C O N T E N T S

PREFACE

This is the second edition of the Caltech-OVRO Forth Manual. It reflects numerous changes that have occurred in the 3 1/2 years since the original publication. Chief among these has been the shift at OVRO toward diverse Forth applications based on PDP-11 systems, many running DEC operating systems.

Both the PDP-11 and PDP-10 systems have been revised to take advantage of a Caltech-developed innovation in the interpreter system. Substantial time and core savings result from using an address interpreter requiring only one machine instruction. The PDP-11 system has been further refined so that only two memory words are required for header information in Forth dictionary entries.

Another development reflected in this Manual is the emergence of a Forth standard vocabulary. Although the AST.01 and AST.01X documents adopted by the Astronomy Forth Users Group in the U.S. are not fully mature language specifications, they do provide useful guidelines for new Forth systems. They help to reduce the chronic problem of Forth installations at various institutions that all have originated from mainstream Forth, but which have diverged under the assault of numerous clever, but non-communicating programmers.

I would like to thank H. W. Hammond and D. H. Rogstad who have been responsible for many of the developments to the PDP-11 Forth at Caltech. I thank D. Dewey, H. W. Hammond, R. B. Leighton, and D. H. Rogstad for reviewing this manuscript. Work at the Owens Valley Radio Observatory is supported in part by the National Science Foundation. This work was also supported in part by the Caltech Jet Propulsion Laboratory.

Martin S. Ewing
3 June 1978

- - - - - - - - - - - - - - - -

This Manual was prepared on the Caltech PDP-10 using the TECO and RUNOFF utilities and a Printronix printer via the VLBI GT44 computer. It is available on machine readable media.

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND.

Rapid acceptance of minicomputers for interactive data acquisition and system control has created a need for matching software systems. High level languages like Fortran, Algol, or PL/1 are not normally effective in environments with limited memory and peripheral devices. Even when these languages can be used, they are designed for batch processing and usually lack features needed for direct interaction with the operator. By default many programmers have been forced to use assembly language. This is efficient for small programs, but there can be great practical difficulties in writing and maintaining larger assembly programs.

The Forth system meets the problem described above; it provides a flexible programming system for minicomputers of moderate size. A machine with 8K 16-bit words and at least one mass storage device can make effective use of Forth. Most new laboratory computers will have at least this size; programming difficulties with smaller machines increasingly outweigh the falling cost of memory and peripherals.

Forth also has important capabilities for the designer of microcomputer systems. Microcomputer development systems typically have some sort of floppy disk storage and so may run the full, minicomputer style Forth. Systems incorporating microcomputers, however, often have minimal peripheral devices. Forth techniques are useful in these cases as a means of writing memory-efficient code and of implementing conversational interaction with the user.

A list of the salient features of Forth will include the following:

1. Incremental compilation and assembly,

2. Push-down stack for parameters and data, natural re-entrancy,

3. Simple language extensibility,

4. On-line editing, rapid compilation,

5. Structured programming encouraged,

6. Typewriter driven system, minimal prompting,

7. Easy trade-off between compact interpretive code and fast machine-language code, and

8. Machine independence for high level programs.

1.2 FORTH AT OVRO.

The Forth system has been adopted for numerous applications by the California Institute of Technology Owens Valley Radio Observatory. These include control of the 3 OVRO telescope systems: the 27 m interferometer, the 40 m telescope, and the new 10 m millimeter telescopes. These systems require Forth's capabilities for real-time control of antenna servos, data acquisition, user interactive control, and easy program maintenance.

Other OVRO applications include more specialized instruments: the Caltech-JPL Mark II VLBI Processor, and a 1024 channel autocorrelation spectrometer. In the former case a heavy real-time control requirement was combined with the need for geometric model calculations of very high (64 bit) accuracy. In all cases, Forth has been used as an intimate and highly flexible hardware debugging tool.

Forth systems at Caltech have been implemented on the PDP-11, PDP-10, and SDS-920 computers. A wide variety of other computers has been used at other institutions; these include the Nova, HP 2100, Varian, and Modcomp machines.

1.3 FORTH DEVELOPMENT HISTORY.

The guiding spirit in the development of Forth has been C. H. Moore, who with E. R. Rather constructed the first Forth systems at the National Radio Astronomy Observatory. Since that time (ca. 1973), they and others have continued as a private company (Forth, Inc., Manhattan Beach, Ca.) to develop the Forth system for a wide variety of applications, both scientific and commercial. The name "Forth" is claimed as a registered trademark by Forth, Inc.

Many other individuals and organizations have adapted Forth to their requirements. Most non-commercial user activity is still in the area of astronomy; astronomy users groups have been established both in Europe* and the U.S.**

Work by the U.S. users group has led to the adoption of a Forth language standard, AST-01, and an extension, AST-01X. The Caltech-OVRO Forth systems vary from that standard to some degree. In cases where there is disagreement, both usages will be given. It is the intention of the Caltech-OVRO group gradually to move to the standard.

NOTE

AST-01 and AST-01X are standards adopted only by the U.S. Forth Astronomy Users Group and have no relationship to products offered by Forth, Inc.

----

*Contact Dr. Peter W. Hill, Observatory, University of St. Andrew, Fife, U.K.

**Contact Dr. Robert W. Milkey, Kitt Peak National Observatory, Tucson, Arizona.

1.4 SCOPE OF THIS MANUAL.

Chapter 2 is intended as 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 vocabularies, the one used in the Caltech-OVRO Forth, AST-01, and AST-01X. The appendices give the implementation details for various Caltech systems. A Bibliography sets out the (rather sparse) publications available.

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 (English or nonsense) that you can type on the keyboard.

NOTE

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

Words are defined in the dictionary, which, like English 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 may type in words (more precisely, word names) to your keyboard terminal. Forth permits a very general and free-form input. With few exceptions, any combination of letters, numbers, or other characters can be used to name a word. One character, normally a "blank", is reserved to delimit words. A few other characters are reserved to let you correct errors in typing. (For Caltech-OVRO PDP-10/11 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 current Caltech-OVRO systems N=4.) N is chosen as 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 "word" that can't be found in the dictionary, Forth sees if the "word" makes sense as a number. If so, the "word" is converted to binary and pushed on the stack (explained below). If a "word" you type 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 "pushes down" older items. When an item is no longer required, it is "popped" off the top of the stack, making older items available again. (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. (Just as do the HP "RPN" calculators -- HP-25, HP-67, etc.) 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 data you supply on the stack, pop their 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". An example

```
( 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. IBM-style magnetic tape, DECtape, cassette tapes, and floppy disks are all usable for this purpose, although a high-speed disk unit is preferable. In any case, a random-access technique is required.*

----

*IBM-compatible magnetic tape is conventionally used for sequential, not random access. Random access (with update in place) can be achieved by using preformatted tape with long inter-record gaps.

The storage device is divided into fixed-length "blocks", normally 512 words = 1024 characters 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 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.

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've 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 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 need 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

----

**Throughout the Manual Forth words and typed input to Forth will be underlined for clarity.

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.

A sample CODE definition follows:

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

Here the Forth word + is defined as three PDP-11 instructions. Their action is to sum the top two stack values and leave the result instead. For further information consult Section 3.8 and the assembler description of your particular machine.

With the word : (colon) you can define Forth words in terms of other Forth words. Colon definitions are fairly 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.

Each function invoked (i.e. word referenced) in a : definition takes one memory word (one byte in some microcomputer versions). This memory word holds a pointer (address) to 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 in the Caltech-OVRO PDP-11/40 version. These figures are comparable to and often somewhat better than equivalent subroutine calls in assembler language.

This is an example of a Forth : definition:

: . 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 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 (INTEGER on older systems) lets you reserve and name such locations. Type

13 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. The number you typed (13) sets the initial contents of the storage location.

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

25 ARRAY DATA

This example reserves 25 memory words 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.

Very often the only way you want to access data in an array is through an index, e.g. the i-th word in an array. The preferred way to define such an array is with the ()DIM word:

16 ()DIM FOO

Like ARRAY, ()DIM reserves the indicated number of memory words under the name FOO. When you wish to access any of the data in FOO, however, you must supply an index. For example,

3 FOO

Here you are specifying the 3rd item in array FOO. What you get back is the memory address of the 3rd word.

()DIM is better than ARRAY because you don't have to worry about the addressing scheme of your computer or about the precision of your data. (In some machines, e.g. PDP-11, adjacent fullwords have addresses differing by 2 because they use byte addressing.)

2.5 STORING AND RETRIEVING DATA IN MEMORY.

The word @ 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

Q @

to get the integer in variable Q (initially 13).

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

<value> <address> !

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

148 Q !

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

Another little program might run

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

In the first line ABC is defined with initial 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 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 (see below) lets you store away the program (in source text form) in a block. To define the word, or collection of words, in the future all you need to do is type

<block#> LOAD

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

Each block to be loaded must end with the special word ;S, which restores the text interpreter to the source previously in effect. Note that LOADs may be nested; a block to be loaded may contain LOADs itself.

A block might contain the following text:

```
2 2 + .
13 LOAD
;S
```

If you were to load this block, 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, block number 13 is loaded (with whatever commands are contained there). Finally the ;S returns control to the calling program (e.g. to the typewriter).

2.7 TYPEWRITER OUTPUT.

Output from Forth normally comes to your terminal (typewriter or CRT). 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 (decimal). Some implementations allow base 16 as well. The special words OCTAL and DECIMAL let you set BASE automatically. The default number base should be decimal, but you should check this on your system.

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

<pointer> <character count> TYPE

The nature of the pointer depends on the system. In the PDP-11, it is simply a byte address that indicates the first character to be typed. For the PDP-10, it is a byte-pointer with the same effect. Beginning with the specified character, TYPE puts out sequential characters until the count is satisfied.

Terminal input and output save space by using the same buffer in main memory. To avoid problems 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 "456 ." is wiped out when "123" is typed.

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 END. Consider the following example:

```
: EXAMPLE 1 BEGIN 1 - DUP END DROP ;
```

BEGIN signals the beginning of a loop. When the program gets to the END (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 END; then the duplicated value is popped from the stack. Since END found a 0, control returns to BEGIN; 1 is again subtracted, leaving -1. END 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 - END 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 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 stack (the return 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 agains 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

----

*Thus data calculated outside the DO - LOOP range can be passed into the range without interfering with loop indices.

off 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. 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 -

```
                  if <value>. eq. 0
                  |-----------------|
                  |                 v
<value>   IF   <true-code>  ELSE  <false-code>   THEN
                             |                     ^
                             |---------------------|
```

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

```
              if <value>.eq.0
             |-------------------|
             |                   v
<value>  IF  <true-code>  THEN
```

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

0 if x .GE. 0, and

(-x)**3 if x .LT. 0. (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 .GE. 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 - ENDs, 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, Forth source code for example. The way you can enter and modify text in Forth blocks is with the Forth text editor.

In the Caltech-OVRO versions of Forth, at least two editors are available. The basic editor (EDIT) is very compact but gives you everything you need to modify text a line at a time. The extended editor (XED) includes flexible string manipulations and lets you search for, insert, or delete text strings anywhere in a block.

For the PDP-11 systems containing a VT-11 vector graphics system (the Caltech-JPL VLBI Processor's GT44 and the OVRO 1024-channel autocorrelator's GT40) there is a special editor called QED. This editor uses the refreshed display to show a block being edited and a cursor within the block. Flexible cursor controls and text manipulations are available. (Refer to Appendix D.)

The standard block length for Forth systems is 512 16-bit words = 1024 8-bit characters. This is conventionally divided into 16 lines of 64 characters.* (The

----

*This format only applies to block to be used for text. Any block may also be used for binary data, in which case you can choose any format.

64th character of a line is logically just before the first character of the next line.)

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

35 BLK !.

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

35 LIST.

As a side effect LIST sets BLK 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 T.

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" 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 BLK ! " FOO BAR" 3 R.

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

10 BLK ! (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 FLUSH 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 provides a more thorough description of the Forth system. The reader is assumed to be familiar with the preceding Chapters and to 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 job 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: one that we can call "standard" Forth (SF). In this language concrete words are defined, such as +, BLOCK, and DO. SF may be compared with other programming languages like Fortran, Basic, or Algol. SF 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

B**2 - 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

----

*In fact a card-oriented Forth for the IBM 360 has been developed at the NRAO.

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).

```
high limit       |...............................|
                 |                               |
                 |  ^                            |
                 |  ^                            |
                 |  ^ return stack               |
                 |...............................|
                 |  v                            |
                 |  v parameter stack            |
                 |  v                            |
             ^   |   .   .   .   .   .   .   .   |
             ^   |                               |
memory       ^   |                               |
  addresses  ^   |   .   .   .   .   .   .   .   |
             ^   |  |                            |
             ^   |  | user application           |
                 |  |   dictionary               |
                 |  |                            |
                 |...............................|
                 |  |                            |
                 |  | "standard" Forth           |
                 |  |   dictionary               |
                 |...............................|
                 |  |                            |
                 |  | Forth object dictionary    |
                 |  |   (kernel)                 |
                 |...............................|
                 |     block buffer 2            |
                 |...............................|
                 |     block buffer 1            |
low limit        |...............................|
```

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 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 sequence of their definition. 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.

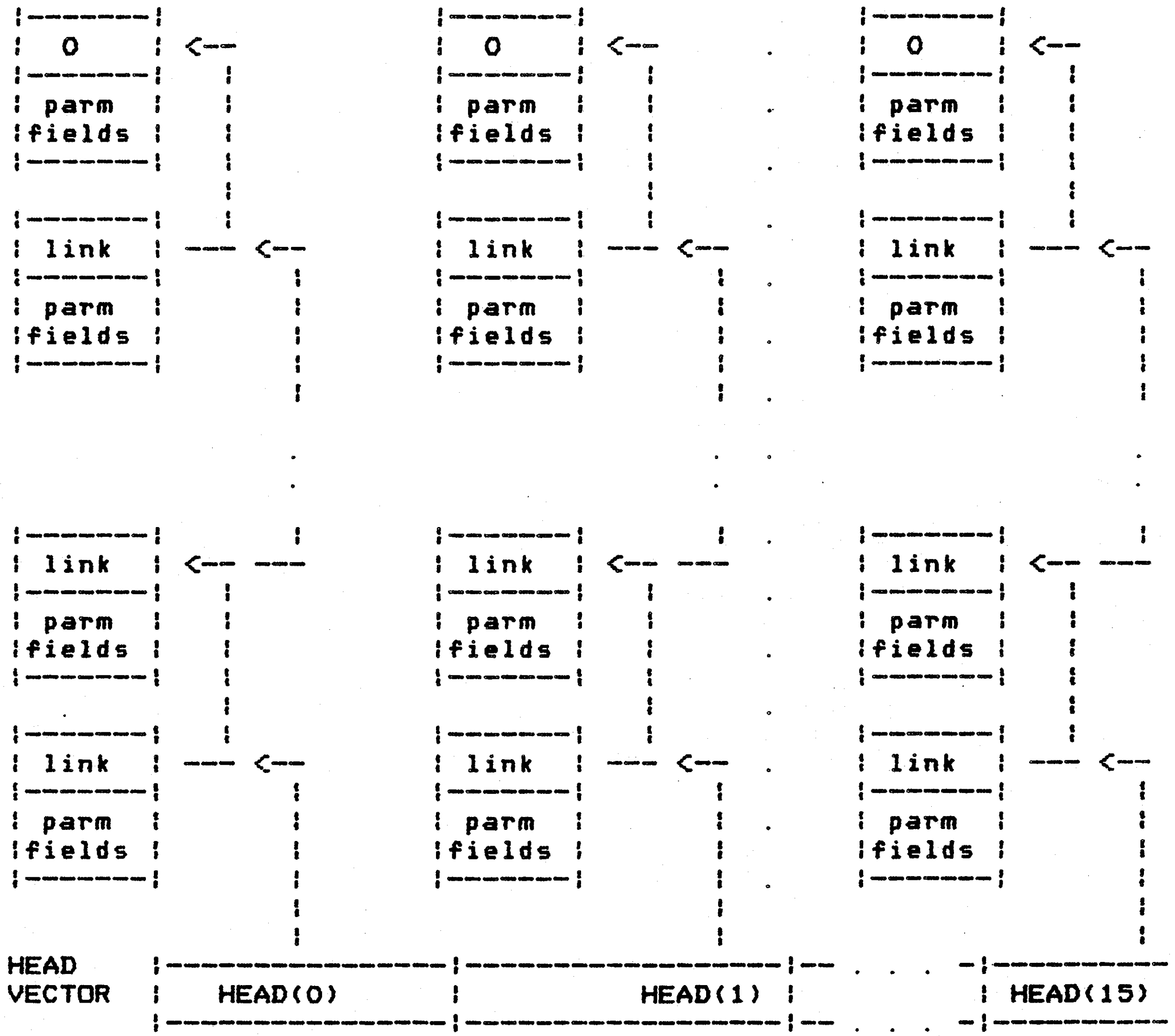

Fig. 3.2 Dictionary Organization.

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 vector of pointers. These pointers aim at the most recently defined word in each branch. A field in each word definition in turn points to the previous word in the same branch. (The exact target of the link may not be the link of the previous work; 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 (but not Caltech-OVRO at this writing). The principle is to divide the dictionary into branches similar to those discussed above. In this scheme however, 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 <u>VOCABULARY</u> and <u>DEFINITIONS</u> control the branching. Figure 3.3 illustrates the <u>VOCABULARY</u> technique.

```
                        |---------------|
                        | central       |
                        | vocabulary    |
                        | (FORTH)       |
                        |               |
                        |---------------|
                                |
                                |----------------------|
                                |                      |
                        |---------------|      |---------------|
                        | more          |      | assembler     |
                        | (FORTH)       |      | vocabulary    |
                        |               |      | (ASSEMBLER)   |
                        |               |      |---------------|
                        |---------------|              ^
                                |                      ^
          |---------------------|                      ^
          |                     |                      ^
|---------------|       |---------------|              ^
| editor        |       | more          |              ^
| vocabulary    |       | (FORTH)       |              ^
| (EDITOR)      |       |---------------|              ^
|---------------|               ^                      ^
          ^                     ^                      ^
          ^                     ^                      ^
 HEAD(EDITOR)             HEAD(FORTH)          HEAD(ASSEMBLER)
```

Fig. 3.3 VOCABULARY branching.

An unlimited number of HEAD pointers can be maintained; each one points to the last defined word 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 will be searched 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 definining 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.*

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

----

*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.

efficiently encoded into two 16-bit memory words as shown in Fig. 3.4.

```
------------- BIT NUMBER ---------------------
 1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0
 5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0
:  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  :
.                                               .
.       LONG LINK FIELD (ONLY IF OFFSET LINK=0) .    WORD 0
.                                               .
:--.--.--.--.--.--.--.--.--.--.--.--.--.--.--.--:
:     :                 :                 :     :
: C1' :       C2        :       C3        : C4  :    WORD 1
:     :                 :                 : high:
:--.--.--.--.--.--.--.--.--.--.--.--.--.--.--.--:
:           :        :  :                       :
:  C4 low   :   L'   :P :  OFFSET  LINK         :    WORD 2
:           :        :  :                       :
:--.--.--.--.--.--.--.--.--.--.--.--.--.--.--.--:
:                                               :
:       BEGIN MACHINE INSTRUCTIONS              :
:               & PARAMETERS                    :
.                                               .
.                                               .
```

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' .neq. 0
  = 4           if L' .eq. 0, C4 .neq. blank
  = 3           if L' .eq. 0, C4 .eq. blank,
                        C3 .neq. blank
  = 2           if L' .eq. 0, C4 .eq. C3 .eq. blank,
                        C2 .neq. blank
  = 1           if L' .eq. 0, C4 .eq. C3 .eq.
                        C2 .eq. 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

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 .neq. 0)

        Previous address = long link field
                (if offset link .eq. 0)
```

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

Fig. 3.4 Dictionary Header for PDP-11 (cont'd)

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. (Many terminals cannot even print lower case, so the restriction is of little importance.) 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. Again, the limitation is slight, as most practical Forth code has few names as long as 10 characters.

The following are examples of distinguishable names:

<u>A</u> <u>B</u> <u>ABCD</u> <u>ABCE</u> <u>ABCE1</u>

However, the following pairs of names are indistinguishable:

<u>ABCD1</u> <u>ABCD2</u>

<u>C1234567890</u> <u>C12345678901</u>

<u>ABCD1234567</u> <u>ABCD0987654321QWERTY</u>

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" 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, i.e. 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.

```
! .     .     .     .     .    !
!                              !
!-----            ------------!
!          HEADER              !
!-------          ------------!
!     (2 OR 3 LOCATIONS)       !
!------------------------------!
!  CODE SECTION                !
!-------          ------------!
!  (1 OR MORE LOCATIONS)       !
!-------          ------------!
.                              .
.                              .
!------------------------------!
.                              .
.   PARAMETER SECTION          .
.        (OPTIONAL)            .
.                              .
! .     .     .     .     .    !
```

Fig. 3.5 General Forth Dictionary Entry.

Every word must contain a <u>code section</u>; 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 <u>:</u> definitions (see Section 3.9). For <u>CODE</u> 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 <u>NEXT,</u> (JMP @(IC)+, see Section 3.4).

Other kinds of words, in particular <u>:</u> words, require an additional <u>parameter section</u> in their dictionary entries. In <u>:</u> words the parameter section contains compiled addresses which direct the execution of the address interpreter. Words defined by <u>VARIABLE</u> or <u>CONSTANT</u> 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 WORD                    COLON WORD
    !----------------!           !---------------!
    !  HEADER        !           !  HEADER       !
    !----------------!           !---------------!
    !                !           ! JSR IC,@#  ...!  <-- 4537(8)
    !  MACHINE       !           !---------------!
    !   INST. CTRS   !           ! 1ST WORD ADR  !
    !                !           !---------------!
    !                !           ! ADDITIONAL    !
    !----------------!           !  WORD         !
    ! JMP @(IC)+     !           !  ADDRESSES    !
    !----------------!           !               !
                                 !---------------!
                                 ! ADR( SEMI )   !
                                 !---------------!

    CONSTANT WORD                VARIABLE WORD
    !----------------!           !---------------!
    !  HEADER        !           !  HEADER       !
    !----------------!           !---------------!
    ! JSR IC,@#  ... !           ! JSR IC,@#  ...!
    !----------------!           !---------------!
    ! ADR( CONST )   !           ! ADR( VAR )    !
    !----------------!           !---------------!
    !  VALUE         !           !  VALUE        !
    !----------------!           !---------------!
```

(CODE SECTIONS 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 scam 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 loaded from disk, e.g. when you type

.R FORTH

under the RT-11 operating system. This initial dictionary and its associated code is called the "object program" or "kernel". For Caltech-OVRO systems the kernel is defined in Macro-11 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

----

*These elegant coding tricks for the PDP-11 were invented by D. H. Rogstad and H. W. Hammond.

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.)

```
                 |---------------|                                |--------------|
                 | HEADER "ABC"  |                                | HEADER "A"   |
                 |---------------|                                |--------------|
                 | JSR IC,@# ... |                  |------>| JSR IC,@# ... |
                 |---------------|                  |             |--------------|
                 | ADDRESS( A )  |--------|             | ADDRESS( AA ) |
                 |---------------|                                |--------------|
(IC)--->| ADDRESS( B )  |--------|             | ADDRESS( AB ) |
                 |---------------|            |                   |--------------|
                 | ADDRESS( C )  |---|        |                   | ADDRESS(SEMI) |
                 |---------------|   |        |                   |--------------|
                 | ADDRESS(SEMI) |   |        |
                 |---------------|   |        |                   |--------------|
                                     |        |                   | HEADER "B"   |
                                     |        |                   |--------------|
                                     |        |------>| JSR IC,@# ... |
Forth definitions:                   |                            |--------------|
                                     |                            | ADDRESS( BA ) |
: A AA AB ;                          |                            |--------------|
: B BA ;                             |                            | ADDRESS(SEMI) |
: C CA ;                             |                            |--------------|
: ABC A B C ;                        |
                                     |                            |--------------|
                                     |                            | HEADER "C"   |
                                     |                            |--------------|
                                     |----------->| JSR IC,@# ... |
                                                                  |--------------|
                                                                  | ADDRESS( CA ) |
                                                                  |--------------|
                                                                  | ADDRESS(SEMI) |
                                                                  |--------------|
```

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.*

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 of this type. 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

----

*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 had by adopting the technique described here.

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 accomplish the diversion of 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. (Done 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 does 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 3 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

----

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

than addresses.

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-English) 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-English 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 block is the input source, TI runs straight through without a pause until ;S is encountered. (And ;S had better be there!)

"Collecting a text string" means scanning the input source until a complete word-name-candidate 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*.

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 collected string 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

----

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

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 "blasted" into 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 FLUSH 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 ERASE-CORE before BLOCK. ERASE-CORE 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 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 (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. Semicolon-S (;S) is the word that must terminate the scan on each block. If ;S is not present, the interpreter will run off the end of the block with unpleasant results.

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 Caltech-OVRO 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 like MACRO-11. 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. Thus compilation is distinct from assembly, which applys 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 <u>:</u> in the text stream changes the state to compilation; it also invokes <u>WORD</u> to collect the next properly delimited word from the text stream. The word name 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 fully defined. To prevent false, premature references to the entry, <u>:</u> also alters the name field slightly so that the name becomes unrecognizable. At the conclusion of the definition, <u>;</u> or <u>;CODE</u> restores the correct name.)

It now remains to create the parameter field of the new <u>:</u> 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 LIT is compiled followed by the literal value of the number. (LIT points to a simple code routine that picks up the number following LIT'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 encountered. 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 <u>;</u>, which compiles SEMI into the dictionary and also resets the compile state. Other compiler directives are <u>IF</u>, <u>THEN</u>, <u>ELSE</u>, <u>;CODE</u>, etc.

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

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

are equivalent.)

3.10 DEFINING WORDS -- ;CODE.

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

1 VARIABLE N.

An initial value (1) is assigned to N. The dictionary entry created for N is shown in Fig. 3.12.

```
!---------------!
!  header       !
!---         ---!
!     "N"       !
!---------------!
!  JSR IC, @#   !
!---------------!
! Address (VAR) !
!---------------!
!  value = 1    !
!---------------!
```

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.

The definition of VARIABLE for the PDP-11 may be given in terms of CONSTANT:

: VARIABLE CONSTANT ;CODE S -) IC MOV, SEMI,.

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 CONSTANT.

The second part of the example begins with ;CODE. ;CODE is a compiler directive that compiles an address (called SCODE), and sets the system state to execution. Following ;CODE are assembly instructions. These instructions define the code (VAR) which will be associated with all VARIABLE words. The dictionary entry for VARIABLE is shown in Fig. 3.13. (Note that the assembler word SEMI, expands into two PDP-11 instructions.)

```
          |---------------|
          |   header      |
          |---         ---|
          | "VARIABLE"    |
          |---------------|
          |   JSR IC,@#   |
          |---------------|
          | Adr(CONSTANT) |
          |---------------|
          | Adr(SCODE)    |
          |---------------|
VAR:      | MOV IC,-(S)   |
          |---------------|
          | MOV (R)+,IC   |
          |---------------|
          | JMP @(IC)+    |   (NEXT)
          |---------------|
```

Fig. 3.13 Dictionary Entry for VARIABLE.

What happens when we execute VARIABLE? First, CONSTANT creates a dictionary entry using the stack value as the constant value and the next word in the input stream as its name. The new dictionary entry has a code section which invokes CONSTANT (see Fig. 3.6), which is inappropriate for a VARIABLE word. It is the purpose of SCODE to establish a different code routine. When this (anonymous) word is executed, the address part of the JSR instruction of the word just defined is reset to point to the machine code part of VARIABLE. Thus the resulting dictionary entry looks like N in Fig. 3.12.

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.) VAR must now restore the IC from the return stack, and execute the NEXT function.

To summarize, ;CODE is 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 ;CODE is ;:. ;: associates a :-level routine with a defining word. The parameter field address is passed on the stack. Thus an alternative definition of VARIABLE would be

: VARIABLE CONSTANT ;: ;

The associated : routine is null in this case.

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 LIT 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 LIT and for SEMI. 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.)

EXEC works by setting up the return stack and instruction counter to execute a word as if it were called from a normal compiled address sequence. After the word finishes, control passes back to the next word following EXEC, either compiled or from the terminal, as appropriate.

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. 3.14.

```
          |---------------|
          |   header      |
          |---         ---|
          |     "0="      |
          |---------------|
          | JSR IC,@#     |
          |---------------|
          | address (XIF) |
          |---------------|
          | address = 1$  |
          |---------------|
          | address (0)   |
          |---------------|
          | address (XSKP)|
          |---------------|
          | address = 2$  |
          |---------------|
  1$:     | address (1)   |
          |---------------|
  2$:     | address (SEMI)|
          |---------------|
```

Fig. 3.14 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 increments 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 SEMI terminates execution of either case.

Other forms of compiled branches work like IF, THEN, etc. Fig. 3.15 is the dictionary entry of a typical DO - LOOP construction:

: LP 4 0 DO RANGE LOOP AFTER ;.

```
        |---------------|
        |   header      |
        |---         ---|
        |   "LP"        |
        |---------------|
        |  JSR IC,@#    |
        |---------------|
        | address(LIT)  |
        |---------------|
        |       4       |
        |---------------|
        | address(0)    |
        |---------------|
        | address(XDO)  |
        |---------------|
1$:     | address(RANGE)|
        |---------------|
        | address(XLOOP)|
        |---------------|
        |     1$        |
        |---------------|
        | address(AFTER)|
        |---------------|
        | address(SEMI) |
        |---------------|
```

Fig. 3.15 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 LIT must be executed with argument 4 to get 4 on the stack. (IC increments after LIT 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 Forthish applications.

Caltech-OVRO systems have been developed both with and without OS support. In this Section we consider some creteria for these choices.

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 (particularly semiconductor memory) is falling rapidly. Low cost peripherals (e.g. 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 <u>all</u> 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; 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 TOPS-10 is generally simpler than as a standalone 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.

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 3.16 illustrates the method.

```
       (recurrent interrupt)
                 |
                 |
   |-----------------|
   |real-time task   |
   |     #1          |
   |-----------------|
                 |
                 |
   |-----------------|
   | interval T2     | (no)
   | elapsed?        |------->|
   |-----------------|        |
    (yes)        |            |
                 |            |
   |-----------------|        |
   |real-time task   |        |
   |     #2          |        |
   |-----------------|        |
                 |            |
                 |            |
   |-----------------|        |
   | interval T3     | (no)   |
   | elapsed?        |------->|
   |-----------------|        |
    (yes)        |            |
                 .            .
                 .            .
                 .            .
                 |            |
   |-----------------|        |
   |real-time task   |        |
   |     #N          |        |
   |-----------------|        |
                 |            |
                 |            |
   |-----------------|        |
   | return from     |        |
   | interrupt       |<-------|
   |-----------------|
```

Fig. 3.16 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 multilevel 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.

This Chapter sets out English definitions for the words in several Forth Vocabularies. Three categories of words exist: words in current Caltech-OVRO use, words in the AST.01 standard, and words in the AST.01X extended standard. There is a large overlap between these categories (in particular AST.01X includes AST.01). There is also no single Caltech-OVRO vocabulary; the vocabulary presented here is weighted toward the PDP-11 system used for the Caltech-JPL VLBI Processor.

4.2 NOTATION.

Notation of this Chapter follows that of the AST.01 document (June, 1977, Terrel Miedaner, Kitt Peak National Observatory). Much of the following text is from that document.

The words appear in essentially the same sequence as their numerically sorted identifier codes. The action of each word is described in abbreviated form: A string of symbols indicating which parameters are to be placed on the stack before executing the word; the word itself; then, any parameters left on the stack by the word. In this notation, the top of the stack is to the right.

Symbols are used as follows:

b   Block 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
u v w	Double-precision (2 cell) numbers.
nnnn	
pppp	The name of a word.
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 action is not related to its action during execution.)

E The word may not normally be compiled within a colon-definition.

K The word is a KPNO word, not currently part of the standard.

L The word causes loading and possible execution of one or more blocks.

N Non-reentrant; may not be used within an interrupt-handler word.

T Tape systems only.

V Caltech-OVRO word, not currently part of the standard. A following number (10, 11, 920, 8080) indicates which type of CPU if not common to all.

X The word is part of the AST.01X extension.

4.3 STANDARD VOCABULARY LIST.

! m p ! Stores m at address p. (V11, V8080: p is a byte address)

!BLOCK b !BLOCK p (Not V) Obtains a core buffer for block b, leaving the first buffer cell address. The block is <u>not</u> read from disk, and is automatically marked as updated.

" " sssss" (Not V) Transmits a message of up to 63 characters delimited by " to the selected output device. Note that a null message (single blank between "s) is not permitted.

" " 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 balnk between "s) is not permitted.

#TER #TER m (X, not V) Returns the physical unit number of the terminal device.

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

' ' nnnn p Leaves the address of the parameter field of nnnn. A compiler directive, ' is executed when encountered in a colon- definition: The address of the following word's parameter field is found immediately (at compilation), and stored in the dictionary (after the address of LIT) as a literal to be placed on the stack at execution time.

> e.g. the sequence: ' nnnn is identical to:
> LIT [' nnnn ,] within a colon-definition.
> (Note: meaning of [differs in V!)

((ssss) Ignores a comment of up to 63 characters delimited by a right parenthesis. A single blank between parentheses is not allowed.

()DIM m ()DIM nnnn (K) Defines an array m+1 cells in length, named nnnn. The sequence; i nnnn leaves the address of the i-th cell on the stack. The index i should be in the range 0 <= i <= m, but no check is made for values outside this range.

`*` m n * q 16-bit integer multiply.

`*/` 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 /.

`+` m n + q 16-bit integer addition.

`+!` m p +! Adds integer m to value at address p.

`+BLOCK` m +BLOCK b (not in V) leaves the sum of m plus the number of the block currently being interpreted.

`+LOOP` m +LOOP (C) Adds m to the loop index. Exit from the loop is made when the resultant index reaches or passes the limit, if m is greater than zero; or when the index is less than (passes) the limit, if m is less than zero. The value m may be a variable.

V8080: This implementation has conditionals that may be executed without compiling. DO, +LOOP remember and restore the Text Interpreter, respectively. The range of the loop must be all in the message buffer (or block) at one time. In practice, you can enclose a sequence of words by DO and LOOP and repetitively interpret them as long as everything can be typed on one line.

`,` m , Stores m into the next available dictionary cell, advancing the dictionary pointer.

`,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 integer subtraction (m-n).

`.` m . Prints the value on the stack as an integer, converted according to the current number base.

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

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

0) m 0) q (X, not V) Inverts (toggles) the most significant bit of m.

0< m 0< f Leaves a true flag if m is negative.

0<= m 0<= f (X) True if m is zero or negative.

0= m 0= f True if m is zero.

0<> m 0<> f (X) True if m is non-zero.

0> m 0> f True if m is positive and non-zero.

0>= m 0>= f (X) True if m is greater than or equal to zero.

0SET p 0SET (V) Store zero at location p.

1+ m 1+ q (X) q = m + 1.

1+! p 1+! (X) Add 1 to the contents of address p.

1- m 1- q (X) q = m - 1.

1SET p 1SET (V) Store one at location p.

2* m 2* q (X) q = 2 * m.

2+ m 2+ q (V) q = m + 2.

2- m 2- q (V) q = m - 2.

2/ m 2/ q (X) q = m / 2.

: : nnnn Create a dictionary entry for a colon-definition, set compilation mode, and set the context vocabulary equivalent to the current vocabulary (V: no vocabularies).

:> :> (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,
(D. H. Rogstad suggests that better notation would be ;< instead of :>, and >: for the reverse function. See >: .)

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

;:   ;: (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 ;: and ; to be executed with the parameter-field address of pppp on the stack. Further explained in Section 3.10. (V11: parameter-field address is not passed at present -- but should be!)

;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.

;EXIT   ;EXIT (X, Not V) Terminate a colon-definition when encountered at execution time; compilation is unaffected.

;S   ;S (E) Stops interpretation of a symbolic block.

<   m n < f True if m less than n. (2's complement, 16 bits)

<=   m n <= f True if m does not exceed n. (2's complement, 16 bits)

=   m n = f True if m = n.

<>   m n <> f True if m not equal to n.

<R   m <R (V) See >R. OVRO has used <R and R> (bra-ket notation) while AST.01 uses >R and R> (arrow notation) to signify moving data to and from the

return stack, respectively.

> m n > f True if m > n. (2's complement, 16 bits)

>: >: (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 True if m not less than n. (2's complement, 16 bits)

>R m >R (C) Pushes m onto the top of the return stack. See I and R>. (V11: <R).

>IM >IM nnnn (not V) Set the precedence bit of the following word, making it a compiler directive.

? p ? (N) Prints the value contained at address p in free format, according to the current base.

?DEF ?DEF nnnn m (Not V) Returns the first memory cell address of nnnn if nnnn can be found in the context vocabulary; zero otherwise.

?TER ?TER c (X, not V) Returns the character code of the last character entered at the terminal, or zero if no character has been typed.

@ p @ q Leaves the contents q of memory address p.

[[(Not V) Stop compilation. The words following the left bracket in a colon-definition are executed, not compiled. Typically, left and right brackets are used in conjunction with the interpreter-level conditionals IFTRUE-IFEND to control compilation.

[[ssss] p q (V) Compile literal string ssss into the dictionary. When control passes to [at execution time, the starting byte address p and character count q are returned on the stack ready for TYPE.

]] (not V) Resume compilation. Words following the right bracket are compiled.

]	] (V) Delimiter for string compiled by [.

^	^ nnnn (Not V) Return the compilation address of the following word; that is, the address which would be compiled in a colon- definition. Abort if nnnn is not found. (V: see %)

AO<	p AO< f (V) Comparison of data with zero (address mode). p is an address pointing to the data.

AO<=	p AO<= f (V) Address mode compare; true if data less than or equal to zero.

AO<>	p AO<> f (V) Address mode compare; true if data not equal to zero.

AO=	p AO= f (V) Address mode compare; true if data equal to zero.

AO>	p AO> f (V) Address mode compare; true if data greater than zero.

AO>=	p AO>= f (V) Address mode compare; true if data greater than or equal to zero.

A<	p q A< f (V) Address mode compare; true if first datum less than second. Equivalent to p @ q @ <.

A<=	p q A<= f (V) Address mode compare; true if first datum less than or equal to second.

A<>	p q A<> f (V) Address mode compare; true if first datum not equal to second.

A>	p q A> f (V) Address mode compare; true if first datum greater than second.

A>=	p q A>= f (V) Address mode compare; true if first datum greater than or equal to second.

AL<	p q AL< f (V) Address mode unsigned compare; true if first datum less than second when considered as 16-bit unsigned integers.

AL<=	p q AL<= f (V) Address mode unsigned compare; true if first datum less than or equal to second.

AL= p q AL= f (V) Address mode compare; true if first datum equal to second. (Better notation would be A=.)

AL> p q AL> f (V) Address mode unsigned compare; true if first datum greater than second.

AL>= p q AL>= f (V) Address mode unsigned compare; true if first datum greater than or equal to second.

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

ABS m ABS q Leaves the absolute value of a number.

ADOPT m ADOPT (C, not V) Stores m into the next available dictionary cell, advancing the dictionary pointer. (See ,.)

AND m n AND q Bitwise logical AND of m and n.

ARRAY m ARRAY nnnn (V) Define a word nnnn that, at execution, will push the starting address of an array of m cells on the stack. The m cells are not initialized and may have random values.

ASSEMBLER
ASSEMBLER (X, not V) 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.

B! m p B! (V) The low order 8 bits of m is stored at the byte address p. (See \!.)

B@ p B@ m (V) The 8-bit byte at address p is returned in the low order part of m. With luck, the high order part of m contains the sign extension of the byte. (I.e. 200(8) --> 177600(8).) (See \@.)

B, n B, (V) Compile the low-order byte of n into the dictionary and increment the dicitionary pointer by 1 byte. (See \,.)

BMOVE m n r BMOVE (V) Move r bytes from area beginning at byte address m to area beginning at byte address n. (See \MOVE.)

BASE BASE p An integer pointing to the current conversion base value.

BEGIN BEGIN (CO+) Mark the start of a BEGIN-END loop. The words between BEGIN and its corresponding END will be repetitively executed until the END-condition is satisfied. Loops may be nested.
V8080: BEGIN and END, like DO and LOOP, may be used at interpreter level -- as long as the enclosed range fits on one line or one block.

BELL BELL (X) Activate terminal bell or noisemaker.

BLK BLK p (N) An integer, pointing to the number of the block being listed or edited.

BLOCK b BLOCK p Leaves the first address of 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.

C m C nnnn (V) Abbreviation for CONSTANT.

CASE m n CASE ... ELSE m ... THEN or
m n CASE ... THEN m
(C2+, X, not V) If m equals n, m is dropped from the stack, and the words immediately following CASE are executed until the next ELSE or THEN. If m does not equal n, m remains on the stack and the words after ELSE (or THEN if no ELSE is used) are executed. The value n is always dropped.

CHAIN CHAIN vvvv (X, not V) 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.

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.

CON — m CON nnnn (X) Abbreviation of CONSTANT.

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 (X, not V) An integer that indicates at which vocabulary dictionary searches are to begin.

CONTINUED — b CONTINUED (not V) Continue interpretation at Block b. The block currently being interpreted is marked as least-recently- accessed, so that its buffer will be used for storage of Block b, and the contents of the alternate block will remain in memory.

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

COUNT — p COUNT (m) n (C) The count-byte n is extracted from the first memory cell of a message string beginning at address p, and left on the stack. The character-address m of the first byte of the message is typically left on the stack or in a register. Whatever, COUNT is to be used in conjunction with a following PRINT or TYPE.

CR — CR Transmit carriage return/line feed codes to the selected output devices.

CURRENT — CURRENT p (X, not V) An integer that indicates the vocabulary into which new words are to be entered.

DECIMAL — DECIMAL Sets the numeric conversion base to decimal mode.

DEFINITIONS

(vvvv) DEFINITIONS (X, not V) Sets the current vocabulary (into which new definitions are placed) to Vocabulary vvvv (the context vocabulary). vvvv need not be specified explicitly.

DIM m n ... p q DIM nnnn (V11) Creates a q dimensional array m+1 by n+1 by ... by p+1 memory words in length.
To access the i, j, ..., k-th element of array nnnn, type i j ... k nnnn; this will leave the appropriate memory address on the stack.
Note: m ()DIM nnn is equivalent to m 1 DIM.

DISCARD DISCARD (N) A null-definition intended for use as a standard REMEMBER word, as some version of DISCARD can always be found in the dictionary.

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.

Loop indices are available to three levels of nesting. Within nested loops, the word I always returns the index of the innermost loop that is being executed, while J returns the index of the next outer loop, and K returns the index of the second outer loop.

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

V8080: DO may be used at interpreter level; see +LOOP.

DP DP p (V) Returns pointer to dictionary pointer.

DPL DPL p (Not V) An integer, pointing to a number-conversion parameter: The number of digits following the fractional point on input or output. A negative value at DPL indicates that no "." was entered on input, or that none is to be printed on output.

DROP m DROP Drop the topmost value from the stack.

DUMP m n DUMP (V) 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.

EDIT b EDIT (LX, Not V) The Editor Vocabulary is loaded, if not already in the dictionary, becoming the context vocabulary. Block b is listed.

EDIT EDIT (V) A constant equal to the block number of the first block of the standard editor. Type "EDIT LOAD" to load the standard editor.

EDITOR EDITOR (X, Not V) 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 (C2) Precedes the false part of an IF-ELSE-THEN conditional or the continuation of a CASE-type conditional.

END f END (C2-) Mark the end of a BEGIN-END loop. If f is true the loop is terminated. If f is false, control returns to the first word after the corresponding BEGIN.
V8080: BEGIN and END may be used at interpreter level. See +LOOP.

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

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

EXIT EXIT (C, Not V) Force termination of a DO-loop at the next opportunity by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered. (V: see TERM)

F F p (KV) An integer pointing to the field length reserved for a number during output conversion.

FLUSH FLUSH Write all blocks that have been flagged as "updated" to disk or tape. Return when output is completed.

FORGET FORGET nnnn Delete 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. For such applications, use a word defined by REMEMBER.

FORTH FORTH (X, not V) The name of the primary vocabulary. Execution makes FORTH the context vocabulary. Since FORTH cannot be chained to anything, it becomes the only vocabulary that is searched for dictionary entries.
Unless additional user vocabularies are defined, new user defintions normally become part of the Forth Vocabulary.

FORTH FORTH b (V) A constant whose value is the number of the first block to be loaded as part of the standard Forth system. Thus after you do a bootstrap load from disk or tape, you type "FORTH LOAD" to load the standard system.

GCH GCH c (Not V) Get a character from the terminal, i.e., return the ASCII code of the next character typed. (V: See TYI.)

GO-TO m GO-TO (EX, Not V) Interrupt interpretation of a block, resume at line m of the current block. GO-TO may only be used during loading of a block.

HEAD HEAD p Returns a pointer to the first location of the last word defined in the current vocabulary.

HERE HERE p Return the address of the next available dictionary location.

HEX HEX (XV8080) 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.

I2 — I2 m (CV11) Equivalent to I 2*. I2 is useful in byte addressing computers to let you index fullwords.

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.

IARRAY — IARRAY nnnn (V) Create a word nnnn that will, at execution time, push the address of its parameter field on the stack. The parameter field is not allocated or initialized. You must initialize these values explicitly, e.g., using ,.

IFEND — IFEND (EX, Not V) Terminates a conditional interpretation sequence begun by IFTRUE.

IFTRUE — f IFTRUE ... OTHERWISE ... IFEND (EX, Not V) 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. See GO-TO.

IM> — IM> nnnn (Not V) Clears the precedence bit of nnnn. Words with the precedence bit set are compiler directives.

IMMEDIATE — IMMEDIATE (CV) Set the precedence bit of the word just defined in the dictionary. Like >IM, but takes no argument.

INTEGER — n INTEGER nnnn (V) Equivalent to VARIABLE.

J — J m (C) Execute J within a nested DO-loop to return the index of the next outer loop.

J2 — J2 m (CV) Equivalent to J 2*.

K — K m (C) Execute K within a nested DO-loop to return the index of the second outer loop.

K2 — K2 m (CV) Equivalent to K 2*.

L< — m n L< f (V) True if m less than n as unsigned numbers. (See U<.)

L<= — m n L<= f (V) True if m less than or equal to n as unsigned numbers. (See U<=.)

L= — m n L= f (V) True if m equals n. (Better notation is =.)

L> — m n L> f (V) True if m greater than n as unsigned numbers. (See U>.)

L>= — m n L>= f (V) True if m greater than or equal to n as unsigned numbers. (See U>=.)

LAST — LAST p An integer pointing to the address of the last dictionary entry made, which is not necessarily a complete or valid entry.

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

LIST — b LIST (VK) List the block b as 16 lines of 64 ASCII characters on the selected output device.

LIT — LIT m (C, Not V) Automatically compiled before each literal encountered in a colon-definition, LIT causes the contents of the next dictionary cell to be pushed on the stack. (V: LIT is anonymous.)

LOAD — b LOAD Begin interpreting block b. The block must terminate with ;S or CONTINUED.

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.
V8080: LOOP can run at interpreter level. See +LOOP.

MAPO — MAPO p (T) An integer pointing to the first location in the tape map.

MAX — m n MAX q Leaves the greater of two numbers.

MESSAGE — n MESSAGE (V) Get line n relative to the first line of block MSGBLK, strip the trailing blanks, and type at the terminal. This word lets you define a large number of messages on disk without tying up main memory.

MIN — m n MIN q Leaves the lesser of two numbers.

MINUS — m MINUS -m Negates a number (2's complement).

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. MK! and MK@ have applications similar to ,CODE and Kitt Peak pseudovariables.

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 memory cells beginning at address p into n cells beginning at address q. The contents of p is moved first; overlapping of data can occur.
(0 10 ! 10 11 4 MOVE clears locations 10 through 14.)

NAND — m NAND n (X, Not V) Logical not-and.

NEXT, — NEXT, (V) An assembler word that may be used to terminate a CODE word. It invokes the Address Interpreter. In V11 and V10 NEXT, assembles a "jump indirect through IC and increment IC" instruction.

NOR — m NOR n (X, Not V) Logical not-or.

NOT — m NOT f (X) 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. See OO.

OO n OO (V) Type n as an unsigned octal number. O. is preferred.

OR m n OR q Bitwise logical inclusive OR.

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

OTHERWISE
OTHERWISE (Not V) An interpreter-level conditional word. See IFTRUE.

OVER m n OVER m n m Push the second stack value.

PAGE PAGE (Not V) Clears the terminal screen or performs a similar action on the current terminal.

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

PCH c PCH (Not V) Transmit a character to the selected output printer device. See TCH. (V: See TYO.)

PRINT m n PRINT (C Not V) Transmit n characters to the selected output printer starting at character address m, which will have been placed on the stack or in an internal register by COUNT.

PRINTER PRINTER (X, Not V) Select a hard-copy printer as the output device for all output directed through PCH or PRINT. See TERMINAL.

QBLOCK b QBLOCK p (X, Not V) Like BLOCK, but may return while previous contents of block are still being written to output device.

R> — R> n (CX) Pop the topmost value from the return stack and push it onto the user stack. See I and >R.

READ-MAP — READ-MAP (T, Not V) Read to the next file mark on tape, constructing a correspondence table in memory (the map) relating physical block position to logical block number. The tape should normally be rewound to its load point before executing READ-MAP.

REMEMBER — REMEMBER nnnn (V920, Not other V) Define a word nnnn which, when executed, will cause nnnn and all subsequently defined words to be deleted from the dictionary. The word nnnn may be compiled into and executed from a colon-definition. The sequence DISCARD REMEMBER DISCARD provides a standardized preface to any group of transient blocks.

REWIND — REWIND (T, Not V) Rewind the tape to its load point, setting CUR=1.

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. (3 ROLL is equivalent to ROT; 1 ROLL is a null operation; 0 ROLL is undefined.)

ROT — m n p ROT n p m Rotate the topmost three stack values.

SEMI, — SEMI, (V11) This word must be used to terminate PDP-11 ;CODE words.

SET — m p SET nnn 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.

SPACE — SPACE (V) Type one space.

SPACES — m SPACES (V) Type m spaces.

SWAB — n SWAB m (V11) Exchange the left and right bytes of n.

SWAP — n m SWAP m n Exchange the topmost two stack values.

TCH — c TCH (Not V) Transmit a character code to the terminal, irrespective of output-device selection. See PCH. (V: see TYO.)

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

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

TYI — TYI c (V) Input one character c, from the keyboard.

TYO — c TYO (V) Output one character to the terminal.

TYPE — m n TYPE (C) Transmits n characters to the terminal, irrespective of output device selection, starting at the character address m. See COUNT, PRINT.

U< — m n U< f (X, Not V) Like <, but unsigned (integer range 0 - 65535). (See L<)

U<= — m n U<= f (X, Not V) Like <=, but unsigned. (See L<=)

U> — m n U> f (X, Not V) Like >, but unsigned. (See L>)

U>= — m n U>= f (X, Not V) Like >=, but unsigned. (See L>=)

UPDATE — UPDATE Flag the most-recently referenced block as updated. The block will subsequently be transferred automatically to disk or tape should its buffer be required for storage of a different block. See FLUSH.

VAR — m VAR nnnn (X) Abbreviation of VARIABLE.

VARIABLE
m VARIABLE nnnn (Not V) Creates a word nnnn which, when executed, pushes the address of a variable (initialized to m) onto the stack. (V: See INTEGER.)

VOCABULARY
VOCABULARY vvvv (EX) Define a vocabulary name. Subsequent use of vvvv will make vvvv the context vocabulary. The sequence vvvv DEFINITIONS will make vvvv the current vocabularly, into which definitions are placed.

WORD c WORD (CN) Read the next word from the input string being interpreted, up to 63 characters or until the delimiter c is found, storing the packed character string beginning at the current dictionary pointer. (V: Delimiter c must be stored in integer DELIM, which is set to blank by WORD.)

XOR m n XOR q The logical exclusive OR.

\! n p \! (V) Store right hand byte of n at byte address p. (B! preferred.)

\, n \, (V) Compile right hand byte of n into next byte of dictionary. Increment DP by 1 byte. (B, preferred.)

\@ p \@ n (V) Return byte at address p in right hand part of n. Left hand part of n may contain the sign extension of the right hand byte. (B@ preferred.)

\MOVE p q r \MOVE (V) Move r bytes beginning at address p to area beginning at address q. (BMOVE is preferred.)

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.

Type EDIT LOAD (SYSTEM DISK EDIT LOAD <user> DISK if file system is loaded) to load the standard editor. Type FORGET EDITOR to release the editor vocabulary.

" " ssss" As described in standard vocabulary above. 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 ".

BLK BLK p An integer that specifies the number of the block you're currently working with.
Example: 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 support of strings; thus the following vocabulary was developed.

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.

Type STRINGS LOAD (STRINGS /LOAD if file system is loaded) to load the strings vocabulary.

^@	p ^@ ssss Get string ssss, located at p, and push it on the string stack. (Byte 0 of the string is its length.)
^!	ssss p ^! Pop ssss from the string stack and store at location p.
^CLR	^CLR Clear the string stack. Note: the string stack is <u>not</u> cleared by ABORT.
^LEN	^LEN n Get length n of top string on string stack.
^-LEN	^-LEN n Get length n of second string on string stack.
^TYPE	ssss ^TYPE Type ssss and pop off string stack.
^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.
^C!	c n ^C! ASCII character c replaces n-th character of top string.
^LEN!	n ^LEN! Set length of top string to n. Equivalent to n 0 ^C!.
^NULL	^NULL ssss Push null string ssss (length 0) on string 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.

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

^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.

-SPACES ssss -SPACES tttt String tttt is ssss with all trailing blanks removed.

^CAT rrrr ssss ^CAT tttt Strings rrrr and ssss are concatenated to form string tttt.

^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.

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

^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.

^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.

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

4.4.3 The Extended Editor.

The Forth Extended Editor (XED) is a superset of the standard editor. 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.

Type XED LOAD (XED /LOAD if file system is loaded) to load the extended editor. XED will automatically load STRINGS. Type FORGET EDITOR to release the XED and STRINGS vocabularies.

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 @ !.

)@ p)@ m Get the contents of address q which is found at location p. Equivalent to p @ @.

)@! p)@! Equivalent to p @ @ 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 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 Double Precision Math.

The Double Precision Math Package (DPMATH) includes operations that deal with 32-bit integers as well as a library of mathematical functions that use 32-bit integers. A double-precision number is represented in (PDP-11) memory by the high-order part in the lower word and the low-order part in the higher word of memory. The left-most bit is the sign, which applies to the full number.

The DPMATH vocabulary was developed first for the 27-m interferometer system (PDP-11/20) by H. W. Hammond. It was carried over to the VLBI Processor (PDP-11/40 = GT44) without major change. The nomenclature followed the then-current Forth usage at other sites. Since that time an improved notation has been adopted.

The original vocabulary uses a "." (period) postfix to indicate double precision. The new system uses a prefixed character to indicate precision and type. "D" and "F" indicate double precision (32-bit) integer and single-precision (32-bit) floating point, respectively. (N.B. The VLBI Processor uses a prefix "F" <u>and</u> postfix "."

to indicate double precision floating point. We have no agreed-upon standard notation for this case, but it seems that a unique one-character prefix ["G" ?] would be preferable to the pre- plus post-fix scheme.) Pure stack operations (<u>SWAP</u>, <u>DUP</u>, etc.) use a prefixed "2" or "4" for double or quadruple word operations. (Such operations may be useful even for single-precision data.)

In the following documentation the original (postfix ".") notation is given first with the newer (preferred) notation second in parentheses.

4.4.5.1 Data Types. -

INTEGER.
: u INTEGER. nnnn Like INTEGER, define nnnn which will push the address of a 32-bit integer (initial value u) on the stack. (2VARIABLE or 2VAR is preferred for new systems.)

C.
: u C. nnnn Like CONSTANT, C. defines a 32-bit constant nnnn which when executed will push value u on the stack. (2CONSTANT, 2CON, or 2C preferred for new systems.)

4.4.5.2 Basic Operations. -

SWAP.
: u v SWAP. v u Exchange top two 32-bit numbers on stack. (2SWAP preferred for new systems.)

DROP.
: u DROP. Get rid of top 32-bit number from stack. (Or, drop 2 16-bit numbers.) (2DROP preferred.)

DUP.
: u DUP. u u Duplicate top 32-bit number on stack. Equivalent to OVER OVER. (2DUP preferred.)

OVER.
: u v OVER. u v u Like OVER for 32-bits.

(2OVER preferred.) !. u p !. Store u at address p.

@.
: p @. u Get the 32-bit integer at location p. (2@ preferred.)

<R. — u <R. Move u to the return stack. (The symbol 2>R is preferred for new systems.)

R>. — R>. u Get u from the return stack. (2R> preferred.)

+. — u v +. w Compute 2's complement sum of u and v. (D+ preferred for new systems.)

-. — u v -. w Compute u - v. (D- preferred.)

MINUS. — u MINUS. -u Negate u. (DMINUS preferred.)

ABS. — u ABS. v Compute absolute value of u. (DABS preferred.)

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.

0SET. — p 0SET. Store a 32-bit zero at p. (20SET is the logically preferred - but confusing alternative notation. Solution: 00SET as a preferred notation?)

1+!. — p 1+!. Add 1 to double precision number at p. (D1+! preferred.)

4.4.5.3 Comparison Operations. -

The following operations provide comparisons equivalent to the words with corresponding names without the terminal ".". These words take 32-bit operands and return a 16-bit logical flag.

0=.	0<>.	0<.	0>.	0<=.	0>=.
L=.	L<.	L>.	L<=.	L>=.	
<>.	<.	>.	<=.	>=.	

Preferred notation:

D0=	D0<>	D0<	D0>	D0<=	D0>=
D=	D<	D>	D<=	D>=	

D<>   D<   D>   D<=   D>=

The following words are comparisons in the address mode which are comparable to their single-precision counterparts. Note: These are used only in the VLBI Processor system. A better notation might use a different prefix ("B" ?).

A0=.   A0<>.   A0<.   A0>.   A0<=.   A0>=.

AL=.   AL<.   AL>.   AL<=.   AL>=.

A<>.   A<.   A>.   A<=.   A>=.

4.4.5.4 Shift Operations. -

In the following an <u>arithmetic</u> shift refers to a shift in which the sign bit never changes when shifting left. When shifting right the sign bit is copied into successive bits to the right. A <u>logical</u> shift treats the sign bit like any other.

ASHIFT — n m ASHIFT r Arithmetic shift, result r = n * 2**(m). If m>0, shift is to left; m<0, to right. (ASH may be preferred.)

LSHIFT — n m LSHIFT r Logical shift left m places. m may be negative. (LSH may be preferred.)

ASHIFT. — u m ASHIFT. v Arithmetic shift like ASHIFT, but for 32-bit integers. (DASH preferred.)

LSHIFT. — u m LSHIFT. v Logical shift like LSHIFT, but for 32-bit integers. (DLSH preferred.)

ROL — n m ROL r Rotate n left m places. Bit 15 (the sign) rotates into bit 0. m may be negative.

LSL — n m LSL r Logical shift left n by m places. (m must be positive.)

LSR — n m LSR r Logical shift right n by m places. (m must be positive.)

ASL n m ASL r Arithmetic shift left by m places. (m must be positive.)

ASR n m ASR r Arithmetic shift right by m places. (m must be positive.)

LSL. u m LSL. v Logical 32-bit shift left by m places. (DLSL preferred, m must be positive.)

LSR. u m LSR. v Logical 32-bit shift right by m places. (DLSR preferred, m must be positive.)

ASL. u m ASL. v Arithmetic 32-bit shift left by m places. (DASL preferred, m must be positive.)

ASR. u m ASR. v Arithmetic 32-bit shift right by m places. (DASR preferred, m must be positive.)

4.4.5.5 Multiplication, Division, And Normalization. -

*. u v *. w Compute w = u*v, low order 32 bits in result. (D* preferred.)

/. u v /. w Compute w = u/v. (D/ preferred.)

Q*. u v Q*. w Computer w = u*v, where u, v, and w are scaled with the binary point to the right of the sign bit, i.e. in the range -1 to +1. The result is the high-order 32 bits of the product. (No obvious preferred notation except to assign a new prefix letter: R* ?)

Q/. u v Q/. w Compute w = u/v, with the same scaling as Q*. (R/ preferred?)

NOR. u NOR. v n Result n is the number of bits u must be shifted left so that bit 15 is different from bit 14; v is the resulting normalized 32-bit number. (DNOR preferred.)

4.4.5.6 Mixed-mode Operations. -

The following words operate on one 32-bit and one 16-bit number.

M* u n M* v Compute the low-order part of the product u*n.

M/MOD u n M/MOD v m Divide 32-bit number u by 16-bit number n to obtain 16-bit remainder m and 32-bit quotient v. Note that remainder and quotient are in reverse order compared to /MOD. (This definition should be changed to correspond.)

M/ u n M/ v Divide 32-bit number u by 16-bit number n to yield 32-bit quotient v.

MOVER u n MOVER u n u Push the 32-bit number on the stack over the 16-bit number.

MSWAP u n MSWAP n u Interchange arguments on the stack.

4.4.5.7 Number Output. -

The following words are provided in the DPMATH package on the 27 m system and on the VLBI Processor:

DD. u n DD. Type variable u with scale factor n. n specifies the number the number of digits to appear to the right of the decimal point. E.g. 123. 1 DD. types 1.23, while 123. 4 DD. types 0.0123.

D.D u D.D Type the decimal value of the 32-bit number u.

OO. u OO. Type the unsigned octal value of u.

The following words are provided on the 10 m system and in some other Forth systems:

D. u D. Type u as a decimal number with as many columns as required.

D.R u x D.R Type u as a decimal number right justified in a field of x columns.

4.4.5.8 Functions. -

SQRT. u SQRT. v Square root. (DSQRT preferred.)

ATAN. u v ATAN. w Arctan(u/v) preserving quadrant information. Result w is in Binary Angular Measure (BAM). (In BAM, 0 degrees = 0, 90 degrees = 40000(8), 180 = -180 = 100000(8), etc.) (DATAN preferred.)

SIN. u SIN. 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. (DSIN preferred.)

COS. u COS. v Compute cosine similar to SIN. (DCOS preferred.)

4.4.6 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. Particularly with the VLBI Processor system, many non-expert persons potentially need to edit blocks. The Forth File System (FFS) is intended to alleviate the problem of disk allocation and protection.

FFS divides the PDP-11 Forth block file (which may be a file within an RT-11 or RSX-11 file structure) 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 in the user file are logical block numbers; FFS maintains a map (User File Directory - UFD) of correspondences between logical and physical block numbers. ("Physical" means numbered in the sense of non-FFS systems; FFS physical block 10 may correspond to an arbitrary hardware disk block when running under RT-11, for example.)

A table of available disk blocks is maintained in block "AVAIL". It is a bit map with each bit signifying 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; i.e. any block reference will cause an error message. The user must assign himself blocks using 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 continuously; 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, 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 by using DISK, e.g., SYSTEM DISK. The file must have been previously defined.

Typical user files will contain software packages such as floating point, VLBI processor software, 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 at 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 group of words that create and manipulate UFDs are accessible through (FILES) LOAD. You must first FORGET FILES, then type (FILES) LOAD. When (FILES) is running, FFS is disabled. It is intended that only system maintainers ("experts") will need to run (FILES).

4.4.6.1 Standard File System Vocabulary. -

The following words are loaded if FFS is implemented in the standard system:

DISK n DISK Set current user file (context) to n. Normally n is provided by a CONSTANT word, e.g. SYSTEM, STRINGS, etc.

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

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

/LOAD n /LOAD Load from block 0 of user file n. The user file which was current before /LOAD is current after /LOAD.

/COPY m n r /COPY Copy block m from user file n to block r of the current user file. Example: MSE DISK 13 DHR 10 /COPY copies block 13 of disk DHR to block 10 of disk MSE.

/EXCHANGE
m n r /EXCHANGE Like /COPY, but the contents of the two blocks are exchanged.

These words are updated to imply references to logical block numbers in the current user file:

LOAD BLOCK LIST SHOW COPY EXCHANGE

4.4.6.2 File Maintenance Vocabulary. -

The following words are accessible by typing FORGET FILES (FILES) LOAD. In this mode, all block references are physical.

UFD UFD Get an available block and designate it as a UFD for a new user file. The file number is typed by UFD. This number should be defined as a constant with the name that will be used to reference the user file.

STAV n STAV Set physical block n "available". (Set corresponding bit in AVAIL to 1.) No check for errors is made.

SNAV n SNAV Set physical block n "not available".

SMFD m n SMFD Store value m in word n in Master File Directory (MFD). Note: n is a word, not byte, address.

LMFD LMFD Dump MFD block.

LUFD n LUFD Dump UFD corresponding to user file n.

XASN m n XASN Define physical block m as logical block n in the current user file. No checks for errors or conflicts are made.

TRANSFER
m n r s TRANSFER Logical block m of file n is transferred to block r of file s. The data are not moved, only ownership is transferred.

APPENDIX A

PDP-11 IMPLEMENTATION.

A.1 GENERAL CHARACTERISTICS.

The DEC PDP-11 is a 16-bit computer architecture that has been realized in many models. OVRO operates 4 distinct types of PDP-11: two PDP-11/40s (VLBI Processor and 10 m telescope control), a PDP-11/20 (27 m interferometer), PDP-11/03s (also known as "LSI-11s", for remote pointing of the 10 m antennas and of the 39 m antenna), and a PDP-11/05 (also known as a "GT40", used for the 1024-channel autocorrelator).

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. Most of the other systems have disk storage and so can run the DEC operating systems. The VLBI Processor and autocorrelator use the RT-11 operating system, while the 10 m control computer runs RSX-11/M.

Forth on the 11/20 is based on a specially formatted 9-track 800 bpi tape. Direct access ("update in place") is possible because long inter-record gaps are written after data blocks. The sequence of records on tape is as follows:

```
      (beginning of tape marker)
                  |
       (long inter-record gap)
                  |
      (12 word label record "1")
                  |
     (standard inter-record gap)
                  |
```

```
        (data record, 1024 bytes)
                   |
                   |
        (long inter-record gap)
       (12 word label record "2")
                   |
      (standard inter-record gap)
                   |
      (data record  2, 1024 bytes)
                   |
                   .
                   .
```

Label records are required to provide indexing so that a new block can be found reliably without rewinding the tape from its current position. The label consists of 12 identical words each containing the number of the data block to follow. (12 words are required so that the label is not treated as a "noise record".)

Data records are found by referring to the labels. An existing data record can be overwritten safely if the tape is positioned by first reading its label record. The long inter-record gaps insure that label records are not erased by updating data.

The direct access tape method is not particularly efficient in use of tape because long interrecord gaps account for about 60% of the tape used. Nevertheless 1000 Forth blocks will fit in 500 feet of tape.

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-A	(RT-11 only) After you stop type out with CTRL-S, you may type CTRL-A to type just one more page of text. This is useful when using CRT terminals or ".GT ON".
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 (but not on the VLBI Processor).
RSX-11: RSX types "MCR>" and you may type any monitor command, such as ABORT. Forth can not be reentered in the current version.

CTRL-O	(RT11 and RSX11) Cancels type out from a running program, but program continues. Allows you to skip lengthy listings. A second CTRL-O turns on type out again.
CTRL-Q	(RT11) After you type CTRL-S to stop type out, you may type CTRL-Q to resume. Type out will not stop again unless you type CTRL-S.
CTRL-S	(RT11) Stops type out from a running program in such a way that no output will be lost. The program continues to run until the output buffer is full. CTRL-Q or CTRL-A may be used to restart output.
CTRL-U	Cancels the entire line you have just typed in. Only effective <u>before</u> 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	B	100000	byte mode (preferred notation)
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) \ INC, INCB 134(R1)
134 1 I) B INC, INCB 134(R1) (preferred notation.)
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 S -) 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, ;
```

Notes:

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

2. Floating point operations are not defined in the basic vocabulary.

APPENDIX B

THE PDP-10 IMPLEMENTATION.

B.1 GENERAL CHARACTERISTICS.

The PDP-10 (DECsystem-10) is a 36-bit computer that uses 7-bit ASCII character codes. The Caltech PDP-10 is operated by the Computing Center and runs the TOPS-10 timesharing system with up to about 45 simultaneous jobs.

Forth for PDP-10 has been written in the MACRO-10 assembly language to run under TOPS-10. Forth relies on the operating system for terminal and disk I/O. It occupies a minimum of 4K words, but may access up to the maximum 56K words normally allowed any (CIT) PDP-10 job.

The character set is the full 7-bit ASCII, with upper- and lower-case characters distinguished. (All standard Forth words are defined in upper case.) Certain control characters are treated specially by the operating system; a partial list of these follows:

<u>Character</u>   <u>Action</u>

CTRL-C (^C)   Stop Forth and return to monitor level. (Two ^Cs will be needed to stop if job is not listening to terminal.)

CTRL-O (^O)   Stop printing at the terminal. Job continues running. A second ^O will resume printing.

CTRL-Q (^Q)   Resume printing after suspended by ^S.

CTRL-R (^R)   Retype current input line. Useful after you've used <rubout> several times.

CTRL-S (^S) Suspend printing at terminal and suspend job. I.e. no output will be lost. Resume with ^Q.

CTRL-T (^T) Monitor types a line giving status of current job: cpu time, core, etc.

CTRL-U (^U) Monitor deletes entire line typed in to date.

<rubout> Monitor deletes last character typed in. Deleted character is echoed after "\" is typed. (<rubout> = <delete>)

Forth's block storage on the PDP-10 is the file found by the file specification: DSK:FORSYS.DAT. The Forth kernel uses "dump-mode" I/O, 2 physical blocks at a time, to retrieve and store Forth blocks. Because a physical block is 128 words long, the Forth block has room for 5 x 256 = 1280 characters in the standard PDP-10 format (left justified, extra bit = 0). So that the last 256 characters are not wasted, the PDP-10 Forth editor operates on 20 (decimal) lines of 64 characters.

Several words peculiar to the PDP-10 environment are provided. SAVE preserves essential Forth information and returns to the monitor. A monitor "SAVE <filespec>" command will then save the Forth core image in such a way that a monitor "RUN <filespec>" command will restart Forth. In this way it is not necessary to use the Forth LOAD each time a program is to be run.

CORE, which takes one argument on the stack, allocates the specified number of 1K word blocks of PDP-10 memory to the Forth job. The stacks are moved up or down, as appropriate. CORE will not allow you to have negative stacks; if you say "0 CORE", you will get the lowest even number of kilowords in which your dictionary plus a modest stack will fit.

A complementary word, CORE?, returns a number on the stack which is the number of memory words unused in the current job, i.e. the distance between the head of the dictionary and the initial stack pointer.

WOPEN is required if you wish to write Forth blocks. As Forth comes up, access to DSK:FORSYS.DAT is read-only. WOPEN opens the file for output. WCLOSE closes the block I/O file for writing, but leaves it open for reading. WOPEN and WCLOSE facilitate sharing of FORSYS.DAT blocks between simultaneous jobs. (TOPS-10 allows only one job at a time to open a file for writing.)

If it is unnecessary to refer to any Forth blocks for a given Forth application, you may type NOFORSYS and then SAVE the core image. When you run the core image, FORSYS.DAT is not opened at all. In this situation the file does not have to be present in your directory. (FORSYS undoes the effect of NOFORSYS.)

The first 16 PDP-10 memory words are special high-speed registers, which are allocated for special Forth functions. CODE words have to respect these allocations at least to the point of restoring critical registers after use. The current register allocations are as follow:

Reg. # (octal)	Name/status
0 - 7	Available
10	V (available)
11	DP (critical)
12	T (critical)
13	TT (available)
14	SP (critical)
15	IC "
16	Available
17	RP (critical)

Register DP is the dictionary pointer; T always contains the same value as the top stack value; TT is an auxilliary register useful in multiply/divide operations; SP is the stack pointer; IC is the Forth instruction counter; and RP is the return stack pointer. Register 16 is left unassigned because it is the register used by Fortran to pass parameters.

B.2 DICTIONARY FORMAT.

A Forth word in the PDP-10 system has the header format shown in Fig. B-1.

```
 -------------------------bit number-----------------------------------
 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1
!-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
!                                 !
!   ABSOLUTE LINK ADDRESS         !             (ZERO)
!                                 !
!-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
!             !             !             !             !             !
! COUNT       ! CHAR.  1    ! CHAR.  2    ! CHAR.  3    ! CHAR.  4    !!
!             !             !             !             !             !
!-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
!
!             CODE SECTION ( MACHINE INSTRUCTION )
!                     1 OR MORE WORDS
!
!-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
!
!             PARAMETER SECTION (OPTIONAL)
!
!-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.
```

Fig. B-1. Forth Word Format for PDP-10.

Word 1 of the header contains only the 18-bit absolute address of the preceding word in the same dictionary branch. The right 18 bits are zeroes, not used in the current version of the system.

Word 2 contains the name of the Forth word, in standard PDP-10 ASCII format: 5 7-bit characters left-justified in the 36-bit word. Actually the first "character" is the character count -- the number of characters in the name. The remaining 4 characters are the first 4 characters of the name. If a name has less than 4 characters, the remaining characters are filled with blanks. The least significant bit of word 2 is used for the precendence bit: a 0 is normal, while a 1 forces execution even in the compile state.

The code section begins in word 3. One or more machine instructions must be present. Optional parameters follow the code section.

B.3 ASSEMBLER.

Most PDP-10 instructions are represented in Forth as "CPU" instructions. Let ADD, be defined by the sequence

270 CPU ADD, .

A complete add instruction may be assembled with the sequence

123 5 ADD, .

This is equivalent to the Macro-10 line

ADD 5,123,

i.e. the contents of location 123 will be added to register 5.

When you execute a CPU word like ADD, , the current stack value is taken as a register specification (possibly including op-code modifiers). The second stack value is taken to be a general address specification -- offset, index register, and indirect bit. These fields are or'ed together with the op-code, the result is stored in the next dictionary location, and the dictionary pointer is incremented.

Some Forth words are defined to assist in specifying the general address value. For example, the sequence

123 6) @) 1 ADD,

pushes octal 26000123, then 1, onto the stack, and assembles the Macro-10 instruction

ADD @123(6)

into the dictionary. The effective address is then the contents of the word whose address is the contents of register 6 plus 123. (The @) adds in 20000000, the indirect bit.)

Op-code modifiers are also defined to reduce the total number of op-codes that are needed to represent the rich PDP-10 instruction set. For example,

123 6) @) 10 M/ ADD,

assembles an instruction equivalent to

ADDM 10,@123(6).

The op-code modifiers correspond to the suffixes used by Macro-10:

I/ - immediate,

M/ - result to memory,

B/ - result to both register and memory, and

S/ - result to self.

Additional modifiers are defined for the halfword MOVE instructions:

HZ/ - fill other half with zeroes,

HO/ - fill other half with ones, and

HE/ - fill other half with extended sign bit.

As an example consider

123 11 M/ HE/ HRR,

which is equivalent to

HRRME 11,123.

A special assembly instruction J, assembles an unconditional jump (JRST) requiring just one stack value, which is the address to which you want to jump.

Arithmetic conditional instructions (e.g. JUMP,, SKIP,, CAI,, CAM,) take modifiers to indicate the sense of the condition:

L/ - .LT.0	G/ - .GT.0
LE/ - .LE.0	GE/ - .GE.0
E/ - .EQ.0	N/ - .NE.0
A/ - always	

If no modifier is used, these instructions will never skip or jump.

The condition to be tested and the register to be tested are determined by stack values at assembly time. The same op-code modifiers used for JUMP,, etc. are used. E.g.

4 LE/ IF, ... THEN,

executes the contained true clause if (at execution time!) the contents of register 4 are less than or equal to zero.

In the case that the current stack value (in register T) is to be tested, some abbreviations are supplied:

IFL, IFLE, IFE, IFA, IFGE, IFG, IFN,.

The definitions go like:

: IFL, T L/ IF, ;.

The following table presents the definitions of the PDP-10 assembler op-codes:

250 CPU EXCH,	251 CPU BLT,	200 CPU MOVE,	210 CPU MOVN,
204 CPU MOVS,	214 CPU MOVM,	500 CPU HLL,	544 CPU HLR,
540 CPU HRR,	504 CPU HRL,	270 CPU ADD,	274 CPU SUB,
220 CPU IMUL,	224 CPU MUL,	230 CPU IDIV,	234 CPU DIV,
400 CPU SETZ,	474 CPU SETO,	424 CPU SETA,	414 CPU SETM,
404 CPU AND,	434 CPU IOR,	430 CPU XOR,	444 CPU EQV,
133 CPU IBP,	135 CPU LDB,	137 CPU DPB,	134 CPU ILDB,
136 CPU IDPB,	264 CPU JSR,	265 CPU JSP,	254 CPU JRST,
266 CPU JSA,	267 CPU JRA,	255 CPU JRCL,	256 CPU XCT,
243 CPU JFFO,	261 CPU PUSH,	262 CPU POP,	260 CPU PUSHJ,
263 CPU POPJ,	240 CPU ASH,	244 CPU ASHC,	241 CPU ROT,

245 CPU ROTC,	242 CPU LSH,	246 CPU LSHC,	252 CPU ABJP,
253 CPU ABJN,	300 CPU CAI,	310 CPU CAM,	320 CPU JUMP,
330 CPU SKIP,	340 CPU AOJ,	360 CPU SOJ,	350 CPU AOS,
370 CPU SOS,	601 CPU TLN,	600 CPU TRN,	621 CPU TLZ,
620 CPU TRZ,	641 CPU TLC,	640 CPU TRC,	661 CPU TLO,
660 CPU TRO,	610 CPU TDN,	611 CPU TSN,	630 CPU TDZ,
631 CPU TSZ,	650 CPU TDC,	651 CPU TSC,	670 CPU TDO,
671 CPU TSO,	047 CPU CALLI,	051 CPU TTCALL,	132 CPU FSC,
144 CPU FADR,	154 CPU FSBR,	164 CPU FMPR,	174 CPU FDVR,
131 CPU DFN,	130 CPU UFA,	140 CPU FAD,	150 CPU FSB,
160 CPU FMP,	170 CPU FDV,		

APPENDIX C

SDS920 IMPLEMENTATION.

C.1 GENERAL CHARACTERISTICS.

The SDS920 (XDS920) is a 24-bit machine using the BCD (6-bit) character set. These two facts set its Forth implementation apart from the more common 16-bit systems. (The only Caltech application of this system is at the 40-m antenna at OVRO.)

Some of the BCD characters cannot easily be represented in this Manual, which is composed on an ASCII system. The representations to be used here, along with the corresponding octal codes, are as follow:

Character	Code
<check>	17
<backsp>	32
<pole>	37
<return>	52
<blank>	60
<tab>	72
<delta>	57
<gull>	75
<fence>	77

Two control devices exist at the 40 m installation: the KSR-35 teletype and the keyboard/Self-Scan Display system. The KSR-35 is a true BCD device while the keyboard/Self-Scan uses the ASCII code. The commonly used characters translate one-to-one between the two codes (a software table is used for this purpose). Some of the less common characters do not map directly; these are listed in the following table:

ASCII		BCD	
Character	Code	Character	Code
@	00	<delta>	57
"	42	\	76
#	43	<check>	17
&	46	<gull>	75
?	77	<pole>	37

In addition the following BCD characters convert to ASCII "~" (34): <backsp>, <tab>, <blank>, <fence>, and <return>.

The following ASCII characters convert to BCD <fence> (77): "~", "{" (or up-arrow), "}" (or left-arrow), "!", and "%".

The Forth word "store" (!) is replaced by = in the '920 system. This is an archaic Forth usage.

The following table summarizes the characters that are recognized from either the keyboard/Self-Scan or the KSR-35 to perform special functions:

Function	ASCII Character	BCD Character
Delete last character typed	RUBOUT	<backsp>
Delete entire line typed	CTRL-SHIFT-K	<fence>
Program interrupt	ALT-MODE	<tab>

Block I/O for the SDS920 is maintained on a 7-track, 556 bpi magnetic tape. The tape is organized in a direct-access format, with a header record preceding every block. The block length is 256 24-bit words. At least 256 blocks are preformatted on the system tape. The tape format is shown in Figure C-1.

```
(Beginning of tape - tape mark)
               |
|--------------------------------|
| Long (3.75 in)                 |
|  Interrecord Gap               |
|--------------------------------|
               |
|--------------------------------|
| Header Record for data         |
|  block # 1                     |
| 50 words, each equal to        |
|       012340001                |
|--------------------------------|
               |
|--------------------------------|
| Normal interrecord gap         |
| (0.75 in)                      |
|--------------------------------|
               |
|--------------------------------|
| Data Block # 1                 |
| 256 words of text or           |
| binary data                    |
|--------------------------------|
               |
|--------------------------------|
| Long Interrecord Gap           |
|                                |
|--------------------------------|
               |
|--------------------------------|
| Header Record for data         |
|  block # 2                     |
| 50 x 12340002                  |
|--------------------------------|
               |
               .
               .
               .
```

Figure C-1 SDS920 System Tape Format.

A set of byte operations for the 920 has been implemented as Programmed Operators (POPs). These are modelled on the byte instructions of the PDP-10.

A data entity called a "byte-pointer" is defined using the following format:

OOB BOO OOO Oww www www www www.

Here ww...w is a normal 14-bit address of a 920 word. BB specifies the 6-bit byte within that word. The left most byte is OO, the right most is 11.

The following POPs all address byte-pointers:

IBP Increment Byte Pointer. Increments the byte pointer by one byte. The word address is incremented if necessary. I.e. byte O of word N+1 follows byte 3 of word N.

DBP Decrement Byte Pointer. As IBP but moves the byte-pointer in reverse ("to the left").

LDBT Load Byte. The byte addressed by the specified byte-pointer is returned in the A register, right-justified.

DPBT Deposit Byte. The right-justified 6-bit byte supplied in A is deposited in the location specified by the byte-pointer. Other bytes in the same word are undisturbed.

ILDB Increment then Load Byte. Increment the byte-pointer then load the byte into A.

IDPB Increment then Deposit Byte. Increment the byte-pointer then deposit the byte in A through the incremented pointer.

Note that these POPs are pseudo-machine operations. As such they are available to the kernel assembly and to <u>CODE</u> words, but not necessarily as Forth dictionary words.

C.2 DICTIONARY FORMAT.

SDS920 Forth is of an older generation than the other Caltech-OVRO systems. Dictionary words do not always have code sections; rather, there is a code address field which points to the code to be executed when the word is referenced.

NEXT in the SDS920 is the routine

```
NEXT LDX *IC
     MIN IC
     BRU *0,2
```

which is effectively a doubly indirect branch through IC.

Figure C.2 demonstrates the format used in the SDS-920 dictionary.

```
                ----------------bit number--------------------
                0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2
                0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
               |-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|
Word 1         | Char.       |  0  |       Link                |
               |  Count      |     |          Address          |
               |-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|
Word 2         | Char.       | Char.      | Char.      | Char.  |
               |    1        |    2       |    3       |    4   |
               |-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|
Word 3         |     0       | P |  0|       Code                |
               |             |   |   |          Address          |
               |-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-|
Word 4         |            Parameter 1                        |
               |               (optional)                      |
               |    .    .    .    .    .    .    .    .    .  |
               .                                               .
               .                                               .
```

Fig. C.2 SDS920 Forth Word Format.

At least three 24-bit words are used for each Forth word. The link is the 14-bit absolute address of the preceding entry in the dictionary. The first dictionary entry has a link of zero.

The first 4 characters of the name of the Forth word are stored in word 2. If the name has less than 4 characters, it is padded on the right with blanks (BCD code 12[8]). The overall length of the name (1 - 64 characters) is contained in bits 0 - 5 of word 1.

The word precedence is contained in bits 6 and 7 of word 3. The absolute address of the code to be executed when the word is referenced is in bits 10 - 23 of word 3. Note that bits 1 and 9 of word 3 (the index and indirect bits) must be left zero.

A Forth word with precedence 2 will be executed at all times when referenced. A word with precedence 0 will be executed when Forth is in the execution state, but compiled when in compile state. The low order bit of the precedence is not used.

C.3 ASSEMBLER.

Four classes of machine instructions are recognized by the SDS920 Forth assembler. The MCPU class includes all memory reference instructions together with others such as shifts and EOMs. The Forth word MCPU takes as input an op-code of up to 12 bits. The op-code is associated with the name following in the input stream. Thus

760 MCPU LDA,

defines the assembler instruction LDA, (load A register) with op-code 0760 (octal). Strictly, the op-code is 76. An extra digit is provided on the right to facilitate the shift instructions.

When referenced, LDA, will take the (then) current stack value, "or" it with 760 * 2**12, and store the result at the next available dictionary location. The dictionary pointer (DP) is then incremented by one. Assume that the value of DP is 1000. The Forth line

4521 LDA,

is equivalent to the Symbol assembler line

LDA 4521

and assembles the octal number 7604521 in location 1000; DP is incremented to 1001. All MCPU-defined words work analogously; only the op-codes differ.

Certain words are defined to assist in specifying the address part of an MCPU-type instruction. The word (X sets the index bit (bit 1) of the current stack word, while (I sets bit 9, the indirect addressing bit. Thus the line

21072 (X (I LDA,

is equivalent to the Symbol line

LDA *21072,2 ;

it assembles the number 27661072 into the dictionary.

The word CPU is used to define all fixed-format instructions that do not reference memory. An example of this class is the register operation CLA, (clear A register). This instruction is defined by the line

04600001 CPU CLA, .

When CLA, is executed, the constant 4600001 is assembled into the current dictionary location.

Two instruction classes have been defined specifically for the W-buffer I/O instructions. WOP is used to define the major EOM instructions. It takes one argument when defining the op-code: the complete EOM code for the number of characters/word and the unit number of the device involved. For example,

00202001 WOP RKB,

defines the RKB, (read keyboard) instruction for the assembler. When referenced, RKB, uses the current and second stack values to determine the unit number and number of characters per word, respectively. Thus

4 1 RKB,

is equivalent to the 920 Symbol expression

RKBW 1,4 ;

it assembles 0202601 into the dictionary.

Finally the TWOP instruction class defines those control EOMs or SKSs which need a unit number but do not have a "C/W" field. An example is TRT, (tape ready test) which is defined

04010411 TWOP TRT, .

The line

2 TRT,

assembles 04010412, the equivalent of the Symbol line

TRTW 2.

Recognized SDS920 assembler codes are given in the following table:

760 MCPU LDA,	350 MCPU STA,	750 MCPU LDB,	360 MCPU STB,
710 MCPU LDX,	370 MCPU STX,	770 MCPU EAX,	620 MCPU XMA,
550 MCPU ADD,	570 MCPU ADC,	630 MCPU ADM,	610 MCPU MIN,
540 MCPU SUB,	560 MCPU SUC,	640 MCPU MUL,	650 MCPU DIV,
140 MCPU ETR,	160 MCPU MRG,	170 MCPU EOR,	010 MCPU BRU,
410 MCPU BRX,	430 MCPU BRM,	510 MCPU BRR,	400 MCPU SKS,
500 MCPU SKE,	730 MCPU SKG,	600 MCPU SKR,	700 MCPU SKM,
530 MCPU SKN,	720 MCPU SKA,	520 MCPU SKB,	740 MCPU SKD,
460 MCPU RCH,	660 MCPU RSH,	662 MCPU RCY,	670 MCPU LSH,
672 MCPU LCY,	671 MCPU NOD,	000 MCPU HLT,	200 MCPU NOP,
230 MCPU EXU,	020 MCPU EOM,	120 MCPU MIW,	320 MCPU WIM,
130 MCPU POT,	330 MCPU PIN,		

4600001 CPU CLA,	4600002 CPU CLB,	4600003 CPU CLR,
4600004 CPU CAB,	4600010 CPU CBA,	4600014 CPU XAB,
4600012 CPU BAC,	4600005 CPU ABC,	24600000 CPU CLX,
4600200 CPU CXA,	24600003 CPU ALL,	4600400 CPU CAX,
4600600 CPU XXA,	4600020 CPU CBX,	4600040 CPU CXB,
4600060 CPU XXB,	4600122 CPU STE,	4600140 CPU LDE,
4600160 CPU XEE,	4601000 CPU CNA,	

```
4020400 CPU BPT1,    4020200 CPU BPT2,    4020100 CPU BPT3,
4020040 CPU BPT4,    4020001 CPU OVT,     0220001 CPU ROV,
0220002 CPU EIR,     0220004 CPU DIR,     4020004 CPU IET,
4020002 CPU IDT,     0250000 CPU ALC,     0200000 CPU DST,
0214000 CPU TOP,     0212000 CPU ASC,     4020010 CPU BET,
4021000 CPU BRT,

202004 WOP RPT,      202044 WOP PPT,      200044 WOP PTL,
203006 WOP RCB,      202006 WOP RCD,      202001 WOP RKB,
202041 WOP TYP,      203011 WOP RTB,      202011 WOP RTD,
203031 WOP SFB,      202031 WOP SFD,      207031 WOP SRB,
206031 WOP SRD,      203051 WOP WTB,      202051 WOP WTD,
203071 WOP EFT,      207071 WOP ERT,

0212006 TWOP SRC,    4012006 TWOP CRT,    4014006 TWOP FCT,
4011006 TWOP CFT,    4010411 TWOP TRT,    4014011 TWOP FPT,
4012011 TWOP BTT,    4011011 TWOP ETT,    0214011 TWOP REW,
4013610 CPU  TFT,    4012610 CPU  TGT,    0214000 CPU  RTS,
0213610 CPU  SRR,
```

The assembler conditionals are listed in the following table:

Word name	test	Word name	test
<u>IFL,</u>	A.LT.0	<u>IFGE,</u>	A.GE.0
<u>IFLE,</u>	A.LE.0	<u>IFG,</u>	A.GT.0
<u>IFE,</u>	A.EQ.0	<u>IFN,</u>	A.NE.0

(Fortran notation for arithmetic comparison is used in the table.) These operations test the value of the A register. For example, <u>IFL,</u> assembles instructions which test for A less than zero. See Section 3.8 for the general <u>IF,</u> <u>ELSE,</u> <u>THEN,</u> constructions.

APPENDIX D

QED - QUICK EDITOR.

Dave Dewey has developed a new editor to take advantage of the high-speed CRT available on the GT40 and GT44 versions of the DEC PDP-11. The following is Dave's description of his editor:

The block being edited is always visible on the screen, so the results of any editing are immediately available to the operator. Most commands are only a few keystrokes, and a cursor indicates the current point of editing. A brief resume of applicable commands appears below the block being edited, and thus this instruction summary is needed only for reference. In fact, QED will be much easier to learn by just reading the first page of this manual and the experimenting with it than by attempting to digest all of its capabilities before trying it out.

A. LOADING THE EDITOR INTO THE DICTIONARY QED has been improved to allow editing with GT ON or OFF. Typical start-up sequence:

```
(Bootstrap the system)
R FORTH
FORTH LOAD

XXX DISK                  (replace XXX with the
appropriate name)
QED /LOAD
```

QED does not redefine the standard system words, so (unlike EDIT or XED) other FORTH programs may be loaded and run on top of QED, as space permits. Even XED may be loaded on top of QED. Interactive editing and debugging is thus hastened. When you no longer need QED, remove it from the dictionary with

    FORGET EDITOR

(If you have also loaded XED, the FORGET line must be typed twice, once for each editor.)

B. LOOK MODE

To look at block NNN, use the command

    NNN Q

(To look at the block most recently listed or edited, just type

    QQ   )

The screen will display this block as well as a summary of the possible commands to QED while in LOOK mode.

> Note: The "^" preceding a character does not mean to type a carat; rather, it means to hold down the CTRL key while typing the character it precedes.

Key Result
^X Display the next block following the current one.
^W Display the block previous to the current one.
^Z Quit--return to FORTH.
^P Prepare to edit--switch to EDIT mode.

One can skim through a series of blocks in search of a particular one exceedingly fast using ^X and ^W.

C. EDIT MODE. br;Assume that you have located the block that you wish to edit, using the aforementioned commands. A ^P will set up EDIT mode, which has these properties:

1. An L-shaped cursor will appear at the beginning of the block. (Future references to the "current" position in this writeup will refer to the cursor's location. It is always between two character locations, and its vertical bar indicates that point.)

2. More commands are now available to the user, and the summary at the bottom of the block grows to reflect this.

3. The ^X, ^W, and ^Z commands perform an extra function while QED is in EDIT mode: the current block is briefly checked for these common mistakes:

    1. no ";S" in the block

2. last char of the block non-blank
3. runons: last char of one line and first char of the next one non-blank.

If there is one of these mistakes, QED will let you know and you may fix them immediately. If, however, this unusual block structure is purposeful, repeating the ^X, ^W, or ^Z will override the error check. If all is well, the block will be immediately flushed to the disk and the traditional ^X, ^W, or ^Z function will occur. (Whenever you go to a new block, QED is reset to LOOK mode.)

C.1 BASIC EDITING COMMANDS

<Text>: Any legal FORTH block character, including space, will be inserted just before the cursor. The cursor and rest of line will move out of the way as needed.

It is conventional in FORTH to indicate that a given line is a continuation from the previous one by indenting the continuation line two spaces (possibly more). The indentation is ordinarily ignored by FORTH just as spaces anywhere else are. (An exception is any field which is interpreted as literal characters, for example [...] or " ... " , in which the spaces are not ignored.) The only reason for the indentation is to make the block easier to read by the programmer.

QED does the "right thing" with attempts to put characters beyond the end of the line. Such a situation can occur in one of two ways:

A. Inserting text when the cursor is at the end of a line.
B. Inserting text in a line whose last character is non-blank.

When QED sees such an attempt, it does one of three things:

1. If the next line is a continuation line and it has room for the word which is about to pop off of the end of the current line, QED pushes that word onto the beginning of the next one. The

indentation is kept the same, and one space is inserted between the just-pushed word and the old contents of that line.

2. If the criteria in #1 are not met, QED attempts to do a <CR> <space> <space> just before the word about to be popped off of the current line. In other words, it starts a new continuation line.

3. If #2 was attempted but no free line was found in the block to do the <CR>, the attempt to insert a character is ignored. An appropriate error message is given, and the block is left in its previous condition.

In all three cases, QED refrains from breaking any words--that is, any string of non-blank characters will be put entirely on one line or the next, instead of starting on the end of a given line and finishing at the beginning of the following one.

As a result of this special treatment, one can insert characters at any point in a block without paying attention to boundary conditions. As long as there is room in the block, QED will shuffle its contents to make room for the text being inserted. The cursor moves in step with such shuffling. One cannot accidentally delete any non-blank characters or lines by inserting text.

<RUBOUT>: This deletes the character preceding the cursor. The rest of the line, as well as the cursor, moves to the left one column. <RUBOUT> is useful not only in fixing just-typed data, but also in deleting any incorrect characters before the cursor.

<CR>: <CR> first makes sure that a blank line follows the current one. If not, it gets one from elsewhere in the block (preferring ones near the bottom) and inserts it. Then <CR> moves the cursor (and any chars which may follow it) to the beginning of the next line. Notice that <CR> will not delete any non-blank lines--if no blank lines are available, it aborts.

The previous commands are all that are needed to create a FORTH block. The following ones are added to ease editing.

C.2 CURSOR MOVEMENT COMMANDS

key command position of cursor

```
^T TOP    just before the first character of the block
^B BOTTOM just after the last character of the block
^Y -LINE to the previous beginning-of-line (on  the  current
line
                unless the cursor is at column 1)
^N +LINE the next beginning-of-line
^H -WORD the previous beginning-of-word, where a word is any
sequence
                of    non-blank     characters.      (      a
                beginning-of-line      counts      as      a
                beginning-of-word, as does the location  one
                space after the last word in a line.)
^L +WORD the next beginning-of-word
^J -CHAR the previous character
^K +CHAR the next character

<TAB> or TAB the next tab stop (Tab  stops  are  permanently
 set every
^I              8 columns as usual)
```

Any attempt to move the cursor beyond a block boundary (beginning or end of block) will result in a position at that boundary.

With the exception of <TAB>, all of the cursor movement commands may be typed with the right hand, allowing the left one to hold down the CTRL key. If the fingers are resting one key to the left of typist's "home" position, the direction and magnitude of movement roughly correspond to the location of the key. (See keyboard diagram.)

C.3 DELETE PREFIX: ^D

A ^D changes the operation of the single character following it. To let the user know that QED is waiting for that second character, ^D causes the cursor to start flashing. It can have two functions:

^D <cursor-moving-key> (DELETE PATH) Instead of moving the cursor, all of the characters along the expected path are deleted. Any lines which end up being all blank by this process are removed.

^D <CR> (CONCATENATE) In effect, this deletes the next CR, to allow concatenation of lines. The next line is moved to the end of the current one. No matter how many leading blanks the following line may have, CONCATENATE inserts exactly one blank between the two segments of the resultant line. (If there is insufficient room at the end of the current line to append all of the next one, as much as will fit is so moved. FORTH words will not be divided.)

At the completion of CONCATENATE, the cursor is positioned between the two resultant segments, at the end of the original first line.

C.4 USE OF THE SAVE BUFFER

Some or all of a FORTH block may be saved, to be later inserted--the contents may be inserted elsewhere in the same block, in a different block, or even in a different disk. The save buffer is particularly useful for changing the order of lines in a block or for duplicating portions of a block. Additionally, one can put a template block in the save buffer to expedite the creation of a series of similar but non-identical blocks.

Three commands manipulate the save buffer:

^F FLAG LINES Flag mode is set. While in flag mode, all lines that the cursor touches are marked at their left edge with a rectangular flag. (The current line is flagged immediately.) The operation of all other commands of QED is unchanged by ^F.

Flag mode, and all flags, are cleared by ^V as well as those commands which initialize a block (^X, ^W, ^R, ^P).

^V SAVE FLAGGED LINES All flagged lines are copied into the save buffer. The block's contents are unaltered, but the previous contents of the save buffer are lost. (Therefore, if there are no flagged lines, ^V will clear the save buffer.) At the completion of ^V, flag mode and all flags are cleared.

Notice that the only way to change the contents of the save buffer is with ^V. Even if you exit from the editor with ^Z, QED faithfully remembers what was last saved. (QED /LOAD initializes the buffer to zero; from then on, it is only altered by ^V.)

If you accidentally flag more lines than you want to save, just hit ^V which clears all the flags. Set the flags that you want and then hit ^V again.

^U UNSAVE The contents of the save buffer are inserted before the current line. The save buffer contents are unchanged. The cursor will then be at the beginning of the line following the last inserted line. (If the last inserted line was at the bottom of the block, the cursor will be at the end of the block.)

^U will abort with a message if there are more saved lines than free lines in the block. (These blank lines need not be contiguous, nor need they be at the cursor. ^U will move the blank lines as needed, without changing the order of non-blank lines.)

C.5 'RESCUE' COMMAND

The fact that editing is so easy and fast with QED means that mis-editing is also easy. After entering edit mode with ^P, one might attempt to delete the first line with ^D ^N, but accidentally type ^D ^B, thus clearing the entire block. The rescue command has been added for just such an occasion. Realize, though, that the block you see before doing a successful ^Z, ^W, or ^X is the block that will be on the disk. Flushing is automatic in that case, and you will have to re-edit the block if it was wrong. Assuming that you have realized your error in time, here is the way out:

^R RESET BLOCK The block is reset to its condition just before EDIT mode was most recently entered: its contents are restored and QED returns to look mode.

D. ERROR HANDLING

QED is designed to be reasonably intelligent, and it should catch any illegal command sequences, responding with an informative message. Attempts to use <CR> and Text when there is insufficient room will be similarly caught. The only way I have found to bomb the system is to hit two ^C's in quick succession. (Incidentally, one ^C will return to RT-11 monitor without altering the previous contents of the current block.

A side benefit is this: if you have inadvertantly hit ^D, but do not wish to delete anything, just hit any text character. QED will give you the error message and ignore both the ^D and the text char.)

D.1 TREATMENT OF BAD AND UNUSUAL BLOCKS

A. UNASSIGNED BLOCKS

Any attempts to look at a block which has not been assigned to the current disk context will be refused, and a message will be given.

B. DISK I/O PROBLEMS Very occasionally FORTH will have trouble reading a block from the disk. QED will most likely crash with a message like "Q?". The picture may remain on the CRT. (An attempt to list such a block will also fail.)

If the disk I/O error occurred as a result of ^X, then the block after the current one is at fault; if it occurred upon ^W, then the previous block is bad.

In any case, it is recommended that a new bootstrap is done to reset any possibly altered parts of FORTH or RT-11.

Most likely the error occurred as a "random" glitch, but it is possible that it is a "hard" error. To check, try fixing the block by copying a good one into it. Hopefully the error will be eliminated and QED will be happy.

If the block is still bad, make note of the disk and block number to let a "system expert" fix the block. In the meantime, avoid any accesses of that block. Do not just release that block and assign another--if you do, some unsuspecting user will wind up with a bad block!

C. BLOCKS WITH UNUSUAL CONTENTS FORTH blocks ordinarily contain only SIXBIT text characters. The SIXBIT codes are the ASCII values 040 through 137 (octal). Here they are in numerical order:

```
(space) !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
```

Note that lower case letters are not SIXBIT.

Some of the earlier FORTH blocks have been initialized to zeros rather than spaces. Therefore, QED will interpret a zero byte as a space. In fact, after you edit a block and exit QED, all such nulls will have been replaced by spaces.

Any other character values are illegal. The display will reveal such illegal characters when you look at such a block: the dotted line which ordinarily marks the position just after column 64 in each line will be located one space to the left for each illegal character in that line. (The illegal characters do not print or take up space in the line.)

QED assumes that the existance of such non-SIXBIT characters indicates that the block is used for numeric data storage (like the Master File Directory block) rather than for character storage. It protects such blocks by making it illegal for you to edit them with QED. If you really do want to edit them, you must eliminate the bad characters by clearing the block, copying another block into it, or editing the bad lines with XED or EDIT.

D.2 ERROR MESSAGES

SORRY, NOT ENOUGH ROOM TO LOAD QED. MUST FORGET SOMETHING FIRST.

This happens if you attempt to do a QED /LOAD when there is not enough room in the dictionary for QED. QED requires about 5200 (decimal) words of memory, and if there isn't 5400 words of space (allowing for stack usage) QED just won't load.

NO SUCH BLOCK EXISTS. This message appears if you ask QED to look at a block which has not been assigned to the current disk context. Causes: ^W ^X (or, from FORTH, Q or QQ)

NOT ENOUGH ROOM. This indicates that you have attempted to insert something into the block, but there isn't enough room. In the case of CONCATENATE, the message means that there is not sufficient free space at the end of the current line to append even one of the words from the next line. Causes: Text <CR> ^U CONCATENATE

CAN'T RUBOUT BEYOND BEGINNING OF LINE. Rubout deletes the character just before the cursor. None exists at the beginning of the line. Cause: <RUBOUT>

<CHAR> IS ILLEGAL-IGNORED. Cause: anything other than a QED command.

<CHAR> WHEN NOT IN EDIT MODE IS ILLEGAL-IGNORED. Cause: anything other than ^G ^W ^X ^Z ^P while in look mode.

<CHAR> AFTER ^D IS ILLEGAL-IGNORED. Cause: anything other than <CR> or a cursor-moving-key (^T ^B ^Y ^N ^H ^J ^K ^L <TAB>), after hitting ^D.

NO SAVED LINES Cause: ^U when SAVED LINES = 0.

NO FLAGGED LINE. Cause: ^V when there are no flagged lines.

WARNING: NO ';S' OR RUNONS FROM ONE LINE TO THE NEXT OR LAST CHAR OF BLOCK NOT BLANK. REPEAT COMMAND TO CONFIRM EXIT.
This message indicates that the current block is not in the typical FORTH format and is therefore likely incorrect. Causes: ^W ^X ^Z (See above, under EDIT MODE.)

NO LINE FOLLOWS THIS ONE TO CONCATENATE. Cause: ^D <CR> while the cursor is at line 16.

NON-CHARACTER DATA FOUND! FIX BLOCK BEFORE EDITING WITH QED. Cause: ^P when current block has non-SIXBIT bytes. (See BLOCKS WITH UNUSUAL CONTENTS, above.)

D.3 ERROR MESSAGE DEFEAT COMMAND Printing of error messages can take an appreciable time, particularly with GT OFF. QED allows the operator to cancel error message printing, although the "beep" associated with an error will still occur, alerting the user that some kind of error has occurred. If error printing is disabled and such a beep comes at an unexpected time, just re-enable error printing and repeat the command that caused the beep.

^G FLIP ERROR PRINTING ENABLE If errors will currently print, disable such printing. If errors will not currently print, enable such printing.

D.4 MISCELLANEOUS ERRORS

Dave Rogstad's correlation program alters various things throughout the PDP-11's memory; some of that alteration is not restored even by a ^C. On occasion I have seen a QED print three <CR><LF>'s upon entering each block while GT was OFF. This indicates that the VLBI program has set the bits in the computer which show GT to be ON, even though it is not. The remedy is to re-bootstrap the PDP-11 and then do the editing. (You may also type the monitor command ".IN" to reset these bits without reloading. - MSE)

E. TECHNIQUES

;To get to the end of a line: use ^N or ^Y as needed to get to the beginning of the following line, then use ^J.

To delete the rest of the block: ^D ^B
(kills whole block if cursor at beginning of block.)

To delete the rest of the line: ^D ^N
(kills whole line if no chars precede cursor.)

To delete the beginning of the line: ^D ^Y
(kills previous line if cursor at column 1)

To compress several short lines of code into a few long lines:
Move the cursor to the first line of the chosen sequence. Do ^D <CR> until there is no more room in the first line, then move to the next one and repeat, etc.

To push words off of the end of the current line and onto the beginning of the next one: Move the cursor to the beginning of the current line with ^Y if it is not already there. Hit <space> repeatedly until the desired number of words has been moved to the next line. Then use <RUBOUT> the same number of times to shift the current line back to its original position.

There are no search commands in QED, since the need for actual searching is so rare. On those few occasions where a search need be made one can either visually scan the blocks with ^X and ^W or use the FT command in XED.

APPENDIX E

FORTH BIBLIOGRAPHY.

1. Anon., Forth Introductory Programmer's Guide, Forth, Inc., Manhattan Beach, California, 1975.

2. Anon., Forth Programmer's Technical Manual, Forth, Inc., Manhattan Beach, California, 1975.

3. Ewing, Martin S., The Caltech Forth Manual, First Edition, Owens Valley Radio Observatory, California Institute of Technology, Pasadena, California, 1974.

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

5. Hollis, Jan M., 36 Foot Telescope Computer System Manual, National Radio Astronomy Observatory, Charlottesville, Virginia, Computer Division Internal Report No. 18, 1975.

6. James, John S., FORTH for Microcomputers, Dr. Dobb's Journal of Computer Calisthenics & Orthodontia, No. 25, 3, Issue 5, pp. 21-27, 1978.

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

8. 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.

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

10. 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.

11. Sachs, Jonathan, An Introduction to Stoic, Technical Report BMEC TR001, Harvard-MIT Program in Health Sciences and Technology, Biomedical Engineering Center for Clinical Instrumentation, June, 1976.

12. Sinclair, W. S., Forth: A Stack Oriented Language, Interface Age, September, 1976.

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

14. Stein, P., The FORTH Dimension: Mini Language Has Many Faces, Computer Decisions, November, p. 10, 1975.