rises. So there is an economic incentive favoring utilization of off-the-shelf software systems when possible. Reinvention of complex scheduling and I/O algorithms is rarely justified.

Technical analysis is more difficult. One (prominent) line of thinking is that much can be done with extremely simple software. Thus Forth standalone systems with minimal multiprogramming, no concurrent I/O, and practically no error recovery capabilities have been very successful. The same thought process leads to the idea that practically all computing can be handled by Forth programming on 16 bit computers with no more than 32K memory words. (Thus the mapping problem for larger memories is avoided.)

With standalone Forth, cross assemblers (such as MetaForth) can be developed that generate systems with nearly identical structure for widely different types of computer. Maintenance and development effort are reduced accordingly.

Technical arguments for Forth running under operating systems have a few major themes: concurrency of large tasks, reliability, and transportability. Programming for many large jobs is simpler when large amounts of memory are available. Memory is cheap, 16 bit computers can give you instant access to 32K words or more; so why not allow each task in the system to use up to

this amount?

The difficulty with large tasks in a multitasking system is that physical memory has to be mapped into the 32K task address space. The mapping problem is fairly severe if you require efficient use of physical memory and CPU time. Vendors' operating systems usually cope with this problem; development of generalized Forth memory mapping software is a nontrivial project.

Concurrency of large tasks may include non-Forth tasks. For example a Forth real-time control task may have to co-exist with Fortran data reduction. This is feasible if both tasks run under a common operating system.

Reliability of a software system is hard to define. One useful principle is that a software fault in one task of the system should be isolated from other tasks. Commonly this feature is provided by memory mapping and by carefully defining user- and system-states of the CPU. Again, it is a major effort to provide these functions in standalone Forth.

Another aspect of the reliability problem is what to do in the event of hardware faults. Large peripheral devices (particularly disks) can be very complex. Many operating and error recovery modes are available. The manufacturer's device driving software (a component of operating systems) becomes correspondingly elaborate and difficult to repeat in Forth.

One hindrance to the wider propagation of Forth has been that many implementations <u>are</u> constructed using the MetaForth cross-compiling scheme. Forth defined in terms of Forth is difficult to learn and difficult to transport to a non-Forth computer. Implementations in the standard assembler code of a

particular machine can easily be transferred to other machines of the same type, particularly if standard file structures and formats are observed.

3.12.2 OS Interfacing Techniques.

Implementation of Forth as a task under an operating system such as RT-11 or VAX/VMS is generally simpler than as a stand-alone system. The OS provides macro instructions for terminal and disk I/O. Buffering and error checking are provided by the OS.

When you have to connect non-standard I/O devices or respond to special hardware interrupts, the situation is a little more complicated. The general purpose operating systems necessarily restrict your freedom of interfacing with external devices, since the system's integrity must be preserved for other system users. In particular for RT-11 you must carefully observe the interrupt protocols with appropriate use of the .INTEN and .SYNCH macros.

Of course any macro defined in the conventional assemblers can be expressed in terms of the Forth assembler. Unfortunately standard Forth lacks a true macro-processing capability, so that it is difficult to define macros with the generality available in the conventional assembler. The problem is not too bad, since you rarely need more than a few types of macro in a given Forth application. VAX/VMS Forth (Chapter 5) has an interesting Forth-based macro capability.

3.13 MULTIPROGRAMMING AND REAL-TIME APPLICATIONS.

In real-time control or data acquisition jobs it is often necessary for a Forth system to interact with external devices on a prescribed time schedule, e.g. sample data every 10 msec or update telescope drives every 0.5 sec. You usually want to be able to converse with Forth in a normal way while the real-time processes are running. In some cases, unrelated users may want to share the computer at the same time.

All such situations require some multiprogramming scheme. Multiprogramming is the general technique of sharing the computer's time, memory, and peripheral devices between multiple job tasks or users. A number of schemes have been used for Forth multiprogramming. Most Caltech-OVRO systems use a multilevel priority scheduling system. Other Forth systems use a round-robin scheduler, especially for multiuser "timesharing" applications. When running under a multiprogramming operating system, independent copies of Forth may be run as separate tasks under the operating system.

3.13.1 Priority Scheduling.

A simplified priority scheduling algorithm is used in several Caltech-OVRO systems. Figure 3.17 illustrates the method.

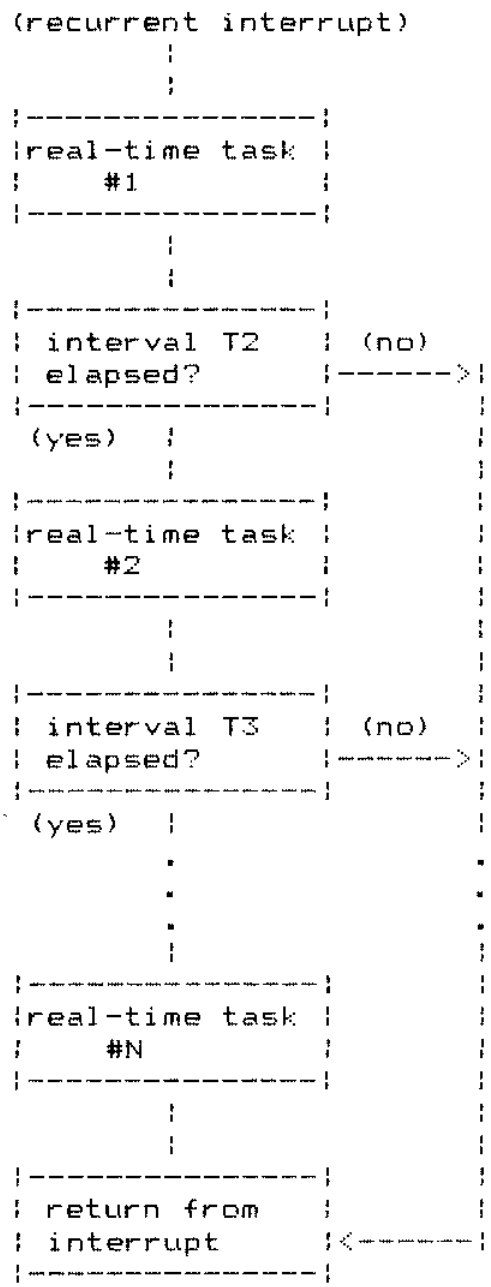


Fig. 3.17 Priority scheduled Multiprogramming.

A recurrent interrupt (say 60 Hz) initiates the "foreground tasks" shown in the figure. Task 1 contains all the functions to be performed every interrupt. When task 1 is completed a counter is examined to see if a predetermined number of interrupts has been processed. If the interval T2 has elapsed, the counter is reset and the lower level task (#2) begins. If T2 has

not elapsed, a return from interrupt instruction is performed: the "background" (e.g. Text Interpreter) then has the use of the machine until the next interrupt.

This multiprogramming technique lets you set up an arbitrary number of execution levels each of which is initiated after a certain integral number of instances of the next higher level. If the interrupt return information is stored carefully, the foreground structure is at least partially reentrant. The level 1 task may interrupt the level 2 task many times before level 2 completes. You must insure that there is enough time for each task level to complete before it is next scheduled to run.

Advantages of this priority scheduling method include the minimal context switching requirements, simplicity, and guaranteed servicing of high priority tasks. The context that has to be preserved when entering a given foreground level is just the general registers including the Forth instruction counter IC, and the hardware instruction counter. If disk and terminal I/O are to be allowed from more that one execution level, then separate buffers must be maintained.

A lower level task in general does not have to be aware of the existence of higher level tasks, except that higher level tasks effectively slow down the computer. If a low level task hangs up in a loop, higher level tasks will still execute.

Problems with the method include the awkwardness of multi-level I/O, the requirement that the basic Forth routines be reentrant, and that the programmer must see that the completion time of an execution level never exceeds its scheduling interval.

3.13.2 Round-robin Scheduling.

A second popular Forth multiprogramming scheme is the round-robin. As the name suggests, the principle is to allow one task to finish, then to begin the next in a chain. After the last task in the chain completes, the first begins again.

The method is well suited to an environment with multiple users all having equal claim to the computer. Performance degrades gracefully as more tasks are added to the loop.

Proper operation of the round-robin requires that tasks be "cooperative", i.e. willing to relinquish rights to the CPU in a timely way. A task does not have to complete its total function before it allows others to execute, but it must release control frequently so that response time to other users is acceptable.

The round-robin is not well matched to real-time situations in which guaranteed response to external events is required. It also lacks "robustness" in the face of any user who wants to monopolize the CPU.

3.13.3 Scheduling Through Operating Systems.

Multiprogramming facilities are available in most general operating systems. These range from simple foreground-background (dual task) systems like DEC's RT-11 to full-scale priority scheduled systems like RSX-11. For a price, the RSX-11 system will give you priority scheduling, time-slicing between tasks of similar priority, and memory protection between tasks. As discussed in the previous Section, you save implementation expense but suffer greater memory and CPU time overheads to implement Forth multiprogramming through operating systems.

CHAPTER 4

FORTH VOCABULARIES.

4.1 INTRODUCTION.

In this chapter we present definitions of some of the most useful and most standardized Forth words. The vocabulary includes the Forth-79 standard, as well as the Double Number Word Set and the Assembler Word Set that were published in the Forth-79 document. Also included are words that are used in the Caltech Forth versions.

4.2 NOTATION.

The style of notation in this chapter follows the AST.01 document (see Bibliography).

Words are listed in "alphabetical" sequence, based on the ASCII character set. The action of each word is described in concise form: A string of symbols that tells which parameters should be placed on the stack before the word is executed; the word itself; then, any parameters that the word leaves on the stack. A parameter appearing to the right of another on the definition line is meant to be above the other on the parameter stack.

The following symbols are used:

b	Block or screen number.
c	7-bit ASCII character code.
f	Flag: 0=False, non-zero=True. All words which return a flag return 0=False or 1=True.
m n p	
q r s	16-bit integers (or addresses)
u v w	Double-precision (2 cell) numbers.
nnnn pppp	Names of words.
ssss	A string of characters.
vvvv	A vocabulary name.

Preceding a verbal description of each word, certain characters may appear in parentheses. These denote some special action or characteristics, as follow:

C The word may be used only within a colon-definition. A following digit (C0 or C2) indicates the number of memory cells used when the word is compiled, if other than one. A following + or - sign indicates that the word either pushes a value onto the stack or removes one from the stack during compilation. (This is not related to its action during execution.)

V The word is not part of Forth-79.

4.3 STANDARD VOCABULARY LIST.

! m p !

Stores word m at address p.

" " sssss"

(V) Enters a string of up to 63 characters into buffer TEXT (or onto string stack in XED) for use by editor. This word is in editor vocabularies only. Note that a null message (single blank between "s) is not permitted.

% % nnnn p

(V) Like ' (below), except returns the address of the code section of nnnn.

' ' nnnn p

Called "tick". Tick leaves the address of the parameter field of nnnn. This is a "smart" word; inside a colon-definition, it produces code that causes the address to appear on the stack at execution time. The colon definition

: eeee ' nnnn ;

is equivalent to

: eeee [' nnnn] LITERAL ;.

((ssss)

Ignores a comment string terminated by a right parenthesis. A single blank between parentheses is not allowed.

`*` m n * q

16-bit integer multiply with sign.

`*/` m n p */ q

Leaves q= (m*n)/p. Retention of an intermediate 32-bit product permits greater accuracy than the otherwise equivalent sequence: m n * p /.

`*/MOD` m n p */MOD r q

Like */, except leaves both remainder (r) and quotient (q). Has full 32-bit intermediate accuracy.

`+` m n + q

16-bit integer addition with sign.

`+!` m p +!

Adds integer m to value at address p.

`+LOOP` m +LOOP

(C) Adds m to the loop index. If m>0, the loop will terminate if the new index equals or passes the limit. If m<0, the loop will terminate if the new index passes the limit. Loop index checking is unsigned; this allows proper operation with 16-bit addresses > 32767.

`,` m ,

Stores m into the next available dictionary cell, advancing the dictionary pointer by two bytes (one word).

`,CODE` m ,CODE nnnn

(V) Begin a code definition named nnnn as for CODE. Allow space for m cells for parameters before beginning machine code. (' nnnn will give the address of the first reserved parameter.)

- m n - q

16-bit signed integer subtraction q=(m-n).

-TRAILING

m n -TRAILING m p

Eliminate trailing blanks from a text string beginning at address (m) with initial length (n). Returns original value (m) with new length (m).

Types the value on the stack as a signed integer, converted according to the current number base (BASE). If the value is negative, types a minus sign; if positive, types no sign.

(C) Transmits a message of up to 127 characters delimited by " to the selected output device. Note that a null message (single blank between "s) is not permitted.

/ m n / q

16-bit integer divide, q=m/n. The quotient is truncated; any remainder is lost.

/MOD m n /MOD r q

16-bit signed integer divide, q=m/n. The quotient (q) is left on top of the stack, the remainder (r) beneath. The remainder has the sign of the dividend (m).

0< m 0< f

Leaves a true flag (f) if (m) is negative.

0<= m 0<= f

(V) Flag (f) is true if (m) is zero or negative.

0<> m 0<> f

(V) Flag (f) is true if (m) is non-zero.

0= m 0= f

Flag (f) is true if (m) is zero.

0> m 0> f

Flag (f) is true if (m) is positive and non-zero.

0>= m 0>= f

(V) Flag (f) is true if (m) is greater than or equal to zero.

0SET p 0SET

(V) Store zero at location p.

1+ m 1+ q

Increment value (m); q=(m+1).

1+! p 1+!

(V) Add 1 to the contents of address p. Equivalent to p 1 SWAP +!.

1- m 1- q

(V) Decrement variable (m); q=(m-1).

1SET p 1SET

(V) Store one at location (p).

2! u p 2!

Store 32-bit variable (u) at location (p).

2* m 2* q

(V) Double variable (m); q=2*m.

2+ m 2+ q

Add two to variable (m); q=m+2.

2- m 2- q

Subtract two from variable (m); q=m-2.

2/ m 2/ q

(V) Halve variable (m); q=m/2.

2<R u 2<R

Move the 32-bit number (u) from the user stack to the return stack.

2@ p 2@ u

Fetch 32-bit value (u) from location (p).

2CONSTANT u 2CONSTANT nnnn

Defines Forth word (nnnn) which will push 32-bit value (u) on the stack.

2DROP u 2DROP

Eliminates a 32-bit variable from the top of the stack. Equivalent to DROP DROP.

2DUP u 2DUP u u

Duplicates a 32-bit variable on the top of the stack. Equivalent to OVER OVER.

2R> 2R> u

Move the 32-bit number (u) from the return stack from the user stack.

2ROT u v w 2ROT v w u

Rotates 3 32-bit variables, similar to ROT.

2SWAP u v 2SWAP v u

Exchange the top two 32-bit variables, similar to SWAP.

2VARIABLE

2VARIABLE nnnn

Define a Forth word (nnnn) which returns the address of a 32-bit quantity contained in the parameter field. Like VARIABLE, except that four bytes are reserved.

79-STANDARD

79-STANDARD

If this word exists, and can be executed successfully, a minimal Forth-79 system is guaranteed to be available. No parameters.

: : nnnn

Create a dictionary entry for a colon-definition, set compilation mode, and set the context vocabulary equivalent to the current vocabulary.

:> :>

(CV) Switch mode from compilation to execution. Compiles a word address that, at execution, will restore IC and branch to the code beginning after :>. If the code ends with NEXT, the return will be correct. Example: : NNNN ... :> ... (assembly instructions) ... NEXT,

; ;

(C) Terminates a colon-definition and stops compilation.

;CODE ;CODE

(C) Stops compilation and terminates a defining word (nnnn). Switch the context vocabulary to ASSEMBLER in

anticipation of a machine-code sequence. When (nnnn) is subsequently executed to define a new word (pppp), the execution-address of (pppp) will point to the machine code sequence following the ;CODE of (nnnn). Then, subsequent use of (pppp) (or any other word defined by nnnn) will cause this machine-code sequence to be executed. The assembly language equivalent of DOES>.

;S ;S

(V) Stops interpretation of a Forth screen.

< m n < f

Flag (f) is true if (m) is less than (n), in the sense of 2's complement, 16-bit arithmetic.

<= m n <= f

(V) Flag (f) is true if (m) does not exceed (n), in the sense of 2's complement, 16-bit arithmetic.

<> m n <> f

Flag (f) is true if (m) is not equal to (n).

= m n = f

Flag (f) is true if (m) is equal to (n).

> m n > f

Flag (f) is true if (m) is greater than (n), in the sense of 2's complement 16-bit arithmetic.

>: >:

(V) Switch mode from execution to compilation. Assembles instructions that save IC and begin the Address Interpreter just after >:. If the compiled

code ends with ;, the return will be correct.

Example: CODE nnnn ... >: ... (compiled Forth words) ... ;

Note that >: and :> can be used freely in either CODE or : definitions.

>= m n >= f

(V) Flag (f) is true if (m) is greater than or equal to (n) in the sense of 2's complement 16-bit arithmetic.

>IN >IN m

Returns the current character offset (m) in the input text stream, range 0 - 1023.

>R m >R

Pushes (m) onto the return stack. See R>.

? p ?

Prints the value contained at address p in free format, according to the current base. Equivalent to p @ . .

?DUP m ?DUP m [m]

If value (m) is non-zero, push a copy of it on the stack.

@ p @ q

Called "fetch", leaves the contents (q) of memory address (p).

ABORT ABORT

Enter the abort sequence, clearing all stacks, printing a simple error message, and returning control to the terminal.

ABS m ABS q

Leaves the absolute value of a number.

ALLOT n ALLOT

Allocate (n) bytes to the parameter field of the most recent Forth definition.

AND m n AND q

Bitwise logical AND of (m) and (n).

ASH n m ASH r

Arithmetic* shift, result (r) = (n) * 2**(m). If m>0, shift is to left; m<0, to the right.

---

*An arithmetic shift is a shift in which the sign bit is "sticky"; it never changes when data are shifted left. When data are shifted right, the sign bit is copied into successive bits to the right, but the sign itself never changes. In a logical shift the sign bit is treated like any other.

ASSEMBLER

ASSEMBLER

Switch the context vocabulary pointer so that dictionary searches will begin at the Assembler Vocabulary. The Assembler Vocabulary is always chained to the current vocabulary.

BASE BASE p

An integer pointing to the current conversion base value.

BEGIN BEGIN

(CO+) Mark the start of a BEGIN - UNTIL or BEGIN - WHILE - REPEAT loop. The words between BEGIN and its corresponding termination will be repetitively executed until the termination condition is satisfied. Loops

may be nested.

BELL BELL

Activate terminal bell or tone.

BLK BLK p

An integer, equal to the number of the block being interpreted or zero if input is coming from the terminal.

BLOCK b BLOCK p

Leaves the address of a buffer containing Block (b). If the block is not already in memory, it is transferred from disk or tape into whichever core buffer has been least recently accessed. If the block occupying that buffer has been updated, it is rewritten on disk or tape before Block (b) is read into the buffer.

BUFFER b BUFFER p

Obtains a core buffer for block b, leaving the first buffer cell address. The block is not read from disk, and is automatically marked as updated.

C! m p C!

The low order 8 bits of (m) is stored at the byte address (p)

C@ p C@ m

The 8-bit byte at address (p) is returned in the low order part of (m). The high order bits are cleared.

C, n C,

Compile the low-order byte of n into the dictionary and increment the dicitionary pointer by one byte.

CHAIN CHAIN vvvv

Connects the current vocabulary to all definitions that might be entered into Vocabulary (vvvv) in the future. The current vocabulary may not be FORTH or ASSEMBLER. Any given vocabulary may be chained only once, but may be the object of any number of chainings. For example, every user-defined vocabulary may include the sequence, CHAIN FORTH.

CMOVE m n r CMOVE

Move (r) bytes from area beginning at byte address (m) to area beginning at byte address (n).

CODE CODE nnnn

Creates a dictionary entry for a code definition named (nnnn), and sets the context vocabulary to Assembler.

COM m COM q

Leaves the one's complement of (m).

COMPILE COMPILE (non-standard parameter!)

This word provides a way to cause specific data to be compiled into the dictionary. When COMPILE executes (it must be called from within a colon definition), the 16-bit word next following in the address sequence is picked up. This data is not pushed on the stack (as LITERAL would do), but it is stored at the next available dictionary location, and the dictionary pointer is incremented accordingly.

Example - if X is defined by : X COMPILE [0 ,] ;, then executing X compiles a "0" at the current

dictionary location.

CONSTANT

m CONSTANT nnnn

Creates a word which when executed pushes (m) onto the stack. (Since the "constant" (m) may be modified by the sequence: q ' nnnn ! it is oftentimes advantageous to define a variable as a constant, particularly if the variable is accessed more often than it is modified.)

CONTEXT CONTEXT p

An integer that indicates in which vocabulary dictionary searches are to begin.

CONVERT u p CONVERT v q

Convert an ASCII string beginning at memory location (p)+1 to a double precision integer according to BASE. Add the result to (u). The sum is returned as (v), and the address of the first character that could not be converted is returned as (q).

COPY m n COPY

(V) Copy the contents of block (m) into block (n) and mark block (n) as updated.

COUNT p COUNT (p+1) n

The count-byte (n) is extracted from the first byte of a message string beginning at address (p), and left on the stack. The string address is incremented by one to point to the first character of text.

CR CR

Transmit carriage return/line feed codes to the selected output device.

CREATE CREATE nnnn

Creates a skeleton word definition with name (nnnn). As initialized, this word will push the address of the parameter field on the stack, although no parameter field space is reserved by CREATE.

CURRENT CURRENT p

An integer that indicates the vocabulary into which new words are to be entered.

D+ u v D+ w

Double precision (32-bit) addition; w=u+v.

D- u v D- w

Double precision (32-bit) subtraction; w=u-v.

D. u D.

Output a 32-bit value according to current value of BASE.

D.R u n D.R

Output a 32-bit value according to current value of BASE. Align output in a field of (n) characters in width.

D0= u D0= f

Return (f) non-zero if (u) is non-zero.

D< u v D< f

Flag (f) is true if (u) is less than (v) in the sense of 32-bit, two's complement integers.

D= u v D= f

Return non-zero value of (f) if u=v.

DABS u DABS v

Returns (v) equal to the absolute value of (u).

DASL u m DASL v

Arithmetic 32-bit shift left by (m) places. (m) must be positive.

DASR u m DASR v

Arithmetic 32-bit shift right by (m) places.

DATAN u v DATAN w

Return (w) = Arctan(u/v) preserving quadrant information. The result is an angle expressed in Binary Angular Measure (BAM).*

---

*In Binary Angular Measure, 0 degrees = 0, 90 degrees = 40000(8), 180 degrees = -180 degrees = 100000(8), etc. In this way, a fraction of a turn is represented with the greatest possible accuracy by a signed integer.

DCOS u DCOS v

Compute (v) = cosine(u), similar to DSIN.

DECIMAL DECIMAL

Sets the numeric conversion base to decimal mode. (Set BASE to ten.)

DEFINITIONS

DEFINITIONS

Sets the current vocabulary (into which new definitions are placed) to the context vocabulary (the vocabulary currently being used for searches).

DEPTH DEPTH n

Leaves the number (n) of 16-bit words currently on the stack, before (n) is pushed on.

DMAX u v DMAX w

Return (w) equal to the larger of (u) and (v), treated as 32-bit two's complement values.

DMIN u v DMIN w

Return (w) equal to the lesser of (u) and (v), treated as 32-bit two's complement values.

DNEGATE u DNEGATE (-u)

Leaves the negative of a 32-bit quantity; two's complement.

DO n m DO

(C) Begin a loop, to be terminated by LOOP or +LOOP. The loop index begins at (m), and may be modified at the end of the loop by any positive or negative value. The loop is terminated when an incremented index reaches or exceeds (n), or when a decremented index becomes less than (n). Within a loop, the word I will place the current index value on the stack.

Execution of DO places three parameters on the return stack: The starting location of the loop, the index limit, and the index.

DOES> DOES>

(C) Terminates a defining word nnnn, which can subsequently be executed to define a new word (pppp). Subsequent use of (pppp) will cause the words between DOES> and ; to be executed with the parameter-field

address of (pppp) on the stack. Further explained in Section 3.10.

DROP m DROP

Drop the topmost value from the stack.

DSIN u DSIN v

Result (v), scaled in the interval -1 -- +1 (binary point to the right of the sign bit), is the sine of angle (u) in BAM.

DSQRT u DSQRT v

Return (v), the square root of (u). (u) must be a positive value.

DU< u v DU< f

Returns (f) non-zero if (u) is less than (v) in the sense of 32-bit unsigned integers.

DUMP m n DUMP

Dump (n) memory cells beginning at address (m). Dump is in current number base.

DUP m DUP m m

Returns a duplicate of the topmost stack value.

EDITOR EDITOR

The name of the Editor Vocabulary. If that vocabulary is loaded, EDITOR establishes it as the context vocabulary, thereby making its definitions accessible.

ELSE ELSE

Precedes the false part of an IF-ELSE-THEN conditional.

EMIT c EMIT

Send character (c) to the current output device.

EMPTY-BUFFERS

EMPTY-BUFFERS

Marks all block-buffers as empty, without affecting their actual contents. Updated blocks are not flushed.

END-CODE END-CODE

Terminate an assembly-language CODE definition or series of definitions, resetting the context vocabulary to CURRENT. See ASSEMBLER.

EXCHANGE

m n EXCHANGE

(V) Exchange the contents of blocks (m) and (n) and flush.

EXECUTE p EXECUTE

Execute the Forth definition whose code address is (p).

EXIT EXIT

Used in a colon definition, EXIT forces an immediate termination of execution of the definition. Not for used in DO - LOOP constructions.

EXPECT p n EXPECT

Look for a sequence of up to (n) characters to be input from the current terminal. Store these beginning at address (p). An input "return" character will terminate the sequence early, if encountered. One or two "null" characters (zero bytes) will be appended to the sequence in memory.

FILL p n m FILL

Treat (m) as a byte value and store (n) copies of it into memory starting at address (p). Do nothing if (n)

is less than or equal to zero.

FIND FIND p

Search for the word whose name is the next "token" in the text input stream. If the word can be found in either CONTEXT or FORTH vocabularies, leave its address (p); otherwise leave zero.

FORGET FORGET nnnn

Delete the word (nnnn) and all dictionary entries following it. Although (nnnn) must be in the context vocabulary to be found, the words that follow it are deleted no matter which vocabulary they belong to. Normally, FORGET should not be used within a colon-definition, as it is not a compiler directive.

FORTH FORTH

Make FORTH the context vocabulary. Since FORTH cannot be chained to anything, it becomes the only vocabulary that is searched for dictionary entries.

HERE HERE p

Leave (p), the address of the next available dictionary location.

HEX HEX

Switch the number base to hexadecimal.

I I m

(C) Push the topmost return stack value onto the user stack without disturbing the return stack. Typically I is used to return the index of an innermost DO-loop, but it can also be used to access values pushed onto

the return stack by >R.

IF f IF ... ELSE ... THEN or
f IF ... THEN

(C2+) IF is the first word of a conditional. If f is true (non-zero), the words following IF are executed and the words following ELSE are not executed. The ELSE part of the conditional is optional. If f is false (zero), words between IF and ELSE, or between IF and THEN when no ELSE is used, are skipped. IF-ELSE-THEN conditionals may be nested.

IFEND IFEND

Terminates a conditional interpretation sequence begun by IFTRUE.

IFTRUE f IFTRUE ... OTHERWISE ... IFEND

Unlike IF-ELSE-THEN, these conditionals may be employed during interpretation. In conjunction with [and], they may be used within a colon-definition to control compilation, although they are not to be compiled. These words cannot be nested.

IMMEDIATE

IMMEDIATE

(CV) Set the precedence bit of the word just defined in the dictionary.

J J m

(C) If the current DO - LOOP is nested within another DO - LOOP, J may be used to obtain the index of the outer loop.

K K m

(C) If the current DO - LOOP is nested within two DO - LOOPS, K will return the index of the outermost loop.

KEY KEY c

Return the next character available from the current input device.

LEAVE LEAVE

Causes the termination of a DO - LOOP (+LOOP) the next time the index is checked. This is done by adjusting the limit value stored on the return stack.

LINE m LINE p

Leaves the character address (p) of the beginning of line (m) for the block whose number is contained at BLK.

LIST b LIST

List the block (b) as 16 lines of 64 ASCII characters on the selected output device. Set SCR to (b).

LITERAL n LITERAL

Compile the address of the code routine called LITERAL followed by the value (n) into the dictionary. When the currently defined word is executed, (n) will be pushed on the stack.

LOAD b LOAD

Begin interpreting block (b).

LOOP LOOP

(C) Increment the DO-loop index by one, terminating the loop if the new index is equal to or greater than the limit.

LSH n m LSH r

Logical shift left (m) places. (m) may be negative.

MAX m n MAX q

Leaves (q), the greater of (m) and (n).

MIN m n MIN q

Leaves (q), the lesser of (m) and (n).

MK! MK!

(V) Mark the present value of DP. Equivalent to HERE MKVAR !. Useful in assembler programming for passing parameter addresses. See MK@.

MK@ MK@ n

(V) Obtain the value of DP that was last marked with MK!. Equivalent to MKVAR @. Example: MK! 123456 , CODE nnnn S -) MK@ P MOV, NEXT, This PDP-11 routine will push 123456 on the stack.

MOD m n MOD r

Leaves the remainder of (m)/(n), with the same sign as (m).

MOVE p q n MOVE

Moves the contents of (n) 16-bit words beginning at address (p) into (n) words beginning at address (q). The order of transfer is lowest-first.

NEGATE n NEGATE (-n)

Leave minus the number (n) on the stack, two's complement.

NEXT, NEXT,

(V) An assembler word that may be used to terminate a

CODE word. It invokes the Address Interpreter. In OVRO PDP-11 versions, NEXT, assembles a "jump indirect through IC and increment IC" instruction.

NOT m NOT f

Invert a boolean condition. Equivalent to 0=.

NUMBER NUMBER

Convert a character string left in the dictionary buffer by WORD as a number, returning the result in registers, internal temporary locations, or on the stack. The appearance of characters that cannot be properly interpreted will cause an error exit.

OCTAL OCTAL

Set the number base to octal.

O. n O.

(V) Type n as an unsigned octal number, regardless of current value of BASE. See OO.

OR m n OR q

Bitwise logical inclusive OR; q=m or n.

OR! m p OR!

(V) Form the logical OR of (m) and the contents of (p). Store at address (p).

OTHERWISE

OTHERWISE

An interpreter-level conditional word. See IFTRUE.

OVER m n OVER m n m

Copy the stack value (m) under the top value (n) onto the top of the stack.

PAD PAD p

Leaves the address (p) of a 64-byte buffer that is used for intermediate storage during some string processing functions.

PAGE PAGE

Clears the terminal screen or performs a similar action on the current terminal.

PICK n PICK m

Returns (m), the (n)-th stack value beneath the current top stack value, not counting (n) itself. (2 PICK is equivalent to OVER.)

PRINT p n PRINT

Transmit (n) characters to the selected output printer starting at character address (p), which will have been placed on the stack or in an internal register by COUNT.

PRINTER PRINTER

Select a hard-copy printer as the output device for all output directed through EMIT or PRINT. See TERMINAL.

QUERY QUERY

Accept input characters from current input device and place into terminal buffer. Input is terminated by a 'return' character or by the transmission of 80 characters.

QUIT QUIT

Clear the return stack, force execution mode, and continue by interpreting text from the terminal. No error message results. This is a "softer" reset than

ABORT.

R> R> n

Pop the topmost value from the return stack and push it onto the user stack. See >R.

R@ R@ n

Take the top value on the return stack and push on the user stack; the return stack is not altered.

REPEAT REPEAT

(C) Signifies the end of a loop in a BEGIN - WHILE - REPEAT loop. Causes control to pass to the point just following BEGIN.

ROL n m ROL r

Rotate (n) left (m) places. Bit 15 (the sign) rotates into bit 0. If (m) is negative, the rotation is to the right.

ROLL u[n] u[n-1] ... u[1] n ROLL u[n-1] ... u[1] u[n]

Extract the (n)-th value from the stack, leaving it on top and moving the remaining values into the vacated position. The depth of the stack is unchanged. (3 ROLL is equivalent to ROT; 2 ROLL is equivalent to SWAP; 1 ROLL is a null operation; 0 ROLL is undefined.)

ROT m n p ROT n p m

Rotate the topmost three stack values so that the previous top value becomes the second; the second becomes the third; and the third becomes the top.

S>D n S>D u

Convert a 16-bit number (n) into a 32-bit number (u) by

extending the sign to the left.

SAVE-BUFFERS

SAVE-BUFFERS

Write all blocks that have been flagged as "updated" to disk or tape.

SCR SCR p

Returns the address (p) of a variable that indicates the most recently edited screen.

SEMI, SEMI,

(V) This word must be used to terminate ;CODE words.

SET m p SET nnnn

(V) Defines a word (nnnn) which, when executed, will cause the value (m) to be stored at address (p).

SHOW m n SHOW

(V) Type blocks (m) through (n) at the terminal, 3 blocks to a page (for hardcopy terminals).

SIGN n SIGN

If (n) < 0, put a minus sign ("-") into the number output string.

SPACE SPACE

Send a blank character to the current output terminal.

SPACES m SPACES

Send (m) blanks to the current output terminal.

STATE STATE p

Returns the address (p) of a variable that contains the flag that indicates whether the input text stream is being compiled or executed. A non-zero value corresponds to the compilation state.

SWAB n SWAB m

(V) Exchange the left and right bytes of (n).

SWAP n m SWAP m n

Exchange the topmost two stack values.

TERMINAL

TERMINAL

Select the terminal as the output device, cancelling any previous selection of the printer.

THEN THEN

(CO-) Terminates an IF-ELSE-THEN conditional sequence.

TYPE m n TYPE

Transmits (n) characters to the current output device, starting at the character address (m). See COUNT, PRINT.

U* m n U* u

Leaves the unsigned 32-bit product, (u), of two unsigned 16-bit numbers, (m) and (n).

U. m U.

Type (m) as a 16-bit unsigned number according the current value of BASE.

U/MOD u n U/MOD r q

Leaves remainder (r) and quotient (q) of the result of an unsigned division of the 32-bit value (u) by the 16-bit value (n).

U< m n U< f

Like <, U< compares (m) and (n) and returns a flag that is true (non-zero) if m < n; however, the comparison

treats the inputs as unsigned values (integers in the range 0 - 65535).

U<= m n U<= f

Like <=, but an unsigned comparison.

U> m n U> f

Like >, but an unsigned comparison.

U>= m n U>= f

Like >=, but an unsigned comparison.

UNTIL f UNTIL

(C) Signals the end of a BEGIN - UNTIL loop in a colon definition. If (f) is true (non-zero), the loop is terminated; if not, execution continues at the first word following the corresponding BEGIN.

UPDATE UPDATE

Flag the most-recently referenced block as updated. The block will later be transferred automatically to disk if its buffer is needed to store a different block. See SAVE-BUFFERS.

VARIABLE

VARIABLE nnnn

Creates a word (nnnn) which, when executed, pushes the address of a variable onto the stack. Two bytes are reserved to hold the variable.

VOCABULARY

VOCABULARY vvvv

Define a vocabulary name. Subsequent use of (vvvv) will make (vvvv) the context vocabulary. The sequence vvvv DEFINITIONS will make (vvvv) the current

vocabulary, into which future definitions are placed.

WHILE f WHILE

If (f) is true (non-zero), execution proceeds normally in a BEGIN ... f WHILE ... REPEAT loop - through to the REPEAT is encountered. After REPEAT, execution loops back to the word following BEGIN. If (f) is false, however, execution skips out of the loop, to the word following REPEAT.

WORD c WORD p

(CN) Read the next word from the input stream, up to 63 characters or until the delimiter (c) is found, storing the packed character string in an internal buffer. The address (p) points to the beginning of the buffer. The first byte of the buffer contains a count of characters in the buffer. The buffer is terminated by an occurence of (c), or by a null (0).

XOR m n XOR q

Leave (q), the bit-wise logical exclusive OR of (m) and (n).

[[

Stop compilation. The words following the left bracket in a colon-definition are executed, not compiled. Permits calculations to be made during compilation.

[COMPILE] [COMPILE] nnnn

(C) Force the compilation of the word (nnnn). This is the way to compile an "immediate" word.

]]

Resume compilation. Words following the right bracket are compiled.

4.4 SPECIAL VOCABULARIES.

Of the vocabularies presented here, only the standard editor is generally used outside of Caltech-OVRO systems. The others, however, are frequently used in our local systems.

4.4.1 Standard Editor.

The "standard" Forth editor is a very simple editor based on substitution of fixed-length lines in the fixed-format block. There are 16 lines of 64 characters in each Forth block. The following vocabulary is available with the standard editor.

" " ssss"

Copies string (ssss) into buffer TEXT. String is padded to the right with blanks as needed to make 64 characters.

((ssss)

Copies string (ssss) into TEXT like ("), except that a right parenthesis [)] serves as the delimiter.

BLK BLK p

An integer that specifies the number of the block (screen) you're currently working with.

Example: Type 144 BLK ! to edit block 144.

BT BT

Type the current block. Equivalent to BLK @ LIST.

D n D

Delete line (n) from the current block and move lines (n)+1, (n)+2, ..., 16 down one line. Line 16 is filled with blanks. The old contents of line (n) are moved into buffer TEXT.

I n I

Lines (n)+1, (n)+2, ..., 15 are moved down one line. (Line 16 is lost.) The contents of TEXT are moved into line (n)+1.

R n R

The contents of TEXT are moved into line (n).

T n T

Type line (n).

4.4.2 Character Strings.

Character string manipulations are a central part of more sophisticated text editors. Standard Forth has no explicit support of strings. The following vocabulary is one approach to providing string handling in Forth.

Variable length character strings (0-63 characters) may be placed on a special string stack (which has a fixed maximum depth). Various operations, prefixed by ("), operate on this stack.

" " ssss"

Push a literal string (ssss) onto string stack. Similar to " in standard editor. In compile mode: Compile ssss into the dictionary with a call to a

string literal routine that will push ssss onto the stack at execution time.

((((ssss)

Like " except the delimiter is). ((lets you enter quotes in a text string.

-TRAILING ssss -TRAILING tttt

String (tttt) is (ssss) with all trailing blanks removed.

=STRINGS

rrrr ssss =STRINGS f

Compare strings (rrrr) and (ssss), return (f)=1 if equal (including in length), 0 otherwise.

"! ssss p "!

Pop (ssss) from the string stack and store at location (p).

"@ p "@ ssss

Get string (ssss), located at (p), and push it on the string stack. (Byte 0 of the string is its length.)

"-LEN "-LEN n

Get length (n) of second string on string stack.

"C! c n "C!

ASCII character (c) replaces (n)-th character of top string.

"C@ n "C@ c

Retrieve (n)-th character from top string, push its ASCII value (c) on Forth stack. Character 0 is the string length.

"CAT rrrr ssss "CAT tttt

Strings (rrrr) and (ssss) are concatenated to form string (tttt).

"CLR "CLR

Clear the string stack. Note: the string stack is not cleared by ABORT.

"INDEX ssss tttt "INDEX m

Search string (ssss) for the first occurrence of (tttt) as a substring. Returns character position of match if found, 0 otherwise.

"LEN "LEN n

Get length (n) of top string on string stack.

"LEN! n "LEN!

Set length of top string to (n). Equivalent to n 0 "C!.

"LINE n "LINE ssss

String (ssss) is drawn from line (n) of the block whose number is in BLK. Trailing spaces are deleted.

"LINE! ssss n "LINE!

String (ssss) is stored in line (n) of BLK. Blanks are added to the right to make 64 characters.

"NULL "NULL ssss

Push null string (ssss) (length 0) on string stack.

"PAD rrrr ssss n "PAD tttt

String (rrrr) is padded to the right using the first character of (ssss) so that the resulting string (tttt) is (n) characters long.

"STRING ssss "STRING nnnn

Like CONSTANT, define (nnnn), which, when executed, will push (ssss) on the string stack.

"SUBSTR

ssss n m "SUBSTR tttt

New string (tttt) is the substring of (ssss) beginning at character (n) and ending with character (m).

"SUBSTR! rrrr ssss n m "SUBSTR! tttt

Result is string (rrrr) with string (ssss) inserted instead of substring (n) through (m) of (rrrr). The length of (ssss) does not have to equal the length of the substring to be replaced.

"TYPE ssss "TYPE

Type (ssss) and pop off string stack.

4.4.3 The Extended Editor.

The Forth Extended Editor (XED) is a superset of the standard editor developed at Caltech. In addition to the line-at-a-time commands, it allows you to search for character strings, alter strings identified by context, etc. XED uses the Character Strings vocabulary described above.

FT ssss FT

Find the first occurrence of (ssss) beginning at the current line number (L#) in the current block (BLK) and type the whole line containing the string. If a match is not found in the current block, continue at BLK + 1 etc. (You have to type 2 CTRL-Cs to stop in RT11 or

RSX11.)

Example: " THIS" FT to find the first occurrence of "THIS" in or after the current block.

FR rrrr ssss FR

Find the first occurrence of (rrrr) in the current block beginning at the current line; replace it with (ssss). The resulting line is truncated at 64 characters.

Example: " THIS" " THAT" FR to replace the first occurrence of "THIS" with "THAT".

FD ssss FD

Find the first occurrence of (ssss) in the current block beginning at the current line; delete this substring of the line. Pad the line back to 64 characters with blanks.

FI rrrr ssss FI

Find the first occurrence of (rrrr) as above; insert (ssss) immediately following (rrrr). Truncate the line at 64 characters.

HT n HT

Hold line (n) of current block on string stack and type.

HR n HR

Replace line (n) with the string on the stack (like R), but save the old contents of line (n) on string stack.

HD n HD

Delete line (n) (like D), but hold its former contents on the string stack.

HI n HI

Insert string on line following (n) (like I), but hold old contents of line 16.

LT LT

Type current line number and line.

BT BT

Type current block. Reset line number to 1.

L? L?

Type current line number.

L1 L1

Set current line to 1.

HOLD n m HOLD

Put lines (n) - (m) of current block on string stack.

UNHOLD n m UNHOLD

Replace lines (n) - (m) from string stack.

+B +B

Increment BLK by 1.

-B -B

Decrement BLK by 1.

ENTER ENTER

Beginning at the current line of the current block, insert text exactly as typed. Each line is terminated by the user typing a carriage return, which fills out the current line with blanks and advances L#. Typing more than 64 characters between carriage returns results in a "bell" and automatic line advance. The line number and a backslash are output before each line is input. Input terminates with a CTRL-Z character.

BLK automatically advances after line 16 of the current block is entered.

CLR-BLK n CLR-BLK

Set block (n) to blanks.

4.4.4 Deferred Operations.

A class of operations modelled on the addressing modes of the PDP-11 has been developed by H. W. Hammond. These are particularly valuable when you need to work with pointers to access successive elements of data structures. Straightforward generalizations to data types other than 16-bit integers are possible.

)! m p)!

Store (m) at the address (q) found at location (p). Equivalent to m p @ !.

)+! m p)+!

Store (m) at address (q) found at location (p), then increment (p) by 2 bytes. (PDP-11 "auto-increment") Equivalent to m p @ ! 2 p +!.

)+@ p)+@ m

Get the contents of (q) found at location (p), then increment (p) by 2 bytes. Equivalent to p @ @ 2 p +!.

-)! m p -)!

Decrement contents of (p) by 2 bytes, then store (m) at location (q) whose address is found at location (p). ("Auto-decrement") Equivalent to -2 p+! p @ !.

)@ p)@ m

Get the contents of address (q) which is found at

location (p). Equivalent to p @ @.

)@! p)@!

Equivalent to p @ @ p !.

-)@ p -)@ m

Decrement contents of (p) by 2, then get contents of location (q) whose address is found at location (p). Equivalent to -2 p +! p @ @.

4.4.5 File System.

The typical Caltech-OVRO Forth system has one "user" at a time, but many users sequentially in time. In this environment, confusion over allocations of block storage is a significant problem. Sometimes, many non-expert persons potentially need to edit blocks on the same disk. The Forth File System (FFS) provides one approach to alleviating the problem of disk allocation and protection. This system is another example of the extensibility of Forth. We provide a brief description of the technique and the vocabulary of FFS.

FFS divides the Forth block file (which may be a file within the file structure of an operating system) into "user files". Each user file may contain up to 512 blocks, numbered 0 - 511. A user refers to his blocks just as in Forth without FFS, i.e., through BLOCK, LIST, etc. Block numbers that the user deals with are considered logical block numbers; FFS maintains a map, called the User
File Directory or UFD, of correspondences between logical and physical block numbers. The "physical" block number refers to the location of a block as it exists relative to the beginning of the

complete block file, as understood by the operating system. Physical blocks may be arbitrarily assigned to a users logical block space (logical disk).

A table of available disk blocks is maintained in a block called "AVAIL". This is a bit map in which each bit signifies the availability (if 1) of a particular physical block. A user, after his UFD is set up, may request up to 512 blocks to be placed in his file. Initially, no blocks are allocated to the user; i.e., any block reference will cause an error message. The user must assign himself blocks using the word ASNBLK. Blocks are assigned one at a time and are given specific logical block numbers in the user's file. Blocks do not have to be assigned contiguously; blocks 0, 1, and 3 may be assigned (using ASNBLK) while block 2 is unsassigned. Thus the user only needs to assign the particular logical blocks he will be using.

An unneeded block can be returned to the available pool with the word RLSBLK.

A user file is specified by a numeric constant (1 - 511). A suitable constant word would normally be defined to specify the file symbolically, e.g., SYSTEM, STRINGS, VLBI, etc. At all times, Forth/FFS maintains a disk "context" which specifies the user file from which all blocks are taken. The user may change user files with the word DISK, e.g., SYSTEM DISK. The file must have been previously defined.

Special user files are defined for software packages such as editors, floating point, diagnostics, etc. A special word has been defined to load such packages: /LOAD. If the user types

DIAGNOSTICS /LOAD, the diagnostics user file is loaded beginning with logical block 0. /LOAD preserves context, i.e. if the current user file is SYSTEM, SYSTEM will be current after a /LOAD command. Thus /LOADs may be nested.

A separate (FILES) vocabulary is available to create and manipulate UFDs. It is intended that only system maintainers ("experts") will need to run (FILES).

A system using FFS has the following small added vocabulary for all users.

/COPY m n r /COPY

Copy block (m) from user file (n) to block (r) of the current user file.

Example: USR1 DISK 13 USR2 10 /COPY

copies block 13 of disk USR2 to block 10 of disk USR1.

/EXCHANGE

m n r /EXCHANGE

This word is like /COPY, but the two blocks are exchanged.

ASNBLK n ASNBLK

Get a block from the available pool, clear it with blanks, and assign in the logical block number (n) in the range 0 - 511 in the current user file. Block (n) must previously have been unassigned.

DISK n DISK

Set the current user file (disk context) to (n). Values of (n) are normally defined by constants giving the symbolic names of the user disks, e.g., SYSTEM,

STRINGS, EDITOR, etc.

RLSBLK n RLSBLK

Deassign logical block (n) from the current user file and return it to the available pool.

In addition to the new words described above, some standard disk-related Forth words are modified to support FFS. These are BLOCK, COPY, EXCHANGE, LIST, LOAD, and SHOW. The modified words refer only to a user's current logical disk.

CHAPTER 5

ADVANCED TOPIC: LARGER FORTH SYSTEMS

5.1 WHY LARGER FORTH SYSTEMS?

The "classical" Forth computer is a minicomputer having 16-bit data words and 16-bit addressing. Typical of such systems would be the DEC PDP-11, the Hewlett-Packard HP1000, the Data General Nova/Eclipse, etc. Common 8-bit microcomputers, such as the 8080 and the Z80, employ 16-bit addressing, and can also be made to perform 16-bit arithmetic.

Addressing capability (the width of address fields) is particularly important for Forth, because Forth intermixes addresses and data on the same stacks. Users are expected to do their own address arithmetic, for example, when indexing data arrays.

The newer generation of 16-bit microcomputers (such as the Motorola 68000 and the Intel 8086) respond to the requirement for address spaces much larger than the 64K bytes allowed by 16-bit addressing. A complete, general address reference for the new microcomputers is typically 32 bits wide. In this regard, they are similar to the larger scale "midicomputers", such as DEC's VAX-11 and Data General's MV/8000, which also have 32-bit addressing.

Two recent Forth systems have faced the addressing problem for the new machines, and adopted a 32-bit word length for all normal Forth data. These are polyFORTH/32 (a trademark of Forth, Inc.) for the 68000 and JPL's Forth for the DEC VAX-11. PolyFORTH/32 is described in the article "Design Considerations

for a 32-bit Forth" by Mike La Manna and Ray Van de Walker in the Proceedings of the 1982 FORML Conference. (See Bibliography.)

5.2 FORTH FOR VAX-11.

The VAX implementation of Forth is interesting in several ways, since it not only confronts the 32-bit addressing problem, but it interacts in an articulate manner with the complex VMS operating system. We will describe the highlights of JPL/VAX Forth as an example of how Forth may be effectively employed in larger computer systems.

5.2.1 TEXT FILES

It is very convenient, when Forth runs under an operating system, to employ "line-structured" files for text and source code. Most larger operating systems represent text files with variable-length records, each of which corresponds to a single printed line. Furthermore, special formatting characters, such as "tab" and "form feed", may be used to position text without redundant blank characters. With a slight increase in the complexity of the Forth system (and some loss in compatibility with smaller Forth systems), line-structured files can replace or supplement the traditional fixed-length block-structured Forth file.

In addition to economizing on disk storage, line-structured files have several further advantages. The files can easily be formatted according to the normal formats of the operating system. Thus all the normal non-Forth system editors can operate on Forth source data. Files can be interchanged with

user programs written in other languages, such as Fortran or Pascal.

In practice, we have found, one of the significant advantages of line-structured files with tab characters and an indefinite number of lines per file is that programmers find it convenient to write nicely indented, logically clear Forth code. This has always been a problem with the standard fixed 16 line screen of earlier Forth systems, in which there is always a temptation to pack definitions tightly into the minimum possible space.

5.2.2 DATA WIDTH

As mentioned above, a 32-bit address must be convenient to manipulate in Forth systems for the newer microcomputers and for the VAX-class midicomputers. It is very awkward to deal with data having mixed lengths on the same stack, so it is natural to consider making a 32-bit stack width standard for these machines.

What are the penalties? There is an obvious penalty in that more memory will be used if all (or most) data take 32 bits when 16 bits might be adequate in many cases. The processor may be slower in operations involving 32-bit data, particularly if the data buses are only 16 bits wide.

Double precision (32-bit) operations on the Intel 8086 are definitely slower than single precision, especially for the 8088 version which has only 8-bit data paths. The memory segmentation scheme of this processor makes it somewhat awkward to deal with data sets greater than 64K bytes in length. For these reasons, a

number of 8086-based Forth systems have chosen to retain 16-bit addressing and not to support the full addressing capacity of the processor.

The Motorola 68000, however, has more of the attributes of a true 32-bit computer, having 32-bit data and address registers, for example. Except for extra bus cycles required, there is not much speed penalty in double precision over single precision operations. In fact, as La Manna and Van de Walker point out, an address interpreter (NEXT function) using 16-bit addresses is considerably slower than the corresponding 32-bit routine because of the lack of an instruction to convert from 16- to 32-bit addresses without sign extension. The full 24-bit address space is available without segmentation. On balance, the 68000 appears well-suited to 32-bit Forth implementations.

The VAX-11 is designed as a true 32-bit computer, having 32- or 64-bit data buses, depending on model number. There is essentially no performance cost in choosing 32- over 16-bit arithmetic, and memory space in the virtual VMS environment is quite inexpensive. The choice of 32-bit data width for JPL/VAX Forth was easy.

5.2.3 ADDRESS INTERPRETER

JPL/VAX Forth has abandoned the Forth address interpreter in favor of using the VAX JSB (jump to subroutine) instruction. This is a major departure from earlier Forth systems, but there are few, if any, cases in which this change is apparent to the user at the colon-definition level. Incidentally, this development has established that the "threaded code" technique

is not fundamentally required in Forth systems.

Why not use the standard address interpreter? In a 32-bit environment, address sequences consist of 32-bit fields, each specifying a particular address from a possible space of 2^{32}, or over 4 gigabytes. Of course, no Forth program will approach this size, and many bits of each address will be zero. The memory "wasted" on wide address fields can be reclaimed, and a substantial performance increased can be gained, by compiling complete VAX instructions instead of 32-bit addresses.

The JSB instruction has several variants. If the distance between the JSB instruction and the routine being called is closer than +127 or -128 bytes, the byte offset form of JSB is used; this requires only two bytes (16 bits) of memory. More commonly, the spacing between call and routine will be greater than 128, and a word offset form of JSB can be used. With this variant, taking three bytes (24 bits), a JSB can call a routine as far away as +32767 or -32768 bytes. If this is insufficient, a longword (32-bit) form is available.

Compiling JSB calls optimized for the shortest lengths compatible with the required offsets allows colon definitions to take less than 32 bits on average. Performance is increased since the NEXT function (address interpreter) is effectively replaced by the one-byte RSB (return from subroutine) instruction.

5.2.4 IN-LINE CODE

Traditional Forth provides three levels of programming for

the user: direct execution from the terminal or from screens via the text interpreter; execution of colon definitions via the address interpreter; and execution of CODE definitions through the address interpreter. JPL/VAX Forth adds a fourth level, the "in-line" definition with two new defining words ICODE and I:.

When you are compiling a word and you refer to a previously-defined in-line word, a JSB instruction is not compiled. Instead a copy of the parameter field of the word you are referring to is placed in the parameter field of the word you are now compiling. This has always been a possible technique for older Forth systems, but without the JPL/VAX JSB technique, there would have been little advantage. But with JSB compilation, a transition from "compiled" sequences of addresses (JSB instructions) to machine code inserted in-line costs nothing, since the VAX CPU interprets either as a valid list of instructions. In-line compilation for functions like +, HERE, DUP, etc, costs very little since these functions often take no more memory than a JSB instruction. Performance is improved because the overhead due to the JSB/RSB instructions is eliminated.

Figure 5.1 summarizes JPL/VAX Forth compilation.

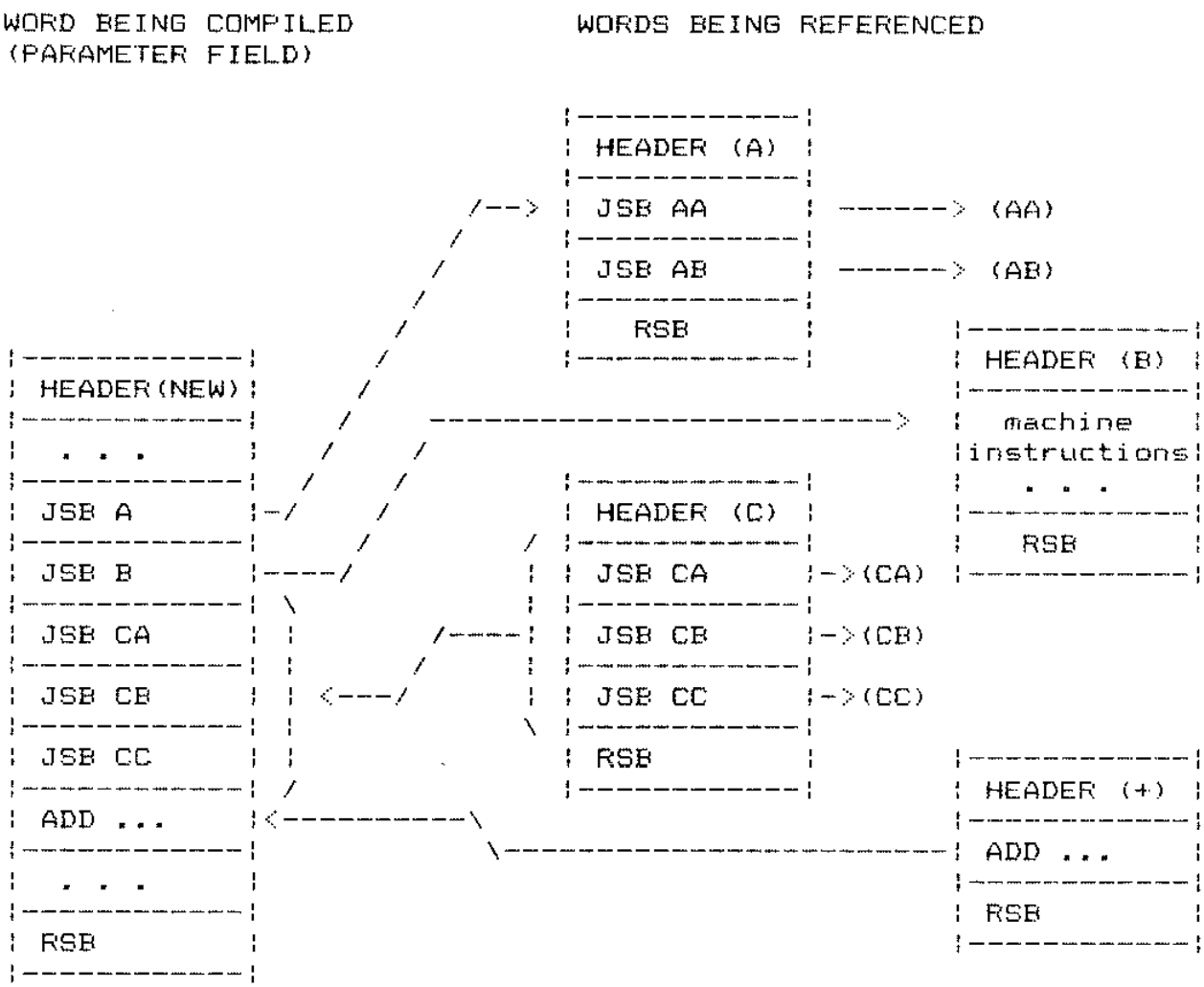


Figure 5.1 VAX Colon Definition with In-line refences.

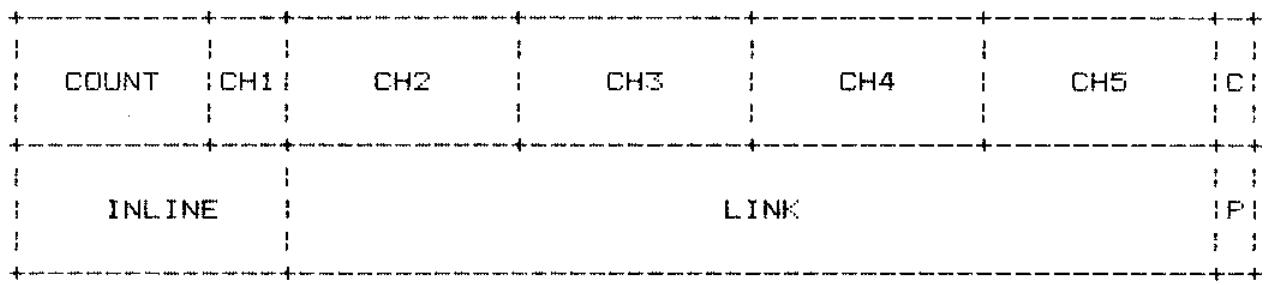
The Figure corresponds to the following definitions:

```
: A  AA AB AC ;
CODE B <machine instructions> NEXT,
I: C CA CB CC ;
ICODE + <source, destination fields> ADD, NEXT,
: NEW <...> A B C + <...> ;
```

Note that the parameter fields (excluding terminal RSBs) of the in-line words C and ± are copied into the parameter field of the new word.

Fields have been allocated in the JPL/VAX Forth word header

to indicate whether a word is an in-line word, and, if so, what the length of its parameter field is. The format of the header is shown in Figure 5.2.



WORD 1:	SIZE (bits)	
COUNT	5	ENTRY LENGTH TRUNCATED AT 32
CH1	2	TWO MSB OF SIX BIT CHARACTER FOUR LOW ORDER BITS ARE A THREAD FOR A 16 WAY INTERLEAVED DICTIONARY
CH2	6	CHARACTER 2 OF ENTRY
CH3	6	CHARACTER 3 OF ENTRY
CH4	6	CHARACTER 4 OF ENTRY
CH5	6	CHARACTER 5 OF ENTRY
C	1	"SMUDGE" BIT, SET DURING ENTRY DEFINITION
WORD 2:		
LINK	24	DISPLACEMENT TO THE LAST DEFINED ENTRY IN THIS THREAD
P	1	PRECEDENCE BIT, SET FOR COMPILER DIRECTIVE

Figure 5.2. VAX definition Header Format.

Definition names are treated much as they are in PDP-11 Forth: seven-bit ASCII characters are compressed into six-bit fields, and the leading character is used as a key into a 16-way threaded dictionary structure.

5.2.5 OPERATING SYSTEM INTERFACE

In a complex environment such as the VAX/VMS operating

system, the user may demand correspondingly more capability from his Forth system. The JPL/VAX Forth system attempts to provide this flexibility at two levels: the user command level, and the system service level. At the user command level, the ability to invoke subprocesses to run any standard VMS utility programs, or even to run other language processors is available through the SPAWN word. Forth text files are normally edited by calling a VMS editor (EDT) with this mechanism. The Forth interpreter can be used as a form of command line interpreter under VMS.

A great variety of system service routines is available through VMS; these include mathematical routines (square root, logarithms, trigonometric functions, etc.), string manipulations, memory management operations, the high level RMS file system, and basic I/O ("QIO") calls. Conventionally, these routines are invoked through a Macro assembler and a complex macro library which translates a programmer's statements like

```
$QIOW_S CHAN=@TTCHAN(R9),EFN=RAB$L_CTX(R3),-
        FUNC=#IO$_SETMODE!IO$M_CTRLYAST,-
        P1=CYAST,P2=R9,P3=#PSL$C_EXEC
```

into a series of MOV and PUSH instructions that set up parameters, selected from a wide range of possible values and formats, finishing with a CALL to the appropriate system routine. References to symbolic values, such as IO$_SETMODE, are evaluated either from the macro library or at the time the object program is linked for execution.

In JPL/VAX Forth, links to most VMS system routines are made through address tables in the Forth kernel. (The tables are filled in by the VMS linker.) Since VMS is a virtual memory system, there is little overhead incurred by linking to many

unused routines in system memory space.

It would be possible to define each possible symbolic system call or parameter value as a separate Forth word, but the number of possible words is quite large. A better scheme has been developed for such references. For example, there are numerous return status codes in VMS prefixed by SS$, such as SS$_NORMAL indicating normal completion of a routine. In Forth, a word SS$ is defined which takes the literal token following in the input stream as a modifier. Thus

SS$ NORMAL

would produce the value corresponding to the VMS SS$_NORMAL symbol. Tables of modifiers and values are established for each prefix type, but these tables are not linked in the dictionary; therefore dictionary search time is not increased, and dictionary headers are not required.

APPENDIX A

PDP-11 IMPLEMENTATION.

A.1 GENERAL CHARACTERISTICS.

The DEC PDP-11 is a popular 16-bit computer architecture that is available in many models. Caltech-OVRO operates 4 types of PDP-11: two PDP-11/40s (VLBI Processor and 10 m telescope control), a PDP-11/20 (27 m telescopes), numerous LSI-11/03s for control of three 10 m antennas, the 40 m antenna, and special equipment, and a PDP-11/05 used for the 1024-channel autocorrelation receiver.

Several Forth systems have been developed for these machines. One (for the 11/20) runs as a standalone system using 9-track magnetic tape for block I/O. Other systems have disk storage and so can run the DEC operating systems, RT-11 and RSX-11/M. The LSI-11 systems are normally operatored without operating systems or disks; they are "down-line loaded" from a larger computer over serial communications lines. PDP-11 Forth is also found running on the 32-bit VAX-11 computers in their PDP-11 compatibility mode.

PDP-11s use the standard 7-bit ASCII character set with one character right-justified in an 8-bit byte. PDP-11 Forth recognizes certain characters for control purposes:

CHARACTER	FUNCTION
CTRL-C	Interrupts execution of any program and returns control to the keyboard. Two CTRL-Cs may be required if the program is not listening to the keyboard. RT-11: RT-11 types "." and you may type any monitor command (e.g. REENTER or RUN). REENTER will let you resume Forth in most cases. RSX-11: RSX types "MCR>" and you may type any monitor command, such as ABORT. Forth can not be reentered after aborting.
CTRL-O	Inhibits terminal output from a running program, but program continues. Allows you to skip lengthy listings. A second CTRL-O turns on output again.
CTRL-Q	After you type CTRL-S to stop type out, you may type CTRL-Q to resume.
CTRL-S	Stops terminal output from a running program in such a way that no output will be lost. The program hangs up after the output buffer is full. CTRL-Q may be used to restart output.
CTRL-U	Cancels the entire line you have just typed in. Only effective before you type "return".
RUBOUT	Cancels the last character you have just typed in. Same as DEL or DELETE.

The 8 PDP-11 registers are allocated according to the following table:

REG.	NAME	FUNCTION
0	-	General Use
1	T	Stack top or General
2	TT	Multiply/Divide or General
3	-	General Use
4	S	Forth Stack Pointer
5	IC	Forth Instruction Counter
6	R	Forth Return Stack Pointer and PDP-11 Hardware Stack Pointer
7	-	PDP-11 Program Counter

A.2 DICTIONARY FORMAT.

The PDP-11 dictionary format was featured in Section 3.3 of this Manual and will not be repeated here.

A.3 ASSEMBLER.

Three types of instructions are supported by PDP-11 Forth: zero-, one-, and two-operand instructions. Forth words 1OP and 2OP are provided to define single and double operand instructions, respectively.

1OP defines words (like CLR,) which require one argument on the stack. The argument specifies the addressing mode and register. For example

```
3 CLR,
```

is equivalent to the Macro-11 line

```
CLR R3,
```

which clears register 3.

For more complicated types of addressing a set of auxilliary words has been provided as follows:

ARGS	SYMBOL	VALUE	ADDRESSING TYPE
r	)	10	register deferred
r	)+	20	auto-increment
r	@)+	30	auto-increment deferred
r	-)	40	auto-decrement
r	@-)	50	auto-decrement deferred
o r	I)	60	indexed
o r	@I)	70	indexed deferred
dst	\	100000	byte mode
dst notation)	B	100000	byte mode (preferred
v	#	27	immediate mode
a	@#	37	absolute mode
a	P	67	relative mode
a	@P	77	relative deferred mode

In this table r stands for any register (0-7), o stands for a 16-bit offset, dst stands for a complete destination specification (e.g. 4)+), v stands for a 16-bit integer value, and a for a 16-bit address.

Examples of typical assembler constructions for single operand instructions follow with their Macro-11 counterparts:

```
3 CLR,          CLR R3
```

Clear register 3 to zero.

```
S  -) TST,      TST -(S)
```

Subtract 2 from register S (4) and test the data at the location to which S now points. This is a simple way to reserve a word on the stack.

```
134 1 I) INC,   INC 134(R1)
```

Increment the data word found at the address 134 + (contents of register 1).

```
134 1 I) B INC, INCB 134(R1)
```

Increment the data byte found at the address 134 + (contents of register 1).

XYZ P CLR, CLR XYZ

Clear the data in variable XYZ. (The assembler uses the relative addressing mode.)

XYZ @# CLR, CLR @#XYZ

Clear the data in variable XYZ. (The assembler uses the absolute addressing mode.) The P and @# modes are equivalent in most cases.

Double operand instructions require both a source and a destination field which can be defined with the mode words as described above. A few examples:

S -) 112 2 I) MOV, MOV 112(R2),-(S)

Move data from address 112 + (contents of register 2) to the stack, after having subtracted 2 from register S (4). (You use the construction -) as a destination to push data on the Forth stack.)

XYZ P -10 # MOV, MOV #-10,XYZ

Move the immediate value (-10) into variable XYZ.

S)+ T MUL, MUL T,(S)+

Multiply register T (1) by the top stack value, pop the stack, and return the product in T (1) and TT (2). Note that the MUL instruction (like DIV, ASH, etc.) may have only a register type "source" field.

Conditional branches (IF, THEN, BEGIN, etc.) are handled through the PDP-11 BR-type instructions. The following Forth words are available as constant definitions:

NE EQ PL MI VC VS CC CS
GE LT GT LE HI LS HS LO

These test the PDP-11 condition codes the same way as the branch instructions Bxx, where xx is replaced by one of the two letter codes.

To make an assembler conditional branch you give the following assembler commands:

<set up condition codes (TST)> xx IF, <true code> THEN,

You first set up the condition codes; this can be a byproduct of some arithmetic (e.g. from an ADD instruction) or the result of an explicit TST or CMP operation. Next give the two letter condition code from the list above, followed by IF,. The IF, will assemble the appropriate branch instruction. (Actually, the branch around the "true code" must occur when the condition you specify is false, so the branch that is assembled is the logical inverse of the condition type you specify.)

An example:

3 2 CMP, EQ IF, FLAG P 1 # MOV, THEN,

This is assembled like the following Macro-11 code:

```
        CMP 2,3
        BNE 1$
        MOV #1,FLAG
    1$:
```

The BEGIN, - END, construction works in a similar way:

BEGIN, <loop code> xx END,

where xx is a condition from the same list. As a concrete example

BEGIN, 0 DEC, MI END,

translates to the following Macro-11 code:

```
1$: DEC 0
    BPL 1$
```

Following is a list of the PDP-11 Forth assembler op-codes:

```
010000 2OP MOV,   020000 2OP CMP,   030000 2OP BIT,
040000 2OP BIC,   050000 2OP BIS,   060000 2OP ADD,
160000 2OP SUB,   070000 2OP MUL,   071000 2OP DIV,
072000 2OP ASH,   073000 2OP ASHC,  074000 2OP XOR,
004000 2OP JSR,

5000 1OP CLR,   5100 1OP COM,   5200 1OP INC,   5300 1OP DEC,
5400 1OP NEG,   5500 1OP ADC,   5600 1OP SBC,   5700 1OP TST,
6000 1OP ROR,   6100 1OP ROL,   6200 1OP ASR,   6300 1OP ASL,
0100 1OP JMP,   0200 1OP RTS,   0300 1OP SWAB,  0240 1OP CLEAR,
0260 1OP SET,   6700 1OP SXT,

: NEXT, IC 30 + JMP, ;  : SEMI, IC R 20 + MOV, NEXT, ;
: CLC, 1 CLEAR, ;  : RTI, 2 , ;  : WAIT, 1 , ;  : HALT, 0 , ;
: SEC, 1 SET, ;  : J, P JMP, ;
```

Note: The following operations are invalid on the PDP-11/04, /05, /10, and /20: ASH, ASHC, XOR, SXT, MUL, DIV, .

APPENDIX B

FORTH BIBLIOGRAPHY.

BOOKS. The following are some contemporary books that describe Forth or Forth-like languages. For the most part they are written in a semi-technical style and are aimed at the small computer user.

1. Brodie, Leo, Starting Forth, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

2. Hogan, Thom, Discover Forth, Osborne/McGraw-Hill, Berkeley, California, 1982.

3. Loeliger, R. G., Threaded Interpretive Languages, Byte Books, Peterborough, New Hampshire, 1981.

JOURNALS. Many of the personal computer journals carry articles on Forth and Forth programs. The following are particularly noteworthy.

1. Byte. A special Forth issue appeared in September, 1980. A collection of Forth reprints from Byte issues is available from the Forth Interest Group (FIG)*.

---

*Forth Interest Group, P.O. Box 1105, San Carlos, California 94070.

---

2. Dr. Dobbs Journal. A special issue appeared in September, 1982.

3. Forth Dimensions. This is a journal specializing in Forth, published by FIG.

CONFERENCE PROCEEDINGS. There have been a number of conferences dealing with Forth issues. The proceedings are a useful source for both theoretical and practical understanding of Forth.

1. FORML (Forth Modification Laboratory) Conference Proceedings, 1980, 1981 (Volumes 1 and 2), and 1982. Available from FIG.

2. Rochester Forth Conference on Databases and Control, Proceedings, May, 1982. Available from FIG.

STANDARDS. The latest available Forth standards document is Forth-79, a publication of the Forth Standards Team, October, 1980, distributed by FIG. A "Forth-83" standard is in preparation.

HISTORICAL REFERENCES. Forth had its beginning in the early 1970s in scientific and astronomical communities. The following are some of the references from that era.

1. Ewing, Martin S., The Caltech Forth Manual, Internal Report, Owens Valley Radio Observatory, California Institute of Technology, Pasadena, California, First Edition, 1974, Second Edition, 1978.

2. Ewing, Martin S., and Hammond, H. Wayne, The Forth Programming System, Proceedings of the Digital Equipment Computer Users Society, Nov., 1974, pp 477 - 482.

3. Miedaner, Terrell, AST-01 and AST-01X Definitions, Memorandum to the Astronomy Forth Users Group, Kitt Peak National Observatory, Tucson, Arizona, 1977.

4. Moore, C. H., and Rather, E. D., The Forth Program for Spectral Line Observing, Proc. I.E.E.E., 61, 9, p. 1346, Sept., 1973.

5. Moore, C. H., Forth: A New Way to Program a Mini-computer, Astronomy and Astrophysics Supplement, 15, pp 497 - 511, 1974.

6. Rather, E. D., Moore, C. H., and Hollis, Jan M., Basic Principles of Forth Language as Applied to a PDP-11 Computer, National Radio Astronomy Observatory, Charlottesville, Virginia, Computer Division Internal Report No. 17, 1974.

7. Sachs, Jonathan, An Introduction to Stoic, Technical Report BMEC TR001, Harvard-MIT Program in Health Sciences and Tech-

nology, Biomedical Engineering Center for Clinical Instrumentation, June, 1976.

8. Sinclair, W. S., The FORTH Approach to Operating Systems, Proc. ACM '76, pp. 233-240, October, 1976.