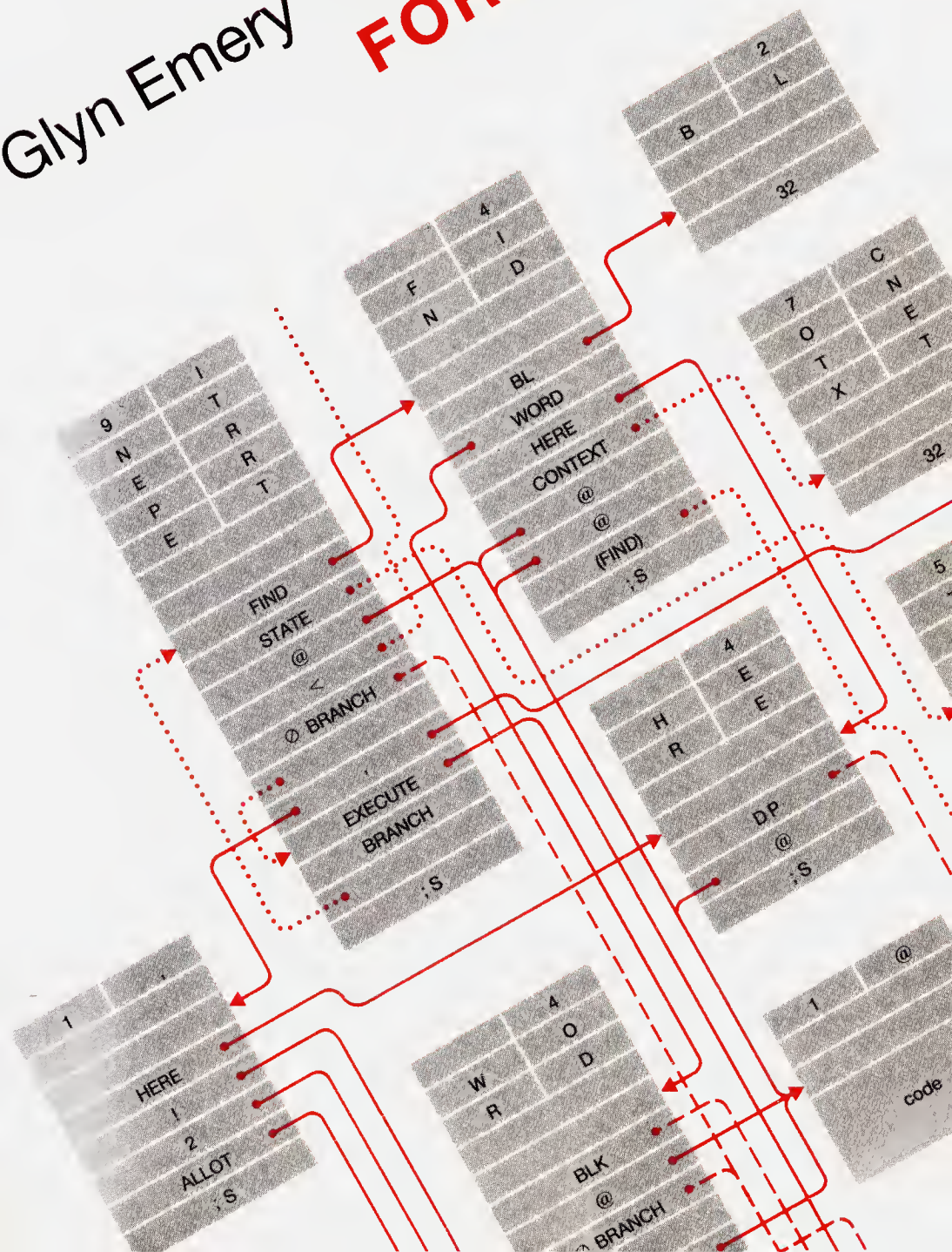
Blackwell Scientific Publications

Computer Science Texts

Glyn Emery

The Students' FORTH



COMPUTER SCIENCE TEXTS

The Students' FORTH

GLYN EMERY
lately Professor of Computer Science,
University College of Wales, Aberystwyth

BLACKWELL SCIENTIFIC PUBLICATIONS
OXFORD LONDON EDINBURGH
BOSTON PALO ALTO MELBOURNE

Blackwell Scientific Publications
Editorial offices:
Osney Mead, Oxford, OX2 0EWL
8 John Street, London, WC1N 2ES
9 Forrest Road, Edinburgh, EH1 2QH
52 Beacon Street, Boston
Massachusetts 02108, USA
667 Lytton Avenue, Palo Alto
California 94301, USA
107 Barry Street, Carlton,
Victoria 3053, Australia

First published 1985

Phototypeset by
Oxford Computer Typesetting

Printed and bound in
Great Britain at
The Alden Press, Oxford

British Library
Cataloguing in Publication Data

Emery, Glyn
The students' FORTH.—(Computer science texts)
1. FORTH (Computer program language)
I. Title II. Series
001.64'24 QA76.73.F24

ISBN 0-632-01436-9

Distributed in North America by
Computer Science Press, Inc.,
11 Taft Court,
Box 6030, Rockville,
Maryland 20850, USA

Contents

Preface

This book is more than an instruction manual on FORTH, for it attempts to provide information not only on how to use the language but also on the way it can be implemented. Indeed the intelligent reader, by the time that he has read to the end, should be in a position to implement his own version of FORTH should he wish to do so. The book is aimed at the reader who has already gained some experience in programming (possibly in BASIC), who is accustomed to thinking in algorithmic terms, and who has realised that the programmer has to make an effort to understand the operation that is really performed by a sequence of commands, in contrast to what its designer intended it to perform.

Although the technical detail may seem frightening to the newcomer, the treatment is progressive. The reader is led into the language step-by-step, so that by the end of Chapter 3 he should be capable of writing quite complex systems. If he wishes, he can postpone reading the technical material until he needs it to answer queries that will inevitably arise in the course of using the language, for a language as close to the machine as FORTH cannot be properly understood without some appreciation of its mechanism. A number of exercises have been included in the text, and model answers are provided at the end of the book. Programming exercises have deliberately been simplified. When he has had some experience the reader can improve upon them to make them more realistic.

FORTH is an unusual concept. It has been heavily criticised by those who do not understand its virtues; but it is not difficult to use once its rules have been mastered; and it does permit systems to be developed interactively. Moreover it gives rise to fast code that occupies much less memory than would be possible with a well-structured compiled language. From an educational standpoint FORTH can provide the same sort of insight into the workings of a binary machine that assembler does; but FORTH is simpler to use than an assembler, being susceptible to examination at every stage without the need for a debugger, and providing its own operating system.

FORTH was invented by C. H. Moore, and was first used, of all things, to control a radio telescope. Moore invented it because what was

available at the time did not give him the ease of use, speed, and storage economy he wanted. Its name arises because it was his fourth attempt, and it is so spelt because it was initially implemented using a version of Fortran that permitted names of up to five characters only. It has spawned a few imitators, such as STOIC and CONVERS; but here we shall be concerned only with implementations that bear the FORTH name.

There is no official authority for FORTH, though there is a FORTH Standards Team that from time to time issues specifications that have much authority. We have followed here in the main the FORTH-83 standard*, which superseded the still widespread 79-standard; but we include a number of words derived from other implemtations as well. In particular we refer to MMSFORTH from Miller Microprocessor Services†, and the multi-user POLYFORTH.

We follow the implementation recommendations of the FORTH Interest Group (fig) when explaining typical ways in which things work. Nevertheless, every implementation differs in detail (doubtless with good reason) from its predecessors; so we have tried to point out the most likely areas of disagreement. The FORTH Standards Team themselves admit that "...the choice to deviate is acknowledged as beneficial and sometimes necessary." We give a glossary at the end of the manual; but the reader is advised to use this in conjunction with the glossary for his own system, and to note carefully any differences.

Aberystwith
January 1985

* Obtainable from Forth Standards Team, P.O. Box 4545, Mountain View, Ca 94040, USA.

† 61 Lake Shore Road, Natick, Mass 01760, USA.

Chapter 1

HSILOP

The conventions we use when writing algebraic expressions have been with us since the sixteenth century. Their only advantage — though it is a very big one — is that we are accustomed to them; but they have disadvantages that we can easily illustrate. The value of the arithmetic expression $3 + 4 \times 5$ is 23; but if you press in turn the keys

$$3 + 4 \times 5$$

on a cheap pocket calculator, you get the "answer" 35. The discrepancy is due to operator precedence. The calculator recognises no precedence. It simply applies the operators in turn as they are keyed in; thus $3 + 4$ gives 7, 7×5 gives 35. In the conventions of ordinary algebra, however, the multiply operator has a higher precedence than the add operator, and should be applied first. If we had really wanted the addition to be done first, then we should have had to use parentheses, thus

$$(3 + 4) \times 5$$

However, we can get the right answer on our calculator by altering the order of evaluation and keying

$$4 \times 5 + 3$$

instead.

The actual evaluation sequence for this last expression might be described as

```
take      4
take      5
multiply
take      3
add
```

We can write this sequence in short as

$$4\ 5 \times 3 +$$

in which we imply the simple evaluation rule:

"working from left to right, apply each operator in turn to the two values preceding it", with the general undertanding that the result of a previous operation is available for use by a succeeding operator.

There are several other ways in which we could have written the same expression to give the same result, for instance

$$3\ 4\ 5\ \times\ +$$

Notice that in this case the 3 is kept in hand until after the multiplication has yielded the single value 20.

EXERCISE 1. Using the same convention give an expression that corresponds to the parenthesised $(3 + 4) \times 5$. What are the steps in its evaluation?

POSTFIX NOTATION

What we have just been doing is evolving an alternative way of writing algebraic expressions, one moreover that corresponds more closely with the actual evaluation process. This new notation is known as postfix, since each operator is written after the values to which it is to be applied, in contrast to the conventional infix notation in which the operator is written between the two values. It is also known as reversed Polish notation, hence the title of this chapter*.

Postfix notation has the added advantage that there is no need for brackets; the order in which the operators are applied is determined solely by their positions in the sequence. Thus the infix expression

$$(A + B) \times (C - D)$$

becomes

$$A\ B\ +\ C\ D\ -\ \times$$

in postfix. Nor is there any need for operator precedence in quite the same way as in infix notation. The infix expression

$$3 + 4 \times 5$$

is capable of two different interpretations as we saw, and as our pocket calculator illustrated; so a precedence rule had to be devised.

* Polish, or prefix, notation, in which the operator precedes the operands to which it is to be applied, is so named in honour of its inventor, the Polish logician Jan Łukasiewicz.

EXERCISE 2. Translate the infix expressions

(i) A × (B − C) + D

(ii) (A + B) × (C − D)

into postfix notation.

STACKS

Let us now attempt to devise an algorithm to evaluate postfix expressions. This is not difficult — indeed there are several pocket calculators on the market that actually require their input to be in postfix form, thereby avoiding anomalies such as the one we referred to earlier. What we need is a data structure that will allow us to store values in the order in which they are supplied (by direct input or as results of earlier calculations) and return them to us in the order in which we want them. In other words, we want to impose a last-in-first-out (LIFO) discipline upon our data. A suitable structure is well known: it is called a *stack*. We may think of a stack as a sequence of items that is accessible at only one end; consequently we can only extract the last item that we put in. For convenience, let us imagine that our stack is vertical with the accessible end at the top. The rules for evaluating postfix expressions then become:

(i) evaluate expression from left to right,
(ii) push each value in turn on to the stack,
(iii) apply each operator to the top two items removing them both,
(iv) leave the result in the top position.

To illustrate these rules, consider the postfix expression

5 4 + 5 2 − ×

(What is the infix equivalent?) Successive configurations of the stack are shown in Fig. 1.1. Notice that we have established here a further

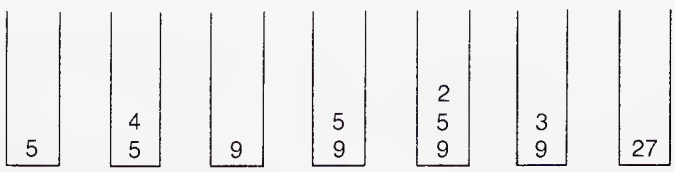


Fig. 1.1.

convention: that in the non-commutative operation of subtraction the operands appear in the same order as they do in normal infix. We shall follow a similar convention regarding division and comparison operators. Notice too that though there are four numbers and two intermediate values only three stack locations are used. This illustrates a further advantage of stack evaluation. Less working space is needed, since values are overwritten when they have been "used up".

EXERCISE 3. Show the successive stack configurations in the evaluation of

(i) 3 2 × 4 − 5 ×

(ii) 7 4 3 + × 2 +

(iii) 9 6 + 2 × +

Quite right. The third example was a cheat; it is not a well-formed postfix expression. With these three operators we need four values, not three. However, a correctly designed processing system should be able to detect an error of this kind and issue a "stack empty" diagnostic message if there are not enough values for the next operator to work on.

FORTH OPERATORS

FORTH is a postfix language. A FORTH program is simply a (usually very complex) postfix expression; and a FORTH system uses a stack for storing data and intermediate results. Following the convention of most modern processors, FORTH makes its stack grow downwards from higher addresses to lower. Nevertheless, we shall continue to refer to the accessible end of the FORTH stack as its "top". In accordance with the ideas developed above, FORTH operators "use up" the values they operate on and replace them with the results they generate. The operators + and − have the same significance in FORTH as in ordinary algebraic notation, the operator − substracting the top stack item from the second, leaving the difference as the new top item and reducing the stack height by one.

As in most programming languages, the FORTH multiply operator is * and the division operator /. These are both integer operators. The operator / generates an integer result by truncation, i.e. by simply dropping the fractional part of the quotient if there is one. Thus

```
5 6 + 3 /
```

leaves 3 on the stack.

There is another division operator MOD which throws away the quotient and yields the remainder instead; so

```
5 6 + 3 MOD
```

leaves 2 on the stack. If we want both the integer quotient and the remainder, FORTH provides an operator /MOD which leaves the quotient as the top stack item with the remainder below it.

No ambiguity arises in division when both divisor and dividend are positive. When either is, or both are, negative, two results are possible. Thus dividing −12 by 5 could give the quotient as −2 with a remainder of −2, or the quotient as −3 with a remainder of +3. In FORTH-83 the second of these two conventions has been chosen. The rule is that the remainder takes the same sign as the divisor. The quotient, which is positive if divisor and dividend have the same sign, and negative otherwise, is truncated to the next lower integer, sometimes referred to as the "floor" of the number. Thus

10 divided by	7 gives	1 with remainder	3
−10	7	−2	4
10	−7	−2	−4
−10	−7	1	−3

The operators + − * / MOD and /MOD are known as *dyadic* operators because they operate on two values. (The word dyadic is preferred to binary for obvious reasons.) Another dyadic operator found in some versions of FORTH is ** , which performs the operation of exponentiation (raising to a power). Thus

```
4 3 **
```

yields 64.

We also need unary or *monadic* operators that operate on only one value. The monadic minus is an example. In ordinary algebraic notation, the same symbol − is used for both the dyadic and monadic operators. It is not possible to do this in postfix notation (why not?), so FORTH uses the symbol NEGATE to perform the monadic operation. Thus the sequence

```
4 NEGATE 5 *
```

leaves −20 on the stack, whereas

```
4 - 5 *
```

should give an error message, since the operator - is dyadic and needs to be supplied with two operands, not just a 4. Some implementations use MINUS instead of NEGATE.

There are also some ternary or *triadic* operators in FORTH. As an example, the operator */ multiplies the second and third stack values together and divides the result by the top item. Thus

```
6 5 4 */
```

yields 7, in contrast to

```
6 5 4 * /
```

which yields what? There is also an operator */MOD , which operates in the same way, except that it yields both quotient and remainder.

A word of caution, however. Most FORTH operators expect signed 16-bit quantities. These will normally be in twos-complement form, which means that the most significant bit is used to indicate the sign. Consequently only 15 bits are available to indicate magnitude. Ordinary operators therefore cannot handle signed numbers outside the range −32768 to 32767. Now the multiplication in */ and */MOD could generate a product outside this range even though the final result may be within it. In such cases some FORTH systems may produce the wrong answer. However if the system has been written in accordance with the FORTH-83 specification, or a later one, it will generate a double-length intermediate result and will produce the correct answer in the end. We shall deal with double-length arithmetic later on.

EXERCISE 4. Evaluate by hand

(i) `40 7 /MOD *`

(ii) `10 9 8 */MOD +`

(iii) `82 3 /MOD 3 * +`

To complete our review of the arithmetic operators that should be built into all FORTH systems, we shall first mention two dyadic operators MAX and MIN whose effect is to leave on the stack the larger (or smaller) of the top two items and drop the other. Thus

```
6 7 5 MAX MIN
```

yields 6. MAX and MIN can be used successively on a sequence of items to find the largest (or smallest); but they cannot always be used in the way just given to find the middle value. There is also a monadic operator ABS , which replaces the top value on the stack with its absolute value. Thus

```
−5 ABS
```

yields 5. In some systems there is an operator +− that negates the second stack item if the top item is negative, and drops the top item.

DISPLAY

Having calculated a value, we shall probably want to display it. We can do this with the "dot" operator . (full stop or point), which causes the top item on the stack to be sent to the standard output device. Most users of FORTH systems will be operating in an interactive environment through a video display unit (VDU) or typewriter console. The standard output device is then the VDU screen or the typewriter platen. The FORTH sequence

```
5 4 * 3 − .
```

should display 17 on the user's terminal in the next position on the device, i.e. the cursor position on a VDU or the head position on a typewriter.

The number output by . is signed single-length, and must therefore be within the range −32768 to 32767 . However there is nothing to stop us from keying in unsigned numbers up to 65535 : though most FORTH systems treat such numbers thereafter as if they were in signed twos-complement form. Numbers greater than 32767 therefore appear as their complements with respect to 65536, for instance typing

```
40000 .
```

should result in the output of −25536 . Notice that in twos-complement the negative range is one more than the positive range; consequently simple negation can put a number out of range. Try

```
−32768 NEGATE .
```

To assist us to tabulate results neatly, FORTH provides another output operator .R (dot-R), which outputs a number right-justified in a field of specified length. In this case the number to be output must actually be the second on the stack, the top item giving the length of the

field. Thus

```
567 8 .R
```

will output the number 567 displaced five spaces to the right of the previous cursor position.

FORTH-83 provides an operator .(referred to as "dot-paren" to display messages. This is one of a few operators that differ from the norm in that they take their operands not from the stack but from the program text. .(expects to be followed by a string, i.e. a sequence of characters that must be terminated with a) immediately after the last character to be displayed. For instance

```
.( GOOD MORNING TO EVERYONE HERE)
```

FORTH-83 makes no specification with regard to the maximum length of string. Most systems allow up to 127 characters of text, though some may allow more.

STACK MANIPULATION

The operators . and .R follow the general rule of FORTH and "use up" the numbers they operate on. Thus the stack is reduced in height by one by the operator . , and by two by the operator .R . If we wish to preserve the top stack item, then we must duplicate it using the operator DUP

```
5 7 * DUP .
```

displays 35 but leaves 35 still on top of the stack. Notice that ABS is the same as

```
DUP +-
```

There is also an operator ?DUP (in some systems −DUP), which duplicates the top stack item only if it is nonzero.

Remember that only the top stack item is directly accessible to a monadic operator, the top two to a dyadic operator, and so on. Consequently it often happens that we find items disposed upon the stack in the wrong order for the operations that we want to perform on them. FORTH provides several operators for stack manipulation, of which DUP and ?DUP are of course examples. Another is DROP which simply removes the top stack item. OVER duplicates the second item on to the top of the stack. SWAP exchanges the top two items; and

ROT rotates the top three items, bringing the third into the top position.

Assuming that the stack originally contained the values 1 2 3 with 3 at the top, we may summarise the effects of the five stack-manipulation operators as follows:

DUP	1 2 3 3
DROP	1 2
OVER	1 2 3 2
SWAP	1 3 2
ROT	2 3 1

in which the stack top is on the right. Thus

```
7 DUP 6 + SWAP 5 - *
```

evaluates to 26. Notice that, apart from DROP , the stack-manipulation operators do not remove values from the stack. Notice too that ROT is a ternary operator.

EXERCISE 5. The top two items on the stack are 567 in top position and 8 below it. Write a FORTH sequence that will output 567 at the right-hand end of an eight-character field but leave the stack as it was before.

EXERCISE 6. What are the full effects of the following sequences (output and final stack configurations)?

```
9 4 - DUP
```

```
7 8 9 ROT + DUP . *
```

Write a sequence of commands to invert the order of the top four stack items.

There are two stack operators of more generality than the foregoing. These are PICK and ROLL.

```
n PICK
```

copies the nth stack value (not counting n itself) and leaves it in place of n on top of the stack. Thus OVER is equivalent to 1 PICK , and DUP to 0 PICK.

```
n ROLL
```

moves the nth value to the top, pushing all the items above it down one place. Thus ROT is 2 ROLL and SWAP is 1 ROLL.

RADIX CONTROL

In our explanation to date we have assumed all input and output to be in decimal radix. This is the situation when the basic FORTH system is first loaded; but FORTH permits alternative number bases to ten. The word HEX causes input and output to be handled as if all numbers were in radix sixteen; and the word DECIMAL causes the radix to revert to decimal. Thus, just after loading FORTH, the sequence

```
22222 HEX . 23AB DECIMAL .
```

will output in turn the numbers 56CE and 9131 . Some systems provide OCTAL as a radix-change operator as well; and all FORTH systems provide a variable BASE that can be set to give any desired radix. However, we shall defer consideration of BASE until we deal in general with variables in the next chapter. Changes induced by HEX and OCTAL persist until the next radix change. Some systems provide H. and O. , which output a single number in hexadecimal or octal and then revert to the previous radix.

EXERCISE 7. What is the full effect of:

```
7 5 HEX * DUP . DECIMAL .
```

Numeric output is of course signed, whatever radix it is in. For instance

```
HEX AB12 . DECIMAL
```

will actually output −54EE . Negative hexadecimal numbers can be thought of as if they are stored as their complements with respect to 10000, which of course is the hexadecimal equivalent of 65536. In general, of course, we do not want hexadecimal information in signed form. Later we shall see how FORTH provides facilities for displaying individual bytes. Negative octal numbers are stored as their complements with respect to 200000.

Chapter 2

The FORTH Language

Before we discuss the language in more detail, it will be useful to look more closely at the operating system of FORTH. FORTH is always in one of two modes: either it is accepting input, or it is executing the last sequence input. Input is terminated by the newline key (RETURN or ENTER depending on the design of keyboard), and this causes the system to start execution. The operators in the sequence just entered are then executed in turn; and this may or may not involve output, or even call for more input. Provided that it does not get hung in a permanent loop, FORTH eventually outputs either OK , to indicate that it has completed the sequence of operations successfully, or a diagnostic message, to show that it has not. In the simplest case, input comes from the keyboard. We shall see later how sequences for execution can be taken alternatively from files.

VARIABLES

So far our operands have all been simple integer constants. But programs need variables; so we must have some way of declaring identifiers. This is provided by the defining word VARIABLE. The sequence

```
VARIABLE X
```

defines a variable called X by allocating sixteen bits of memory to it. Once we have assigned a value to it, we can use X in arithmetic expressions; but be careful. Simply quoting the identifier X has the effect of placing on the stack not its value but the address of the location containing it. It is necessary to do this of course because we may be quoting X not as the source of a numeric value but as a destination for storing a new value. In some older implementations VARIABLE automatically initialises the value to the top value on the stack. This is not true of the FORTH-83 standard.

Identifiers can be of any length up to a system-defined limit; and they may consist of any printable graphic characters. However, there are a few micro-based systems in which, with the idea of saving memory, only the length of the identifier and its first three characters are stored.

The design of FORTH reflects the design of the machines on which it is most commonly implemented. Consequently addresses refer to bytes even though most operands are two bytes long. In some implementations variables may be stored only at even addresses.

If we want the *value* of a variable in contrast to its address, we use the "contents" operator, which is @ . Thus the sequence

```
X @ 7 + .
```

adds seven to the current value of X and outputs the result.

To change the stored value of a variable, we use the "store" operator ! . This puts the second stack item into the location whose address is given as the top item. The sequence

```
X @ Y !
```

copies the value of X into Y .

As well as the variables that the user defines for his own private use, there are a number of variables that are provided by the system, some of which are public in multi-user systems. These are known as *user variables*. An example is BASE which contains the radix that is to be used for input or output of data. The sequence

```
25 7 BASE ! . DECIMAL
```

will change the value of BASE to 7 and then output 34, which is the base-7 representation of decimal 25. Notice the use here of DECIMAL to restore the original radix. In our exercises and examples we shall assume that the value of BASE is initially ten, unless the contrary is specified.

The letters from G onwards are available for use as digits when the radix is greater than sixteen. Thus

```
19 20 BASE ! . DECIMAL
```

will output J . Unfortunately J cannot be used alone as a digit on input, since, as we shall see, it has a quite different significance in FORTH; but it should be accepted if preceded by zero: i.e.

```
20 BASE ! 0J DECIMAL .
```

should output 19. Indeed it is good practice to use 0 to preface all numbers that do not start with a decimal digit, since otherwise the system may attempt to treat the number as an operator or character string.

Some implementations provide a word SET that can be used to

define a new word that will set a variable to a particular value. Thus if our system did not have a word OCTAL we could define it by

```
8 BASE SET OCTAL
```

VARIABLE stores the name of the new variable in a memory structure known as the *dictionary*, and assigns space close to the name to hold the variable value. This arrangement is perfectly satisfactory for private user-defined variables, and even for user variables in single-user systems. However, in multi-user systems the basic dictionary, which contains all the user variables, should be held in common. Unfortunately system variables such as BASE can have different values for different users, and cannot therefore be common. What most versions of FORTH do is establish a private user area in memory and store user variables there. The dictionary then holds not the value of the variable but an offset giving the position of the variable's value relative to the base of the user area, all individual user values being held in the same relative position in the user areas for the different users. Reference to the name of a user variable, like reference to that of a private variable, places the address on the stack; but in the former case the address is computed by adding the offset to the base of the user area.

Some systems provide a word USER that enables users to define new variables in the user area. USER takes the top stack item as the offset and stores this in the dictionary. Remember though that addresses in FORTH refer to bytes not 16-bit items. Thus the declaration

```
50 USER Y
```

establishes Y as an identifier for the twenty-sixth 16-bit item (0 is the first) in the user area. If you decide to use USER you will have to find first which locations have been pre-empted by the system.

If we wish to increase the stored value of a variable X, by seven say, we can write

```
X DUP @ 7 + SWAP !
```

However, this is rather a longwinded sequence for such an obviously useful operation; so most FORTH systems provide a single operator +! to add values into memory. Thus

```
7 X +!
```

is a much neater way to do the same thing.

Another useful shorthand operator is the query ? which outputs the content of the address at the top of the stack. In other words, ? is the

same as @ . which is just what one would expect it to mean. The sequence

```
VARIABLE Y 9 Y ! Y ?
```

should define Y , initialise it to 9, and then output its value. Notice that BASE ? will always output 10, since that is the value of any number in its own radix.

CONSTANTS

Programs need constants as well as variables, and FORTH provides a defining word CONSTANT . Thus

```
12 CONSTANT IN/FT
```

associates a memory location with the identifier IN/FT and places the value 12 in it. There is one important difference between variables and constants in FORTH: quoting a constant identifier places not the address but its stored value on the stack. Thus we might convert feet to inches using the sequence

```
FEET @ IN/FT * INCHES !
```

FEET holding the original value in feet and the new value being stored in INCHES.

We shall see later how to get hold of the address of a constant, and how to change it, should this be necessary. Of course there is nothing to stop us from writing

```
25 IN/FT !
```

but this does not do at all what we want, since it stores 25 not in the location allocated to IN/FT but in location 12. If 12 happens to be in read-only memory, then no harm is done; but, if not, then the system itself could be corrupted. Moral — do not use ! (or +! or for that matter any other of the "store" operators that we shall meet) unless you are quite sure you know what you are doing. Notice too that ? should be used with caution on constants.

```
IN/FT ?
```

prints not 12 but the content of location 12. And another caveat — some systems define as variables what others define as constants. Make sure you know which are which.

EXERCISE 8. Assuming that the declarations

```
46 USER Y 5 CONSTANT K
```

have been made, and that a variable X has been initialised to 9, what effects have the following if exccuted in turn as written?

```
8 BASE ! X @ . DECIMAL

7 Y ! K Y @ + X !

X DUP ? DUP @ K * OVER !
```

WORDS

We have been talking of operators; but the term used in FORTH for all functional components of the language, whether they be operators in the conventional sense or identifiers, is "word". Thus * − */ are words in FORTH, as are DUP OVER and VARIABLE . So too, once we have defined them, are the variables and constants X IN/FT and so on. FORTH words are strings of any printable characters, and they must be separated from one another by at least one space, since writing two operators contiguously implies that together they form a single word.

FORTH itself is a FORTH word. It has the effect of calling the basic vocabulary of FORTH, and is thus in effect executed on startup. In this book we describe FORTH-83, which has a standard required word set together with a controlled reference set of words and several extension sets. The controlled reference set comprises words that, while not being obligatory in the standard, must not be included unless they operate according to the definition given. The glossary in this book includes the whole of the required word set and most of the controlled reference set together with most of the extensions. Some implementations include a word FORTH-83, which will only execute provided that the implementation complies with the standard. A fundamental feature of FORTH is the facility to define new words. We have seen how this can be done using the defining words CONSTANT, USER and VARIABLE ; but we are able too to define new operators. This is done using the defining word : (colon). Thus the sequence

```
: SQUARE DUP * ;
```

defines a new word SQUARE to be equivalent to the two-word sequ-

ence DUP * . When we have defined SQUARE we can write for instance

```
3 SQUARE .
```

which has the same effect as writing

```
3 DUP * .
```

and prints 9.

The word SQUARE identifies what in other languages might be called a subroutine. The word : starts compiling the "subroutine"; and the word ; (semicolon) terminates the compilation. The result is to add SQUARE to the store of words in FORTH's dictionary, so that thereafter it can be used just like any other FORTH word. In contrast to most other languages, FORTH subroutines do not have arguments as such. All parameters are passed via the stack, and results are returned in the same way. Since quoting a variable name places its address and not its value on the stack, we should note that therefore variable parameters are called by reference, which means that they can be changed as a side-effect of the call.

The facility for defining new words makes FORTH indefinitely extensible. Most of the words provided in the basic vocabulary are built by "colon definition" from a relatively small number of primitives; but they can themselves be regarded as primitives for any special-purpose system that the user may wish to build on top of them. The obligation to proceed by successive definition forces the user to obey a discipline of program structuring more strictly than with other programming languages of a similar degree of complexity, such as early dialects of BASIC. This in turn makes FORTH programs relatively easy to understand and consequently to maintain.

EXERCISE 9. Define new words

(i) CUBE2 to form the cube of the second stack item without deleting it,

(ii) H. to output a single number in hexadecimal,

(iii) DUODECIMAL to change the radix.

DOUBLE LENGTH

We mentioned earlier that FORTH has a double-length or double-precision feature. This treats two adjacent stack items as one double-

length signed quantity with 32-bit precision. In FORTH-83 the item nearer the stack top is the high-order half; though you may find systems in which the opposite convention is followed. In a minimal FORTH system there should be at least three double-precision arithmetic operators D+ DNEGATE and D< . The first of these treats the top four staek items as two double-length numbers, and forms their sum. DNEGATE , which may alternatively appear as DMINUS , replaces a double-length number by its negativc. D< enables two double-length quantities to be compared.

The system may have other double-length operators as well. D− is used for subtraction, D>, D= and D0= are used for comparing quantities, and D. and D.R output the full double-length number in the radix given in BASE according to the conventions of . and .R . Many systems provide other double-length facilities such DABS , DMAX and DMIN. An unsigned double-length comparator DU< may also appear. Some double-length operators are prefaced by a 2 instead of a D . Examples are 2! , 2@ , 2CONSTANT , 2DROP , 2DUP , 2DROP , 2OVER , 2ROT , 2SWAP , 2VARIABLE etc.; but it should be possible to make up any double-length operations with the limited facilities provided in the reference set. There may also be an operator S−>D that converts from single to double length by extending the sign bit. Notice by the way that double-length duplication is simply OVER OVER . By judicious use of whatever features are available, it is possible to perform arithmetic to quite high precision without too great a sacrifice of computing speed.

To fix the structure of double-length numbers in the mind, thc following sequences

```
50000 0 50000 0 D+ D.
```

and

```
50000 0 50000 0 D+ . .
```

respectively output 100000 and 1 followed by −31072.

EXERCISE 10. With the aid of no other doublc-length arithmetic operator than the basic three, dcfine the operations D−, 2SWAP and D> .

The fig-FORTH implementation derives the double-length operators, and even some single-length operators, from a set of *unsigned* double-length primitives. This approach is possiblc of coursc only because the sign convention in use is twos-complement. Signed and unsigned addition and subtraction are identical operations in twos-complement, provided of course that the result remains within range, the only difference being in the way the quantities are interpreted, for instance

for output. There is an unsigned multiply UM* which forms the double-length product of two unsigned single-length numbers in the range 0 to 65535. The single-length twos-complement signed product is then formed simply by dropping the high order half of the result, provided of course that it is within range. There is also a divide operator UM/MOD that divides a double-length unsigned dividend by a single-length unsigned divisor to give single-length quotient and remainder. For technical reasons, this may only accept a 31-bit dividend. Unsigned double-length relations U< , U> and U= can also be found, as can unsigned output operators U. and U.R , which operate in the same way as . and .R .

From the unsigned primitives, we can derive not only the signed single-length operators but also signed mixed-mode operators. Thus M* multiplies two single-length signed numbers to give a signed double-length product; and M/ divides a signed double-length number by a signed single-length number to give a signed single-length quotient. Some systems also have a signed M/MOD .

CONDITIONAL BRANCHES

Programming languages need control structures. so that the course of a program can be changed during execution in accordance with conditions relating to the data being processed. In common with most programming languages, FORTH provides three types of control structure: an IF branch, a counting loop, and an indefinite repeat loop. The simplest IF structure is of the form

```
IF ..... THEN
```

or on some systems

```
IF ..... ENDIF
```

Notice that these structures operate according to the spirit of FORTH. The word IF tests the value of the top item on the stack. dropping it in accordance with the usual FORTH convention. If the condition is satisfied, i.e. if the tested value was nonzero. then the sequence between IF and THEN (or ENDIF) is executed: if it is not satisfied. i.e. if the value was zero, then control passes direct to the sequence following THEN (or ENDIF).

Following usual practice, FORTH has an alternative structure

```
IF ..... ELSE ..... THEN
```

or IF ..... ELSE ..... ENDIF

In this ease, the sequenee between IF and ELSE is exeeuted if the top value on the staek was nonzero; and the sequenee between ELSE and THEN (or ENDIF) is executed if it was zero. In either ease control passes afterwards to the sequenee following THEN (or ENDIF).

RELATIONS

Notiee that the operation of IF described here implies that FORTH treats any nonzero quantity as being equivalent to logical "true" and zero alone as equivalent to "false". FORTH also provides the comparison operators (relations) > < and = to compute truth values. In FORTH they are of eourse postfix operators, and they return 0 if the relation is false, and the standard all-ones eonfiguration (equivalent to −1) if it is true. The order of operands is the same as in infix notation. Thus

```
7 5 >
```

returns the same result as infix

```
7 > 5
```

whieh is true:

```
9 4 <
```

returns the same as infix

```
9 < 4
```

whieh is false. Following the normal eonvention of FORTH, the two items eompared are "used up", and are replaeed by the truth value. There is strietly no need for an inequality eomparator. If we want an inequality test we simply subtraet the two operands, whieh leaves zero (false) if they are equal, and nonzero (true) if they are unequal. Nevertheless some systems do provide a word <> to test for inequality and leave the standard values for "true" or "false".

FORTH provides three further eomparison operators, 0< 0> and 0= . The first of these replaees the top staek item by −1 (true) if it was negative (less than zero), and by 0 if it was positive. Thus

```
X @ 0<
```

is equivalent to

```
X @ 0 <
```

or, if you prefer,

```
0 X @ SWAP <
```

0> performs the inverse operation, except that of course both 0> and 0< return 0 (false) if the number was zero.

The word 0= replaces the top stack item by −1 if it was zero and by 0 if it was nonzero. It can therefore be used to reverse the truth value of the top stack item, that is, perform the NOT operation. Indeed in some systems the word NOT is actually used instead of 0= . As an example of the foregoing, consider the definition of MIN (MAX is trivially different).

```
: MIN OVER OVER > IF SWAP ENDIF DROP ;
```

EXERCISE 11. How would you perform the non-strict inequality operations >= and <= ?

EXERCISE 12. Define a word D= that will test the equality of two double-length numbers, assuming that it does not already exist. (You may use D− defined in exercise 9.)

LOGICAL OPERATORS

Relations can be combined together by using logical operators. FORTH provides at least AND OR and XOR . These perform bitwise operations upon the top two stack items, leaving respectively their intersection, union and exclusive union. Thus

28 14 AND yields 12

40 10 OR yields 42

40 10 XOR yields 34

(Write the numbers in full binary if you need convincing.) Some systems provide NOR and NAND logical operators as well. AND and OR work correctly if the top two items on the stack are being interpreted as truth values rather than bit sequences. For instance

nonzero zero AND yields zero

nonzero zero OR yields nonzero

Thus for instance one can write

```
X Y > Z Y = AND IF ....
```

The operator XOR may yield zero or nonzero, depending on the actual positions of the bits; but if we keep strictly to −1 as the only representation for "true", then this operator too will work for truth values. NOT (or 0=) converts any pattern of ones to a zero word and converts a zero word to −1. A word can be complemented (1s replaced by 0s and 0s by 1s) by forming the twos complement (NEGATE) and subtracting unity; though some systems do provide a complementation operator COM .

To assist with masking operations, a word MASK is sometimes provided. This replaces the top stack item with that number of ones, left-aligned if the number is positive, right-aligned otherwise. Thus

```
B @ 4 MASK AND
```

would select the first digit of a BCD number stored at B . A variant of this is the operator @BITS . The top stack item is the mask, and the second item is the address of the unmasked word. Thus

```
B 4 MASK @BITS
```

would have the same effect as the sequence just given. A word !BITS in some systems masks the third word on the stack by the top word, and stores it in the address given in the second. For example

```
B @ C 4 MASK !BITS
```

stores the first digit of B in C .

When handling bit strings, it is useful to have shift operators as well. SHIFT shifts the second stack item a number of places specified by the first — left if positive, right if negative. An arithmetic shift operator ASHIFT is sometimes available that takes account of the sign of the number that is being shifted.

EXERCISE 13. Define a word that will split a byte into two BCD digits, and store them in two consecutive (16-bit) locations.

INDEFINITE LOOPS

The simplest form of indefinite loop is

```
BEGIN ..... UNTIL
```

The sequence between BEGIN and UNTIL is executed repeatedly until the top stack item at the end of the sequence (i.e. just before UNTIL is

encountered) is nonzero (true). In some systems UNTIL is replaced by END . Notice that UNTIL (or END) "uses up" the truth value generated by the word preceding it, whether or not the condition is satisfied. If the condition is satisfied, control passes to the sequence following UNTIL . To take a simple example, the following definition generates a word that will output a number of Xs controlled by the number on top of the stack when it is called

```
: EXES BEGIN 1 - ." X" DUP 0= UNTIL ;
```

we might call the new word as

```
7 EXES
```

to give the output XXXXXXX (7 of them). Notice the use here of the word ." to print a single X. ." operates like .(except that it is compiled to do its work at execution time. It prints the string following it, which must be terminated by a double quote.

In place of the two words 1 and - , we might have used the FORTH word 1- . Most FORTH implementations provide 1+ , 2+ and 2- as well, to increment and decrement the top stack item. Some systems provide operators such as 1+! and 1-! to increment or decrement memory locations, 2* and 2/ to double or halve the top stack item, and so on.

To give a more complex example of an indefinite loop: to find the highest power of one number contained in a second we might define a word PWRIN as follows

```
: PWRIN OVER SWAP BEGIN ROT ROT OVER *
ROT OVER OVER > UNTIL DROP SWAP  ;
```

Thus the call

```
3 90 PWRIN
```

leaves the result 81 .

The BEGIN ... UNTIL structure requires the looping test to be made at the end of the cycle; consequently the loop is always executed at least once. It is sometimes preferable to make the test at or near the beginning so that the words within the loop need not be executed at all. This is in fact the case with our example PWRIN , since the definition just given produces the wrong result if the first number happens to be greater than the second. An alternative form of indefinite loop

```
BEGIN ..... WHILE ..... REPEAT
```

is provided therefore. WHILE tests (and drops) the top item on the stack. If this is zero (false) then control passes direct to the sequence following REPEAT ; otherwise the sequence following WHILE is executed, REPEAT unconditionally returning control to the start of the loop. Thus we may rewrite our definition of PWRIN correctly as

```
: PWRIN SWAP DUP BEGIN ROT OVER OVER <
WHILE ROT ROT OVER * REPEAT DROP SWAP / ;
```

To give a simpler example to illustrate the operation of this structure, the following word produces a sequence of Ys and Zs starting and ending with a Y .

```
: WYZE BEGIN 1+ ." Y" DUP WHILE ." Z" REPEAT
DROP ;
```

Thus the call

```
4 WYZE
```

produces the output YZYZYZY . This same structure appears in several guises in different sytems, such as

```
BEGIN ..... IF ..... AGAIN
```

```
WHILE ..... PERFORM ... PEND
```

As a final example, consider Euclid's algorithm for finding the highest common factor of two numbers. This is simply

```
: EUCLID BEGIN SWAP OVER MOD ?DUP 0= END ;
```

EXERCISE 14. Define a word that will halve the top stack item if it is even, and hence define a word that will find the largest (not necessarily prime) odd divisor of the top stack item.

COUNTING LOOPS

We also need a counting loop. In FORTH this takes the form

```
DO ..... LOOP
```

The top item of the stack on entry to the loop is the starting value of the index; and the item below this is the terminating value. The index is increased by unity every time the word LOOP is encountered; and control is returned to the DO unless the limit has been reached or exceeded. Thus we might redefine the word EXES defined above as

```
: EXES 0 DO ." X" LOOP ;
```

Here zero is the starting value for the loop index. So

```
7 EXES
```

gives the same output as before.

To allow the index to be increased by values other than unity, the word LOOP can be replaced by +LOOP , which causes the top item on the stack at that point to be added to the index before it is tested. +LOOP operates quite generally, and if we wish we may write the sequence so that the index is increased by a different amount each time round. We shall give an example of the use of +LOOP later.

It often happens that we want to terminate a counting loop on some condition other than completion of the count. For instance, suppose that we wish to skip up to twenty items from an input device, stopping either when we have counted to twenty or if the last character read was a newline. Assuming that we have already defined a word READ to place the next input character on the stack, we could define a word GET thus

```
: GET 0 DO READ 13 = IF LEAVE THEN LOOP ;
```

Since 13 (decimal) is the ASCII code for return, the call 20 GET will read up to 20 characters, terminating if a return character is read before the twentieth.

A word of warning, though. The word LEAVE operates in a rather unexpected way. All it does is alter the count so that it becomes at least as great as the terminator. The remainder of the loop is then executed in the normal way up to the word LOOP . The LOOP test now fails, so that control is not returned to the start. The loop is indeed "left", but not necessarily, it must be understood, at the point where LEAVE is situated.

The index of a DO loop is maintained by the system: but it is accessible to the user. The word I in most systems causes a copy of the current index to be pushed on to the stack. Thus the sequence

```
DO ..... I . LOOP
```

prints out the index on each cycle — a useful debugging technique. Notice that DO appears to "use up" the initial index and terminator. What it does in fact is transfer them temporarily to a second stack, called the *return stack,* where they are manipulated by LOOP . We shall have a lot more to say about the return stack later.

As a final example, consider the loop

```
DO I +LOOP
```

This doubles the index each time around so that, if the index stands initially, at 1, the number of cycles is the base-2 logarithm of the terminator.

The use of a second stack for holding index and terminator makes it possible to nest DO loops to an arbitrary (system-dependent) depth. The word I of course always refers to the index of the innermost loop. FORTH-83 also specifies a word J , which permits the user access to the index of the next enclosing loop, thus

```
DO ..... DO ... I ... J ... LOOP ... I ... LOOP
```

This facility is useful when handling two-dimensional arrays. Notice that the first I in the examples produces the index of the inner loop, while the second produces that of the enclosing loop, i.e. it gives the same value as the preceding J . Some systems provide a word K that yields the index of a second enclosing loop.

COMPILING WORDS

There is one very important point that must be made about all the foregoing control structures: the words DO IF BEGIN and the other words associated with them differ from most of the words we have discussed previously in that they are compiling words; that is, not only do they cause operations to be performed when they are executed, but they also cause modifications to be made to the program when they are compiled. This means that they cannot be used directly. We cannot for instance write

```
DUP 0< IF . ELSE DROP THEN newline
```

and expect that this will print the top stack item if it is negative.

Compiling words must be introduced with a "colon" or other code definition, for instance

```
: PRNEG DUP 0< IF . ELSE DROP THEN ;
```

is quite acceptable. The definition gives an opportunity for IF to do the compiling part of its job when PRNEG is compiled, and to do the condition-testing part later when PRNEG is called. The reason for this will be evident when we come to discuss the compilation process. In the meantime you will have to accept the rule and obey it.

THE RETURN STACK

The return stack is an essential component of FORTH's system for transfer of control between words in the dictionary: its use in controlling loops is incidental. It is accessible to the ordinary user through three special operators. These are >R R> and R@ (referred to as to-R, R-from and R-fetch). The first moves the top item of the main stack on to the return stack — moves, notice, in contrast to copies, which is what I does; the second moves the top item of the return stack back to the main stack; and the third copies the top item without affecting the return stack. In most systems, therefore, I is a synonym for R@; but this cannot be guaranteed, since there is nothing in the FORTH-83 specification to oblige the loop index and terminator to be stored in a particular order. Try a loop containing

```
I R@ = .
```

on your own system.

These three words are commonly employed when the return stack is used for temporary storage; but be very careful when you are doing this, otherwise a loop or other control structure may find its parameters changed in some unforeseen way. As an example of this application consider how mixed-mode signed multiplication M* can be defined from mixed-mode unsigned multiplication.

```
: M* OVER OVER XOR >R ABS SWAP ABS U* R>
D+- ;
```

In this definition we use the return stack as temporary store. We are interested only in the sign of what is stored there. This is negative if the signs of the two operands are different. It is to be used subsequently to adjust the sign of the product.

EXERCISE 15. Define words

(i) to output the largest of the top four items on the stack without deleting them,
(ii) to copy the fourth stack item to the top without using PICK,
(iii) to swap two double-length items.

DO loops can be nested within other control structures; but this must be proper nesting: they must not interlace. It would be wrong, for instance, to write

```
: WRONG DO READ IF LOOP THEN ;
```

As a final example on the use of DO loops, the word FIBO defined below outputs a sequence of Fibonacci numbers as specified by the top stack item. (The numbers in the Fibonacci series are characterised by the fact that each is the sum of the previous two, the first two numbers in the series both being 1.)

```
: FIBO 0 1 ROT 0 DO DUP 10 .R CR SWAP OVER +
LOOP ;
```

The word CR causes a newline on the output device; consequently the numbers in the sequence are printed in a neat column. In fact FIBO may give an overflow diagnostic: the twenty-third Fibonacci number is the largest within single-length range.

EXERCISE 16. Define a word that will calculate the factorial of n (i.e. $n \times (n-1) \times \ldots \times 2$).

BLOCK MOVES

FORTH has facilities for handling blocks of bytes. A block of memory can be filled with any desired byte value by the use of the word FILL . Here the character to be used is the top stack item, the number of bytes and the starting address being the second and third respectively. From this we can define a word ERASE that fills a number of bytes in memory with zeros, thus

```
: ERASE 0 FILL ;
```

The number is given as the top stack item; and the starting (lowest) address as the second. We can define a word BLANKS that works in a similar way, except that the block is filled with ASCII space characters (hex 20, decimal 32).

```
: BLANKS 32 FILL ;
```

There is also a block-move operator CMOVE that transfers a number of bytes from one place to another in memory. Again the byte count is the top stack item, the starting address of the destination is the second, and that of the source is the third; for example

```
FROM TO 100 CMOVE
```

If there is a need for a word-move operator for 16-bit items, it could be defined as

```
: WMOVE DUP + CMOVE ;
```

Notice that CMOVE operates from low address to high address; consequently, if the destination is above the source and the areas overlap, some information can be lost. This fact can actually be made use of in the definition of FILL .

```
: FILL SWAP >R OVER C! DUP 1+ R> 1 - CMOVE ;
```

Some systems provide an additional move CMOVE> that operates from high address to low. C! is defined below.

EXERCISE 17. Define a word MOVE that cannot overwrite even if the two areas do overlap. (Use CMOVE and CMOVE> .)
N.B. Some systems use MOVE to denote a 16-bit move operator.

BYTE OPERATORS

Practically all FORTH systems are implemented on byte-addressed machines. What this means is that the FORTH 16-bit numeric item is implemented as two consecutive bytes and stored in two consecutive addresses. FORTH-83 does not specify the byte order within a 16-bit field in memory, and implentors are likely to follow the natural conventions of the processors they are using. Thus Motorola microprocessors store the high-order byte in the lower of a pair of addresses: Intel and Zilog machines in contrast store the high-order byte in the higher address. The latter may look at first sight to be more logical; but the former is more convenient when handling strings of text.

FORTH provides facilities for handling individual bytes or characters. These are for the most part analogues of the integer-handling facilities. They operate on byte addresses, so their precise effects depend on the byte order in which data is stored. C@ (C-fetch) replaces an address at the top of the stack by the byte stored in that address. The byte is placed in the low-order half of the top stack item (which, as we have seen, is not necessarily the lower byte address of the pair), the remainder being made zero. Thus the sequence

```
HEX ABCD X ! X DUP C@ . 1+ C@ .
```

would output AB and CD but in an order dependent upon implementation details. It is worth while trying this sequence out on your own system to see what happens.

There is also a word C! (C-store) to store bytes. The address for storage is the top stack item; and the byte to be stored is the low-order half of the second item. For example

```
Y C@ X 1+ C!
```

would copy the byte of Y stored in the lower address into the higher byte address of X . Many systems have more byte-handling facilities than these — such as C? CVARIABLE and so on. Some systems provide a byte-swap operator >< and an operator >MOVE< that moves a block of 16-bit words performing a byte-swap operation on the way. This is useful when transferring data from, say, an Intel system to a Motorola.

It is worth noting that some systems require 16-bit items to be stored in an even-odd pair of memory locations. Thus the sequence

```
10 X C! 10 X 1+ !
```

would give an "odd address" diagnostic on such a system, as would a word move if given an odd address.

Here is a useful definition, which will give a hexadecimal dump of successive bytes. The number of bytes to be output is placed on top of the stack, with the starting address below it.

```
: DUMP HEX 0 DO DUP I + C@ . LOOP DECIMAL ;
```

EXERCISE 18. Rewrite DUMP to print the address as well as the contents, neatly tabulated.

Chapter 3

Transput and Files

Byte information is usually intended to be interpreted as a sequence of ASCII codes for printing. The ASCII code is reproduced in an appendix. The word EMIT (in some systems ECHO) outputs the low-order byte of the top stack item not as a number but as a graphic character. Thus

```
HEX 7A42 EMIT
```

outputs the letter B . EMIT may also increment a user variable OUT , which refers to the cursor position and can be used if desired for line-length formatting.

The inverse of EMIT is KEY . When KEY is executed, it puts the ASCII value of the next key pressed into the low order position of the top stack item, and clears the high-order half. Thus by executing

```
KEY
```

you can find the ASCII value of any key on your keyboard. Despite its name, KEY should have the same effect whatever the input device. Thus it should operate also when loading from file. Some systems have an additional word ASCII that skips blanks until it finds a non-blank character. There is a lot of divergence between systems in this area.

There are also one or two useful ASCII output words. CR has already been mentioned. It can be used in contexts in which the use of the newline key would have a different significance. SPACE outputs a space. SPACES outputs spaces to a number given as the top stack item. There may also be a word BELL that operates the beeper on the console if this is the current output device. Other specialised words may be provided to make use of whatever features the hardware offers. On computers with timers there might be a word MS to interpose a delay of a number of milliseconds as given by the top stack item.

STRING OUTPUT

The form of text output performed by the words .(and ." is suitable for constant strings — user communication, headings, diagnostic messages,

and the like. Variable strings that have been processed by program and are held in working storage are best output using the word TYPE which incidentally is called by .(and ." . The number of characters to be output by TYPE is the top stack item, and the starting address of the string the second item. However, strings of characters are usually "counted", i.e. held in memory together with their length count, which is the first byte. The word COUNT takes a string stored in this way, with the starting address (i.e. the address of the count byte) supplied as the top stack item; it stacks the count above the address, and then advances the address by unity. Parameters are then in the right form for the string to be output by TYPE .

A full definition of TYPE is a nice illustration of the use of several of the words we have considered so far. It is

```
: TYPE ?DUP IF OVER + SWAP DO I C@ EMIT LOOP
ELSE DROP THEN ;
```

The heart of this is the DO loop. The limits set when this loop is entered are actually the starting and finishing addresses of the string to be typed. Consequently I gives on each cycle the address of the next character to be output. The loop terminates when the last character has been emitted.

TYPE displays text starting at the current position of the cursor, and leaves the cursor position just after the last character typed. If you want newlines, then you must either put in a CR command or else include the ASCII return and linefeed codes in the string. TYPE outputs everything it is given, including trailing blanks if there are any. To remove trailing blanks FORTH provides a word −TRAILING , which adjusts the count to ignore them, the address and count having been placed on the stack as required by TYPE . A possible definition of −TRAILING is

```
: −TRAILING DUP 0 DO OVER OVER + 1− C@ BL −
                    IF LEAVE ELSE 1− THEN LOOP ;
```

The sequence OVER OVER + 1− generates the address of the last character in the string. If this is a blank, then the count is reduced by one, and the (new) last item is tested. This continues until the last item is non-blank.

STRING INPUT

Input of character strings is achieved using the word EXPECT and WORD . EXPECT reads from the terminal a number of characters as

specified by the top stack item, and stores them starting at the address given as the second item. EXPECT refers only to input from the terminal; input from file is handled differently. In case the actual number of characters is not known beforehand, input may be terminated with a keyboard "return", the count on the stack then serving only to set a maximum limit. The actual number of characters received is stored in a user variable SPAN .

EXPECT is used by the operating system to read commands into the terminal input buffer for interpretation. The start address of this buffer is held in standard systems in a user variable TIB , and the number of characters it contains in a user variable #TIB . The following FORTH program displays itself on the terminal:

```
TIB @ #TIB @ TYPE
```

EXERCISE 19. Define words that will read up to 20 characters from the keyboard, and output them again

(i) with all blanks suppressed,

(ii) in reverse order.

(You may assumne that a scratchpad containing at least 20 bytes is available, that its address is generated by the word PAD . and that there is a constant BL containing the ASCII blank.)

Before we discuss the operation of WORD it will be helpful to consider briefly how FORTH words and their definitions are stored. The dictionary is a simple linear list that grows from low addresses to high. The next address above the current top of the dictionary is stored in the dictionary pointer DP (in some systems H) which is usually available as a user variable. The content of DP, i.e. the address of the first free location, is referred to as HERE . Thus HERE is equivalent to

```
DP @ .
```

The dictionary structure is shown in Fig. 3.1. Notice that the convention used throughout this book is to show memory with the lowest addresses at the top of the page and the highest at the bottom. The top stack item can be stored direct into the dictionary at HERE by the operator , (comma), which also increases HERE by two. We shall see the function of , more clearly later when we discuss compiling. There is also a byte operator C, for storing individual bytes in the dictionary. The conventions are similar to those for C! .

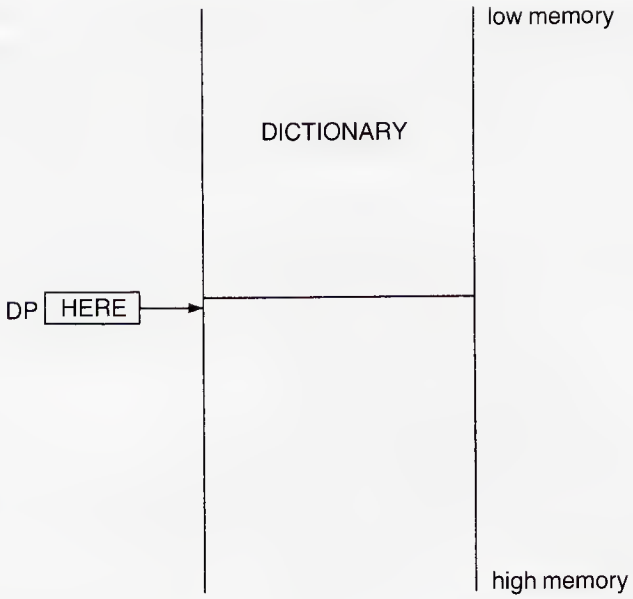


Fig. 3.1. The FORTH dictionary.

Another useful word is ALLOT which adds the top stack item to HERE so as to leave space in the dictionary, for instance to store an array. To leave space for an array of, say, ten integers (20 bytes) one would write

```
20 ALLOT
```

The definition of ALLOT is simply

```
: ALLOT DP +! ;
```

The principal use of the word WORD is to read names when constructing new dictionary entries. To assist in this, all the characters read by WORD are stored at the top of the dictionary and are preceded by the character count, which is consequently stored at HERE . WORD does not read direct from the input device, but from a buffer associated with it. The number of the buffer currently being used is held in a user variable BLK . The number zero is associated with the terminal buffer: numbers greater than zero are associated with file buffers. WORD starts reading at a point in the buffer defined by an offset contained in a user variable >IN (just IN in some systems); and it advances the content of >IN after reading a character. It reads until it encounters a delimiter.

The actual delimiter can be selected by the user, who puts its ASCII value in the top stack position before calling WORD. To simplify things, WORD is designed to ignore any leading occurrences of the delimiter, which will normally be a space. WORD counts characters as it reads them, and stores the count at HERE, i.e. just ahead of the character string.

Figure 3.2 shows the situation after WORD has transferred the letters "INPUT" from the buffer to the top of the dictionary. We show

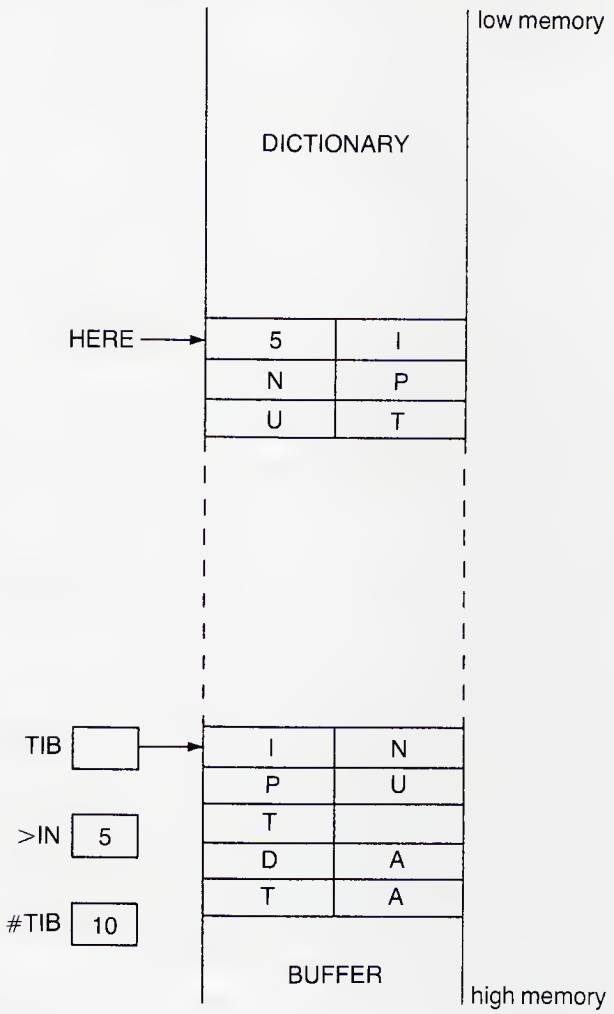


Fig. 3.2. Operation of WORD .

the memory as if it were two bytes wide, the better to illustrate the 16-bit structure of FORTH. The low-order byte is on the left, so that text appears in the normal European left-to-right sequence. As a further example of the use of WORD consider the definition of the word (, which introduces a comment

```
: ( 41 WORD ;
```

41 is the decimal ASCII value of ")" so WORD continues to read from the input device until it reches the closing delimiter of the comment. Since the string is merely a comment, there is nothing else to be done with it.

FILING

FORTH data and source-program text are stored in blocks on whatever backing device is supplied with the system. The first machines on which FORTH was implemented had a file block of 128 bytes, which was an inconveniently small unit for editing. It had the further disadvantage that, where tape was used, it led to inefficient utilisation. Consequently a larger data unit had to be used as well. This is 1024 bytes. Since 1024 bytes can conveniently be displayed simultaneously on a VDU, it is referred to as a *screen*. Because the storage block on early systems was much smaller than a screen, a constant B/SCR was provided to make the conversion easier. In newer implementations the size of a file sector is usually made the same as a screen, so the value of this constant is unity; the terms "block" and "screen" are then synonyms, and will be so used here.

Techniques for handling transfers to and from file naturally depend on what file-handling facilities the operating system provides. The majority of FORTH implementations are on small machines with random file devices (usually diskette) and a simple two-level memory. Microdrives may be considered to be slow diskettes; but other types of tape filing, such as cassettes, give rise to rather specialised schemes; and we shall hardly concern ourselves with them here. Machines with virtual memory, can handle files of the order of 100 screens as part of the program space. Very long files, or multiple files, are again handled in a system-dependent manner that likewise we cannot consider here.

FORTH incorporates a rudimentary virtual-memory scheme, which operates very efficiently in practice. In the standard system, the virtual memory consists of a set of blocks, 32 being regarded as a minimum, numbered from 0. A small number of buffers (usually three screens'

worth) is provided to receive data transferred from file. A constant B/BUF in some implementations stacks the number of bytes in a buffer, normaly 1024. The word BLOCK searches these buffers to determine whether or not the block whose number is on top of the stack is already there. If it is not, then a read command is issued to the backing device in order to retrieve it. It is then read into one of the buffers. In either case BLOCK returns the start address of the buffer on top of the stack. Special care must be taken when using cassette-based systems. Since stepping back is not normally possible, manual intervention is necessary to retrieve a block with a number earlier than that of the block last loaded.

Several algorithms are possible for deciding which buffer to overwrite next. A simple (and popular) one is to take the buffers strictly in turn, but avoid the one most recently used. The address of this is held in a variable PREV , while that of the next buffer to use is held in a variable USE . This will normally be the "stalest", i.e. the one that was least recently taken from file. If the original contents of this buffer have not been altered since they were read, then an exact copy still exists on file, and no further action is necessary. If, on the other hand, the buffer has been updated, then the new version must first be written back to file. In neither case does the user have to concern himself with the details. All "page turning" is performed by the system.

A block whose contents have been changed can be tagged by using the word UPDATE , which marks the most recently used buffer (i.e. the one whose address is in PREV) typically by putting a 1 in the most significant position of the first word. There is also a word EMPTY-BUFFERS , which removes the update marker from all buffers, preventing them from being written back and thereby erasing them from the system. The main use of this word is on initialisation. The word SAVE-BUFFERS (in some systems FLUSH) writes all updated buffers back to file store.

EDITING

All FORTH systems incorporate an editor, which is not necessarily part of the main system, and may have to be loaded separately in some system-dependent manner. Editors differ widely; but all of them take care of much of the detail involved in handling screens of text; for instance an editor will automatically mark as updated any screen that it writes to.

The word LIST retrieves from backing storage the screen whose

serial number is the top item on the stack, displays it on the current output device, and copies the serial number to a user variable SCR . If a printer is fitted, and the output is currently switched to it, the screen will be printed. LIST formats text in sixteen lines of 64 characters, with line numbers inserted on the left. Some editors have a word LINE that places on the stack the buffer location of the start of the nth line of the screen designated by SCR , n being the top stack item.

Some editors include a word CLEAR that makes reusable the buffer containing the block whose number is the top stack item. The old contents of the buffer are lost, though a (possibly earlier) version of its contents may still exist on file. Operating in a similar manner, there is often a word COPY that copies one buffer to another. The second stack item specifies the number of the screen to be copied, while the top item gives the screen number to be associated with the copy. CLEAR is used to create a new block when editing. CLEAR and COPY together can be used for renumbering screens.

Some systems have facilities for displaying a sequence of three screens, known as a triad. The display device in this case is normally a printer. The word TRIAD displays the triad containing the screen whose number is the top stack item, and will start with the lowest numbered of the three, whose serial number is consequently a multiple of three. Another useful facility sometimes found is a word INDEX that lists the top line of every screen, conventionally containing a title.

LOADING PROGRAMS

If a screen contains FORTH code then it is presumably intended to be compiled or executed at some stage. The word LOAD treats the screen whose serial number is on top of the stack as if it were being entered from the standard input device, and causes it to be compiled or executed in thc same way. The number of the last block loaded is held in a user variable BLK . The word LINELOAD on some systems enables loading to start with a particular line of a screen, the line number having been placed on the stack.

All buffers, including the terminal buffer, end with a null character (ASCII 00 hex.). This is actually a dictionary entry which is executed to terminate loading. However most FORTH programs run to more than one screen. To make it easier to load screens that are numbered consecutively, FORTH provides a "next-screen" operator −−> , which can be written at the end of one screen if the next in sequence is to be loaded after it. In case the screens to be loaded are not sequential, a

sequence of LOAD commands can be issued, or a LOAD command can be placed at the end of one screen to ensure loading of the next in logical sequence. Some older versions of FORTH had a word THRU to control sequential loading. Thus

```
20 25 THRU
```

loads blocks 20 to 25 inclusive.

To save having to remember the numbers of the screens where our different programs are stored, we can define a constant of the appropriate value. Thus, for instance, if we had written a system caled WORD-PROCESSOR starting on screen 96, then we could place a constant definition

```
96 CONSTANT WORD-PROCESSOR
```

in some standard screen of first loading. Subsequently then

```
WORD-PROCESSOR LOAD
```

would cause screen 96 and any subsequent "next" screens to be loaded. Other systems have a word LOADS such that

```
96 LOADS WORD-PROCESSOR
```

has a similar effect simply on a call of WORD-PROCESSOR . Some versions of LOADS will also cause a change of vocabulary (see below).

WRITING TO FILE

Remember that TYPE is designed to feed the standard output stream. By default this will be the console, though text can usually be copied or redirected to a printer. Most systems also permit redirection to a named file. For users who wish to program their own file transfers, there is a word BLOCK that obtains the block whose number is the top stack item, and then leaves in its place on the stack the first-word address of the buffer containing it. If the desired block is already in memory, this is a trivial operation: if not, the block is transferred from backing store, with a possible need for a rewrite.

New blocks are created using the word BUFFER . The number of the block to be created must be at the top of the stack. A buffer is allocated to it; and the previous contents of that buffer are written to file if necessary. The buffer address is left on the stack, so BUFFER operates rather like BLOCK except that there is no file search if the block is not already in memory. Strings can be moved to the new block with

CMOVE or CMOVE> . This way of working enables the file to be formatted as the user wishes, but leaves the responsibility of writing back to file to the FORTH system, provided of course that the block has been marked by using UPDATE .

FORMATTING OUTPUT

FORTH systems provide a text scratchpad for general use during input and for formatting output. This occupies space above the current top of the dictionary (HERE) and access to it is gained by using the word PAD , which is defined in FORTH-83 as

```
: PAD HERE 84 + ;
```

The scratchpad can extend upward until it meets the main stack, which grows into the same memory area (see Fig. 3.3). The "top" of the stack

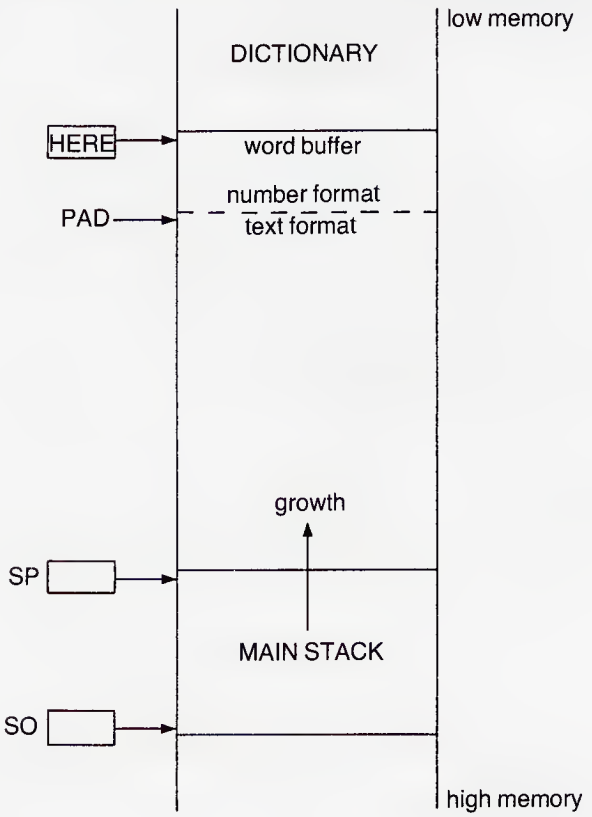


Fig. 3.3. PAD .

is held in a user variable SP ; and many systems provide a word SP@ (SP-fetch) that returns the address of the top of the stack. Consequently the amount of memory available for scratchpad use is output by

```
SP@ HERE - .
```

There may also be a word SP! for initialising the stack pointer on reset.

If SP is not available, the word DEPTH used in conjunction with the address of the stack base can be used to generate the stack pointer. The stack base is usually referred to as S0 . In FORTH-83 this is treated as a constant yielding the base address direct: in other implementations it is a user variable, so it is obtained by writing S0 @ . In many implementations S0 is the same as TIB , the start of the input buffer.

The addresses above PAD are used by certain elements of the FORTH system notably the editor. The addresses below it are used for numeric output. When outputting a numeric item, we need to carry out a conversion from binary into the radix given in BASE . This is done by successively dividing by the base; but this of course generates the digits in the reverse order to that in which they are output — hence the need to work downward from PAD , so that the digits are left in the correct sequence to be output using TYPE .

FORTH provides simple number-formatting facilities. These are initiated by the word <# (less-sharp) and terminated by the word #> (sharp-greater). The operator # (sharp) generates a single digit by dividing the number at the top of the stack by the base, converting the remainder to ASCII code, and storing it below PAD . The quotient remains on the stack. The word #S (sharp-S) converts whatever remains on the stack into ASCII characters dividing successively by the base until it can do so no longer. The word HOLD adds one ASCII character to the formatted string, and can therefore be used to insert other symbols such as a point.

In FORTH-83 the word SIGN places an ASCII minus on the top of the stack if the previous top quantity was negative, though on some systems it works slight differently. The following pictured output sequence

```
<# # # 46 HOLD #S 36 HOLD  #>  TYPE
```

outputs the top stack item with precisely two digits after the point, and a currency sign at the start; i.e. it outputs an integral number of pence (cents) in the normal form for pounds and pence (dollars and cents). FORTH-83 requires # and #S tc operate on double-length quantities; but in some implemtations they may be found to work only for 16-bit

numbers.

EXERCISE 20. Write a pictured output sequence to prepare a length in inches for output in the form of yards, feet and inches.

A typical mechanism for all this is to make the word <# copy PAD into a temporary variable HLD , which then acts as a pointer during the conversion. As each digit is converted, it is deposited in the address given in HLD and HLD is decremented by unity. Under this scheme, the definition of HOLD becomes

```
: HOLD -1 HLD +! HLD @ C! ;
```

which implies that HOLD may be used only after <# . The word #> restores initial conditions, except of course that the item output has been dropped from the stack, and leaves the appropriate address (i.e. the final value of HLD) and the character count on the top of the stack ready for the string to be output by TYPE .

NUMERIC INPUT

The word CONVERT converts numbers during input into binary using the radix given in BASE . CONVERT converts a numeric ASCII string, stored starting at the address given as the top stack item, into double-length binary, accumulating the result into the second stack item. Conversion proceeds until a character is encountered that cannot be a digit in the radix of the base. Thus input numeric quantities can be delimited by spaces, newlines, file terminators or digits outside the range of the base.

EXERCISE 21. Design a simple calculator that will accept unsigned decimal integers when keyed in, output them again in a neat column with the digits spaced in threes, and output their sum when an = sign is keyed in, all items input to be delimited by newlines.

EXERCISE 22. Modify exercise 21 to become a "cash register" handling quantities with two digits after the point.

Chapter 4

The FORTH Dictionary

To conserve memory space, many microprocessor versions of FORTH record only the first three characters of a word together with the character count. Thus the words CATASTROPHE and CATERPILLAR become equivalent, both appearing as 11CAT in the dictionary. Although a good FORTH system will warn you if you redefine an existing dictionary name, there is nothing to stop you from doing this; and indeed it is often most useful, since thereby one can obtain something of the convenience of local identifiers in block-structured languages. FORTH however goes further than this, and operates a system whereby the same word can have different interpretations in different contexts. This is achieved by offering the user a choice from several *vocabularies*. The main vocabulary is named FORTH ; but most systems provide additional vocabularies named EDITOR and ASSEMBLER respectively; and the user can define further vocabularies of his own.

When a word is compiled into a new definition, or when it is executed direct from input, it is sought within the dictionary, starting with the most recent entry in the vocabulary pointed to by a user variable CONTEXT . Thus for instance the word I , which yields the loop index in the FORTH vocabularly, would probably cause insertion of new text in the context of an EDITOR vocabulary, and might refer to an index register if the context happened to be ASSEMBLER . A list of the words in the context vocabulary can be obtained by typing VLIST ; on some systems the word CATALOG is used instead. This lists the words in reverse order of definition.

Simply typing the name of a vocabulary makes that the context; thus

```
EDITOR VLIST
```

makes EDITOR the new context vocabulary and then lists the words in this context. There are certain other words that can ehange the context automatically; for instance, on some systems the word LIST automatically puts the system into EDITOR context, and CODE or ;CODE puts it into ASSEMBLER context.

When a new word is defined, it is put into the vocabulary pointed to

by the variable CURRENT . This is not necessarily the same as the context vocabulary. In fact in most systems the current vocabulary is searched if the desired word cannot be found in the context. Normally however the current and context vocabularies are the same, in which case the two operations CONTEXT ? and CURRENT ? will output the same address. If CURRENT and CONTEXT do not indicate the same vocabulary, they can be made to do so with the word DEFINITIONS which sets CURRENT from CONTEXT . Thus

```
EDITOR DEFINITIONS
```

first sets the context to EDITOR and then sets CURRENT to be the same. If we wish to go back to the original context, we merely type FORTH , leaving EDITOR still the current vocabulary for new definitions. However, one effect of the defining colon may be to set CONTEXT to CURRENT, since the compiler refers to CONTEXT not CURRENT to find the parameters for constructing new words. Details in this area are not standardised.

Since CONTEXT is the first vocabulary to be searched when compiling or executing direct from input, it is best for this to be the one containing the commonest words. If a word cannot be found in the context vocabulary, most systems search the vocabulary within which this context was first defined, and so on recursively. For most vocabularies the "parent" is the FORTH vocabulary. If the word still cannot be found after a recursive search of both context and current vocabularies, an attempt is made to treat it as an integer in the radix given in BASE. Only if this is still not possible does the system issue an error message.

USER VOCABULARIES

We can define new vocabularies with the defining word VOCABULARY . This adds the new vocabulary name to the current vocabulary, and associates it with a pointer that will chain to words defined in the new vocabulary. In some systems it is also added to a chain headed by user variable VOC-LINK .

Once created, the new vocabulary can be made the context in the normal way by quoting it; and it can then be made current by using DEFINITIONS . For instance, we might write

```
VOCABULARY FRENCH FRENCH DEFINITIONS
```

and then go on to define new words such as

```
: MITTERAND ;
```

or even new vocabularies

```
VOCABULARY LANGUEDOC
```

which will be added to the FRENCH vocabulary.

Finally the word FORGET may be used to delete groups of definitions, or even complete vocabularies from the dictionary

```
FORGET MENOT
```

deletes from the dictionary the word MENOT and all words defined after it, regardless of what vocabulary they are in. This only happens, of course, provided that MENOT is present in an accessible vocabulary. To avoid the chaos that would ensue if the user happened by accident to attempt to FORGET part of the basic vocabulary, a variable FENCE is provided giving an address below which FORGET is not permitted to operate. However this precaution is not necessary if the basic FORTH system is in ROM. One can protect the whole dictionary defined to date by writing

```
LATEST FENCE !
```

The main purpose of FORGET is to release memory in situations in which space is at a premium. Many FORTH users make a dummy definition such as

```
: TASK ;
```

at the beginning of a session, so that a subsequent

```
FORGET TASK
```

will delete everything added that session. Some systems provide a word EMPTY to do the same thing. Others provide a word REMEMBER, which can be used to define a word that, when executed, will delete itself and all words defined after it.

EXERCISE 23. What are the effects of the following sequences of words executed in turn?

```
VOCABULARY FRENCH FRENCH DEFINITIONS
: DEGAULLE ; FORTH
VOCABULARY GERMAN : GAUSS ;
FORGET DEGAULLE
```

DICTIONARY STRUCTURE

In most systems every FORTH dictionary item comprises four fields as shown in Fig. 4.1. The first is the name field. On small systems this is exactly four bytes long, the first giving the full length of the name, and the remainder the first three characters. If the name has fewer than three characters, then the field is filled with spaces. On larger systems the full name is stored. The length byte also carries some tags whose functions we shall consider later. In Fig. 4.1, following the convention of the book, we show memory with the lowest address at the top. This

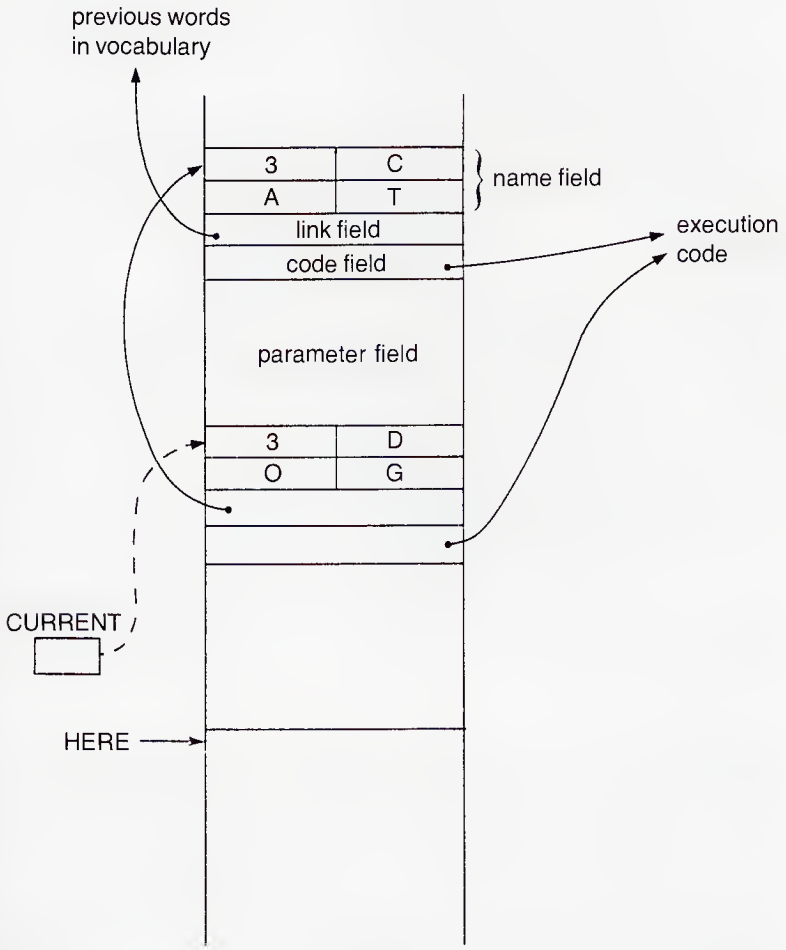


Fig. 4.1. Dictionary entry.

makes the name fields more readable. We also show the structure as being two bytes in width, since the remaining fields are made up of two-byte items. To this end we have chosen words with odd numbers of characters in our example.

The second field in every dictionary entry is known as the link field. It is two bytes long, and it carries a link to the beginning of the name field of the previous word defined in the current vocabulary. It is by travelling down a chain of such links that the system is able to look up words in the dictionary. The third field is the code field. It also contains an address, which points to the code that is to be invoked when the word is executed. All dictionary entries created by the same defining word have the same pointer in their code field; and it is the code-field address that is placed upon the stack when a word has been found in the dictionary.

The first three fields are sometimes known as the *head* of the dictionary entry. The fourth is the parameter field or *body* of the entry. Its length is variable; and it gives a full definition of the word. In most implementations (including that of fig-FORTH) the body is contiguous with the head, though there is nothing in the standard that makes this obligatory. The simplest bodies belong to constants and variables, and are only two bytes long. The body of an entry created by the word CONSTANT is simply the constant value itself; and the code field points to a routine to load this value on the stack. The body of an entry created by VARIABLE is similar; but the code in this case loads the parameter-field address, not the value, on the stack. The body of an entry created by USER , if the system provides this word, contains an offset giving the relative position of the variable in the user area. The code executed in this case adds this offset to the pointer to the start of the user area. Figure 4.2 shows these three cases, the dictionary entry being on the left, with the stack configuration obtained by executing the defined words on the right.

THREADED CODE

The parameter field of an entry created by a colon definition depends on the definition itself. It provides the threads needed in the threaded-code execution of FORTH. Threaded code can take one of several forms. The form adopted in FORTH is a sequence of compilation addresses. In the fig-FORTH model these are pointers to the code fields of the words in the sequence to be executed. Thus for instance when the definition

```
: MAX OVER OVER < IF SWAP THEN DROP ;
```

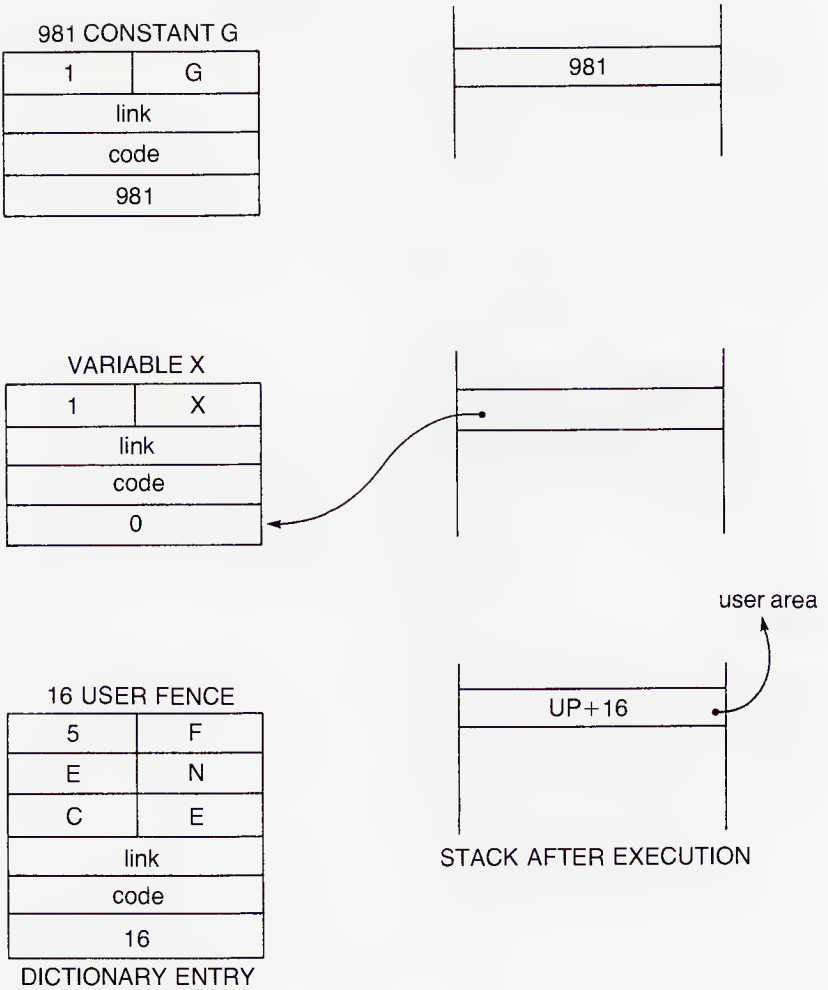


Fig. 4.2. Constant, variable and user entries.

is compiled, the parameter field is filled with pointers to the code fields of OVER < and so on in turn. Some of these words themselves have threads in their parameter fields; for instance < could be defined as

```
: < - 0< ;
```

Eventually a thread leads to a word defined wholly in machine code. This is executed, and control returns to the next thread in the same definition. Notice that the process is recursive: words are executed in

turn until a semicolon returns control to the next higher level. Figure 4.3 illustrates the example, and assumes that OVER and − are primitives defined in code, but that < is colon-defined.

This simple form of threaded code is possible because FORTH is a postfix language, and words are executed strictly in the sequence in which they are written. There is an exception to this rule when a word is used as data for a preceding word, as for instance strings output by ." or identifiers in VARIABLE definitions. Otherwise the sequence of control is strictly observed.

The need for sequential execution also explains why structures like

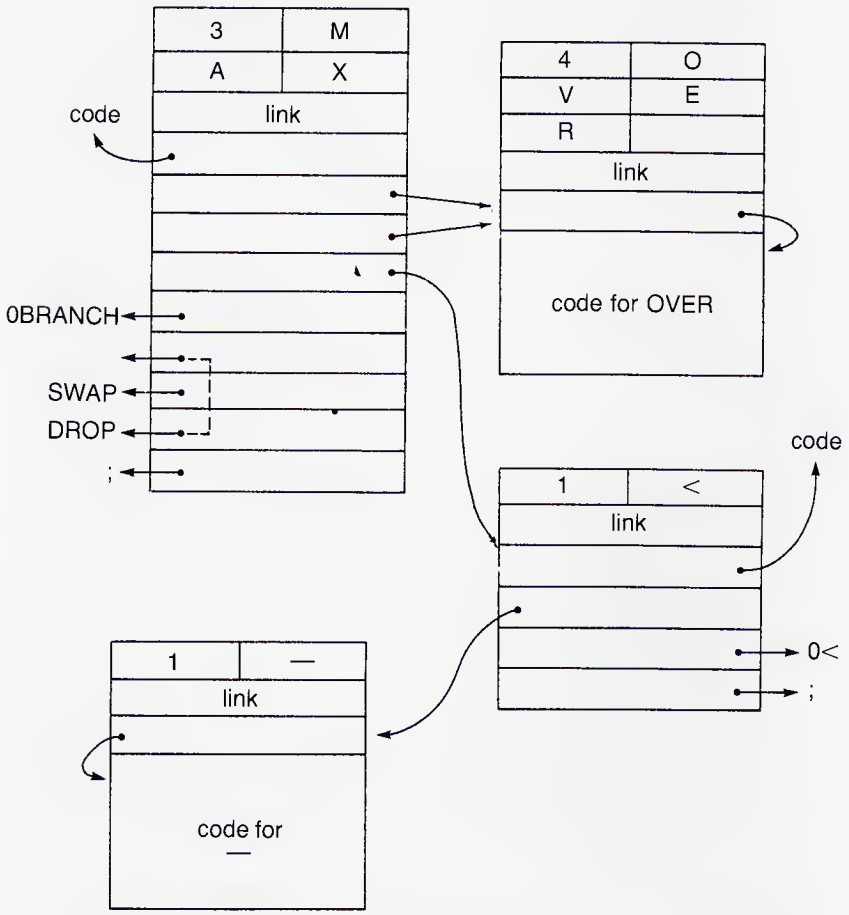


Fig. 4.3. Threading.

IF ... THEN may occur only in compiled definitions, and cannot be executed directly on input. When IF is executed, it may find the condition false, in which case it has to transfer control to THEN . It might have been possible to design a FORTH interpreter that scanned forward in this case until it found a THEN in the input stream; but this would have made it impossible to nest IF ... THEN sequences. What actually happens is that during compilation IF leaves on the stack an address pointing (typically) to an empty location in the parameter field of the word being compiled. Later during the same compilation, THEN stores there a pointer to its own location. We shall consider this process in more detail later on.

The word ' (single quote mark, or "tick") is very powerful despite its size. It searches the dictionary for the next word in the input stream, and puts its compilation address on top of the stack. It is therefore like ." and VARIABLE in requiring to be followed by its parameter. It is used principally in the word ['] , which retrieves thread pointers during compilation and places them in the parameter field of the word being defined. However, the word ' is available for general use, and can be used for instance to change the value of a constant in the dictionary, constants being stored as we have seen in the parameter fields of the relevant dicitionary entries. To do this we must have some way of converting from compilation address to parameter-field address. This is achieved by the word >BODY , which in the fig. model simply has to add 2. Thus if some lunatic government should decide to decimalise the foot, we could make the necessary adjustment by executing

```
10 ' IN/FT >BODY !
```

Notice that the tick is followed by its argument only during interpretation. To *compile* a word to change a constant value, either use LIT

```
: NEWIN/FT LIT IN/FT >BODY ! ;
```

or else delay quoting the name until you want to perform the change

```
: CHANGE ' >BODY ! ;
10 CHANGE IN/FT
```

Having found the compilation address of the desired word, we can convert it to the link-field address or to the starting address of the name field. Some systems make available words LFA and NFA to perform such conversions. There may also be a word PFA that converts the name-field address to the parameter-field address. Another convention uses >LINK and LINK> , >NAME and NAME> to convert be-

tween the compilation address and the addresses of the other fields.

The word TRAVERSE may also appear in the vocabulary. This can be used to move from one end to the other of a name field of arbitrary length, the starting address being given as the second stack item, and the sign of the top stack item specifying the direction of traverse. In the fig-FORTH implementation, TRAVERSE uses the convention that the top (128) bit of the first (length) byte and of the last byte in the name-field are ones. There is also a word LATEST that stacks the name-field address of the last word defined in the current vocabulary; though in some implementations it gives the last word defined — possibly in a vocabulary current earlier on. Another useful word sometimes found is ID. which prints the name of a dictionary entry, given its name-field address on the stack. Thus

```
LATEST PFA LFA @ ID.
```

will output the name of the last word but one. In some systems LATEST is contained in a user variable LAST . LAST should be the same as CURRENT @ except during actual compilation.

USER-DEFINED TYPES

We are now in a position to describe one of the most useful features of FORTH, one that makes it as powerful as many much larger languages. The basic FORTH vocabulary provides only a small number of different data types. There are variables, constants, and what might be regarded as functions, the last being created by using colon definitions. But FORTH also permits new types of object to be defined as well as new instances of existing types. This is achieved by using the words CREATE and DOES> . The principal function of CREATE is to create a new dictionary entry; but it is used in this special way as well. The syntax when it is used in conjunction with DOES> is

```
: type-name CREATE compile-time actions DOES>
execution-time actions ;
```

When the new type name is used in a definition of the form

```
type-name name-of-instance parameters ;
```

it creates a new instance of the type. It does this by using the word CREATE to build a new dictionary entry. CREATE puts the name-of-instance in the name field, enters an appropriate value into the link

field, enters a pointer to the code for constant definitions into the code field, and puts zero into the first location in the parameter field. In some systems <BUILDS is a synonym for CREATE .

DOES> modifies the code field to point to a special code segment designed to handle definitions of this kind, and replaces the zero in the first parameter location by a pointer to the sequence of execution-time actions given in the original type definition. Consequently, in the final execution, when the name-of-instance is invoked, control passes direct to those execution-time actions. These operate on the current stack contents, and any other structures built during the definition of the instance, to produce the desired result. CREATE and DOES> are executed when the new instance is compiled. Further compile-time actions can be specified by additional words between them. For instance, we might wish to allocate more space in the parameter field of the new object, or link the parameter field to a structure elsewhere in memory.

A simple example may help to make the description clearer. Suppose that we wish to store dates in the form

```
day month year
```

and to have the date displayed when called for. We can do this by defining a new word to define DATE as follows

```
: DATE CREATE , , , DOES>
DUP 2+ DUP 2+ ? ? ? CR ;
```

Having compiled the word DATE we can use it to define actual dates, as for instance

```
5 1 85 DATE TODAY
```

or `4 7 1776 DATE INDEPENDENCE`

CREATE in the definition of DATE creates the necessary dictionary entry with TODAY or INDEPENDENCE in its name field; and the three commas put the three numeric items previously stacked into the parameter field of this entry.

When TODAY is called, the code associated with DOES> stacks the first parameter-field address, and then invokes the part of DATE following DOES>. This then constructs the next two parameter-field addresses, stacking them above the first. The three ? operators then cause the three components of the stored date to be displayed; CR gives a newline, and the semicolon returns control in the normal way. DOES> has an exact or near synonym ;: in some implementations.

EXERCISE 24. Define a word TRACE such that the call

TRACE AA

will cause a copy of the top stack item when TRACE is called to be saved in the dictionary and displayed when AA is called subsequently.

EXERCISE 25. In Chapter 2 we introduced a word SET that defines a word that executes to store a value in a given address. Use CREATE ... DOES> to define SET .

THREADING MECHANISM

The threading mechanism is usually based on the use of an instruction pointer (IP) and a working pointer (W), which together determine the flow of control. Ideally these two will be implemented in machine registers, though neither will of course be the processor's own program counter. Suppose that we are just completing a code sequence associated with parameter (n−1) in the current definition. IP is then pointing to the parameter (n), which in turn is pointing to the code field of the next word to be obeyed, as in Fig. 4.4.

At the end of the code associated with parameter (n−1) we obey a standard code sequence, which we shall refer to as NEXT . This transfers the new parameter (n) to W, advances IP and W by one 16-bit address, and jumps to the appropriate code segment. The position is then as shown in Fig. 4.5, PC being the processor's program counter.

At this point W is pointing to the first parameter of the new word; so we have several possibilities, depending on the type of object that this word represents. If it is a constant, then the code associated with it will simply stack the parameter pointed to by W and obey NEXT again, which will advance to the next parameter at the original level. If it is a variable, then the process is similar, except that the address not the value is stacked. If the word is one that was defined in code as part of

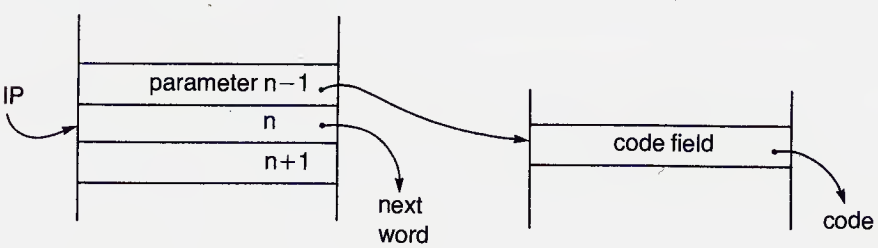


Fig. 4.4. Threading mechanism.

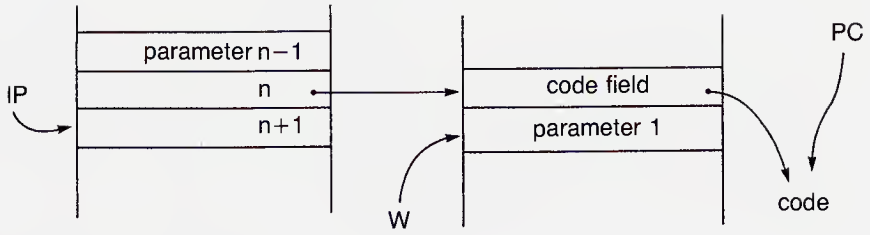


Fig. 4.5. Effect of NEXT .

the FORTH nucleus, then in most implementations its code field points to the first location in its parameter field, which in this case consists of the code sequence. The sequence ends with NEXT which returns control in the same way as for constants and variables.

If however the new word pointed to by parameter (n) was created by a : (colon) definition, then the process is slightly more complicated. It is here that the return stack comes into its own. The code associated with colon definitions saves the old value of IP on the return stack, and the value of W is copied to IP . The position is then as shown in Fig. 4.6.

The colon code ends as usual with NEXT which advances IP to the second parameter of the new word and enters the code sequence associated with the first parameter. The situation is then the same as it was originally (Fig. 4.4), except that the system is now operating at a lower threading level. Since the return address is saved on the return stack on every regression to a lower level, the process is recursive; regressions

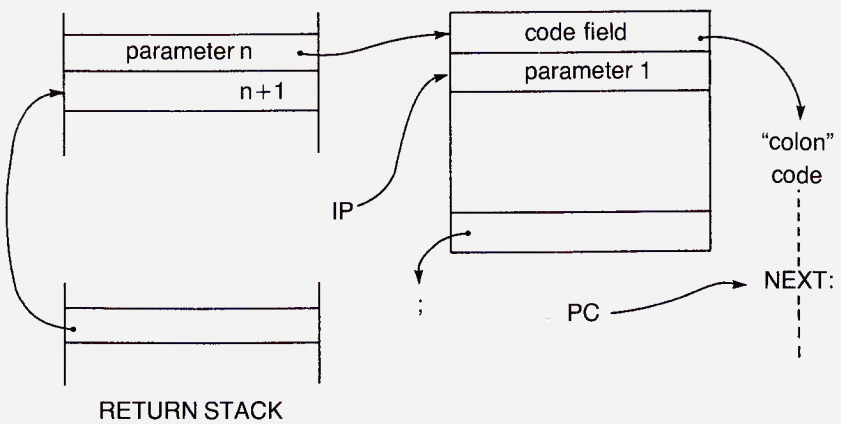


Fig. 4.6. Effect of : .

can be retraced when necessary. This is the function of the semicolon. The code associated with this simply restores IP from the return stack, and enters NEXT to get the code associated with the next parameter at the higher level. Notice that one consequence is that the return stack is available for communication between words only if they are at the same level. Any attempt to communicate between words at different levels using the return stack results in corruption of the return address.

Words containing CREATE and DOES> are also defining words,

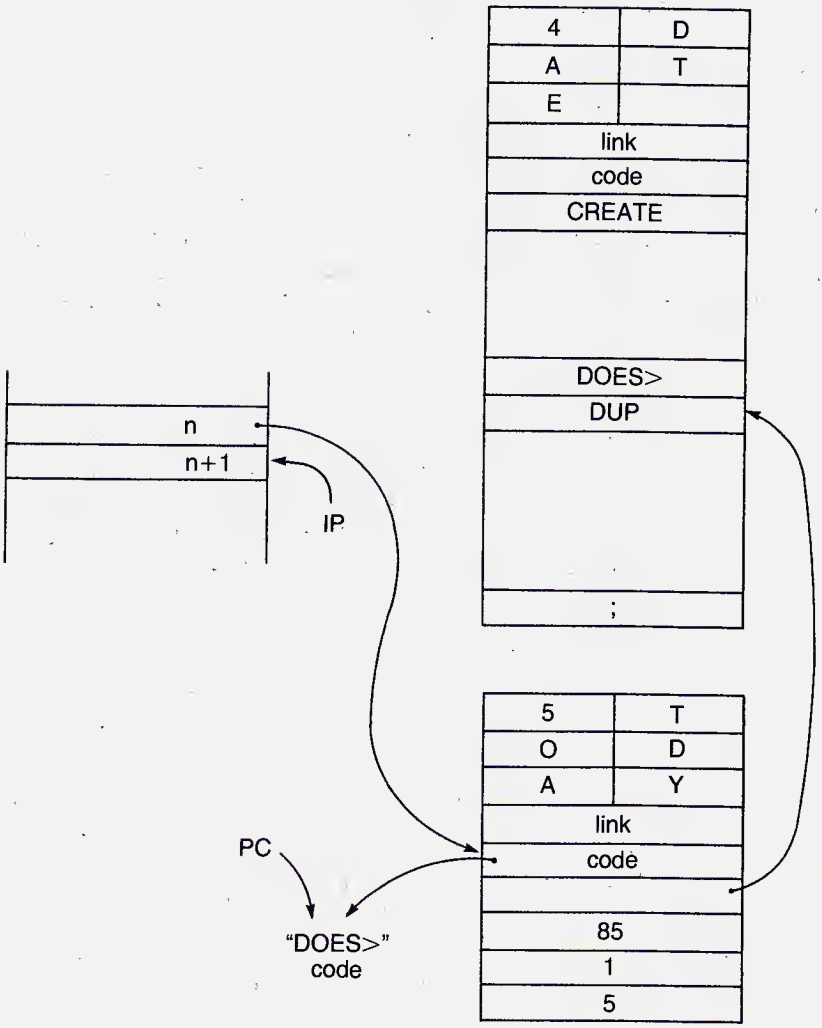


Fig. 4.7. A user-defined type.

and have their own sequence of operations at execution time. Take for instance the example we just gave of a defining word DATE that is used to define a new word TODAY . The situation just before TODAY is called as a parameter of another word, or as one of a sequence of words entered direct from the keyboard is shown in Fig. 4.7.

When TODAY was defined by DATE its first parameter was set to point to the first parameter in the DOES> section of DATE . The code field of TODAY points to a segment of code associated with the definition of the word DOES> . This first transfers the current IP to the return stack, then points IP to the first DOES> parameter of DATE , and finally puts the address of the second parameter of TODAY on the main stack. This parameter of course contains the year of today's date. The situation now is as shown in Fig. 4.8.

The DOES> code ends in the usual way with the sequence NEXT which transfers control to the first of the DOES> parameters of DATE and advances IP . The parameters are then executed one by one until eventually a semicolon returns control to the address stored on the return stack. This address refers of course not to a parameter of TODAY but to a parameter in the sequence that called TODAY .

VOCABULARY MECHANISM

There are a number of different mechanisms for handling vocabularies in existing FORTH systems. We shall describe here the mechanisms used in the fig-FORTH implementations, which is probably the one found most frequently. The operation of VOCABULARY may conveniently be described using the CREATE ... DOES> feature, thus

```
: VOCABULARY CREATE A081 ,
                    CURRENT @ 2- ,
                    HERE VOC-LINK @ ,
                      VOC-LINK ! DOES>
                      2+ CONTEXT ! ;
```

The first parameter, set by the first comma operator, is A081 (given in hex for clarity) which is equivalent to the name field for a word "space"*. The second parameter (second comma) will eventually form the head of the chain of link fields for all the words to be defined in this

* Since the convention used here is that the low byte precedes the high byte, 81 is the length byte, the 8 providing the bit that marks the start of the name field. The single-character name is "space", or 20 (hex), which becomes A0 when the end-of-field bit is added.

vocabulary. It is set initially to point to the starting point of the vocabulary within which the new vocabulary was defined (in this case FORTH), thereby ensuring that the new vocabulary grows as a branch of the old one. The DOES> part of the definition of VOCABULARY ensures that its address becomes the CONTEXT pointer when the new vocabulary is called.

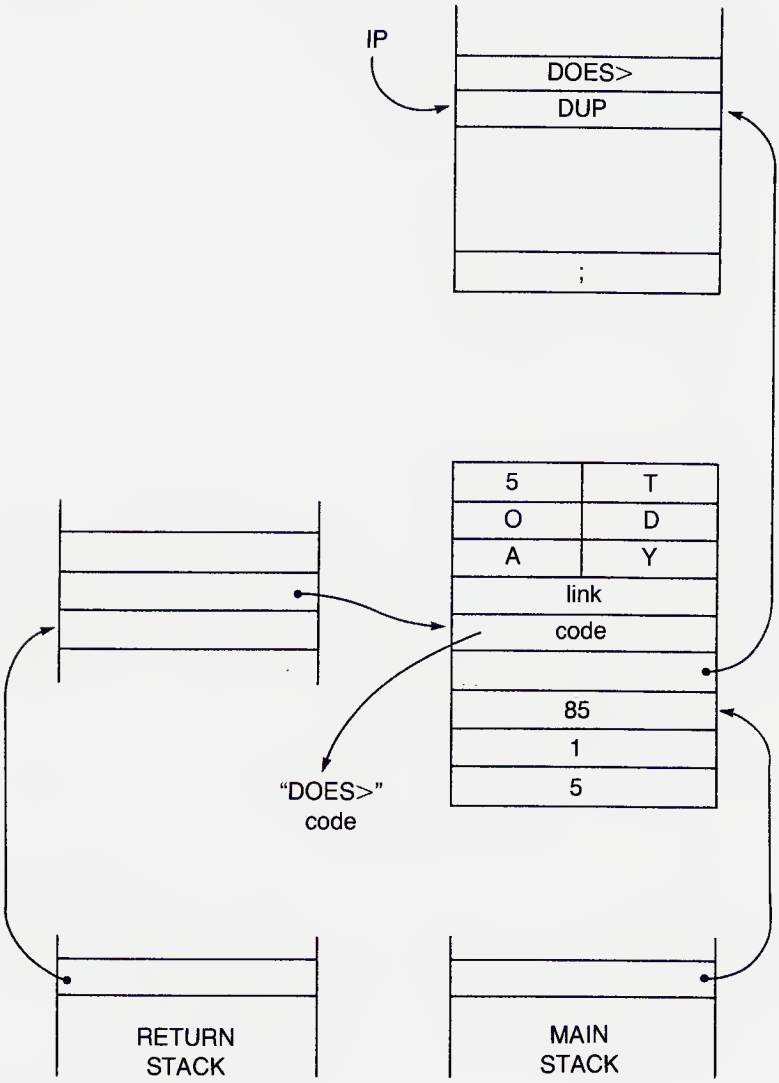


Fig. 4.8. Calling a user-defined type.

Finally, the third parameter (third comma and !) forms part of a chain that connects all the vocabularies together. Figure 4.9 shows the structure just after a new vocabulary LATIN has been defined within the context of FORTH (i.e. LATIN is the latest word defined in the FORTH vocabulary). Figure 4.10 shows the structure after LATIN has been made current, i.e. after LATIN DEFINITIONS has been executed, and AMO has been defined as the first word in this vocabulary. AMO is now at the head of the current chain; and, being so far the only word in the vocabulary, it is linked direct to the end, which

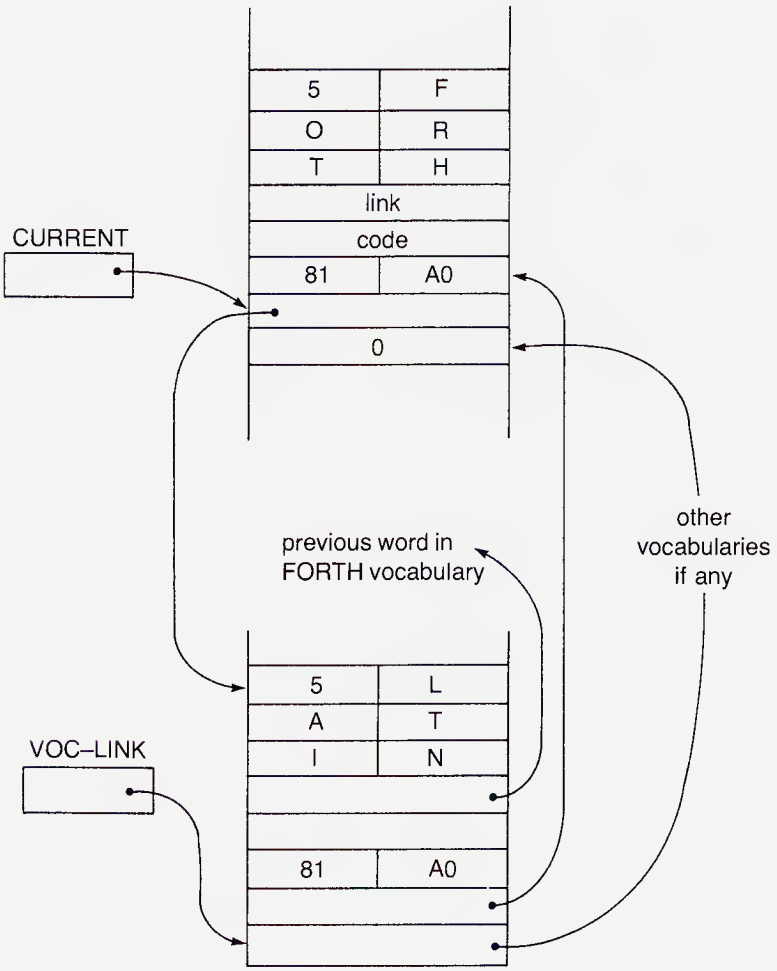


Fig. 4.9. A new vocabulary.

is the FORTH entry.

Now if LATIN is the context, any directory search will look first at the words in the LATIN vocabulary. If the word sought is not found in LATIN, then, on reaching the earliest word defined in this vocabulary (AMO in our example) the search will pass to the pseudo-word "space" in the parameter field of the entry for FORTH. This will actually look like an ordinary dictionary entry with a link field that is chained to the latest word that was defined in the FORTH vocabulary (which could have been LATIN itself). The search therefore continues (backward)

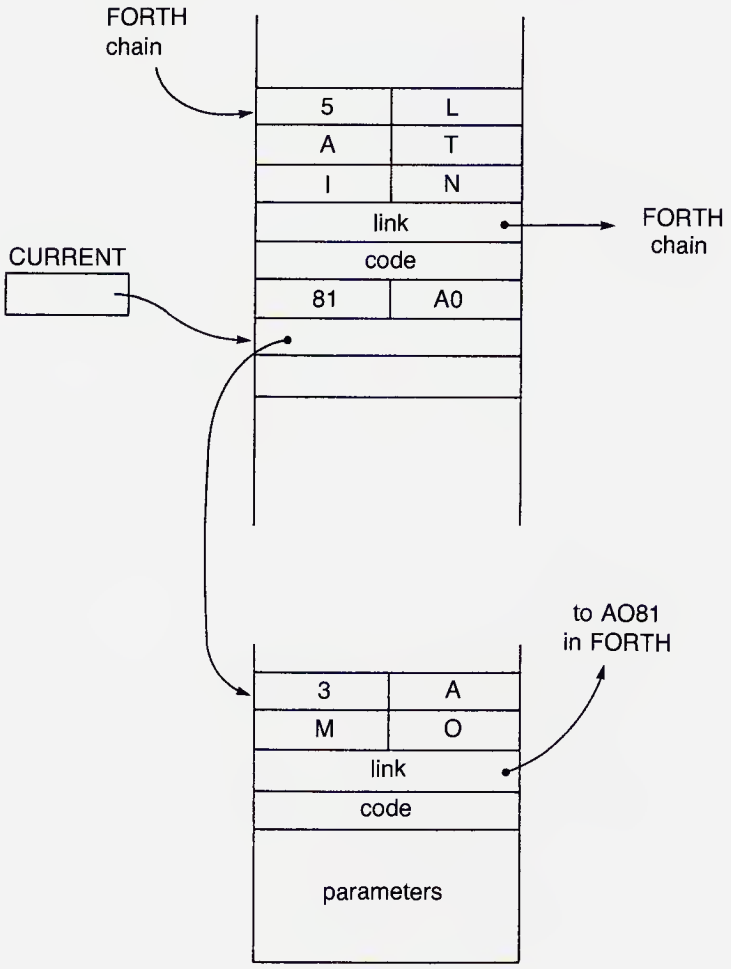


Fig. 4.10. Adding a word.

through the whole FORTH vocabulary. The first word of all to be defined in the FORTH vocabulary, and hence the last in the search, (LIT in fig-FORTH) has zero in its link field; and this serves to halt the search. Thus in fig-FORTH the vocabularies form a tree, with FORTH at the root, which is searched recursively. Since searching terminates as soon as a match is found, words can be multiply defined, the definition chosen in any given context being that nearest to the start of the search. The tree structure is not universal. For instance, some systems organise the dictionary not as a linked list but as a hashed set. This greatly speeds up searching. But searching, remember, occurs only during compilation or direct interpretation, so its effect on precompiled programs is likely to be small. Moreover, hashing makes it difficult to organise a flexible scheme for redefinition.

Chapter 5

Compiling and Executing FORTH

Two processes are involved in running FORTH programs — compilation and execution. Of course, compiling is just executing a defining word, but it is convenient to consider the two processes as separate. During "colon" compilation, words in the input stream are looked up in turn in the dictionary, and their addresses are threaded together by entering them into the parameter field of the new dictionary entry. Thus the time-consuming part of the job, the dictionary lookup, is carried out once only, at compilation time. Subsequently, when the word is executed, control is rapidly passed down the threads, with no need for further reference to the dictionary. It is this that makes FORTH, and for that matter any other threaded language, so fast in comparison with fully interpreted languages like BASIC, in which every symbol has to be looked up anew every time it is used.

A small nucleus of primitive FORTH operations is implemented directly in code; these are used to build up more complex functions by threading them together in different combinations. Thus the amount of raw machine code in a FORTH system is quite small. This is in contrast to a fully compiled language, in which a new copy of each primitive is compiled every time it appears in the program. Certainly some time is consumed in simply following the FORTH threads; but this is a small price to pay for a system that can pack quite a complex program into very little space.

COMPILING

Compiling then is simply executing a defining word, i.e. any word that puts a new word into the dictionary. The header for a dictionary entry is built by the word CREATE . This is a complex and system-dependent operation. In the fig. model it reads the length and characters of the new word into the next free area in the dictionary, using WORD , so generating the name field of the new entry directly on top of the dictionary. It issues a warning if a word is being redefined within the same context. An important function of CREATE is to "smudge" the first byte (the length byte) of the name field by setting the 32-bit. This

has the effect of making the word unrecognisable should it be referred to within its own definition. Were this not done, the compiler might enter a recursive loop from which it could not escape. CREATE chains the link field to the previous word defined in the current vocabulary, and plants a link to its name field in the parameter field of the current vocabulary. It sets the code field initially to zero. Later this will be made to point to the code for that particular type of entry. This is often held in the parameter field of the defining word that called CREATE . Finally HERE is updated to point to the first address of what will eventually form the new parameter field. The structure at this stage is shown in Fig. 5.1.

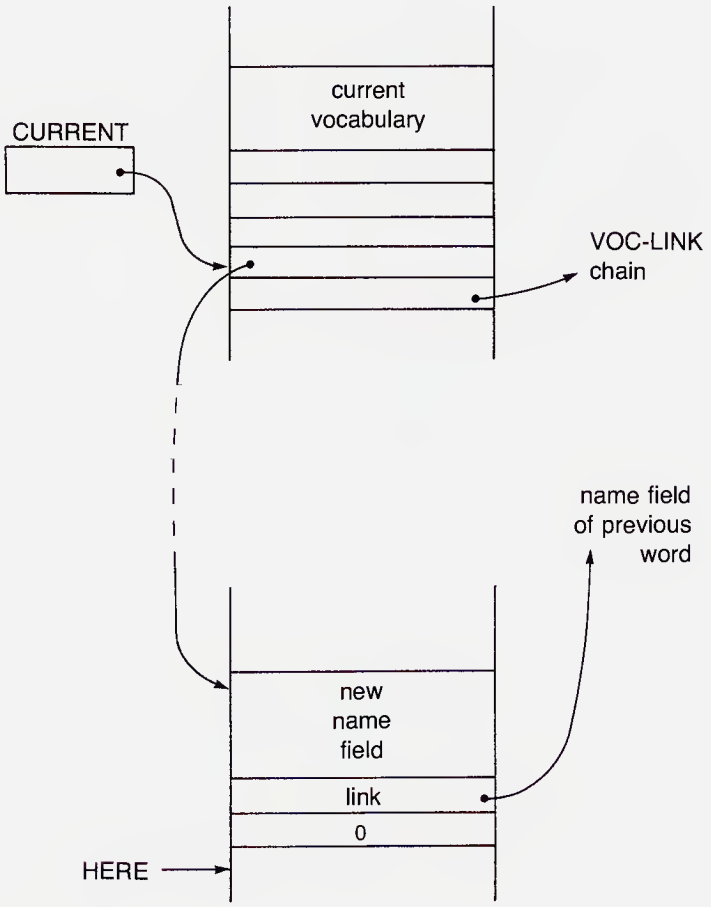


Fig. 5.1. Effect of CREATE .

The parameter field of a constant entry is simply the value of the constant, which at this stage is the top item on the stack. The definition of CONSTANT is thus

```
: CONSTANT CREATE SMUDGE , ;CODE code for
                                    constants
```

SMUDGE toggles the "smudge" bit, in this instance of course setting it back to zero, CREATE having earlier set it to one. The definition of VARIABLE is then simply

```
: VARIABLE 0 CONSTANT ;CODE code for variables
```

and that for USER is

```
: USER CONSTANT ;CODE code for user variables
```

with a code sequence to perform the required addition of offset and base. The word ;CODE operates just like ; except that it also links the code field of the new word to the code that follows it in the definition of the compiling word. We shall have more to say about this later.

COLON COMPILATION

Colon compilation is controlled by a user variable STATE . Every item in the dictionary carries a precedence code. If this is lower than the value of STATE then the item is compiled as a component of the current dictionary entry. If it is equal or higher, then the word is said to be IMMEDIATE and is executed. In early versions of FORTH the precedence code was held at the high end of the link field. But this limited address size, so modern implementations carry the precedence code as the 64 bit in the length byte of the name field. Since the top bit (128 bit) of this byte is a 1 , the variable STATE is usually set to either 0 or 192 (C0 hex) to enable a direct comparison to be made without the need to mask out the remaining bits.

On entering a colon definition, the system therefore alters STATE to 192. This value is set by the word] , defined as

```
: ] 192 STATE ! ;
```

STATE is eventually set back to 0 by the word [. The definitions of] and [thus make it possible to include a directly executed interlude within a compiled sequence simply by enclosing it within square brackets, e.g.

```
: JOE ." EXECUTING JOE" . . . [ .( COMPILING
                                JOE) ] . . . ;
```

We give the definition of the defining word : here as if it were itself a colon definition; but in fact it will be precompiled in any working system.

```
: : ?EXEC !CSP CURRENT @ CONTEXT ! CREATE ]
                          ;CODE colon code IMMEDIATE
```

Thus : first checks to make sure that the system is in execution mode (one cannot compile a thread to a colon sequence). It then sets the context to CURRENT to ensure that the components of the new definition are accessible to the vocabulary in which it is being defined; and calls CREATE to compile the appropriate header. Finally the right square bracket sets the system into compile mode. The word !CSP at the start stores the current stack pointer in a special vairable CSP so that its value can be checked at the end of the compilation.

IMMEDIATE WORDS

Certain words have to be executed, not compiled, whichever state the system is in, : being one. Other examples are (and .(. Vocabulary names too, which change the context, must normally be executed, not compiled. There is also a class of words that require some operations to be performed at compile time and others at execution time. Words like IF and DO are in this category, as is ; . All these words have the precedence bit set to 1 and are said to be immediate. Any user-defined word can be made immediate by putting the word IMMEDIATE after the definition. IMMEDIATE however is not immediate.

The word ; may be defined as

```
: ; ?CSP COMPILE EXIT SMUDGE [ ; IMMEDIATE
```

It threads only one word for subsequent execution, namely EXIT , which does the exiting at the end of the definition to which the semicolon refers. The word EXIT has to be compiled not executed. The use of COMPILE ensures this, even though the word ; is immediate. All the other words are executed at compile time. ?CSP checks the stack pointer that was saved in CSP at the beginning of the compilation, SMUDGE toggles the 32-bit back to zero in the length byte of the name field of the word being compiled, and [returns the system to the execute state.

When it is eventually obeyed during execution, EXIT restores the thread for the next level up, which has temporarily been held on the return stack. In some implementations EXIT has a synonym ;S : in others it is constrained to operate only during compilation, and makes use of ;S after calling ?COMP . ;S can be called at the top level. Its effect then is to terminate interpretation of the current line and call for the next line from the terminal. If the system is currently loading from file, then the loading of that block is prematurely terminated.

The word COMPILE must be used in the definition of all words that are partly executed at compile time, and in particular in words involving transfer of Control. Consider, for instance, the definition

```
: SHEEP 0 DO ." BAA " LOOP ;
```

Thus, for instance, executing 3 SHEEP would print BAA BAA BAA .

Now the definitions of DO and LOOP are typically

```
: DO COMPILE (DO) <MARK 3 ; IMMEDIATE
```

and

```
: LOOP 3 ?PAIRS COMPILE (LOOP)
                        <RESOLVE ; IMMEDIATE
```

During the compilation of SHEEP, DO , being immediate, starts to execute. However, the first thing it does is compile the word (DO) whose purpose is to transfer the terminator and initial index to the return stack when SHEEP is executed. Having threaded (DO) in the parameter field of SHEEP the system marks the current position in the parameter field of SHEEP by putting the address on the stack. It then places the number 3 on the stack. The parameter field of SHEEP then receives a pointer to ." , which is not immediate, and the string 3BAA.

LOOP again is immediate, so it places 3 on the stack and executes ?PAIRS to ensure that DO and LOOP have been properly paired up. ?PAIRS issues an error message if it does not find two identical values in the top two stack positions, in this case a pair of 3s. The word (LOOP) is then compiled. At execution time this will increment the index, compare it with the terminator, and jump back if the first is less than the second. Finally <RESOLVE takes the address previously planted on the stack by <MARK and computes an offset that will be used by (LOOP) at execution time to perform the jump back. <RESOLVE is called BACK in some systems. Figure 5.2 shows the eventual dictionary structure of the entry SHEEP.

Words containing IF and THEN are compiled in a similar way. The

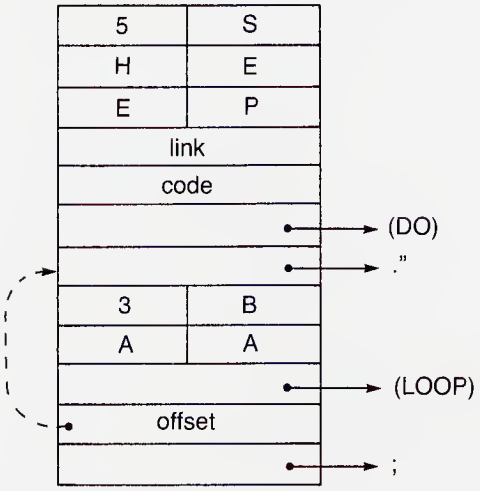


Fig. 5.2. Structure of sheep.

definition of IF is

```
: IF COMPILE ?BRANCH >MARK 2 ; IMMEDIATE
```

Thus, after compiling a thread to ?BRANCH , which will do the actual testing at execution time, the compiler executes >MARK , which puts the current dictionary address on the stack and stores zero at that address. Notice that >MARK is different from <MARK in that the former allots space in the dictionary.

The definition of THEN is

```
: THEN ?COMP 2 ?PAIRS >RESOLVE ; IMMEDIATE
```

which first checks that the system is compiling (?COMP) and that THEN has been properly paired with an IF . It then executes >RESOLVE , which stores an offset (difference between THEN and IF locations) in the parameter following ?BRANCH , so that ?BRANCH can perform the exit jump correctly. >RESOLVE may be defined

```
: >RESOLVE HERE OVER - SWAP ! ;
```

THEN does all its work at compile time. There is no execution-time activity, so no need for COMPILE .

EXERCISE 26. Devise a word BREAK that will cause immediate exit from a DO loop in contrast to LEAVE , which continues to execute the

loop. (You may have to devise an alternative form of LOOP to go with it).

The word COMPILE may be defined as

```
: COMPILE ?COMP R> DUP 2+ >R @ , ;
```

The word ?COMP calls an error diagnostic unless the system is currently compiling. The address on the return stack at this stage is pointing to the next parameter, which in turn points to the word to be compiled. It is incremented to point to the next word, and its original value is then used to obtain the address of that word's code field and store it in the dictionary, i.e. compile it. Figure 5.3 shows the execution of COMPILE when compiling (DO) within DO .

COMPILE is used to force a word to be compiled when the word containing it is immediate and therefore being executed. There is another word [COMPILE] that can be used to force compilation of an immediate word when the word containing it is being compiled. The difference is subtle but important. For one thing, [COMPILE] is immediate, while COMPILE need not be, since it is normally used within immediate words. A typical use of [COMPILE] is in the definition of aliases of immediate words; for example

```
: ENDIF [COMPILE] THEN ; IMMEDIATE
```

THEN is immediate; but it has to be compiled as part of ENDIF .

A better example is ELSE which illustrates both COMPILE and [COMPILE]. The definition is

```
: ELSE 2 ?PAIRS COMPILE BRANCH >MARK
                SWAP 2 [COMPILE] THEN 2 ; IMMEDIATE
```

Between checking that it is paired with an IF and setting a 2 to ensure pairing with a THEN it compiles an unconditional branch that will cause the "else" sequence to be skipped if the "if" condition has been satisfied. >MARK then allots space to hold the offset for BRANCH just as it does in IF . Finally, THEN is compiled as part of ELSE , and of course executed when ELSE is (eventually) executed. The SWAP is there to make sure that the >RESOLVE inside this THEN operates on the mark set by IF , the second mark being set for the benefit of the final THEN . Notice the extra 2 which is needed for the "pairs" check in THEN. COMPILE is used here because BRANCH is immediate, [COMPILE] because ELSE is immediate. The reader will probably understand the foregoing better if he considers in detail the compilation

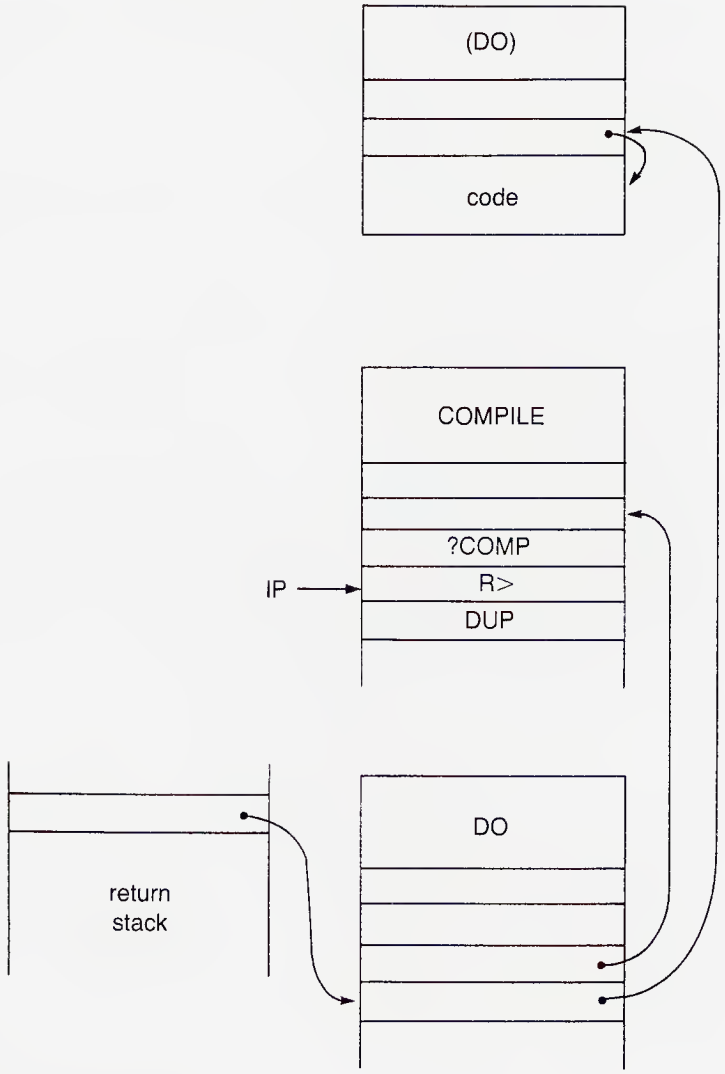


Fig. 5.3. Compiling (DO) .

of a simple word such as

```
: .TRUTH IF ." TRUE" ELSE ." FALSE " THEN CR ;
```

[COMPILE] may be defined

```
: [COMPILE] FIND 0= 0 ?ERROR DROP CFA , ;
                                        IMMEDIATE
```

Thus it simply searches for the next word in the input, registers an error if it cannot be found, and compiles a thread to it if it can.

CONDITIONAL COMPILATION

Some implementations provide facilities for conditional compilation similar to those offered by macro assemblers. They are obtained by using the sequence

```
IFTRUE .... OTHERWISE .... IFEND
```

For instance, if it were necessary to compile different sequences depending on the amount of memory available, we might include within a colon definition

```
[ S0 HERE - I000 > IFTRUE big memory sequence
                  OTHERWISE small memory sequence IFEND ]
```

Notice that IFTRUE and OTHERWISE can operate at the top level, in contrast to IF and ELSE . IFTRUE simply ignores text up to OTHERWISE if the condition is false, and OTHERWISE ignores text up to IFEND if it is true. One consequence of this is of course that IFTRUE sequences cannot be nested.

CODE

The bodies of the defining words are rather unusual, in that in most systems they contain the actual machine code that will be accessed from the code fields of the words that they define. Thus one effect of a defining word is to enter into the code field of the word being defined a pointer to part of its own parameter field. In the fig-FORTH implementation, the word ;CODE is used to do this at compile time. ;CODE also separates the words executed at compile time from the code section.

Any word can of course be defined entirely in machine code, provided of course that an assembler is available: in fact all the primitive kernel operations are. Some implementations permit the introduction of a code definition by the use of the word CODE . CODE first creates a new dictionary entry, as does any defining word, and places in its code field a simple pointer to the next memory location, i.e. the first location in its parameter field. Thus when the word is executed the code is entered through the normal mechanism. Every code sequence is terminated with a machine-code jump that returns control to the next piece of

code to be executed. This is compiled by the word END-CODE , which should terminate every code interlude; though in most implementations a newline is an alternative terminator. The difference between ;CODE and CODE is illustrated in Fig. 5.4.

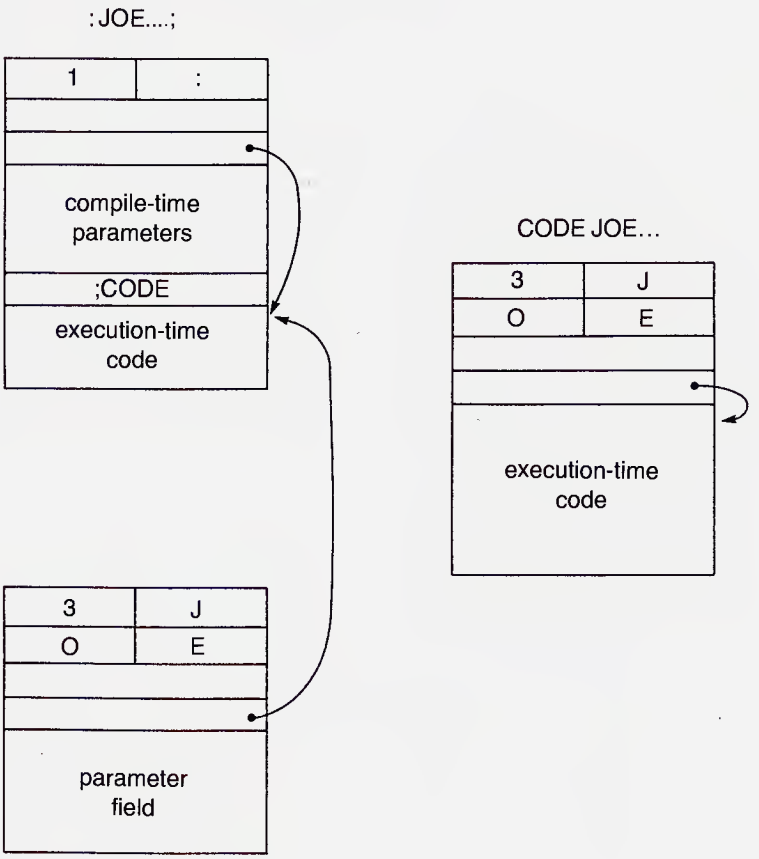


Fig. 5.4. ;CODE and CODE .

ASSEMBLER

Practically all versions of FORTH provide an assembler, usually as a separate vocabulary. The principle is quite simple: all the symbols in the assembly language are defined as immediate FORTH words that store the appropriate codes in the dictionary. The complexity of the assembler depends of course on the complexity of the assembly code. For instance, if the value to be given to an assembly mnemonic depends

upon the parameters associated with it, then its definition in FORTH is dependent in a similar manner and is correspondingly complex. To make it easier to resolve such dependences, FORTH assemblers usually require parameters to precede function codes in the normal postfix manner; and this can be disconcerting to someone accustomed to conventional assemblers. Address expressions too must naturally be in postfix form.

Apart from these, there may well be other differences in detail between a FORTH assembler and a conventional assembler for the same code. The assembler can be greatly simplified if different mnemonics are used for different addressing modes; and advantage is often taken of this fact to reduce the size of the assembler. Thus the user may have rather more work to do when writing assembly-language sequences for FORTH than when writing them for a conventional assembler. Against this, the FORTH assembler may be able to make use of some high-level constructs of FORTH itself, such as IF .... THEN. One problem does arise — that of setting labels. Most FORTH assemblers include a word LABEL that sets the following name as a constant, with value equal to the current program location. Forward jumps, to labels that have not yet been set, do constitute a difficulty that can normally be resolved only by avoiding the issue and using some construct such as IF .... THEN .

RECURSION

FORTH is not a recursive language, although the use of a stack during compilation provides the potential for recursion. Auto-recursion, i.e. the power of a word to call itself, is expressly prevented by the "smudging" process, which deliberately corrupts the name field of a word in the dictionary until after it has been defined. This prevents the system from entering an infinite loop, which might occur during compilation if the system tried to call a word for immediate execution while trying to compile it, or during execution if a word tried to jump back to itself. Mutual recursion, one word calling another word that in turn, either directly or indirectly, calls the first, cannot be performed in a simple manner in FORTH, since the definition of a word can only include words that have already been entered into the dictionary.

There are, nevertheless, ways of getting round limitations of the language. For instance, the following word can be incorporated into any definition, and will stack the code-field address of the word being defined.

```
: THISCODE CURRENT @ @ PFA CFA ; IMMEDIATE
```

The sequence of words

```
THISCODE LITERAL EXECUTE
```

will then compile an auto-recursive call. For example, the following definition provides a truly recursive computation of a factorial.

```
: FAC DUP 2 = IF ELSE DUP 1 - THISCODE LITERAL
                                    EXECUTE * THEN ;
```

Some systems provide the equivalent of this triplet in the form of the word RECURSE .

IMPLEMENTATION

Any FORTH implementation will consist of several layers. At the bottom comes the nucleus, consisting of those words that are to be regarded as system primitives. Indeed the nucleus could be regarded as comprising two layers — those words that are implemented wholly in code and those that are defined in terms of other words but are precompiled, since they are needed for the construction of the compiler.

A third layer is concerned with input, output and filing. The interpreter provides a fourth layer. Once the words in this layer have been added (in precompiled form) the system is capable of executing sequences of words from an input device. However, it is not until the fifth layer, the compiler layer, is complete that new words can be added to the dictionary. Above the compiler layer the user can add layers of his own to cover his own application.

The layer structure described here corresponds roughly to the way in which a system is built up for a new system; though there may be exceptions. For instance, some of the device layer can be left until after the compiler is working, since only one input and one output device is necessary initially. Inevitably, though, a policy of graceful implementation conflicts with brevity and execution speed. Particularly in the compiler layer, there will be forward references, which have to be entered initially as zero, and reassembled when the layer is complete.

The actual method of implementation will depend heavily upon the facilities that the assembler provides. For instance, a powerful macro-generator can be invaluable in creating and linking the headings of the dictionary entries. The FORTH Interest Group offers an implementation manual. The kernel of this is written in 6502 code; but the remain-

der is in FORTH. It is particularly valuable in suggesting a logical order in which words can be defined.

MEMORY MAP

Figure 5.5 shows the fig-implementation memory map. Most single-user FORTH implementations will be similar to this, unless the processor has special characteristics, and employs certain pages for special purposes. In addition to its private memory environment, FORTH makes use of whatever monitor or operating-sytem routines are provided elsewhere in memory. The low-address end of the FORTH memory area typically holds the "boot-up" literals, i.e. the default values that are loaded into the major variables on initialisation. Above these comes the dictionary, in more or less layered order. There are two sets of these: the nucleus of primitives defined in code, and the precompiled colon and other definitions. The top of the dictionary is marked by the dictionary pointer DP , which may or may not be held in a user variable.

The main stack grows into the same area of memory as the dictionary. Whenever a new word is defined, the system checks that adequate space is available for it, for the stack, and for any data that is being saved relative to PAD . 64 bytes are regarded as sufficient for the main stack; and 2000 bytes at least should be available to the user for his own additions to the dictionary. The base of the stack is marked by S0 and its top by SP . S0 is often directly available to the user as a constant; but SP is variable, so may not be in view of the damage that can easily be caused by manipulating it unwisely. The difference, S0−SP , is stacked by the word DEPTH .

The addresses just beyond the base of the main stack are occupied by the terminal input buffer, which normally has a capacity of 80 characters. Beyond this again is the return stack, growing from high to low address in the normal way. Its size should not be less than 48 bytes. The base of the return stack (pointer R0) usually coincides with the base of the user area (pointer UP). The size of the user area is fixed at FORTH load time; and the user may have some trouble to change it. The user area is bounded by the area allocated to the file buffers (typically three) each of which occupies 1024 bytes. The variable LIMIT points to the first address above the FORTH memory area. Some of the pointers shown in Fig. 5.5 will be available in FORTH user variables; others will exist only as assembly-time identifiers, and thus be inaccessible to the user of FORTH . However they can usually be redefined, with a little ingenuity. For instance, if we wanted a pointer to the start of the

user area, we could write

```
0 USER UP UP DUP !
```

to define it and its contents.

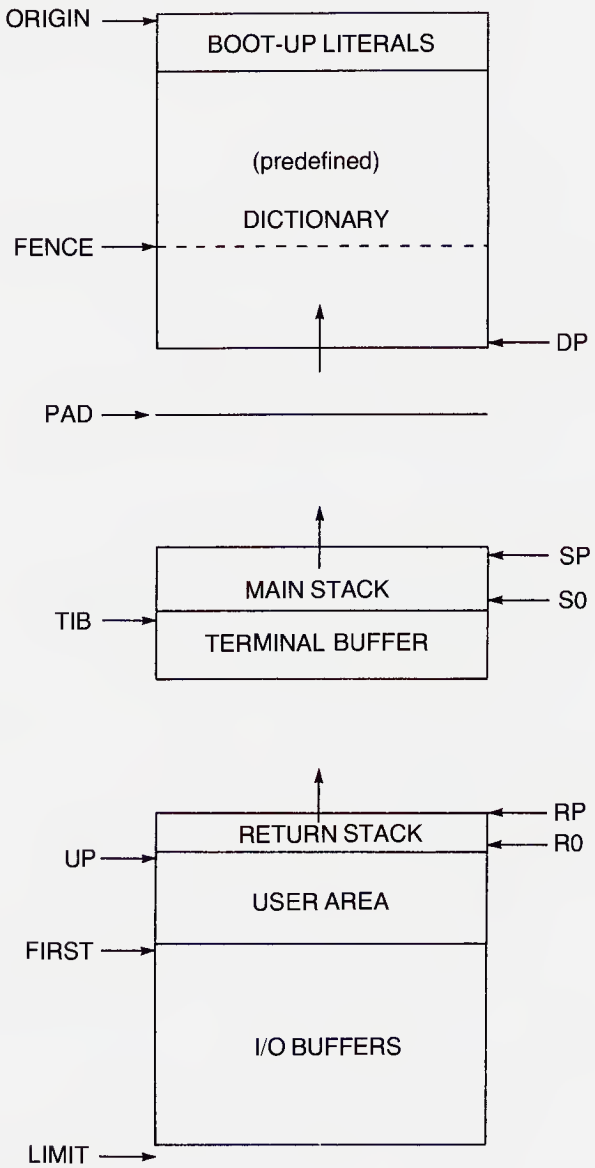


Fig. 5.5. Memory map.

There are a few multi-user versions of FORTH. The principal difference between these and the single-user versions is that the predefined part of the dictionary is shared between all users to save memory space. It must therefore consist of pure procedural code, and not be write-accessible. The boot-up literals are the same for all users; but some of them are relative and must be relocated in the individual user memory areas.

THE SOURCE INTERPRETER

The converse of COMPILE is EXECUTE , which is one of the kernel words, and has the effect of passing control to the word whose code-field address is on the top of the stack. It is called repeatedly by the source interpreter, which comes into operation on start-up, and controls the execution of sequences either typed in direct or loaded from file. Thus in immediate-execution mode the system is simply searching for words in the dictionary, stacking their code-field addresses, and then executing them.

The precise details of startup for FORTH are, of course, machine-dependent; but eventually the system executes a word QUIT , which really embraces the whole interpretive process. QUIT is so named because it is called in the event of unrecoverable error. It thus represents a "warm" restart of the system. The core of QUIT is a loop of the form

```
BEGIN RP! CR QUERY INTERPRET STATE @ 0=
                        IF ." OK" THEN REPEAT
```

The effect of RP! is to restore the return stack to its proper starting condition, the main stack having been reset earlier either on startup or by the error routine that called QUIT . CR sends a newline to the interpretive echo device, and QUERY accepts a line of up to 80 characters from the keyboard, placing them in the terminal buffer. The word INTERPRET is then called to execute in sequence the words in the line. Provided that the line contains a complete sequence of operations, i.e. provided that the state at the end of the line is not compiling, the system prints OK . In any case, control is returned to the start of the loop to read more program. If the system is compiling then it will continue to compile text from the new line.

INTERPRET may be defined as follows

```
: INTERPRET
  BEGIN BL WORD FIND ?DUP
```

```
        IF STATE + IF EXECUTE
                   ELSE , THEN ?STACK
  ELSE NUMBER DPL @ 1+
      IF DLITERAL
      ELSE DROP LITERAL THEN
    ?STACK THEN
UNTIL ;
```

The word FIND searches the dictionary for the next word in the buffer, after WORD has read it on to the top of the dictionary. If it fails to match it, then "false" is left on the stack; otherwise its compilation address is substituted for the address of the original string, and "true" is stacked above it. If the word is immediate, then "true" is represented by 1, otherwise it is represented by −1. If the word is not immediate and the value of STATE is 1, then the word is threaded, i.e. its compilation address is stored in the dictionary entry currently being compiled: otherwise it is immediately executed. FIND is a rather specialised compiling word. A word −′ in some implementations operates in a broadly similar manner.

If the word cannot be found in the dictionary, then an attempt is made to convert it into a number according to the radix in BASE . NUMBER is not one of the FORTH-83 required word set; but it calls one of the required words, CONVERT , which we reproduce here for the sake of the techniques it illustrates.

```
: CONVERT
  BEGIN 1+ DUP >R C@ BASE @ DIGIT
    WHILE SWAP BASE @ U* DROP ROT
                BASE @ U 63 D+ DPL @ 1+
                IF 1 DPL +! THEN R>
  REPEAT R> ;
```

This word is more general than NUMBER , as it converts the number pointed to by the top stack item, accumulating it in the second stack item, and leaving the top of the stack pointing to the first unconverted character. DIGIT is one of the kernel words. It converts an ASCII character lying second on the stack in accordance with a base placed on top of the stack if this is possible, returning "true" on the stack top with the digit value below it; otherwise it returns "false" on the stack and drops the ASCII character.

NUMBER uses CONVERT to convert the string read by WORD into a double-length number on the stack. It returns the position of its

point in user variable DPL , the value of DPL being −1 if there was no point. INTERPRET handles the number as a double-length only if it contained a point; otherwise the high-order half is dropped. If the word cannot be interpreted as a number, then NUMBER calls the error processor.

The words LITERAL and DLITERAL used in INTERPRET are immediate. LITERAL is defined

```
: LITERAL STATE @ IF COMPILE LIT , THEN ;
```

so that it is simply a dummy operation when the system is in the execution state (STATE zero) and leaves the number unchanged on the stack. If the system is compiling, however, LITERAL compiles the word LIT and then stores the number in the dictionary after it. Subsequently at execution time LIT will reload the number on to the stack from the parameter field of the word being executed. DLITERAL works similarly.

INTERPRETATION FROM FILE

When the word LOAD is executed, text is interpreted from file instead of the keyboard, the number of the block to be interpreted having previously been placed on the stack. LOAD is defined

```
: LOAD BLK @ >R >IN @ >R 0 >IN ! BLK !
                              INTEPRET R> >IN ! R> BLK ! ;
```

Thus the current block number (in BLK) and position pointer within the block (in >IN) are saved on the return stack, making LOAD recursive. This means that it is possible to initiate a new LOAD within a block that is already in process of loading.

The task of reading a new block into memory is carried out by WORD when called from INTERPRET . WORD is a much more complex operator than we implied in Chapter 3. It first tests the value of BLK ; if this is zero, then it takes text from the terminal input buffer, whose address is in user variable TIB . If it is nonzero, then it calls BLOCK to ascertain whether or not the desired block is already in memory, and to transfer it if it is not. BLOCK leaves the start address of the appropriate buffer on the stack. The interpreter stops automatically when execution comes to the end of a buffer, whether file or terminal. This is because every buffer ends with a null byte (00 hex, which is not the same as either ASCII zero or space). There is actually a dictionary entry for "null", which is usually referred to as X in the glossaries, since

ASCII null is non-printable. The effect of "null" is to drop the current return address from the return stack, thus ensuring that control is returned to the next higher level, i.e. not to the word following EXECUTE , as would normally be the case, but to the word following INTERPRET (normally inside QUIT). The definition of X is quite complex, since its operation depends on whether the code is coming from file or a terminal.

ERROR CHECKS

All versions of FORTH provide error checks within the system; and these are available for users to incorporate in their own extensions. The definition of INTERPRET given above includes a word ?STACK that checks for stack underflow and overflow. We saw earlier how the word ?PAIRS could be used to check correct pairing of words like DO and LOOP . There are also words ?COMP and ?EXEC that call the error processor unless the system is compiling or executing respectively. A common cause of error in stack processing is to assume that the stack has returned to a certain earlier state when it has not. The word !CSP stores the current stack pointer in a variable CSP ; and the word ?CSP enters the error processor if the present stack pointer is not the same as the value stored in CSP . This is an invaluable aid in development.

Techniques for error handling may differ from system to system. Typically the error-test words stack "true" or "false" with a characteristic error number above it, for instance

```
: ?PAIRS - 13 ?ERROR ;
```

The pair of items are differenced. If they were equal then zero (false) is passed as a parameter to ?ERROR , which then simply tests the truth value and enters the full error routine if an error has occurred, dropping the error number it not. ?ERROR is defined as

```
: ?ERROR SWAP IF ERROR ELSE DROP THEN ;
```

The word ERROR is thus called with an error number on top of the stack. It prints a query mark, and then calls the word MESSAGE , which operates differently according as the system is or is not fitted with disc filing. If there is no disc, the message printed might simply be the word "error" followed by the error number. If a disc file is available, however, the error number is taken as the number of a line on a particular screen (typically 4 or 5), and the content of that line is printed. The presence or absence of a disc file is recorded by a 1 or 0 in a

variable WARNING . If this variable is made negative, the error process is by-passed and the word (ABORT) is called. This word calls the predefined ABORT , which resets certain parameters and then calls the system-dependent QUIT . The idea of providing (ABORT) as well as ABORT is to enable the user to define his own rejection procedure if he wishes. (ABORT) calls ABORT by default. There is also a word ABORT" , which tests the top stack item and, if it represents "true", prints a string of (diagnostic) text before termination, such text being terminated in the program by a double quote.

Answers to Exercises

The following model answers have been checked using a Sinclair QL with COMPUTER ONE software. They are not guaranteed to work for other systems, especially if the 83 standard has not been followed.

1. 3 4 + 5 ×

2. (i) A B C − × D +

   (ii) A B + C D − ×

3. (i)
```
   2   4   5
3 3 6 6 2 2 10
```

   (ii)
```
     3
   4 4 7     2
7 7 7 7 49 49 51
```

   (iii)
```
   6     2
9 9 15 15 30 stack error
```

5. OVER OVER SWAP .R

6. SWAP ROT 4 ROLL

9. (i) : CUBE2 OVER DUP DUP * * ;

   (ii) : H. HEX . DECIMAL ;

   (iii) : DUODECIMAL 12 BASE ! ;

10. : D− DNEGATE D+ ;
    : 2SWAP 3 ROLL 3 ROLL ;
    : D> 2SWAP D< ;

11. : >= < 0= ;
    : <= > 0= ;

12. `: D= D- 0 SWAP 0 D+ + 0= ;`

    Note that `D- + 0=` gives the wrong answer if the two halves of the difference happen to be complementary.

13. Assume stack configuration on call is
    `---` byte address

```
: BCD! OVER 15 AND OVER ! SWAP -4 SHIFT
                                15 AND SWAP 2+ ! ;
```

14.
```
: EVEN/2 DUP 2 / OVER OVER DUP + =
                                IF SWAP THEN DROP ;
: ODDIV BEGIN DUP EVEN/2 DUP ROT = UNTIL ;
```

15. (i) `: 4MAX >R >R MAX R> MAX R> MAX . ;`

    (ii)
```
: 4PICK >R >R >R OVER R> SWAP R> SWAP
                                R> SWAP ;
```

    (iii) `2SWAP >R ROT ROT R> ROT ROT ;`

16. `: FAC 1 SWAP 1+ 1 DO I * LOOP . ;`

17.
```
: MOVE >R OVER @R + OVER <
            IF R< CMOVE ELSE R< <CMOVE THEN ;
```

18.
```
: DUMP HEX 0 DO DUP I + DUP 10 .R
                        C@ 5 .R CR LOOP DECIMAL ;
```

19.
```
: BLANKS BL FILL ;
: >PAD PAD 20 BLANKS PAD 20 EXPECT ;
: NOBL PAD DUP 20 + SWAP
    DO I C@ DUP BL =
    IF DROP ELSE EMIT THEN LOOP CR ;
: REV PAD 19 + 20 0 DO DUP I - C@
    EMIT LOOP CR ;
```

    N.B. The location of PAD changes if you define a new word. You may find that your system inserts a null at the end of the input.

20. : CONV #S DROP 32 HOLD ;
YDFTIN 12 UM/MOD 3 /MOD SWAP ROT 0
<# CONV CONV #S #> ;

21. : 3DIGITS # # # 32 HOLD ;
: PRCOLS OVER OVER CR <# 3DIGITS 3DIGITS #S #
> TYPE ;
: READ TIB @ 10 EXPECT ;
: ACCUMULATE 0 0 TIB @ 1+ CONVERT ;
: CALCULATOR 0 0 BEGIN READ ACCUMULATE C@
61 <>WHILE PRCOLS D+ REPEAT 2DROP PRCOLS ;

Will handle totals up to 4294 967 295.

22. : PRCOLS OVER OVER CR <# # # 46 HOLD
3DIGITS 3DIGITS # # 96 HOLD #> TYPE ;

and recompile CALCULATOR.

24. : TRACE CREATE DUP , DOES> ? ;

25. : SET CREATE , , DOES> DUP 2+ @ SWAP @ ! ;

Appendix

The ASCII Character Set

hex	decimal	character	key
00	0	null	
01	1	SOH	CTRL/A
02	2	STX	CTRL/D
03	3	ETX	CTRL/C
04	4	EOT	CTRL/D
05	5	ENQ	CTRL/E
06	6	ACK	CTRL/F
07	7	BEL	CTRL/G
08	8	BS	CTRL/H
09	9	H TAB	CTRL/I
0A	10	LF	CTRL/J
0B	11	V TAB	CTRL/K
0C	12	FF	CTRL/L
0D	13	CR	CTRL/M
0E	14	SO	CTRL/N
0F	15	SI	CTRL/O
10	16	DLE	CTRL/P
11	17	DC1	CTRL/Q
12	18	DC2	CTRL/R
13	19	DC3	CTRL/S
14	20	DC4	CTRL/T
15	21	NAK	CTRL/U
16	22	SYN	CTRL/V
17	23	ETB	CTRL/W
18	24	CAN	CTRL/X
19	25	EM	CTRL/Y
1A	26	SUB	CTRL/Z
1B	27	ESC	SHIFT CTRL/K
1C	28	FS	SHIFT CTRL/L
1D	29	GS	SHIFT CTRL/M
1E	30	RS	SHIFT CTRL/N
1F	31	US	SHIFT CTRL/O

20	32		space bar
21	33	!	SHIFT !
22	34	"	" or SHIFT "
23	35	#	SHIFT #
24	36	$	SHIFT $
25	37	%	SHIFT %
26	38	&	SHIFT &
27	39	'	' or SHIFT '
28	40	(	SHIFT (
29	41	)	SHIFT)
2A	42	*	SHIFT *
2B	43	+	SHIFT +
2C	44	,	,
2D	45	-	-
2E	46	.	.
2F	47	/	/
30	48	0	0
31	49	1	1
32	50	2	2
33	51	3	3
34	52	4	4
35	53	5	5
36	54	6	6
37	55	7	7
38	56	8	8
39	57	9	9
3A	58	:	SHIFT :
3B	59	;	;
3C	60	<	SHIFT <
3D	61	=	=
3E	62	>	SHIFT >
3F	63	?	SHIFT ?
40	64	@	SHIFT @
41	65	A	SHIFT A
42	66	B	SHIFT B
43	67	C	SHIFT C
44	68	D	SHIFT D
45	69	E	SHIFT E
46	70	F	SHIFT F
47	71	G	SHIFT G
48	72	H	SHIFT H

49	73	I	SHIFT I
4A	74	J	SHIFT J
4B	75	K	SHIFT K
4C	76	L	SHIFT L
4D	77	M	SHIFT M
4E	78	N	SHIFT N
4F	79	O	SHIFT O
50	80	P	SHIFT P
51	81	Q	SHIFT Q
52	82	R	SHIFT R
53	83	S	SHIFT S
54	84	T	SHIFT T
55	85	U	SHIFT U
56	86	V	SHIFT V
57	87	W	SHIFT W
58	88	X	SHIFT X
9	89	Y	SHIFT Y
5A	90	Z	SHIFT Z
5B	91	[	[
5C	92	\	\
5D	93	]	SHIFT]
5E	94	^	SHIFT ^
5F	95	_	_
60	96	£ or `	£ or `
61	97	a	A
62	98	b	B
63	99	c	C
64	100	d	D
65	101	e	E
66	102	f	F
67	103	g	G
68	104	h	H
69	105	i	I
6A	106	j	J
6B	107	k	K
6C	108	l	L
6D	109	m	M
6E	110	n	N
6F	111	o	O
70	112	p	P
71	113	q	Q

72	114	r	R
73	115	s	S
74	116	t	T
75	117	u	U
76	118	v	V
77	119	w	W
78	120	x	X
79	121	y	Y
7A	122	z	Z
7B	123	{	{
7C	124	|	SHIFT |
7D	125	}	SHIFT }
7E	126	~	SHIFT ~
7F	127	DEL	DEL

Legend:

SOH	start of header	NAK	neg acknowledge
STX	start of text	ETB	end of text block
ETX	end of text	CAN	cancel
EOT	end of transmission	EM	end of medium
ENQ	enquiry	SUB	substitute
ACK	acknowledge	FS	file separator
SO	shift out	GS	group separator
SI	shift in	RS	record separator
DLE	data-link escape	US	unit separator
DC	device control		

N.B. The codes given here are probably the commonest, but by no means universal.

Glossary and Index

The items in this glossary are arranged in ASCII order. Each entry is made up as follows:

The word, and its page reference(s)

A short description — refer to main text for fuller explanation.

A picture of the stack top before and after execution,
<condition before> --- <condition after>
The highest stack item is on the right.

Symbols used as follows:

n	16-bit number
d	double-length number
addr	16-bit address
t/f	logical flag
b	byte (low-order in 16-bit field)

Special points (displayed at end of entry), coded as follows:

R	FORTH-83 required word
O	one of the controlled reference words
E	in a FORTH-83 extension set
C	permitted within colon def only
I	immediate
U	user variable

! 12
Store n at address
n addr --- R

!BITS 21
Store n1 masked by n2 in addr
n1 addr n2 ---

!CSP 63
Store stack pointer in CSP.

40
Generate next digit from an unsigned double number by dividing by current BASE and put in output string. Keep quotient. May only be used between <# and #> .
d1 --- d2 R

#> 40
End pictured numeric output conversion. Drop top stack item and leave address and count of output sequence.
d --- addr n R

#S 40
Convert remaining digits of unsigned double number adding each

to pictured output string. May only be used between <# and #> .
d −−− 0 0 R

#TIB 32
Leave address of word containing the number of characters in the terminal buffer.
−−− addr R, U

' 49
Leave compilation address of next word in input stream.
−−− addr R

(35
Acccept comment up to)
−−− R

(ABORT) 78
User-definable reset sequence. Predefined as ABORT

(DO) 64
Execution-time actions of DO.

(LOOP) 64
Execution-time actions of LOOP.

* 4
Leave single-length product of top two items.
n1 n2 −−− n3 R

** 5
Raise n1 to power n2.
n1 n2 −−− n3

*/ 6
Multiply n1 by n2 giving double-length product, and divide by n3 giving single-length result.
n1 n2 n3 −−− n4 R

*/MOD 6
As */ but leave remainder n4 and quotient n5.
n1 n2 n3 −−− n4 n5 R

+ 4
Leave single-length sum of top two items.
n1 n2 −−− n3 R

+! 13
Add n to store at addr.
n addr −−− R

+− 7
Leave n1, negated if n2 was negative
n1 n2 −−− (−)n1

+LOOP 24
Add signed increment n to loop index. Return to DO if result is less than limit; otherwise discard loop parameters and exit.
n −−− R, I, C

, 32
Copy 16-bit number from stack to dictionary. Increase HERE by 2.
n −−− R, I

− 4
Subtract n2 from n1 and leave difference.
n1 n2 −−− n3 R

−' 75
Leave parameter-field address below "false" of next name in input stream. If not present, leave "true".
−−− addr f
−−− t

−−> 37
Continue loading next screen.
−−− I,O

−DUP 8
Duplicate n iff it is nonzero.
n −−− n n (n <> 0)

0 --- 0

-TRAILING 31
Adjust character count n1 of a string beginning at addr to exclude trailing blanks.
addr n1 --- addr n2 R

. 7
Print n according to BASE.
n --- R

." 22
Accept text up to next " and compile so that later execution will display it.
--- R

.(8
Accept and immediately display text up to next).
--- R, I

.R 7
Print n1 right-aligned in a field of length n2.
n1 n2 --- O

/ 4
Integer divide n1 by n2 and leave quotient.
n1 n2 --- n3 R

/MOD 5
Integer divide n1 by n2 to leave remainder n3 and quotient n4.
n1 n2 --- n3 n4 R

0< 19
Test n. If negative leave "true"
n --- t/f R

0= 19
Test n. If zero leave "true"
n --- t/f R

0> 19
Test n. If positive and nonzero leave "true"
n --- t/f R

1+ 22
Increase n by unity.
n --- n+1 R

1+! 22
Add unity into memory location pointed to by addr.
addr ---

1- 22
Decrease n by unity.
n --- n-1 R

1-! 22
Subtract unity from memory location pointed to by addr.
addr ---

2! 17
Store d at addr.
d addr --- E

2* 22
Double n (neglecting overflow).
n --- 2n O

2+ 22
Add two to n.
n --- n+2 R

2- 22
Subtract two from n.
n --- n-2 R

2/ 22
Shift n1 right one bit.
n1 --- n2 R

2@ 17
Leave double-length number stored at addr.
addr --- d E

2CONSTANT 17
Define next input string as the name of a double-length constant, initialised to the value of d.
d ––– E

2DROP 17
Remove a double-length number from the stack.
d ––– E

2DUP 17
Duplicate a double-length number.
d ––– d d E

2OVER 17
Copy d1 to the stack top.
d1 d2 ––– d1 d2 d1 E

2ROT 17
Move d1 to the stack top.
d1 d2 d3 ––– d2 d3 d1 E

2SWAP 17
Exchange two double-length items.
d1 d2 ––– d2 d1 E

2VARIABLE 17
Assign next word in input stream to a new double-length variable.
––– E

79-STANDARD
Output a confirming message if the FORTH-79 standard is available; otherwise treat as an error.
–––

: 15, 62
Create a new dictionary entry, using the next input string as the name. Compile threads to following words up to the next semicolon. If input stream is exhausted before a semicolon, report an error.
––– R

; 16
Terminate current colon definition and stop compilation.
––– R, I, C

:: 51
Introduce code segment in a compiling word.
–––

:CODE 42, 62, 68
Used to terminate the definition of a compiling word and introduce the code obeyed by all words defined by this word.
––– E

:S 64
Execution-time actions associated with the semicolon.
–––

< 19, 47
Leave "true" if n1 is less than n2.
n1 n2 ––– t/f R

<> 19
Leave "true" if top two items are unequal.
n1 n2 ––– t/f

<# 40
Initialise pictured numeric output.
––– R

<MARK 64
Mark destination of a backward branch during compilation.
––– addr C, I, E

<RESOLVE 64
Compile offset at source of backward branch.
addr ––– C, I, E

= 19

Leave "true" if top two items are equal.
n1 n2 --- t/f R

> 19
leave "true" if n1 is greater than n2.
n1 n2 --- t/f

>< 29
Swap bytes.
n1 --- n2

>BODY 49
Convert compilation address to parameter-field address.
addr1 --- addr2 R

>IN 33
Leave address of a variable containing a pointer to the current position in the input buffer.
--- addr U, R

>MARK 65
Allot space for offset and mark source of forward branch.
--- addr C, I, E

>MOVE< 29
Move block of n 16-bit words starting at addr1 to addr2 performing a byte swap on each.
addr1 addr2 n ---

>R 26
Move top item to top of return stack.
n --- R

>RESOLVE 65
Compile offset for forward branch and store it at marked source location.
addr --- C, I, E

? 13
Display the number in addr, using current radix.
addr ---
R

?BRANCH 65
Transfer control to location given by offset immediately following if flag is true.
t/f --- C, E

?CSP 63, 77
Check that the current stack pointer has the same value as that previously stored by !CSP. Error if unequal.

?DUP 8
Duplicate top item if it is nonzero.
n --- n n (n nonzero)
0 --- 0 R

?ERROR 77
Issue error message number n if flag represents "true".
t/f n ---

?PAIRS 64, 77
Issue error message if top two stack items are unequal.
n1 n2 ---

?STACK 77
Issue error message if the stack is out of bounds.

@ 12
Replace addr by its contents.
addr --- n R

@BITS 21
Return contents of addr masked by n1.
addr n1 --- n2

ABORT 78
Warm restart.

C, 32
Store b at top of dictionary, and advance pointer.
b ––– O

C! 28
Store b in addr.
b addr ––– R

C@ 28
Leave contents of byte addr.
addr ––– b R

CATALOG 42
List the names in the context vocabularies.
–––

CLEAR 37
Write all updated buffers back to mass store.
–––

CMOVE 27
Move n bytes from addr1 to addr2.
addr1 addr2 n ––– R

CMOVE> 28
Move n bytes from addr1 to addr2 starting at the highest address (addr1 + n – 1).
addr1 addr2 n ––– R

CODE 42, 68
Create a dictionary entry to be defined wholly in assembler code.
––– E

COM 21
Leave ones complement of n1.
n1 ––– n2

COMPILE 64, 66
Compile compilation address of next word in input string into dictionary, and advance pointer.
C, R

CONSTANT 14, 46, 62
Define next string in input as the name of a constant whose value is n.
n –––

CONTEXT 42
Leave address of variable specifying vocabulary for initial search.
––– addr U, E

CONVERT 41, 75
Numeric conversion according to current base of string pointed to by addr1. Accumulate into d1. Leave address of first non-convertible character.
d1 addr1 ––– d2 addr2 R

COPY 37
Copy contents of screen n1 to screen n2.
n1 n2 –––

COUNT 31
Leave address of first byte and the character count of string pointed to by addr.
addr ––– addr+1 n R

CR 27, 30
Output newline.
––– R

CREATE 50, 60
Create a dictionary entry with next word in input string as the name.
R

CURRENT 43
Leave address of a variable specifying vocabulary for new definitions.
––– addr U, E

CSP 63
Leave address of variable temporarily storing the stack pointer.

--- addr U

D+ 17
Leave sum of two double-length numbers.
d1 d2 --- d3 R

D- 17
Leave difference between two double-length numbers.
d1 d2 --- d3 E

D. 17
Display signed double-length number according to current base.
d --- E

D.R 17
Display d right-aligned in a field n characters wide at current cursor position.
d n --- E

D0= 17
Leave "true" if d is identically zero.
d --- t/f E

D2/
Shift d1 right one place.
d1 --- d2 E

D< 17
Leave "true" if d1 is less than d2.
d1 d2 --- t/f R

D> 17
Leave "true" if d1 is greater than d2.
d1 d2 --- t/f

D= 17
Leave "true" if two d-l items are equal.
d1 d2 --- t/f E

DABS 17
If d1 is negative, leave its twos complement.
d1 --- d2 E

DECIMAL 10
Set conversion base to ten.
--- R

DEFINITIONS 43
Set content of CURRENT equal to that of CONTEXT.
--- R

DEPTH 72
Leave number equal to the number of items on the stack.
--- n R

DIGIT 75
Convert low byte of n1 (assumed ASCII) according to base given as n2, leaving binary equivalent and "true". If not possible leave only "false".
n1 n2 --- n3 true
n1 n2 --- false

DLITERAL 76
If executing, do nothing. If compiling, compile d as a double-length literal in the dictionary.
d ---

DMAX 17
Leave larger of d1 and d2.
d1 d2 --- d3 E

DMIN 17
Leave smaller of d1 and d2.
d1 d2 --- d3 E

DMINUS 17
Replace d by its twos complement.
d --- -d

DNEGATE 17
Replace d by its twos complement.

d --- -d R

DO 23, 64
Begin a counting loop.
I, C, R

DP 32, 72
Leave address of variable containing dictionary pointer.
--- addr U

DPL 76
Leave address of variable containing a count of the number of digits to the right of the last decimal point input.
--- addr U

DOES> 50
Terminate compile-time actions and commence run-time actions in defining a new defining word.
--- I, C, R

DROP 8
Delete top stack item.
n --- R

DU< 17
Leave "true" if d1 is less than d2, both treated as unsigned.
d1 d2 --- t/f E

DUP 8
Duplicate top stack item.
n --- n n R

DUMP 29
List contents of n locations starting at addr.
addr n --- O

ECHO 30
Output low byte to top item as an ASCII code.
n ---

EDITOR 42
Change context vocabulary to editor.
--- O

ELSE 18, 66
Terminate true part and start false part of conditional statement.
--- I, C, R

EMIT 30
Output low byte of top item as an ASCII code.
n --- R

EMPTY 44
Delete all dictionary entries to FENCE.

EMPTY-BUFFERS 36
Mark all file buffers as empty.
--- O

END 22
Test condition in repeated block. Exit if true.
--- I, C

END-CODE 68
Terminate a definition introduced by CODE or ;CODE.
--- E

ENDIF 18, 66
Terminate conditional sequence.
--- I, C

ERASE 27
Transfer n zero bytes to memory starting at addr.
addr n --- O

ERROR 77
Notify error number n, and execute warm restart.
n ---

EXECUTE 74
Execute dictionary entry whose compilation address is addr.
addr ––– R

EXIT 63
When executing a colon-defined word, terminate execution.
––– C, R

EXPECT 31
Read from terminal, storing at addr. Stop on newline or when n characters have been transferred.
addr n ––– R

FENCE 44
Places on stack the address of a variable containing the dictionary address below which entries should not be deleted.
––– addr U

FILL 27, 28
Fill memory with n instances of b starting at addr.
addr n b ––– R

FIND 75
Place on stack the compilation address of the word starting at addr1, with "true" above it. If not found, leave original address and "false".
addr1 ––– addr2 t/f R

FLUSH 36
Write to mass storage all blocks marked as updated, and delete "update" markers.
––– R

FORGET 45
Delete from dictionary all entries up to and including next word in input stream.
––– R

FORTH 42
Return to prime vocabulary.
––– I, R

FORTH-83
Output confirming message if system conforms to FORTH-83 standard.
––– R

H. 10
Output one number in hexadecimal, returning to original base.
–––

HERE 32
Return address of next available dictionary location.
––– R

HEX 10
Change number base to sixteen.
––– O

HLD 41
Return address of a variable holding address of latest text character during output conversion.
––– addr U

HOLD 40
Insert b as a character in a pictured output stream.
b ––– R

I 24
Return index of innermost loop.
––– n C, R

I'
Return index of innermost loop.
––– n C

ID. 50
Print a dictionary entry name, given name-field address.
addr –––

IF 18, 23, 49, 65
Test flag and execute conditional statement.
t/f ––– I, C, R

IFEND 68
Terminate conditional interpretation.
–––

IFTRUE 68
Introduce conditional interpretation.
–––

IMMEDIATE 63
Mark latest dictionary entry to be executed immediately if encountered during compilation.
––– R

IN 33
Leave address of a variable containing a pointer to the current position in the input buffer.
––– addr U

INDEX 37
Print first line of every screen over range given by n1 and n2.
n1 n2 –––

INTERPRET 74
Begin interpretation in block whose number is given in BLK starting at character indexed by >IN.
––– O

J 25
Return index of next enclosing loop.
––– n C, R

K 25
Return index of second enclosing loop.
––– n C, O

KEY 30
Return ASCII value of next character from input device.
––– b R

LAST 50
Leave address of a variable containing the name-field address of the last dictionary entry.
––– addr U

LATEST 50
Leave name-field address of last word in the current vocabulary.
––– addr

LEAVE 24
Alter loop limit to value of current index so forcing termination on completion of loop.
––– C, R

LFA 49
Convert a parameter-field address to the corresponding link-field address.
addr ––– addr

LIMIT 72
Constant equal to one greater than the largest available memory address.
––– addr

LINE 37
Leave physical address of the beginning of line n of the screen whose number is in SCR.
Range is 0 .. 15.
––– addr

LINELOAD 37
Start interpretation at line n1 in screen n2.
n1 n2 –––

LIST 36

Display screen n.
n --- O

LIT 59, 76
Place contents of next dictionary parameter on the stack.
--- n

LITERAL 76
Compiling ... compile LIT followed by the numeric value of the top stack item;
Interpreting ... ignore.
n --- R

LOAD 37, 76
Begin interpreting screen n.
n --- R

LOADS 38
Define next word in input stream as a command to load screen n.
n ---

LOOP 23, 64
Compile words to increment and test loop index. If less than limit branch back to DO, otherwise exit.
--- C, R

M* 18, 26
Multiply two signed single-length items n1 and n2 to give a double-length result.
n1 n2 --- d

M/ 18
Divide double-length signed quantity d by single-length signed quantity n1 to leave quotient (N.B. quot may be d-l in some systems.)
d n1 --- n2

MASK 21
Return mask of n1 ones, left-aligned if positive, right if negative.
n1 --- n2

MAX 6, 20, 46
Return greater of two numbers.
n1 n2 --- n3 R

MESSAGE 77
Print line n of message screen.
n ---

MIN 6, 20
Return lesser of two numbers.
n1 n2 --- n3 R

MINUS 5
Return twos complement of top item.
n --- -n

M/MOD 18
Divide double-length signed item by single-length signed item. Leave double quotient d2 with remainder n2.
d1 n1 --- n2 d2

MOD 5
Divide n1 by n2, leaving remainder with same sign as dividend.
n1 n2 --- n3 R

MOVE 28
Move n bytes from addr1 to addr2 without overwriting.
addr1 addr2 n ---

MS 30
Wait n milliseconds.
n ---

NAND 20
Perform NAND operation on top two items.
n1 n2 --- n3

NEGATE 5, 21
Return twos complement of top item.
n --- -n R

NFA 49
Convert parameter-field address to corresponding name-field address.
addr --- addr

NOR 20
Perform NOR operation on top two items.
n1 n2 --- n3

NOT 20
Leave ones complement of top item.
n1 --- n2 R

NUMBER 75
Convert a character string starting at addr, with preceding count, to a signed double number according to current base.
addr --- d

O. 10
Output one number in octal and return to original base.
n ---

OCTAL 10
Reset current base to eight.
--- O

OR 20
Perform bitwise OR on top two items.
n1 n2 --- n3 R

OTHERWISE 68
Terminate "true" sequence and start "false" sequence in conditional execution.

OVER 8
Return a copy of the second stack item.
n1 n2 --- n1 n2 n1 R

PAD 32, 39
Return top address of scratch area above the dictionary. Normally 84 bytes long.
--- addr R

PFA 49
Convert name-field address to corresponding parameter-field address.
addr1 --- addr2

PICK 9
Return contents of n1th stack item, excluding n1.
n1 --- n2 R

PREV 36
Return address of variable containing address of most recently used buffer.
--- addr U

QUERY 74
Accept up to 80 characters from the terminal into terminal buffer.
--- O

QUIT 74
Warm restart.
--- R

R> 26
Move top item of return stack to top of main stack.
--- n R

R@ 26
Return top item of return stack.
--- n R

R0 72
Leave address of variable containing base address of return stack.
--- addr U

RECURSE 71
Compile compilation address of current word so that it may be executed recursively.
--- C, I, O

REMEMBER 44
Define next input word as a command that deletes itself and all words defined subsequently from the dictionary.

REPEAT 22
Terminate repetitive loop.
--- I, C, R

ROLL 9
Move nth item of stack (not counting n itself) to the top, moving intermediate items down.
n1 n2 ... nn n --- n2 ... nn n1 R

ROT 8
Move third stack value to the top.
n1 n2 n3 --- n2 n3 n1 R

RP! 74
Initialise return stack.

S->D 17
Sign-extend a single-length number to form a double.
n --- d

S0 40, 72
Return address of a user variable containing base address of main stack.
--- addr U

SAVE-BUFFERS 36
Write all updated buffers to mass storage.
--- R

SCR 37
Return address of a variable containing the number of the screen most recently listed.
--- addr U, O

SET 12
Define the next input word as a command that will store n in the given address.
n addr ---

SHIFT 21
Perform logical shift of n1 by n2 bits, left if positive, right if negative.
n1 n2 --- n3

SIGN 40
Store a minus sign in the next position in a pictured output sequence if n is negative.
n --- R

SMUDGE 62
Toggle the third bit in the name field of a word being compiled.

SP 40, 72
Leave address of a variable containing the stack pointer.
--- addr U

SP! 40
Initialise the main stack pointer.

SP@ 40
Return address of top of main stack just before SP@ was executed.
--- addr O

SPACE 30
Output an ASCII space.
--- R

SPACES 30
Output n spaces.
n --- R

SPAN 32
Leave address of variable containing the count of characters received during the last EXPECT.
--- addr U, R

STATE 62
Leave address of a variable containing the compilation state.
--- addr U, R

SWAP 8
Exchange top two stack values.
n1 n2 --- n2 n1 R

THEN 18, 65
Terminate conditional sequence.
--- I, C, R

THRU 38
Load consecutively blocks n1 through n2.
n1 n2 --- O

TIB 32, 73
Leave address of a variable containing the address of the terminal buffer.
--- addr U, R

TRAVERSE 50
Move across a name field starting at addr1, to right if n is positive, left if negative. Leave address of end of field.
addr1 n --- addr2

TYPE 31
Send n characters starting at addr to current output device.
addr n --- R

U. 18
Print an unsigned number according to current base.
n --- R

U.R 18
Print n1 as an unsigned number right justified in a field of n2 bytes.
n1 n2 --- O

U< 18
Leave "true" if n1 < n2 both treated as unsigned.
n1 n2 --- t/f R

U> 18
Leave "true" if n1 > n2 both unsigned.
n1 n2 --- t/f

UM* 18
Form unsigned double-length product of n1 and n2.
n1 n2 --- d R

UM/MOD 18
Divide d by n1 (unsigned) leaving unsigned remainder n2 and unsigned quotient n3.
d n1 --- n2 n3 R

UNTIL 21
Compile words to test top stack item. Loop if false, exit from loop if true.
t/f --- I, C, R

UPDATE 36
Mark most recently referenced block so that it will be written back to mass store before being overwritten.
--- R

USE 36
Leave address of variable containing address of least recently used buffer.

--- addr U

USER 13, 46, 62
Assign the next word in the input stream as the name of a user variable with offset n.
n ---

VARIABLE 11, 46, 62
Assign the next word in the input stream to a new variable entry in the dictionary.
--- R

VLIST 42
List the names in the context vocabularies.

VOCABULARY 43, 55
Assign the next word in the input stream as the name of a new vocabulary that will become the context when the name is quoted.
--- R

VOC-LINK 43
Leave the address of a variable containing the head of a chain linking all vocabularies.
--- addr U

WHILE 22
Compile code to test a flag and exit from a repetitive loop if false, otherwise continue to obey loop.
t/f --- I, C, R

WORD 33, 76
Copy characters from the input buffer to the top of the dictionary until a delimiter given by b is encountered. Ignore leading instances of the delimiter. Insert the character count as the first byte of string, and leave its address on the stack.
b --- addr R

X 76
Pseudonym for the dictionary entry whose name is ASCII null. Its effect is to terminate text interpretation.

XOR 20
Perform exclusive OR on top two items.
n1 n2 --- n3 R

[62
End compilation mode.
--- I, R

['] 49
Compile compilation address of next word in input stream as a literal into the dictionary.
--- C, I, R

[COMPILE] 66
Force compilation of the following word in the input stream.
--- I, C, R

] 62
Set compilation mode.
--- R