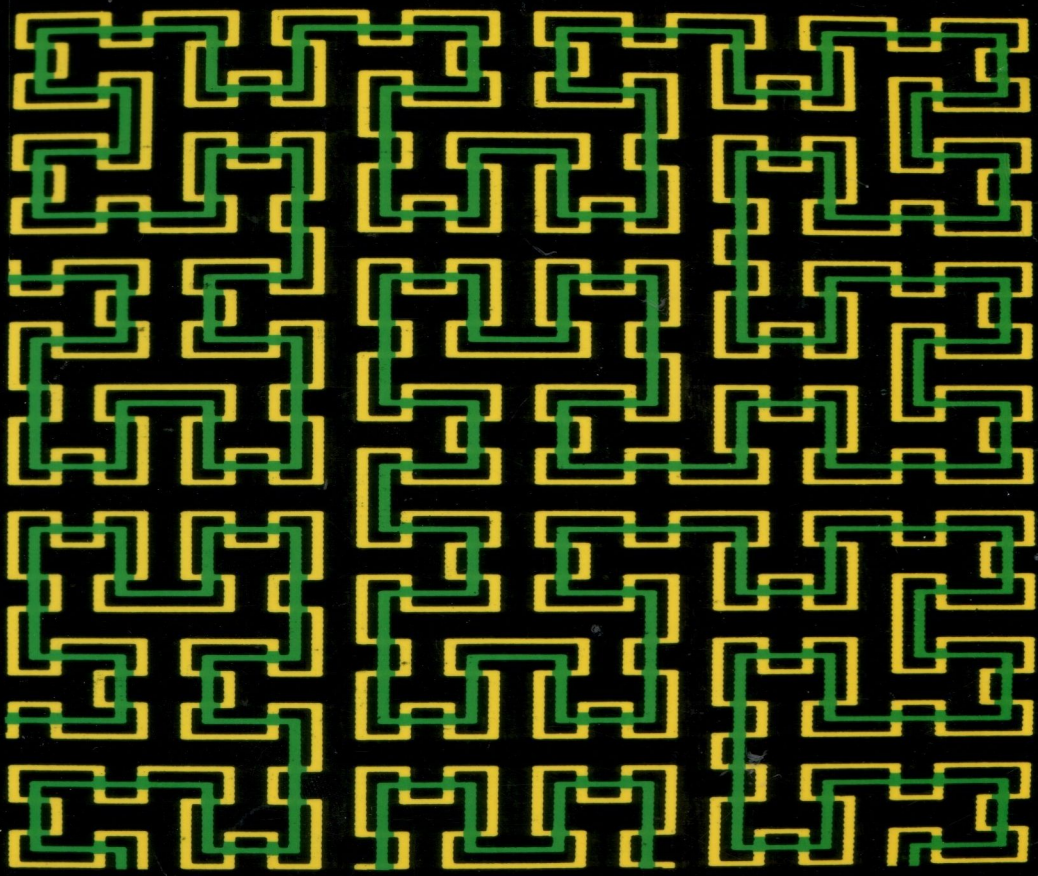
FORTH

on the BBC Microcomputer

RICHARD DE GRANDIS-HARRISON



FORTH

on the BBC Microcomputer

RICHARD DE GRANDIS-HARRISON

ACORNSOFT

Acknowledgements

This manual and the associated implementation of FORTH could not have been produced without the assistance and advice of many people.

Firstly, I must thank David Johnson-Davies of Acornsoft for the generous way he has offered both facilities and information, greatly easing my task. He also provided the factorial calculations of chapter 11. Jeremy Bennett, Philippa Bush, and Simon Hughes of Acornsoft were all involved in the preparation of this manual and suggested a number of improvements to the final text.

Many of my fellow members of the FORTH Interest Group UK helped with discussion, encouragement, and advice. I would especially like to thank Ken King who tested the provisional system, and Harry Dobson who provided the machine code version of the Editor MATCH routine.

The greatest common divisor calculator of chapter 6 is by R L Smith and the random number generator used in chapter 8 is by J E Rickenbacker. They appeared in Forth Dimensions Vol 2 pages 167 and 34 respectively.

Acornsoft FORTH owes much to the public domain publications of the Forth Interest Group and in particular to the work of Bill Ragsdale. Finally I wish to express my gratitude to Charles Moore of Forth Inc. without whose efforts the language would not exist.

R. deG-H 17/1/83

FIRST EDITION

ISBN 0 907876 06 4

First Published by Acornsoft Limited, 4a Market Hill, Cambridge, CB2 3NJ, England

Contents

1 About this manual

Although this manual is written to explain the operation of FORTH on the BBC Microcomputer, most of the contents are applicable to any version of the FORTH language. This implementation follows the FORTH-79 standard.

Individual addresses of, for example, registers and subroutines and the details of the memory map will be different on different machines, and the machine code sections are necessarily concerned with the 6502 microprocessor. In the descriptions of FORTH code, only the tape interface (chapter 8) and the graphics (chapter 12) are likely to be significantly different in other versions.

In this manual all FORTH words are written in upper-case letters, exactly as they are typed in and appear on the display. Since FORTH may use any character that can be typed on the keyboard, there may occasionally be confusion between FORTH words and punctuation marks. In any cases where such confusion may arise, FORTH words are placed in angle brackets, e.g. <.> , <,> and <."> .

In examples which contain text, both typed at the keyboard and produced by the computer, the underlined sections represent the computer's output, for example:

```
2 3 + .
5 OK
```

All keyboard input must be terminated by pressing the RETURN key, and this will not normally be shown explicitly.

All numbers appearing in the text will, unless otherwise stated, be given in decimal base.

2 About FORTH

FORTH was invented around 1969 by Charles H Moore. It was originally created as a convenient means of controlling equipment by computer. Most high-level languages that can be used on mini- and micro- computers (for example BASIC) are too slow for such control and the only other alternative is to use machine code routines. These, however, are very tedious to write and enter.

FORTH solves many of these problems by allowing fast-executing programs to be written in a high-level language. It also has the great advantage on small systems of using very little memory for program storage. One further advantage, which will become apparent as you use the language, is that FORTH encourages the writing of well-structured programs.

The speed of FORTH is largely due to the fact that it is a compiled language, so that the stored program is in a form very close to machine code. Unlike most other compiled languages, however, FORTH is interactive, which means that each new word can be tested as soon as it has been entered. If it does not do what you want, it can be changed immediately until you are satisfied.

Perhaps the most powerful feature of FORTH is that it is an extensible language. When you define a new word in FORTH, it becomes an integral part of the language and can be used to produce further definitions in exactly the same way as the words resident in the basic system. This allows the production of short, neat solutions to complex problems.

You may be beginning to realise, from what has been said so far, that writing programs in FORTH is very different from writing in languages like BASIC. A FORTH

program consists of a series of definitions of actions, each represented by a 'word'. These words are then combined in further definitions until the required action of the whole program is represented by a single word. The program can then be executed by typing this single word at the keyboard.

The procedure for writing a program in FORTH begins with a specification of the overall action of the program. This is then broken down into a sequence of small tasks and these, if necessary, are further divided into simpler tasks. Eventually the tasks are reduced to the point where each is very easy to write in FORTH code. The program is then written, starting with these simple routines and building back up to the full program. Testing can be carried out at each stage, greatly reducing the chance of errors in the final program.

As an example we can consider the task of controlling a domestic washing machine. The whole program might be represented by the word WASHING which could be defined as:

```
: WASHING
WASH RINSE DRY ;
```

The words WASH, RINSE and DRY could themselves be defined as:

```
: WASH
FILL HEAT SOAP AGITATE SPIN ;

: RINSE
FILL AGITATE SPIN ;

: DRY
SPIN SPIN ;
```

At the next lower level the words FILL and HEAT, for example, could be written:

```
: FILL
TAP ON
BEGIN ?FULL UNTIL
TAP OFF ;
```

and

```
: HEAT
HEATER ON
BEGIN ?TEMPERATURE UNTIL
HEATER OFF ;
```

Coding could then begin with the definitions of the actions of the words TAP, HEATER, ON, OFF, ?FULL etc. The action of each word would be checked, with a simulation of the machinery and sensors of the washing machine, until the program is completed by the definition of WASHING.

This example illustrates that, in FORTH, the problem to be solved at any stage is simple and well-defined. Note also that many of the words appear several times; once a word is defined it may be used in a number of different situations, greatly easing the programming load.

FORTH is an example of threaded code. The words in a FORTH program can be imagined to be strung together like beads on a thread. From one word the thread loops to pass through all the words in its definition and, if necessary, further loops include the words of lower level definitions. Ultimately the thread returns to the highest level word of the sequence.

FORTH is actually implemented as indirect threaded code, where each 'bead' on the thread is not the routine itself but the address of the routine. In the dictionary, therefore, each word consists of a list of the addresses of the words out of which it is built.

So far there have been many references to 'words' in FORTH, so it is about time to define what can be used as a FORTH word.

A word is defined as any combination of characters, separated by one or more spaces from another word. Any character that can be typed on the keyboard, including non-printing characters and control codes, is allowed. The only characters that cannot be used in a FORTH word are a space, which is reserved as a delimiter to separate successive words, and a null (ASCII zero), which is used to mark the end of input text. In Acornsoft FORTH a word may be of any length up to a maximum of 31 characters.

The following are examples of valid words:

```
.
FORTH
."
+!
EMPTY-BUFFERS
```

The following are not valid:

```
EMPTY BUFFERS (includes a space)
THIS-WORD-CAN'T-BE-USED-IN-FORTH (32 characters)
```

3 Starting FORTH

3.1 Loading instructions

Acornsoft FORTH is supplied on disk, ROM, or cassette.

For the disk-based system refer to the pack for loading instructions.

If using the ROM system the command *FORTH will put you into the FORTH system.

To load and run the tape version, place the cassette in the cassette recorder, fully rewound, type

```
CHAIN "FORTH"
```

and press <RETURN>; the 'Searching' message should appear on the screen as you do this. Press the PLAY button on the casssette recorder and wait for the program to load. Loading will take more than five minutes; note that several files will be loaded, and all are called FORTH. Then there will be a short pause while FORTH re-locates.

When loading is complete a copyright message, and the sign-on prompt

```
Acornsoft FORTH
OK
```

will be displayed on the screen.

The Editor, Assembler, and Graphics Demonstration are loaded from within FORTH - loading instructions are supplied with the casette, disk, or ROM pack.

In all but the ROM-based version of Acornsoft FORTH, pressing the BREAK key will take you out of FORTH.

3.2 ESCAPE

In all versions of Acornsoft FORTH the ESCAPE key can be used to leave the current task and return control to the keyboard. It will terminate the execution of any FORTH routine immediately and the word ESCAPE will be displayed. The only exception is if an endless machine code loop is being executed, in which case execution may be terminated by pressing the BREAK key.

3.3 BREAK

FORTH has its own operating system for saving to and loading from cassette or disk, so monitor commands will not be used again, unless you decide to use them. The only exceptions are if your program crashes, or if you press the BREAK key, when control will return to the operating system. The following example illustrates the procedure to verify your return to FORTH.

Once FORTH is loaded and the sign-on prompt has appeared type in the following:

```
: STARS BEGIN 42 EMIT 2 SPACES AGAIN ;
```

After you press the RETURN key, FORTH will respond with OK. This is the standard response to all correct operations. If you do not get the OK response, check that you have typed the example correctly - it is important to leave at least one space between each word.

You have caused a new definition named STARS to be entered in the FORTH dictionary. This can be checked by typing

```
VLIST
```

and then <RETURN>. This will give a list of all the words present in the dictionary. The listing may be stopped at any time by pressing the TAB key. It may then be restarted by pressing the Space Bar - any other key will abort the listing and return control to the keyboard.

The first word in the list will be STARS, showing that it is now present in the dictionary.

When the message OK appears (either after the listing is complete or when you have aborted the listing by, for example, pressing the TAB key twice) execute the word by typing STARS (don't forget to press RETURN) and the system will type an endless display of stars. You will obtain no response from any key except BREAK (or ESCAPE) since an endless loop is being executed.

Press the BREAK key to get the BASIC prompt '>' and then press function key 8 when the sign-on prompt will again appear. Typing in

STARS

will give the same response as before, showing that STARS is still present in the FORTH dictionary. This is an example of a 'warm' start, in which all current dictionary entries are retained. Since FORTH is relocated on loading the warm start entry point may change, but function key 8 will always start at the correct address.

Repeat the sequence of executing STARS and pressing the BREAK key, but this time re-enter FORTH by pressing function key 9.

The sign-on prompt will appear again but this time you will find that a VLIST no longer includes the word STARS, showing that it is not in the dictionary. An attempt to execute STARS will give:

STARS ?

The question mark is displayed whenever a word is not recognised by FORTH. Restarting FORTH with function key 9 gives a 'cold' start which forgets everything except the nucleus dictionary. A cold start can be performed from within FORTH by typing the word COLD, and a warm start by typing the word WARM.

When either a cold or warm start is executed the sign-on prompt is printed. The copyright message is only given in the case of a cold start. Initially, FORTH will accept numbers in decimal and both stacks (see chapter 4) are cleared. New definitions will be added to the FORTH vocabulary.

4 Stacks of arithmetic

4.1 Stacks

Most high-level languages use one or more stacks for their internal operations, for example for storing intermediate values during the calculation of the result of an arithmetical expression. Languages such as FORTRAN and BASIC are designed so that the user needs no knowledge of the internal structure of the computer, and they therefore keep the stacks well out of sight.

A FORTH programmer has direct access to the stack with full control of the values stored and their manipulation. Most words in FORTH will place values on the stack or expect to find values there. It is essential, therefore, to understand the structure and operation of stacks.

The type of stack used by FORTH is one known as a last-in first-out (LIFO) stack where the value most recently placed on the stack is the one that is most accessible. The action is similar to the pop-up pile of plates that is sometimes seen in restaurants. If a plate is placed on the top of the pile it moves down until the new plate is at counter level. When a plate is removed the pile rises so that the plate which was underneath becomes the new top of the pile. Because of this similarity the structure is also known as a push-down stack.

Here's an illustration of the action of a LIFO stack:

```
TOP-> 27        TOP-> 15        TOP-> -3
      -3              27              19
      19              -3               4
       4              19
                       4

      (a)             (b)             (c)
```

(a) is the initial state of the stack, (b) is the state after the value 15 has been 'pushed' onto the top, and (c) is the final state after the values 15 and then 27 have been 'popped' from the top of the stack.

This is the conventional view of the LIFO stack, in which the top of the stack is always found at the same memory location. The contents of the stack are moved to make room for a new top item, or to replace an item that has been removed.

This would be very slow in operation because of the need to move the entire contents of the stack for each addition or removal. A more efficient method is to leave the contents in the same positions in memory and then change the pointer to the top of the stack when items are added or removed. In FORTH it is convenient to make the stack grow downwards in memory so that the 'top' of the stack is at the lowest memory location used by the stack contents.

The scheme now appears like this:

```
       4               4               4
      19              19              19
      -3              -3       TOP-> -3
TOP-> 27              27
                TOP-> 15

      (a)             (b)             (c)
```

In this manual, unless otherwise stated, we use the conventional wording in all descriptions, so that the 'top' stack item is always the one that is most accessible.

In FORTH the top item of the stack is found at an address given by the variable called SP (stack pointer). Each single-precision item in the stack is stored as a 16-bit number, using two bytes of memory. Thus the top item on the stack is found at address SP, the first item from the top is at SP+2, the second from the top at SP+4, etc. The address of the Nth item from the top of the stack is simply SP+2*N.

FORTH uses two stacks known as the 'computation stack' (sometimes called the 'parameter stack') and the 'return stack'. The programmer will generally use only the computation stack. This stack is used for all arithmetic operations and to transfer information from one FORTH word to another in the execution of a program. In this manual the computation stack will be referred to as 'the stack' unless confusion is possible.

The return stack is mainly used by the system:

a) to store the address of the routine to which control is returned after execution of the current word

b) to store the current loop index in a DO ... LOOP

In addition the return stack may be used by the programmer, with caution, as a temporary store for values from the computation stack. This is one possible method of gaining access to a stack value which is not at the top of the computation stack. In Acornsoft FORTH the computation stack can hold up to 36 single-precision numbers, and the return stack up to 44 addresses or single-precision numbers.

4.2 Arithmetic

Arithmetic in FORTH is performed on integers rather than floating-point numbers. There is no reason why floating-point arithmetic should not be used but this would reduce the operating speed. The integer operations in FORTH are designed to allow fast and accurate arithmetic, without the need to use a floating-point format. It has been said that if you need to use floating-point arithmetic in FORTH, you do not fully understand your application! This is rather an extreme viewpoint but makes the point that there are very few problems that cannot be solved by the use of FORTH's integer operations.

All the arithmetic operators in FORTH expect to find their values on the stack and replace them by their result. A consequence of this is that the numeric values must be placed on the stack before the operator is used.

Thus to add the numbers 2 and 3, the following sequence should be typed at the keyboard:

```
2 3 +
```

where 2 places the number 2 on the stack
3 places the number 3 on the stack
+ removes the top two items from the stack, adds them and places the result on the stack.

The FORTH word <.> removes the top item from the stack and prints it on the display so the following result should be found:

```
2 3 + .
5 OK
```

(Don't forget to press RETURN after the <.>.)

Placing the operator after the numbers on which they act is known as postfix, or reverse-Polish, notation and will be familiar to anyone who has used a Hewlett-Packard calculator. The normal method of

writing arithmetic operations is known as infix notation. One advantage of using postfix notation is that there is no need to use brackets to indicate the order of evaluation as the order is completely unambiguous.

4.3 Single-precision operations

4.3.1 Single-precision numbers

In FORTH, single-precision numbers are of 16 bits (2 bytes) with the most-significant byte at the lower address. Unsigned numbers are in the range 0 to 65535 inclusive. Signed numbers are stored in two's-complement form and are in the range -32768 to +32767 inclusive (see Appendix D).

A number may be placed on the stack simply by typing it at the keyboard and following it by <RETURN> . The top stack item may be removed from the stack and printed on the VDU by the use of the word <.> (dot). This word interprets the number as a signed integer. To show the action of <.> , try the following examples:

```
17 .
17 OK
-21 .
-21 OK
32767 .
32767 OK
32768 .
-32768 OK
```

In single precision, numbers greater than 32767 are interpreted by <.> as being negative.

Numbers greater than 32767 can be printed as unsigned integers using the word <U.> as, for example:

```
32768 U.
32768 OK
```

4.3.2 Single-precision arithmetic

FORTH does not provide an exhaustive set of arithmetic operators, since the needs of different applications vary widely. There is, however, a sufficiently large range of general-purpose operators so that any required operation can be defined by the user.

The following list contains all the single-precision arithmetic operators provided in FORTH. In the stack action the notation is (stack before ... stack after) with the top of the stack to the right, and the items separated by '\':

Table 1

WORD	Stack action	Description
+	(n1\n2 ... sum: n1+n2)	Add
-	(n1\n2 ... difference: n1-n2)	Subtract
*	(n1\n2 ... product: n1*n2)	Multiply
/	(n1\n2 ... quotient: n1/n2)	Divide (integer)
MOD	(n1\n2 ... remainder)	Remainder of n1/n2
/MOD	(n1\n2 ... rem\quotient)	Leave quotient with remainder beneath
*/	(n1\n2\n3 ... n1*n2/n3)	Intermediate product n1*n2 is stored in double precision
*/MOD	(n1\n2\n3 ... rem\n1*n2/n3)	As */ but also leave remainder beneath
NEGATE	(n1 ... -n1)	Change sign
ABS	(n1 ... |n1|)	Absolute value
+-	(n1\n2 ... n3)	Leave, as n3, the value of n1 with the sign of n2
1+	(n1 ... n2)	Add 1 to the top stack item
1-	(n1 ... n2)	Subtract 1 from the top stack item
2+	(n1 ... n2)	Add 2 to the top stack item
2-	(n1 ... n2)	Subtract 2 from the top stack item
2*	(n1 ... n2)	Fast multiply by 2
2/	(n1 ... n2)	Fast divide by 2

The following list gives examples of the use of the first four of these in postfix notation, compared with the corresponding infix form:

Infix	Postfix
2 * 3	2 3 *
9 / 4	9 4 /
2 - (3 * 5)	2 3 5 * -
(2 + 3) * 5	2 3 + 5 * (or 5 2 3 + *)

If you are not familiar with postfix notation you may find it useful to try these examples at the keyboard, using the <.> word to print the result. Try a few examples of your own, using <.> to check if the result is what you expected. If the operators run out of numbers to work on, FORTH will give error message 1 (empty stack), but there will be no indication if too many numbers are left on the stack at the end. Once you have completed a calculation keep using <.> until error message 1 is given, to make sure that there are no numbers unexpectedly remaining.

FORTH can be used to calculate a formula, such as the value of the quadratic expression

3x^2 - 5x + 4

for various values of x. If, for example, x has the value 2 the result could be found as follows:

```
2 2 * 3 * 2 5 * - 4 + .
6 OK
```

This involved typing 2 as the value of x in three places. It can be improved upon by using the stack operators described in the following section.

Try using the other words in the list. Use each one with a range of numerical values, both large and small, positive and negative, to become familiar with their actions.

4.3.3 Single-precision stack operators

There are several words in FORTH which act directly on the numbers on the stack:

Table 2

WORD	Stack action	Description
DROP	(n ...)	Remove the top stack item
DUP	(n ... n\n)	Duplicate the top item
?DUP	(n ... n\n) or (n... n)	Duplicate the top item if it is non-zero, otherwise do nothing
OVER	(n1\n2 ... n1\n2\n1)	Copy the second item over the top item
SWAP	(n1\n2 ... n2\n1)	Exchange the top 2 items
ROT	(n1\n2\n3 ... n2\n3\n1)	Rotate the top 3 items, so that the 3rd item moves to the top

There are also two words which act on numbers further down the stack:

PICK used as n PICK to make a copy, on the top of the stack, of the nth number in the stack, for example:

1 PICK is equivalent to DUP
and 2 PICK is equivalent to OVER

ROLL used as n ROLL to rotate the top n items on the stack, bringing the nth item to the top, for example:

2 ROLL is equivalent to SWAP
and 3 ROLL is equivalent to ROT

These two words are useful to extract a needed number that is some depth below the top of the stack, but are relatively slow in their operation and should be used sparingly.

A further three words act to transfer numbers between the computation stack and the return stack (see section 4.1):

R@ Copy the top item of the return stack to the computation stack (the return stack is unchanged)

>R Transfer the top item of the computation stack to the return stack

R> Transfer the top item of the return stack to the computation stack

Since these last two words modify the contents of the return stack, which is used for system control, they should be used with caution. They should never be executed directly from the keyboard, and, within a definition, they should normally be used only as a pair. This will ensure that the state of the return stack is unchanged between entry and exit when the new definition is later executed. The main use of >R and R> is as a temporary store for the top value on the computation stack when a calculation needs to use the number(s) below it.

The stack contents may be manipulated by the use of several of the stack operators in succession. The following list includes a number of useful stack manipulations which require two words:

Stack before	Stack After	Words
1 2	1 1 2	OVER SWAP
1 2	2 1 2	SWAP OVER
1 2	2 1 1	SWAP DUP
1 2	1 2 1 1	OVER DUP
1 2 3	2 1 3	ROT SWAP
1 2 3	3 2 1	SWAP ROT

It is worthwhile understanding the solutions for this list. If you are not sure about a solution, try it out at the keyboard. Remember that <.> prints the top of the stack first so that the correct response for the first of these is:

```
1 2 OVER SWAP . . .
2 1 1 OK
```

If we now return to the earlier problem of calculating the value of the quadratic function

$3x^2 - 5x + 4$

we can see that it is possible to perform the calculation in such a way that the value of x needs to be typed once only. The following example shows how this can be done. It is broken down into several sections so that the stack contents can be shown at each stage (remember that the top stack item is on the right). The stack contents can be shown at each stage by use of <.S> which displays the stack without altering its contents:

```
                  Stack contents

2                  2
DUP DUP 3          2  2   2   3
* *                2 12
SWAP 5            12  2   5
*                 12 10
- 4                2  4
+                  6
```

The advantage of this is that everything except the value of x can be made into a definition (see chapter 5):

```
: QUADRATIC
     DUP DUP 3 * *
     SWAP 5 * - 4 + ;
```

This can then be used with many values of x. It expects to find the value of x as the top stack item and replaces it by the value of

3x^2 - 5x + 4

4.3.4 Relational and logical operators

Most of the relational operators in FORTH apply a test to the top one or two stack items, returning a true or false value, depending on the result of the test. A false result is indicated by zero and a true result by a non-zero value being left on the stack. As usual the words replace the arguments with the result, in this case a true or false flag.

The relational operators provided in FORTH are listed below; unless otherwise stated they all act on signed numbers:

0= Leave true if the top stack item is zero, otherwise false

0< Leave true if the top stack item is negative

0> Leave true if the top stack item is positive

= Leave true if the top two stack numbers are equal

< Leave true if the second stack item is less than the top item, for example, 2 3 < leaves true

> Leave true if the second stack item is greater than the top item, for example, 3 2 > leaves true

U< As < , but the two numbers are treated as unsigned integers

MAX Leave the larger of the top two numbers on the stack

MIN Leave the smaller of the top two numbers on the stack

Note the difference between < , which compares two signed integers in the range -32768 to +32767, and U<. They act identically on numbers in the range 0 to 32767 but will give different results outside this range.

U> is not supplied initially but can be defined as

```
: U>   SWAP U< ;
```

The logical operations in FORTH usually act on the top two numbers on the stack and are:

AND Leaves a bit-by-bit logical AND of the top two stack numbers

OR Leaves a bit-by-bit logical OR of the top two stack numbers

XOR Leaves a bit-by-bit logical Exclusive-OR of the top two stack numbers

TOGGLE Performs a bit-by-bit Exclusive-OR of the low order byte of the top stack number with the byte whose address is second on the stack; the result is replaced at this address.

One application of XOR is to determine the sign of the product of two numbers and is used in this way for many of the multiplication words in FORTH. A negative number has the most significant bit set to 1. The Exclusive-OR of two numbers of the same sign (i.e. whose most significant bits are both 0 or both 1) will be a number with a zero most significant bit, indicating a positive result. With two numbers of opposite sign the Exclusive-OR will leave a number with most significant bit 1, showing the result to be negative. Note that the value of the result has no meaning in this context. An example of this use of XOR is the definition of MD* in the next section.

The use of TOGGLE is illustrated in the definition of SMUDGE (defined in hexadecimal base):

```
: SMUDGE LAST 20 TOGGLE ;
```

LAST returns the address of the name header of the most recently defined word in the dictionary, and 20 TOGGLE changes the 'smudge' bit in the header to allow or prevent the word from being found in a dictionary search.

4.4 Higher-precision arithmetic

In addition to single-precision, FORTH also supports double-precision arithmetic. Double-precision numbers are stored in 32 bits, using four successive bytes of memory, with the least significant byte at the lowest address. Again, two's-complement form is used, giving a range of values from -2147483648 to +2147483647 inclusive.

Note that because of the way the LIFO stack is implemented in FORTH a double-precision number on the stack has its 2 high-order bytes 'above' the 2 low-order bytes.

A double-precision number may be entered from the keyboard by including a decimal point as the last character of the number. Thus typing in

```
12.
5832478.
```

will place the numbers 12 or 5832478 on the stack, in double-precision form.

There is one purely double-precision arithmetic word:

D+ Double-precision add

In addition there are six mixed-precision operators:

M* (n1\n2 ... nd)
Multiply two signed single-precision numbers to give a signed double-precision product.

U* (u1\u2 ... ud)
As M* , but all numbers are unsigned; this is the multiplication primitive (machine code).

U/ (ud\u1 ... u2\u3)
Divide the double-precision number second on the stack by the single-precision number on the top; a single-precision quotient is left above a single-precision remainder and all numbers are unsigned. This is the division primitive (no error checks are made so this word should be used when a fast division is essential).

U/MOD (ud\u1 ... u2\u3)
As U/ , except that an error message is given if division by zero is attempted. All other division operators use U/MOD and are therefore similarly protected against division by zero.

M/ (nd\n1 ... n2\n3)
As U/MOD, except all numbers are signed.

M/MOD (ud1\u1 ... u2\ud2)
As U/MOD, but leaving a double-precision quotient; all numbers are unsigned.

There are also three sign-changing words for double-precision numbers:

DNEGATE Change the sign of a double-precision number

DABS Leave the absolute value

D+- Apply the sign of the single-precision number on the top of the stack to the double-precision number beneath.

There are four stack operators in double-precision:

2DROP Remove the double-precision top stack item

2DUP Duplicate the double-precision top stack item

2OVER Copy the second double-precision stack item over the top double-precision item

2SWAP Exchange the top two double-precision items

Note that these words can also be used to act on the top two single-precision numbers, i.e. 2DROP is equivalent to DROP DROP and 2DUP is equivalent to OVER OVER.

Finally, there is one relational operation provided for double-precision numbers:

D< Leave true if the second double-precision stack item is less than the top double-precision item, for example, 1. 2. D< leaves true, else leaves false.

As an example of the use of some of the double-precision words consider the definition of the word MD* , which also illustrates the use of XOR to determine the sign of a product, as discussed in section 4.3.4.

The word MD* performs a mixed-precision multiplication which leaves the signed double-precision product nd2 of the signed double-precision number nd1 and the signed single-precision number n:

```
: MD* ( nd1\n ... nd2 )
   2DUP XOR >R       ( keep sign of product )
   ABS >R DABS R>    ( MOD of multiplicand & multiplier )
   DUP ROT * >R      ( high order product )
   U* R> +           ( low product, plus high product )
   R> D+-            ( apply sign to product )
;
```

This word is used in the factorial routine in chapter 11 to allow the calculation of the factorial of numbers up to 12.

5 FORTH definitions

5.1 Introduction

Programs written in FORTH are usually, and more accurately, known as applications. The idea of a program implies the generation of a sequence of actions, distinct from the set of instructions which form the language in which the program is written. In FORTH the distinction between the language and the 'program' is far less clear. The sequence of actions are created as additional words in the FORTH vocabulary, and can be used in exactly the same way as the original words to produce a more complex process. In effect the language is simply being extended.

Many people argue that FORTH should not be described as a language since it contains no rules or structures that cannot be changed by the user. Whether this is a valid argument or not, the fact remains that the great power of FORTH lies in its ability of extension to cope with any situation that may arise.

There are several ways in which new words can be placed in the dictionary. These use defining words, of which the most common are:

: (colon)
CONSTANT
VARIABLE
USER
CREATE
VOCABULARY

The exact formats of words created by each of the above in Acornsoft FORTH are given in more detail in Appendix B. It will, however, be useful to give here a general description of the construction of a typical dictionary entry.

The exact format of a dictionary entry is dependent on the method of implementation and may well be different in different versions of FORTH. Although the method used in Acornsoft FORTH is a common one, it should not be assumed to apply to all other versions of FORTH.

All dictionary entries consist of two parts, the head and the body. The head contains:

a) the name of the entry (variable length)

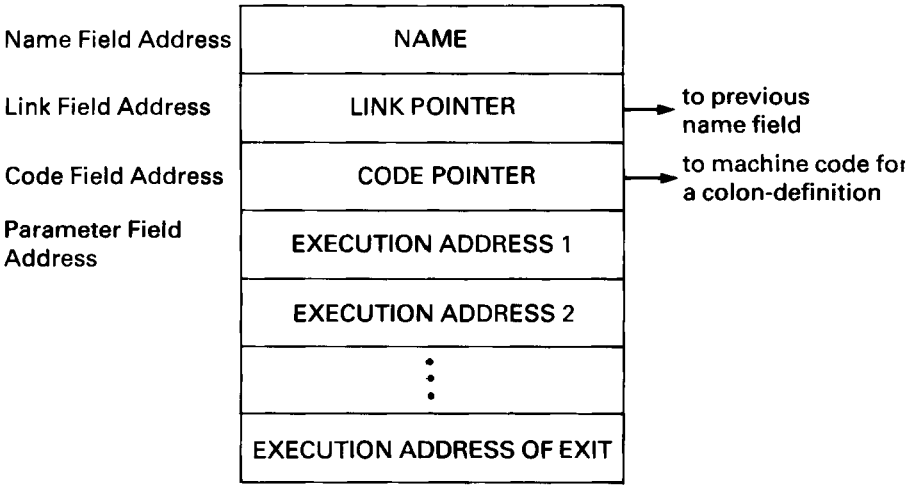
b) a link pointer to the name of the previous entry

c) a code pointer to the machine code used in the execution of the entry.

The starting addresses of these fields are known as the name field address, the link field address, and the code field (or execution) address respectively.

The body of the entry, also known as the parameter field, contains the information which defines the action of that particular entry. The nature of this information differs according to the defining word which was used in its creation. For a colon-definition, for example, the parameter field contains a list of the execution addresses of the words in the definition, terminated by the execution address of EXIT which causes an exit from the word.

The diagram below illustrates these points for a dictionary entry created by a colon-definition:



In this and all such diagrams in this manual, memory addresses increase from top to bottom.

There are a number of words supplied which allow the address of one of these fields to be converted into the address of another. The initial address is expected on the stack, and it is replaced by the new address:

Word	Action
PFA	Convert the name field address to the parameter field address
CFA	Convert the parameter field address to the code field address
LFA	Convert the parameter field address to the link field address
NFA	Convert the parameter field address to the name field address

There are no words provided for the conversion of the code field or link field addresses. This is not normally a problem since any search for a word will usually return either the parameter field address or the name field address. If such a conversion is required it is, however, very simple in Acornsoft FORTH since

```
code field address      = link field address + 2
parameter field address = code field address + 2
```

5.2 Colon-definitions

5.2.1 Form

The colon-definition is the most frequently used way of defining a new action in FORTH and has been used in several of the examples in earlier chapters. The form of a colon-definition is:

```
: NAME ... ;
```

The colon indicates the start of the definition of a new dictionary entry for the word NAME. The NAME is followed by a sequence of actions in terms of FORTH words which have been previously defined, and the definition is terminated by a semicolon <;> . Once defined the word can be executed by typing its NAME at the keyboard. Since colon-definitions are used extensively throughout this manual, no specific examples are given here.

5.2.2 Separating applications

When starting to write a new application it is useful to first make a null definition such as:

```
: TASK ;
```

This is a word which has no function except to mark the start of the application - executing TASK will do

nothing. When the application is no longer required, however, typing

FORGET TASK

will erase TASK and all subsequently defined words, clearing the dictionary for a new application.

5.3 CONSTANT, VARIABLE and USER

5.3.1 CONSTANT

Numerical values may be compiled into a colon-definition as literal values. An alternative is to define them as constants. The sequence

10 CONSTANT LENGTH

will create a dictionary entry for a constant with the name LENGTH and value 10. The entry has the single-precision value of the constant in its parameter field, and the code field contains a pointer to machine code which will copy the value from the parameter field to the stack. Thus, when LENGTH is later executed it will place the value 10 on the stack, just as if the number 10 had itself been typed, so that typing

LENGTH . will give
10 OK

There are two advantages in using constants rather than literal values:

a) When used in a colon-definition LENGTH will compile its two-byte execution address, whereas the literal value requires four bytes - two bytes for the address of the literal handling routine and two bytes for the value. If the value is used many times there is a net saving in memory space, despite the space needed for the definition of LENGTH .

b) If it is necessary to change the value at some later time it is simpler to change the definition

of LENGTH , rather than every occurrence of the literal value.

To change the value of a CONSTANT, the operators <'> (tick) and <!> (store) are used. <'> followed by the name of the CONSTANT leaves its parameter field address on the stack. <!> uses two values from the stack; a numeric value with an address above it. It acts to store the numeric value in the two bytes starting at the address. Thus

```
100 ' LENGTH !
```

changes the value of LENGTH to 100, leaving the stack unchanged. Typing

```
LENGTH .  will now give
100 OK
```

5.3.2 VARIABLE

A dictionary entry for a variable may be created by typing, for example

```
VARIABLE XLENGTH
```

This will create a variable with name XLENGTH and initialised to zero. The dictionary entry will contain the single-precision value of the variable in its parameter field.

The difference between a CONSTANT and a VARIABLE is that, on execution, the CONSTANT places its value on the stack but the VARIABLE places on the stack the address of the memory containing the value. The value of the variable is returned by the <@> (fetch) operator. This takes the address from the stack, replacing it with the two-byte value fetched from the corresponding location. Hence

```
XLENGTH @
```

puts the value of XLENGTH on the stack. This method is chosen so that the storing of a new value in the variable is made simple by, for example:

```
40 XLENGTH !
```

This replaces the old value of XLENGTH by the new value, 40.

The value of a variable can be incremented using +! , for example:

```
1 XLENGTH +!
```

will increment the value of XLENGTH by 1. The increment may be positive or negative, and of any magnitude (within the valid range for single-precision numbers).

The value of a variable can be printed by using <?>. Its action is as one would expect from the definition:

```
: ?   @ . ;
```

5.3.3 USER

This word is provided to allow system modifications, and will not be used in most applications. A user variable may be created by, for example, the sequence:

```
50 USER TERMINAL
```

In this case the value of the variable is not initialised. The above sequence creates a new user variable whose name, TERMINAL , is stored in the dictionary, but the value of the variable will be stored in a separate user variable area. The number (50) is used as an offset in the user area from the value of the user variable pointer, UP . The above sequence will therefore reserve two bytes of memory, 50 bytes above the start of the user area (see the memory map in Appendix C).

The user area is reserved for system variables, many of which are initialised on a cold start of FORTH, and should not be used for variables in an ordinary application. The user variable area provided has its base at address 400 hex with a maximum offset of &C hex (60 decimal). The first unused offset is 32 hex (50 decimal), allowing up to five additional user variables to be defined.

The user variables provided in the system are:

S0	The start address of the computation stack
R0	The start address of the return stack
TIB	The start address of the terminal input buffer
WIDTH	The maximum width of a dictionary entry name, normally 31
WARNING	Error message control, normally 0 (not used by system words)
FENCE	Lower limit for FORGET
DP	Dictionary pointer
VOC-LINK	Points to the most recently defined vocabulary
BLK	If 0, input is from terminal, otherwise from mass storage buffer
>IN	Current offset into the input buffer
OUT	Number of characters output (not used by system words)
SCR	Current tape/disk screen number
OFFSET	Screen offset for mass storage, normally
CONTEXT	Vocabulary pointer for dictionary searches (see section 5.5)
CURRENT	Vocabulary pointer for new definitions (see section 5.5)
STATE	Indicates the compilation state, non-zero when compiling
BASE	Contains current numeric conversion base
DPL	Position of decimal point (not used by system)

CSP Current stack pointer value - used in compiler security

R# Pointer to the editing cursor

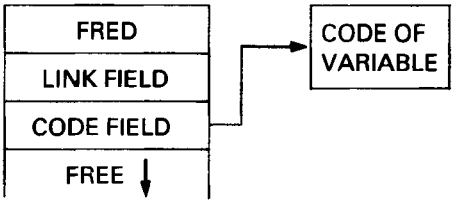
HLD The address of the last converted character during numeric output conversion.

5.4 **CREATE**

CREATE is a word that will produce a dictionary head whose code field contains the address of the machine code used by VARIABLE. Thus when a dictionary entry formed by CREATE is later executed it will, like VARIABLE, leave on the stack the address of the first byte of its parameter area. Unlike VARIABLE, however, no parameter space is allocated. If we

```
CREATE FRED
```

the dictionary entry will appear as shown in the following diagram:



Assuming that FRED is the last word in the dictionary, executing FRED will leave the address of the first free byte in the dictionary (the same address as returned by HERE). We could now ALLOT some space in the dictionary for some values in FRED's parameter area. Thus if, after creating FRED, we then type

```
6 ALLOT
```

we shall reserve six bytes of memory. The address of the first of these bytes is left on the stack whenever FRED is executed. This is a useful way of defining a simple form of array. The sequence

```
CREATE FRED          6 ALLOT
```

as used above produces the equivalent of an array to hold three single-precision numbers. The addresses of the first, second and third of these numbers are left on the stack by

```
FRED
FRED 2+
FRED 4 +
```

respectively. Values may be placed at these three addresses, by the use of <!>, or read back by the use of <@>. The values are not initalised when space is reserved by the use of ALLOT.

We may often want to intialise the values at the time of allocating space in the parameter area. For example, the word VARIABLE not only allocates a two-byte parameter area, but it also initialises it to contain zero. A definition of VARIABLE can be written in terms of CREATE as follows:

```
: VARIABLE   CREATE   0 , ;
```

and has the same action as the version provided.

It is when CREATE executes, that it accepts the name of the dictionary entry it is to make. This means that the name should not be included in a colon-definition using CREATE, since in this situation CREATE is being compiled and does not execute. The name is required when the definition including CREATE (for example VARIABLE) is executed. Thus, when we use VARIABLE as

```
VARIABLE JIM
```

the dictionary head for JIM is formed by CREATE and two bytes of parameter area are allocated and set to zero by <0 ,>.

As a further example we can form a new defining word, 2VARIABLE which will create a variable capable of holding a double-precision value. The definition is simply:

```
: 2VARIABLE   CREATE  0 ,  0 , ;
```

When a double-precision variable is created by, for example

```
2VARIABLE LOTS
```

the execution of CREATE produces the dictionary head for LOTS and then four bytes of dictionary space are reserved and set to zero.

The efficient use of double-precision variables requires the further definitions of 2@ and 2!, the double-precision versions of @ and ! respectively. These are most effectively defined as machine code routines and their definitions are given in section 10.7 on the FORTH Assembler.

The word 2VARIABLE generates the name header and reserves the first four bytes of the parameter area. A single-byte form of variable is not so easy to define in terms of VARIABLE but is simple in terms of CREATE. It is, of course

```
: CVARIABLE   CREATE 0 C, ;
```

where only one byte of parameter area is allocated. A single-byte variable can then be defined like this:

```
CVARIABLE TINY
```

Fetching and storing numbers in the case of single-byte variables must not, of course, use @ or ! since these work on two-byte numbers. The single-byte equivalents C@ and C! must be used instead. The value in TINY can be put on the stack by

```
TINY C@
```

and a value placed in TINY by, for example

```
15 TINY C!
```

If you use single-byte or double-precision variables it is your responsibility to ensure that you use the correct versions of the fetch and store operations. Using <!> on a single-byte variable, for example, will alter part of the following dictionary entry. In the best case it will mean that the following word cannot

be found; in the worst case it can result in a system crash that will require the whole system to be reloaded.

So far all the examples have used the fact that a word defined using CREATE will leave the address of the first byte of the parameter area on the stack. CREATE becomes more useful if this action can be changed. In fact all the defining words use CREATE to produce the name headers but then change the contents of the code field address of the new word so that it points to a different machine code sequence. As an example of this we can examine the definition of a word which will allow short machine code routines to be written without the use of an assembler.

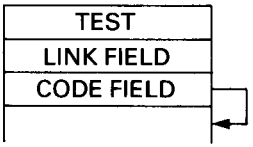
The code field of a machine code primitive must contain the address of the start of the machine code itself. This code is normally placed in the parameter area of the word. In Acornsoft FORTH this area starts with the byte immediately following the code field. What is needed, therefore, is to rewrite the contents of the code field so that it contains the address of the following byte. This can be done with the following definition:

```
: CODE  CREATE  HERE  -2  ALLOT  , ;
```

When code is executed as for example,

```
CODE TEST
```

it generates a normal dictionary header for TEST as described earlier. Then HERE returns the address of the first free byte in the dictionary (which will be the first byte of the parameter area of TEST). The words -2 ALLOT step back over the two bytes of the code field and <,> then rewrites the code field to contain the address returned by HERE. The structure of the entry for TEST will, at this stage, be as shown in the following diagram:



So far we have not supplied any machine code for TEST to use. Executing TEST at this stage is liable to be fatal for the system.

Suppose, as a simple example, we want TEST to drop three items from the stack. There is a section of machine code within the system which will drop two items. In our Assembler the start of this code is labelled POPTWO and it is, as explained in chapter 10, also a valid exit from a machine code routine. The address of the start of POPTWO is also given there; it is &1DD +ORIGIN.

The routine we wish to write need, therefore remove only one item (two bytes) from the stack before jumping to POPTWO. The address of the most accessible stack item is held in the 6502 X-register, so to drop this 16-bit value we need to increment the X-register by two. The code to increment the X-register is &E8. The sequence (in hex)

```
E8  C,  E8  C,
```

will therefore compile the code to drop one stack item.

All that is now required to complete the entry is to compile the jump to POPTWO. The code for a jump is &4C followed by the address (low byte first). This is done by the sequence (again in hex)

```
4C  C,  &1DD +ORIGIN ,
```

Since <,> puts the low (least significant) byte into the dictionary first, the two bytes of the address are put in the correct order automatically.

The full sequence to generate this definition is

```
HEX
   CODE  TEST
    E8  C,  E8  C,
    4C  C,
    1DD +ORIGIN  ,
DECIMAL
```

and produces code which, in conventional assembly language, would be written as

```
INX
INX
JMP POPTWO
```

The word TEST may now be executed (provided the stack has three numbers on it which can be dropped).

This way of writing machine code requires a knowledge of the machine language operation codes and of all addresses used. It is also very tedious and does not give very readable definitions! It is, however, very useful when you want to include one or two short machine code routines without first having to load the Assembler. A further example is given in chapter 8 where a hand-assembled machine code version of the Editor word MATCH is given. This is much faster than the high-level versions provided in the Editor screens. Before using this method to produce your own machine code routines it would be advisable to read chapter 10 which gives essential information on the rules to be obeyed when writing machine code in FORTH.

There is one further major use of CREATE which involves the CREATE ... DOES> pair. This gives an extremely powerful technique for generating new data structures and is the entire subject of chapter 9.

5.5 VOCABULARY

A VOCABULARY is a subset of the dictionary, and the VOCABULARY structure of FORTH is the means by which the order of a dictionary search is controlled. Normally, if an existing word is redefined, a dictionary search will find only the latest definition. The old word will still be used in earlier definitions, but only the most recent version will be available for new definitions. The following example illustrates this:

```
: QUOTE   ." THIS IS A LITERAL STRING" ;
: PRINT   QUOTE CR ;
```

Executing PRINT will type out the message of QUOTE. The

word CR performs a carriage return and line feed on the display. If QUOTE is redefined as

```
: QUOTE   ." A DIFFERENT MESSAGE";
```

MSG # 4 will be given, warning that QUOTE is already in the dictionary. The new definition of QUOTE can then be made and used in:

```
: NEWPRINT   QUOTE CR ;
```

Now, NEWPRINT will type out the new message. However, PRINT will still respond with the original message. Executing a VLIST will show that there are now two entries for QUOTE .

Typing FORGET QUOTE and then executing a VLIST will show that the second QUOTE (and NEWPRINT) will have disappeared from the dictionary but the earlier definition of QUOTE will still remain. It is necessary to type FORGET QUOTE a second time to remove both versions from the dictionary. If the two versions of QUOTE are defined in different vocabularies it is possible, by changing the dictionary search order, to select which version will be used in further definitions.

The order of search and the determination of which vocabulary a new definition will be entered in is controlled by the two user variables CONTEXT and CURRENT , each of which points to the most recently defined word in a vocabulary. CONTEXT points to the VOCABULARY that is first searched by a dictionary search, in either a VLIST or a search for a word to compile into a colon-definition. CURRENT points to the VOCABULARY into which new definitions are placed. These two are usually, but not necessarily, the same.

A new vocabulary is created by, for example:

```
VOCABULARY TEST-VOC IMMEDIATE
```

This creates a new vocabulary with the name TEST-VOC (by convention all VOCABULARY words are IMMEDIATE). This vocabulary can be made the CONTEXT vocabulary by executing TEST-VOC . The CURRENT vocabulary remains as that from which the new vocabulary was created (this

would normally be the FORTH vocabulary). Thus, after creating and then executing TEST-VOC , a dictionary search will start in TEST-VOC , but new definitions will still be entered into the old vocabulary. The process of making a new definition automatically sets CONTEXT to be equal to CURRENT so that after the sequence (assuming the initial vocabulary to be FORTH):

```
VOCABULARY TEST-VOC IMMEDIATE ( create TEST-VOC )
TEST-VOC                      ( set CONTEXT to TEST-VOC )
: NAME ;                      ( define NAME )
```

the word NAME will be in the FORTH vocabulary, which will now also be the CONTEXT vocabulary.

The word DEFINITIONS sets the CURRENT vocabulary to be the same as the CONTEXT vocabulary, so the sequence

```
VOCABULARY TEST-VOC IMMEDIATE
TEST-VOC DEFINITIONS
: NAME ;
```

will place the definition of NAME in the TEST-VOC vocabulary which will then be the CURRENT (and CONTEXT) vocabulary. Note that TEST-VOC itself is created in the FORTH vocabulary. Typing

```
FORGET TEST-VOC
```

will still work correctly since the vocabulary TEST-VOC links to FORTH. Each vocabulary eventually links back into the 'parent' vocabulary (the CURRENT vocabulary at the time of its creation). It is normal to ensure that each new VOCABULARY links directly into the FORTH vocabulary, because, although it is possible to chain vocabularies, this can result in a complicated and confusing search sequence and is therefore not recommended.

In addition to separating the words of one application from those of another, one of the main uses of the VOCABULARY structure, is to allow the use of the same word to represent several different actions and still be able to find earlier definitions.

If we use the earlier example and type the following:

```
FORTH DEFINITIONS (make FORTH the current vocabulary)
: QUOTE    ." THIS IS A LITERAL STRING " ;
VOCABULARY TEST-VOC IMMEDIATE
TEST-VOC DEFINITIONS
: QUOTE    ." A DIFFERENT MESSAGE " ;
```

the warning of a duplicate entry will still be given but now the first version is in the FORTH vocabulary and the second version is in the TEST-VOC vocabulary. Typing

```
FORTH QUOTE          will give
THIS IS A LITERAL STRING OK
```

whereas

```
TEST-VOC QUOTE       will give
A DIFFERENT MESSAGE OK
```

At this point VLIST will start in the vocabulary TEST-VOC .

The ability to use the same word for two different actions is used in the EDITOR vocabulary where, for example, the word I is used to insert a line of edited text. In the FORTH vocabulary the word I is used to return the loop index from within a DO ... LOOP (see section 6.3).

5.6 The compilation of a colon-definition

5.6.1 Normal action

When a FORTH word is typed at the keyboard it is usually executed as soon as the RETURN key is pressed. In the compilation mode, during the creation of a colon-definition, the response is quite different. The word is not executed but its execution address is added to the list of addresses in the dictionary entry being constructed. This continues until the terminating semi-colon is found, whereupon normal execution is resumed.

5.6.2 IMMEDIATE words

It is often necessary to define a word that will execute even in the compilation mode. Examples include the conditional words IF , ELSE , THEN , which must execute in order to calculate the offsets for their branches, and the vocabulary words e.g. FORTH , EDITOR , which allow the changing of the CONTEXT vocabulary to include words from other vocabularies in the current definition.

The response to these words is identical in both execution and compilation modes - they are always executed. They are classed as IMMEDIATE words and are made so by including the word IMMEDIATE at the end of their definitions, for example

```
: DO-IT-NOW   CR ." I AM EXECUTING" CR ; IMMEDIATE
```

If this is now used in another definition, for example

```
: TEST
    CR ." I HAVE BEEN COMPILED" CR
    DO-IT-NOW ;
```

the message I AM EXECUTING will appear as soon as the RETURN key is pressed after typing in DO-IT-NOW. Executing TEST will produce the message I HAVE BEEN COMPILED, but not the message of DO-IT-NOW, since this was executed and not compiled. Note that any type of word, not just colon-definitions, may be made IMMEDIATE.

5.6.3 Making a normal word IMMEDIATE

It may be necessary, during the formation of a colon-definition, to execute one or more words which would normally be compiled. This is useful, for example, for the calculation of a numerical value, or to change the numeric base, during the compilation of a colon-definition, and is accomplished by the use of the words [, which is itself IMMEDIATE , and]. The action of [is to terminate compilation and enter the execution mode, while] has the opposite effect. They are usually, but not necessarily, used as a pair (see section 9.6.2). Their use in a colon-definition is:

```
: NAME ... these words are compiled as usual ...
     [ ... these words are executed ... ]
       ... compilation continues ... ;
```

The use of [and] can clarify the meaning of source code, for example,

```
... [ #BUF @ BUFSZ * ] LITERAL ...
```

conveys more meaning than

```
... 2056 ...
```

though both compile identically.

5.6.4 Compiler security

It is important to ensure that the sequence of words involving [and], or any IMMEDIATE word, does not change the number of items on the computation stack. In the above example the number £BUF @ BUFSZ * (i.e. 2056) is added to the stack and then removed by LITERAL. Part of the compiler security system is to check that the number of items on the stack is unchanged across a colon-definition. If the actions involving [and] have the net effect of adding or removing stack items, an error message will be given when <;> is reached and the definition will be left in an incomplete form.

It is also important to realise that the words IF , ELSE , DO , BEGIN leave a number on the stack, to be checked and removed by the corresponding THEN , LOOP , UNTIL etc. If these numbers are changed or removed by any IMMEDIATE action the compiler security system will again give an error message and the definition will be incorrectly terminated.

5.6.5 Forcing the compilation of IMMEDIATE words

It may occasionally be necessary to compile a word that is marked as IMMEDIATE . You may wish, for example, to delay a change of CONTEXT vocabulary until a word is executed rather than the change taking place during the definition of the word, as normal. Each IMMEDIATE word can be forced to compile by preceeding it with the word [COMPILE] .

As an example, we can compile the IMMEDIATE definition DO-IT-NOW of section 5.6.2:

```
: DO-IT-LATER
    [COMPILE] DO-IT-NOW ;
```

The message of DO-IT-NOW is no longer displayed during the definition and will only appear when DO-IT-LATER is executed.

The word [COMPILE] is, as indicated by the square brackets, an IMMEDIATE word and not itself compiled. Its only action is to force compilation of the following word. You may however, if necessary, force its compilation by:

```
... [COMPILE] [COMPILE] ...
```

5.6.6 Compiling into another word

The word COMPILE (without square brackets) is not an IMMEDIATE word and will therefore be compiled as normal into a colon-definition. It is used in the form:

```
: NAME ... COMPILE WORD ... ;
```

In this sequence COMPILE and WORD are both compiled into the dictionary entry for NAME . When NAME is executed, COMPILE will act to place the execution address of WORD into the next free dictionary space i.e. to compile it. WORD is not executed during the execution of NAME .

5.6.7 An example

A useful example of the use of IMMEDIATE words, [COMPILE] and COMPILE occurs in the literal numeric handler of FORTH.

The word LITERAL is used by the keyboard interpreter to compile a literal numeric value into a colon-definition:

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

In the execution of LITERAL

STATE @	returns a true value if compiling and a false value if the keyboard input is being executed
IF	tests this value and skips to THEN on a false result (see chapter 6), so LITERAL has no action in execution mode
COMPILE LIT	if compiling a new colon-definition, compiles the literal handler LIT into the new definition
	adds the numeric value from the top of the stack into the definition

The new definition will now include the sequence

... LIT (value) ...

and when it is later executed, LIT will act to put the following value onto the stack, as required.

Note that LITERAL has to be an IMMEDIATE word so that it will execute whenever a numeric value is to be included within a definition. The word DLITERAL is used by the keyboard interpreter to compile a double-precision numeric value into a definition, and uses LITERAL twice. LITERAL must, however, be compiled into the definition of DLITERAL by the use of [COMPILE] .

```
: DLITERAL
    STATE @
       IF SWAP
            [COMPILE] LITERAL (low part)
            [COMPILE] LITERAL (high part)
       THEN
; IMMEDIATE
```

Like LITERAL , DLITERAL has no action in execution

mode. A double-precision value is stored on the stack with its high-order part above the low-order part so that on execution of DLITERAL in compilation mode,

SWAP places the low-order part above the high-order part

LITERAL compiles the low-order part into the definition

LITERAL then compiles the high-order part

The new definition will now contain the sequence:

... LIT (low part) LIT (high part) ...

and when it is later executed the double-precision number will be pushed onto the stack with its two parts in the correct order.

6 Conditionals and loops

6.1 Introduction

FORTH is a highly-structured language, in which all transfers of control are accomplished without the use of GOTO statements. This requires the writing of applications in a modular style, where each module has only one entry and one exit point. Although very different from writing a BASIC program, it is not too difficult since it is almost impossible to write a FORTH application in any other way. Once you have grasped the underlying ideas, writing structured programs becomes natural and soon you begin to wonder how you ever managed to do anything in an unstructured language.

6.2 Conditional branches

The simplest conditional branch in FORTH uses

... IF ... THEN ...

These words may look familiar, but in FORTH their actions are somewhat unusual. In BASIC the action of a statement such as

IF X=2 THEN GOTO 2137

is interpreted as:

IF this test is true THEN do this statement, otherwise go on to the next line (and just what does line number 2137 do, anyway?).

Just like operators, the IF in FORTH is postfix, so the value to be tested comes before the IF . The IF ... THEN structure in FORTH is interpreted as:

IF the result of the test was true, do this sequence, otherwise skip it, and in either case THEN continue with the following sequence.

Some FORTH systems attempt to make this clearer by using IFTRUE to replace IF , and ENDIF to replace THEN.

One restriction in FORTH is that the branch words, and the loop words of the following sections, can only be used inside a colon-definition and may not be directly executed from the keyboard. The way in which they are used is:

```
: EXAMPLE
    ?TEST IF DO-THIS THEN CONTINUE ;
```

A FORTH definition with the same function as the BASIC statement in the earlier example could appear as:

```
: =2?  2 = IF ." VALUE WAS TWO " THEN ;
```

Where has X gone? The sequence 2 = will test the top number on the stack and leave a true (non-zero) result if it were equal to 2 and a false (zero) result otherwise. In FORTH it is often not necessary to use an explicitly-named variable; as long as the appropriate value is placed on the stack at the right time it doesn't matter how it got there.

In the above example the value to be tested can be entered directly from the keyboard, for example:

```
2 =2?
VALUE WAS TWO OK
3 =2?
OK
```

Note the use of the word WAS. The general rule in FORTH is that words remove from the stack the numbers they use. The word <=> will remove the 2, placed on the stack by =2? , and the number being tested, leaving only a true/false flag. The flag is removed by IF . If the value being tested is needed again DUP must be used before =2? .

In many cases you may wish to execute one sequence if

the test is true and a different sequence if the test is false. The sequence

IF ... ELSE ... THEN

will, if the result of the test was true, execute the words between IF and ELSE and skip to THEN . If the result of the test was false it will skip to ELSE, and execute the words between ELSE and THEN . The words (if any) after THEN will be executed in either case.

To illustrate this we can

```
FORGET =2?
OK
```

and redefine it as follows:

```
: =2?
    2 = IF    ." VALUE WAS TWO "
        ELSE ." VALUE WAS NOT TWO "
        THEN ;
```

(The layout is irrelevant - type it any way you like - as long as you do not press the RETURN key in the middle of a word, or in the middle of ." ..." , all will be well. The above layout looks good and makes the structure clearer.)

Trying the new version gives, for example,

```
2 =2?
VALUE WAS TWO OK
3 =2?
VALUE WAS NOT TWO OK
```

Incidentally, the definition can be rewritten to use less memory. FORGET the old definition and replace it by:

```
: =2?
    ." VALUE WAS "
    2 - IF ." NOT " THEN
    ." TWO" ;
```

This has exactly the same effect as the earlier, longer, version (remember that a true result may be represented by any non-zero value).

Often, the number being tested for truth by IF may be needed for a calculation in the IF ... THEN sequence, but not needed otherwise. One way of doing this is:

```
DUP IF ... ELSE DROP THEN ...
```

DUP duplicates the number to be tested, and the copy is discarded by DROP if it is false.

A neater solution is to use ?DUP , which will only duplicate a number if it is non-zero. Thus the sequence is equivalent to

```
?DUP IF ... THEN ...
```

If the number is zero it is not duplicated and there is obviously no need then to DROP it, since the only copy is removed by IF .

The IF ... THEN and IF ... ELSE ... THEN forms may be nested to any required depth, provided that the nested structure lies completely within the outer structure. The following are examples of valid nestings (the nested structure is underlined for clarity):

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

Too many levels of nesting, however, make the definition hard to understand and should be avoided. It is much clearer if a long definition with many levels of nesting is split up into a number of short definitions. The nested structure of the second case given above is much clearer if it is written as:

```
: NESTED-IF
            IF   ... THEN ;

: OUTER-IF
            IF   ... NESTED-IF ...
            ELSE ...
            THEN ... ;
```

A general rule in FORTH is 'long definitions = bad definitions' - keep them short!

6.3 Definite loops

A loop whose number of repetitions is known before entry will use the DO ... LOOP structure. DO takes two values from the stack, the start index and the loop limit. If we take as an example the definition:

```
: TENCOUNT   10 0 DO I . LOOP ;
```

then executing TENCOUNT will give:

```
TENCOUNT
0 1 2 3 4 5 6 7 8 9 OK
```

The word I , which should only be used within a loop, places the current loop index on the stack. Note that in this example, I is immediately followed by <.> which types (and removes from the stack) the value left by I .

The loop index is post-incremented; in other words the increment occurs in LOOP , after the body of the loop is executed. The loop will terminate when the (incremented) loop index equals or exceeds the loop limit. This has two important consequences:

a) Regardless of the value of the loop limit and the starting index, the body of the loop will be executed at least once.

b) The last execution of the loop body will be with an index which is one less than the loop limit. Thus in the example, the loop was executed ten times but the last execution was with a loop index of 9.

Programming errors that cause a net change in the number of items on the stack inside the body of the loop can lead to a stack overflow resulting in a system crash. One of the easiest ways of crashing the system is to execute the following definition (not recommended):

```
: CRASH 100 0 DO I LOOP ;
```

Small stack overflows will result in error message 7 (stack full). Larger overflows may cause message 1 (stack empty) if the stack contents extend beyond the bottom of page zero and 'wrap round' to the top. Even large stack overflows, such as that in the above example, will not cause the loss of the system. Pressing the BREAK key and restarting at the warm entry point as described in chapter 3, should cause a successful recovery.

As long as there are two values on the stack for DO ... LOOP , it does not matter how they got there. In the examples so far the values have been placed on the stack within the definition. They may, however, be entered directly from the keyboard:

```
: DELAYS 0 DO LOOP ;
```

In this example, only the starting index (zero) is put on the stack within the definition. The loop limit may be entered from the keyboard so that

```
10000 DELAYS
(pause) OK

30000 DELAYS
(longer pause) OK
```

can be used to give a variable length delay.

```
15500 DELAYS
```

will give approximately a one-second delay.

Of course both values may be entered from the keyboard. The definition

```
: COUNTER DO I . LOOP ;
```

will allow the following:

```
8 0 COUNTER
0 1 2 3 4 5 6 7 OK
52 47 COUNTER
47 48 49 50 51 OK
```

and so on.

It is awkward to have to remember to type the values in reverse order and to have to add one to the last required value. The routine can be made more 'user friendly' by defining

```
: COUNTS   1+ SWAP DO I . LOOP ;
```

This can then be used in a more sensible way:

```
1 7 COUNTS
1 2 3 4 5 6 7 OK
10 14 COUNTS
10 11 12 13 14 OK
```

It can even be used to count in hexadecimal:

```
10 16 HEX COUNTS
A B C D E F 10 OK
```

Remember to change the base back to DECIMAL .

For increment values other than 1 the DO ... +LOOP structure is used. +LOOP expects to find its incremental value on the stack. This value can be given in the definition as in the following example:

```
: 3-COUNT 1+ SWAP DO I . 3 +LOOP ;

0 15 3-COUNT
0 3 6 9 12 15 OK
```

The increment could, if you feel confident, be calculated within the loop to give a variable increment, for example:

```
: SEQUENCE   1+ SWAP DO I DUP . 2* +LOOP ;

1 27 SEQUENCE
1 3 9 27 OK
```

By the use of a negative increment, the loop can count backwards:

```
: BACKWARDS   DO I . -1 +LOOP ;

0 6 BACKWARDS
6 5 4 3 2 1 0 OK
```

Note that with a negative increment the loop terminates when the incremented index passes the loop limit. This is an exception to the usual termination of a loop and is a requirement of the FORTH-79 standard for historical reasons.

Loops may be nested, provided that the inner loop is completely enclosed by the outer one. This is illustrated by the following example, which is laid out in such a way that the nested structure is made clear:

```
: 100-COUNT
    10 0 DO I 10 *
            10 0 DO DUP I + . LOOP
            DROP CR
         LOOP ;
```

The word I is used twice, once in each loop. The first I will therefore leave the outer loop index and the second I will leave that of the inner loop.

The sequence I 10 * leaves on the stack the tens value for the count. In the inner loop this is first duplicated, so that it remains available for the next time round the loop. The inner loop index (the unit's value) is then added to it and the resulting value is printed. On leaving the inner loop the old tens value is dropped, and a CR ensures that the final display is 'tidy'. Executing 100-COUNT will then display 100 integers, from 0 to 99 inclusive.

There is no reason why loops and conditional branches should not be nested, again provided that the inner

structure is completely enclosed by the outer. Definitions in which the structures overlap, such as

DO ... IF LOOP ... THEN

are not allowed.

In the following example it is assumed that the word ALIST has been previously defined to leave on the stack the starting address under the number of single-precision (16-bit) items in a table of values (the method of doing this is discussed in section 9.4.3.). The word LOOK-UP will search the table for a particular value, initially on the stack, and will leave either

a) the item offset within the table under a true flag, if the item is found, or

b) only a false flag if the item is not found in the table.

It is used in the form:

n ALIST LOOK-UP

where n is the value to be found.

The definition may be tested without the need to define ALIST since all it needs is three values on the stack. It can be used to search any region of memory for a particular (16-bit) word by giving it the value to find, a starting address and the length, in words (2-byte units), of the region to be searched. For example,

5678 0 +ORIGIN 4096 LOOK-UP

will search the first 8K of the dictionary for the value 5678 (and fail to find it).

1234 0 500 LOOK-UP

should find the value with an offset of 3077.

```
: LOOK-UP      ( val\addr\count ... offset\1 ) ( found )
               ( val\addr\count ... 0 ) ( not found )
   0 DO        ( loop limit is count )
      2DUP     ( value under base addr )
      I 2* +   ( add byte offset to addr )
      @ =      ( table item = val? )
      IF       ( equal )
       I 0 LEAVE ( item offset under 0, and exit loop )
      THEN
   LOOP        ( top of stack is 0 if found, or )
               ( addr [assumed non-zero] if not )
   IF          ( not found )
      DROP 0   ( DROP value, leave 0)
   ELSE        ( found )
      ROT ROT 2DROP 1   ( DROP val and addr, leave 1 )
   THEN ;
```

The word LEAVE, when executed, causes an exit from the loop. The exit is not immediate but will occur when LOOP (or +LOOP) is next encountered. The action of LEAVE is to change the loop limit to be equal to the current value of the loop index, which is not changed. The words, if any, between LEAVE and LOOP will be executed once before exiting the loop.

An interesting variation on LOOK-UP is the following alternative definition. It has exactly the same effect, but searches the region of memory from high addresses to low. There is, however, one word fewer to be executed in the loop for an unsuccessful match, so it is slightly faster:

```
: LOOK-UP      ( val\addr\count ... offset\1 ) ( found )
               ( val\addr\count ... 0 ) ( not found )
   OVER >R     ( save addr for later )
   2* OVER +   ( calculate endaddr in table )
   0 ROT ROT   ( put 0 under addr and endaddr )
   DO
    OVER I @ = ( table item = val? )
    IF         ( found )
      DROP     ( DROP the 0 )
      I        ( table address [assumed non-zero] )
      I NEGATE ( and its complement for +LOOP )
    ELSE
      -2       ( increment for +LOOP )
    THEN
   +LOOP
```

```
SWAP DROP   ( val )
R>          ( recover addr )
OVER        ( copy of either 0 or table address )
IF          ( not the 0 )
 - 2/ 1     ( calculate table offset under 1 )
ELSE
 DROP       ( DROP addr but leave the 0 )
THEN ;
```

An unsuccessful search of 8192 bytes of memory, i.e. 4096 comparisons, takes about 2.5 seconds using the first method and about 2.2 seconds by the second method.

The method of leaving the loop on a successful match does not use LEAVE. For an unsuccessful match the loop index is decremented by -2, but on a successful match the complement of the loop index is left for +LOOP. This will guarantee that the increment will cause the loop limit to be exceeded, thus terminating the loop.

It is a useful exercise to try to modify either (or both) of these routines to search for a particular byte (8-bit) in memory. Not too many alterations are needed. A further useful modification would be to change the input stack requirements from val\addr\count to val\addr\endaddr.

The word J can be used to leave, in an inner loop, the loop index of the outer loop. Its action is demonstrated by the following definition:

```
: JTEST
    CR ." J ( OUTER ) I ( INNER )"
    CR 7 3  DO
            3 0 DO    CR J .
                      10 SPACES I .
                  LOOP CR
            LOOP ;
```

Executing JTEST produces a table of the inner and outer indices.

Since the loop index and limit are kept on the return stack, which is also used to keep track of the level of nesting of colon-definitions, I and J cannot be

used in a separate definition inside a DO ... LOOP. The following sequence, for example, will not have the required effect:

```
: INNER   3 0 DO J . I . LOOP ;

: OUTER   CR 7 3 DO INNER CR LOOP ;
```

Executing OUTER will give the correct operation for I, but the value of J will not be what was intended.

If DO ... LOOPs are nested to a depth of 3 then, from the innermost loop:

I leaves the index of the inner loop

J leaves the index of the middle loop

K leaves the index of the outer loop

The word K is not provided in the nucleus dictionary. Its definition is:

```
: K   RP@ 9 + @ ;
```

The idea can be extended to further levels by defining L , M , etc. For each extra level the definition is similar to that of K , except that the number to be added is increased by 4, for example:

```
: L   RP@ 13 + @ ;
```

6.4 Indefinite loops

There are three forms of indefinite loop:

BEGIN ... AGAIN

BEGIN ... UNTIL

BEGIN ... WHILE ... REPEAT

In each case BEGIN marks the start of the sequence of words to be repeated.

The word AGAIN causes a branch back to the

corresponding BEGIN so that the intervening words are repeated endlessly. This form of loop was used in the definition of STARS in chapter 3 to create an application whose execution could only be terminated by pressing the BREAK or ESCAPE key. A BEGIN ... AGAIN loop is used only if it is to initiate a repetitive sequence of actions which are to continue until the machine is switched off. It is useful for turnkey applications where the user is not expected to know, or wish to alter, the method of operation.

In the FORTH system it is, for example, used for the keyboard interpreter which interprets all input to the computer. While FORTH is in action all operations are at a more or less deep level of nesting from within the keyboard interpreter, to which control must ultimately return (when OK is displayed).

The remaining two forms may be terminated by the result of a test made within the loop.

In the case of BEGIN ... UNTIL , the word UNTIL tests the top item on the stack. If this value is false (zero) a branch will occur to the corresponding BEGIN . If the value is true (non-zero) the loop will be left and the words following UNTIL will be executed. Consider the following definition:

```
: PAUSE
    CR BEGIN ?TAB UNTIL
    ." TAB KEY PRESSED" CR ;
```

This will loop until the TAB key is pressed since ?TAB leaves a false value on the stack unless the TAB key is pressed, then it leaves a true value.

If we define

```
0 CONSTANT HELL-FREEZES-OVER
```

then the definition

```
: WAIT
    BEGIN HELL-FREEZES-OVER UNTIL ;
```

will, on execution, wait until the condition is satisfied!

On a slightly more useful level, the definitions

```
: GCD    ( n1\n2 ... gcd )
    BEGIN
         SWAP OVER MOD ?DUP 0=
    UNTIL ·

: G-C-D  ( n1\n2 ... )
    GCD CR ." THE G-C-D IS " . ;
```

will calculate and display the greatest common divisor of the numbers n1 and n2. For example,

```
15 25 G-C-D
```

will respond:

```
THE G-C-D IS 5 OK
```

Note how in these definitions, the calculation of the result and the display routines are placed in separate words. This means that GCD can be used by itself as part of a longer calculation where the value is not required to be printed, and we are saved the trouble of rewriting its definition.When the value is to be displayed, however, the word G-C-D can be used, as in the above example.

The BEGIN ... WHILE ... REPEAT structure will terminate as the result of a test which should be made immediately before WHILE . This word expects a true/false flag on the stack but in this case will terminate the loop when the value is false. If the value is true, execution will continue with the following words up to REPEAT which, like AGAIN , causes an unconditional branch back to the corresponding BEGIN . For a false value the words between WHILE and REPEAT are skipped, the loop terminates, and then the words after REPEAT are executed. An example of the use of BEGIN ... WHILE ... REPEAT is the INPUT routine in section 7.1.3.

Indefinite loops may, of course, be nested with any of the other structures to any reasonable depth, provided that the nested routine is totally enclosed within the outer structure.

7 The Ins and Outs of FORTH

This chapter deals with the methods of controlling input and output in FORTH. So far we have met two output operations, <.> and <."> , which display a number (in the current numeric base) and a literal character string respectively. All input, whether text or numeric, has used the keyboard interpreter. The following sections give specific methods of input and output.

7.1 Input

7.1.1 Character input

A single character may be input by the use of KEY . This word waits for a key to be pressed and leaves the corresponding ASCII code on the stack. The definition

```
: SHOWASCII KEY . ;
```

will accept a character from the keyboard and display its ASCII code in the current numeric base.

The sequence

```
KEY DROP
```

is a useful way of causing a wait until any key is pressed.

7.1.2 Text input

The word QUERY will accept any sequence of characters typed on the keyboard, up to a limit of 80 characters, or until RETURN is pressed. The characters are stored in the terminal input buffer, followed by one or more zeroes.

Text may be transferred from the input buffer to the word buffer, by the use of WORD . This expects to find a delimiter character, usually a space (ASCII code 32) on the stack.

Leading delimiter characters are ignored, and text up to the next delimiter is transferred to a 256-byte buffer whose start address, given by WBFR, is left on the stack. (Those who have used other versions of FORTH should note that the text is not transferred to HERE.) The first byte of the buffer contains the length of the following text string. The end-of-text delimiter found in the input is stored at the end of the string but is not included in the string length byte.

The text string may then be moved to another region of memory or further manipulated, depending on what is required. As an example, the definition

```
: .STRING
    QUERY 32 WORD COUNT -TRAILING TYPE ;
```

will accept text from the keyboard and type it on the display. To use this definition, type in .STRING and press <RETURN>. Any characters typed in after this will be redisplayed after the next <RETURN>.

It is important to realise that the keyboard interpreter itself uses WORD so that all keyboard input is transferred, word by word, to the word buffer, overwriting the previous contents. The implication of this is that all uses should be from within a definition so that its execution does not involve the keyboard interpreter. Simply executing

```
QUERY 32 WORD COUNT -TRAILING TYPE
```

will not give the intended result.

The delimiter character need not always be a space. The word <."> , for example, uses ASCII code 34 (&22) which is a double quote mark (") as its delimiter. Since <."> is a FORTH word it must be separated by a space from the text on which it operates. The closing double quote mark (") is not a FORTH word but only a delimiter and so does not need a space separating it from the

text. If a space is left, however, it will be included in the text string.

The word <."> is one of the most common ways of entering literal text into a definition, for display when the definition is executed. Outside a definition <."> will cause the immediate display of the input text.

It is not possible to use <."> to enter text from a running application. The reason is that it will not wait for input to be entered from the keyboard but assumes that the required text is already present in the input buffer, immediately after <.">. This is also true of the word STRING which is used as, for example

```
39 STRING BEANS'
```

STRING expects to find a delimiter character (in this case the code for a single quote) on the stack. It accepts text which must be in the input buffer as in the above example, before STRING itself executes. The text is up to the next occurrence of the delimiter (which should not be a null (ASCII 0) or a space) or until the next <RETURN>. The text is not moved from the input buffer but its start address and length are left on the stack. If the input buffer is exhausted when STRING is executed (for example by pressing RETURN immediately after typing the word STRING) only a zero is left on the stack. Strictly speaking this is an error condition since it is wrong to use STRING except when followed by text. It is possible to include STRING in a definition, but the text should still be present in the input buffer when STRING executes. It is, for example, used in the definition of OS' which accepts commands to be passed to the operating system, such as:

```
OS' TV 0,1'
```

This puts the display into a non-interlaced mode (remember this will only come into effect at the next MODE change by typing, say, 6 MODE). Note that the string to be used must still be present in the input buffer before OS' (and therefore STRING) is executed.

Inputting a string from a running application requires a technique such as that used in the .STRING example

earlier. For example, we could define

```
: $IN                ( c ... addr )
   CR ." ?$ "        ( give a prompt )
   QUERY WORD        ( accept text to word buffer )
;                        .
```

This expects a delimiter character on the stack and waits for text to be typed at the keyboard. The text of the string is placed in the WORD buffer with a starting count byte whose address is left on the stack.

It may sometimes be convenient to accept string input without it being transferred to the WORD buffer. For this purpose we can replace WORD in the above definition by (WORD) . The action is similar to that of WORD except that the text is not moved from the input buffer, and that the address of the first byte of the text and its length are left. A definition of such a string input word would be:

```
: $INPUT             ( c ... addr\count )
     CR ." ?$ "
     QUERY (WORD) ;
```

Don't forget with both of these words to make sure the required delimiter is on the stack. This will not normally be a space (ASCII code 32 or &20) so that spaces can be included in the string. Again, because of the use of the input buffer and the word buffer by the system, it is wise to use them from within a colon-definition.

7.1.3 Numeric input

Most applications do not require special numeric input routines. Since, in general, words expect to find their numeric data on the stack, this can be placed there by use of the keyboard interpreter before the word is executed. This is illustrated by the way G-C-D was used at the end of the last chapter.

Occasionally it may be necessary to wait for numeric input during the execution of a word, and for this the following definition may be used:

```
: NUMIN
    CR ." ? "                                   ( give prompt )
    QUERY                              ( accept text to buffer )
    32 WORD  ( transfer characters to WBFR with space
                                    - ASCII 32 - as delimiter )
    NUMBER                          ( convert to double number )
    DROP                                 ( make single number )
;
```

This routine leaves a single-precision number on the stack.

The disadvantage of this routine is that a standard error message is given if a non-valid character is present in the input. This causes execution to stop and the stack is cleared - a rather drastic action for a mistyped input!

Valid characters are:

a) an optional minus sign as the first character

b) an optional decimal point as the last character

c) all characters that may be interpreted as digits in the current numeric base; in hex , for example, valid characters are 0 to 9 and A to F inclusive.

It would be possible to define an alternative error-handling routine and change the ABORT vector to use this new routine (see the section on vectored execution in chapter 11).

An alternative solution is to use the word CONVERT.

There are two main differences between NUMBER and CONVERT. Firstly, CONVERT does not generate an error message on detecting a non-valid character, but simply leaves the address of the first unconvertible character in the input text. Secondly, it does not test the first character of the input for the presence of a minus sign. In addition, CONVERT requires a dummy double number on the stack, into which the input value is built.

The following definition generates its own error message and will not continue until a valid number is

entered. Note that the base is HEX so 20 WORD is the same as the 32 WORD in NUMIN , which was defined in DECIMAL base.

```
HEX
: INPUT
     BEGIN  CR  ." ? "
     QUERY  20  WORD (input text to WBFR)
     DUP 1+ C@ 2D =  (first char is minus sign?)
     DUP >R +        (save result & skip sign if there)
     0 0 ROT CONVERT (convert to double number)
     C@ DUP BL =     (next char a space?)
     OVER 2E = OR    (or a decimal point?)
     OVER 0= OR 0=   (or a null i.e. end of line?)
    WHILE            (if not)
     R> 2DROP 2DROP  (clean up stack)
     ." NOT VALID"   (report error)
    REPEAT           (and try again)
    R> IF            (recover sign)
    >R DNEGATE R> THEN      (change sign if necessary)
    2E - IF DROP THEN       (convert to single number)
    ;                       (if no decimal point)
DECIMAL
```

The input prompt and the error message may, of course, be changed to whatever you prefer. The result of a successful conversion is identical to that of the keyboard interpreter.

7.1.4 Manipulating blocks of memory

The examples of the last two sections make frequent use of WORD , which transfers a block of data from the input buffer to the word buffer. At this point it is worth examining the words which allow such transfers to be made.

The usual way of transferring a block of bytes from one region of memory to another is by the use of CMOVE . This word uses three values from the stack: the starting address of the source block, the starting address of the destination block and the number of bytes to be moved. The stack action is therefore:

```
CMOVE ( from\to\count ... )
```

The byte with the lowest address is moved first and the

transfer proceeds in the order of increasing address. There is never a problem if the destination address is less than the source address, but a difficulty arises if the destination address is higher than that of the source and the two regions overlap. Consider, for example, that the five bytes starting at FROM contain the characters F O R T H and it is required to move them one byte forwards in the memory. It might be thought that the following sequence would do the job:

```
FROM DUP 1+ 5 CMOVE
```

This illustration shows what would happen:

```
FROM      > F   F   F   F   F   F
            O > F   F   F   F   F
            R   R > F   F   F   F
            T   T   T > F   F   F
            H   H   H   H > F   F
                                F

Moves       0  1st 2nd 3rd 4th 5th
```

In each column the arrow head indicates the character that will be moved to produce the next column. The final result is that the whole region of memory is filled with the first character - probably not the required effect! In order to avoid this the word <CMOVE can be defined. Its overall action is the same as that of CMOVE except that the byte with the highest address is moved first and the transfer proceeds in order of decreasing address. A high-level definition is:

```
: <CMOVE
    1-  0  SWAP  DO  OVER  I + C@
                     OVER  I + C!
                -1  +LOOP  ;
```

The sequence

```
FROM  DUP  1+  5  <CMOVE
```

will produce the following result:

```
FROM     F   F   F   F > F   F
         O   O   O > O   O   F
         R   R > R   R   O   O
         T > T   T   R   R   R
       > H   H   T   T   T   T
             H   H   H   H   H

Moves    0  1st 2nd 3rd 4th 5th
```

If you don't want to have to worry about which of the two versions to use, you can define an 'intelligent' version which will select the correct one for you:

```
: CMOVE ( from\to\count ... )
    >R 2DUP R> ROT ROT -
    IF <CMOVE ELSE CMOVE THEN
;
```

In addition to the block transfers discussed above it is often necessary to fill a region of memory with a given character. This may be used, for example, to initialise the contents of an array or to clear the contents of a buffer. The words provided for this purpose are:

FILL (addr\n\b ...)	Fill n bytes of memory starting at addr with byte b
ERASE (addr\n ...)	Fill n bytes of memory starting at addr with ASCII null
BLANKS (addr\n ...)	Fill n bytes of memory starting at addr with ASCII space

The definition of FILL is interesting as it uses the overlaying feature of CMOVE that was eliminated by the use of <CMOVE . The definition is worth examination and is given without comment:

```
: FILL ( addr\n\b ... )
    SWAP >R OVER C!
    DUP 1+ R> 1- CMOVE
;
```

The definitions of ERASE and BLANKS are very simple:

```
: ERASE   0 FILL ;
: BLANKS   BL FILL ;
```

BL is a constant whose value is 32, the ASCII code for a space (or blank).

7.1.5 **Testing the keyboard**

It is likely that, within many applications, you will need to test the keyboard to see if a key is being pressed. The basic way to do this is to use the word ?KEY which expects to find a number on the stack.

This number will be interpreted in one of two ways, depending on whether it is positive or negative. If it is positive (i.e. in the range 0 to 32767 inclusive) the system will wait for that number of hundredths of a second, continuously testing the keyboard for a key being pressed. If a key is pressed within the time limit the ASCII code of the key is left on the stack. If no key is pressed then a negative value is left. As an example:

```
: ?KBD
   CR  ." PRESS A KEY (BE QUICK)"
   CR  50 ?KEY DUP 0<
   IF   ." YOU WERE TOO SLOW"
   ELSE ." YOU PRESSED A " EMIT
   THEN CR  ;
```

If the number is negative then ?KEY will test the keyboard to see if a particular key is being pressed at the instant that ?KEY is called. The value of the number determines which key is tested (see the table given on page 275 of the BBC Microcomputer User Guide). A value of 1 will be left on the stack if the key was pressed, otherwise a zero is left. These values may be regarded as true and false flags respectively. This form is used by the word ?TAB which tests if the TAB key is being pressed and is used, for example, by VLIST to pause the listing.

During the execution of ?KEY the keyboard buffer is emptied, and the auto-repeat is disabled, but re-enabled on exit. To prevent the auto-repeat causing

characters to be fed to the input buffer between the exit from ?KEY and removing your finger from the key, you may want to use a definition such as:

```
: TABTEST
   ?TAB DUP
   IF
    BEGIN ?TAB NOT UNTIL
   THEN  ;
```

This will act exactly like ?TAB except that the routine will not be left until your finger is removed from the TAB key.

7.2 Number bases

All numeric input and output is converted according to the current value of the user variable BASE . Any value of BASE may be used, subject to the restriction that it should lie in the range 2 to 255. A practical upper limit is 36, to avoid the use of non-alphanumeric characters. The FORTH-79 standard specifies a range of 2 to 70 for BASE .

The internal numeric handling of FORTH is always in binary, irrespective of the value of BASE , so there are no time overheads to working in any base you choose.

The two numeric bases DECIMAL and HEX are provided with the system. Any other base can be defined as follows:

```
: BINARY   2 BASE ! ;
: OCTAL    8 BASE ! ;
: BASE-36   36 BASE ! ;
etc.
```

On first entry to the system, or after executing COLD or WARM , the base will always be DECIMAL .

Many decimal-to-hex routines have been published in BASIC, with a greater or lesser degree of complexity. In FORTH such a routine is simply:

```
DECIMAL ( make sure you start in decimal )
: D->H
    HEX . DECIMAL ;
```

Here is an example of its use:

```
31 D->H
1F OK
```

The routine can be modified to translate between any two bases.

7.3 Output

7.3.1 Character output

To output a single character, the word EMIT can be used used. This will display the character whose ASCII code is on the stack.

Examples:

```
65 EMIT
A OK
49 EMIT
1 OK
```

It can also be used to execute control codes (see the BBC Microcomputer User Guide, page 378) from within a definition, for example

```
: BELL 7 EMIT ;
```

A better method, however, is to use >VDU as described in section 12.1.2.

7.3.2 Text output

Text strings in FORTH are stored with a preceding length count byte, as mentioned in section 7.1.2. Access to a string is usually via the address of this byte.

The display of a string is performed by TYPE which expects on the stack the address of the first character, under a length count. The conversion to

this form from the address of the count byte is done by the word COUNT. Thus

```
WBFR COUNT TYPE
```

will display the string starting with its count byte at the address given by WBFR. The character count may include a number of blank spaces at the end. These can be removed by the use of -TRAILING, which deletes all trailing spaces from the string, for example:

```
WBFR COUNT -TRAILING TYPE
```

Remember that the use of strings stored at WBFR should only be from within a colon-definition to avoid their being overwritten by the keyboard interpreter.

The FORTH system does not provide string handling facilities but they are fairly easy to include if required. For example the following definitions provide mid-, left and right string extraction. They all expect to find as addr1, the address of the count byte of the string from which characters are to be extracted.

Both LEFT$ and RIGHT$ require one further value, the number of characters to be extracted. MID$ requires two values, the position in the string of the first character to be copied, as n1, and the number of characters.

The definition of MID$ contains a number of safeguards so that the characters to be copied will not extend beyond the limits of the original string. An error message is generated if silly values are chosen so that a negative character count results. In all cases the selection of a range totally outside the limits of the original string will normally leave a string of one (first or last) character. The starting address and length count of the extracted string are left on the stack.

```
: MID$      ( addr1\offset\number ... addr2\count )
    >R              ( save number )
    1  MAX          ( not before first char )
    2DUP
    R> +            ( last char+1 )
    SWAP C@ 1+      ( max length available+1 )
    MIN >R          ( dont go past end )
    OVER C@ MIN     ( nor for starting char )
    R> OVER -       ( how many to extract )
    >R + R>         ( get address of first char )
    DUP 0< 5 ?ERROR ( negative count not allowed )
;

: LEFT$             ( addr1\n1 ... addr2\n2 )
    1 SWAP MID$  ;  ( offset is 1 )

: RIGHT$
OVER C@ OVER - 1+   ( calculate offset )
SWAP MID$  ;
```

All three of these leave the stack in a state ready to TYPE the selected character string. This is illustrated in the following example which uses the first version of string input, $IN (see section 7.1.2):

```
: STRINGS
    39  $IN              ( input string )
    DUP  COUNT TYPE      ( display it )
    DUP 7 LEFT$          ( get first 7 chars )
    CR TYPE              ( display them )
    DUP 12 3 MID$        ( get 3 middle chars )
    TYPE                 ( display them )
    6 RIGHT$             ( get last 6 chars )
    TYPE                 ( display them )
    CR  ;
```

Then execute STRINGS as follows:

```
STRINGS <RETURN>
?$ THIS IS NOT A SHORT STRING' <RETURN>
THIS IS NOT A SHORT STRING
THIS IS A STRING
OK
```

Further discussion of strings is deferred to section 9.5.

7.3.3 Numeric output

The numeric output operators provided in FORTH are:

.	(n ...)	Display the signed number n followed by one space
.R	(n1\n2 ...)	Display the signed number n1 at the right of a field n2 characters wide; no following space is printed
D.	(nd ...)	Display the signed double number nd in the format of <.>
D.R	(nd\n ...)	Display the signed double number nd to the right of a field n characters wide; no following space is printed
U.	(un ...)	Display the unsigned number un in the format of <.>
H.	(n ...)	Display in hexadecimal base in the format of <.>
DEC.	(n ...)	Display in decimal base in the format of <.>

The words .R and D.R are useful for tabulating information. Their use is illustrated in the following routine which will dump 64 bytes of memory, given its starting address. When the listing stops, pressing the Space Bar will display a further block. Pressing any other key will terminate the routine. The display is in hex, regardless of the initial value of BASE , which is restored on exit from the routine:

```
: DUMP                                          ( addr ... )
    BASE @ SWAP                          ( save current base )
    HEX
    BEGIN
     DUP  64  +  SWAP        ( set address of next block)
        8 0 DO CR
            DUP I 8 * +
            DUP 0 4 D.R SPACE              ( show address )
            8 0 DO
                DUP I + C@                   ( get a byte )
                3 .R                         ( display it )
                LOOP
            DROP
            LOOP
        DROP CR
        KEY BL -          ( wait for key press and test )
    UNTIL                       ( if space, repeat loop )
    DROP
    BASE !                                  ( restore base )
    CR
;
```

To illustrate the action of U. try the following:

```
30000 .
30000 OK
30000 U.
30000 OK
```

This shows that U. and<.> give the same result for positive signed numbers.

```
-30000 .
-30000 OK
-30000 U.
35536 OK
```

The number -30000 is interpreted by<.> as a signed integer in the range -32768 to +32767, but by U. as an unsigned integer in the range 0 to 65535. Whether a number is to be treated as signed or unsigned is a matter of context.

The word U. is simply defined as < 0 D. > ; in other words it prints the value as a double-precision number with a high order part of zero.

The sequence

```
0 4 D.R
```

in DUMP also uses this idea to display a 2 byte (4 hexadecimal digits) address as an unsigned number.

7.3.4 Numeric output formatting

The numeric output operations of the previous section enable the use of two formats: the printing of a number at the current cursor position (using <.>) and the placing of a number at the right of a field of specified width (using .R and D.R).

Other formats may be produced by use of the special numeric output formatting words:

<# Set up for numeric conversion

#> Terminate numeric conversion

Convert one digit

#S Convert the remaining digits

SIGN Insert a minus sign in the converted string.

HOLD Insert the specified character in the converted string

The words # , #S , SIGN , HOLD may only be used between <# and #>, and all act on a double-precision number on top of the stack. On completion of the conversion, the word #> leaves the number ready to be displayed by TYPE .

The first example will display a double-precision number as pounds and pence. The original number is the value in pence and the routine will handle amounts up to 21474836.47, which should be enough for most purposes!

```
: .POUNDS          ( nd ... )
    DUP ROT ROT    ( keep high part, including sign )
    DABS           ( make positive )
    <#             ( start conversion )
    ##             ( convert 2 digits to pence )
    46 HOLD        ( insert decimal point )
    #S             ( convert remaining digits )
    SIGN           ( insert sign if needed )
    96 HOLD        ( insert # )
    #>             ( end conversion )
    TYPE SPACE     ( display converted number )
;

-12345. .POUNDS
£-123.45 OK
```

The following example will display a double number with the decimal point in the position indicated by the value stored in DPL. If DPL is zero or negative the decimal point will be at the extreme right-hand side of the number.

```
: .REAL
    DUP ROT ROT DABS
    DPL @ 0 MAX          ( make sure not less than 0 )
    <#
    ?DUP IF              ( if non-zero )
        0 DO # LOOP      ( convert DPL digits )
    THEN
    46 HOLD
    #S SIGN #>
    TYPE SPACE
;
```

Note that all numeric conversion starts with the least significant digit and proceeds towards the more significant digits. The conversion process produces the string of output characters in a scratchpad area whose start address is placed on the stack by the word PAD . Numeric strings are built up starting at PAD and working towards the lower addresses. The characters in the string are therefore in the correct order (most-significant digit first) for display by the numeric output words.

8 Mass storage and the Editor

8.1 Introduction

In most of the examples so far it has been assumed that they would be typed in at the keyboard and executed directly. You will probably by now have made at least one mistake during the typing of a definition and will, therefore, have discovered that the only way to correct the mistake is to retype the definition. Provided the original text is still displayed on the screen the correct parts can be entered by means of the COPY and cursor control keys. If the text of the definition has disappeared from the screen it must all be typed again.

A program written in BASIC is stored in a form very close to the way in which it was originally typed, and so it is possible to list the program and edit lines very simply. Since FORTH applications are compiled as soon as the RETURN key is pressed, the original form of the definition (the source code) is lost and only the compiled form (the object code) is retained. One solution to this problem would be to use an application which can reconstruct the source code from the compiled object code. The reconstructed text could be edited and then recompiled.

A simpler approach is to keep a copy of the source text in a reserved area of memory. If the definition does not work correctly the text is readily available for modification and recompilation. This is the main purpose of the Editor vocabulary provided as an application with Acornsoft FORTH. Once the definitions are working correctly they may be transferred to, say, tape or disk for use on another occasion. They are saved in source form directly from the reserved memory which is, in fact, the mass storage buffer area.

8.2 Mass storage

Exactly where large amounts of data are stored will depend on the filing system currently in use. It may be on tape, disk, Econet files etc. To avoid having to specify which system is in use we can refer to all of them by the term 'mass storage'.

Mass storage is organised as a number of 'screens' or 'blocks', each of which contains 1024 (1K) bytes. A screen is conventionally divided into 16 lines of 64 characters. Each screen is identified by a number which ranges from zero to a maximum which will depend on the particular filing system in use. The simplest way of obtaining access to the contents of a screen is by use of the word BLOCK . Thus

```
3 BLOCK
```

will bring the contents of screen 3 into the mass storage buffer area and leave on the stack the address of the first byte of the data area of the buffer.

In a tape-based system the tape must be positioned to the start of the appropriate screen. Screen 3 is the start of the Editor application and is stored on the tape immediately after the end of FORTH. The normal 'Searching' and 'Loading' messages will be given, with the file name being the screen number written as a four-digit hexadecimal number.

The first line of screen 3 may be displayed by

```
3 BLOCK   64 TYPE
```

In this case, since screen 3 is already in the buffer, it will not be loaded again and the line of text will be displayed immediately. The whole of screen 3 can be displayed using the command

```
3 LIST
```

Again the screen will not be reloaded. If you now type

```
4 LIST
```

screen 4 will be read from mass storage into the buffer

area before being displayed. Screen 3 will still be present in the buffer area, as can be seen by LISTing it again. The buffers provided when Acornsoft FORTH is first loaded will store two screens. This is the absolute minimum to ensure their correct operation, but the buffer area may be increased by the user.

Each buffer occupies 1028 (&404) bytes of memory (the value is returned by the constant BUFSZ). This odd value is needed because two extra bytes are used both before and after the 1024 bytes of data. The first pair is used for holding the number of the screen whose data is currently occupying the buffer. The contents of these two bytes are regarded as a 16-bit number. If the most significant bit is set the contents of the buffer are marked as having been changed since they were read in from mass storage (i.e. UPDATEd). In this case the data will automatically be written back to storage before the buffer is used for another screen of data. The final two bytes always contain zero and are used to ensure that interpretation of the screen will stop at this point, even if the data completely fills the buffer.

The contents of the buffers can be written to mass storage by the use of SAVE-BUFFERS or FLUSH. These two words will only write the buffer contents if they have been marked as changed (section 8.3.6 describes how this is done). For the moment you will find that typing SAVE-BUFFERS will have no effect. FLUSH behaves similarly to SAVE-BUFFERS except that in addition to writing changed screens to mass storage it also marks all the buffers as empty. If you experiment with FLUSH you will have to read screens 3 and 4 back into the buffers, by using either BLOCK or LIST, before continuing.

8.2.1 Reconfiguring the mass-storage buffers

This section is concerned with changing the number and location of the mass-storage buffers. You may like to skip it on a first reading.

The number of buffers may be changed simply by altering the separation between the addresses held in the constants FIRST (the first byte of the buffer area) and LIMIT (the first unused byte after the end of the

buffers). The difference between these two addresses must always be an exact multiple of BUFSZ.

Before making any change to the buffers it is wise to execute either SAVE-BUFFERS or FLUSH so that there is no danger of losing any data which has been changed since the last read from mass storage.

Entering the following code will then make the system use three buffers:

```
FIRST  BUFSZ -    ( new address for FIRST )
' FIRST !         ( store it )
```

There is, of course, a corresponding reduction in the remaining memory available for the dictionary. It is your responsibility to make sure that the gap between the top of the dictionary (the address left by HERE) and the start of the buffer area (at FIRST) is sufficient for your application. As a general rule, 2-3K should be enough for most purposes.

An easier method of increasing the number of buffers is to put the number of buffers you require in the variable #BUF and then execute SETBUF ; for example:

```
5 #BUF !   SETBUF
```

will give you five buffers. The contents of the buffers are not initialised by SETBUF . To clear all the buffers you must then type EMPTY-BUFFERS .

The end of the buffer area is normally set to &5800 and memory above this address is reserved for high-resolution graphics (modes 4 to 7 are available). If your application does not need to use the higher-resolution modes you may change the value of LIMIT to make some of this memory available to FORTH. The relevant addresses are shown in the memory map in Appendix C.

The original buffer area, containing two buffers, can be restored at any time by use of INITBUF , which will also mark the buffers as empty. It will not, however, reset LIMIT to its original value, should you have changed it.

8.3 The Editor

8.3.1 Loading the Editor

You may have noticed from the listings of screens 3 and 4 that they contain the text of definitions exactly similar to those you have so far been typing at the keyboard. The contents of these screens may be interpreted, as though from the keyboard, using LOAD . Each of the two screens you have listed ends with the word --> which causes interpretation to continue with the next screen. Thus, if we load screen 3 it will, when interpreted, start the interpretation of screen 4, and so on. Now load the Editor by typing

```
3 LOAD
```

(If you have listed screens 3 and 4, as described in section 8.2, you will find that they will be LOADed immediately, since they are still present in the buffers; it is not until the loading of screen 5 begins that more information is read from mass storage.) The process will continue without the need of any intervention from the user until a screen does not contain a -->. Don't worry about the error message 'I MSG # 4' that appears during the loading. This is simply a warning that the FORTH word I has been defined to have another meaning within the Editor vocabulary.

Now that the Editor is loaded we can see how it is used to produce your own applications. The following description refers to the use of the Editor in a tape-based system. There are a few small differences for use with disks and these are described in section 8.4. The differences are merely for the sake of convenience and all commands described in this chapter will work with any filing system.

8.3.2 A sample editing session

The best way to learn the actions of the editing facilities is, as with the rest of FORTH, by using them. The following is an example of how the Editor vocabulary can be used to write, modify and save an application.

Before the Editor can be used a blank editing screen must be set up and the Editor vocabulary declared. If we attempt to list a blank screen by typing

```
30 LIST
```

the normal action will be, as seen earlier, to fetch screen 30 from mass storage. If screen 30 has not yet been written it is unlikely that a search of the tape will be very successful! To solve this problem the word PROGRAM has been included in the editor application. This should be used to initialise the system for writing a new application in one or more consecutive screens. It prompts for a starting screen number as shown in the following example.

It uses the Editor word P which puts a line of text into the screen. The line number should be on the stack and the command is normally used as

```
n P text for line n
```

Since P is a FORTH word it must be separated from the text by a space. The text for each line is terminated by RETURN. It is good practice to leave a space after P, even if you then just press RETURN (for example to clear a line of text) - otherwise an ASCII null may be placed in the line and this will stop interpretation of the screen at that point.

It is conventional for line 0 of every screen to contain a comment giving the contents of that screen. The FORTH word <(> is used to start a comment and causes all text up to a right parenthesis <)>, or to the end of the current line, to be ignored. Since <(> is a FORTH word it must be separated by at least one space from any following text. The right parenthesis is simply a delimiter and so need not be separated by a space from the end of the comment. It is usual, however, to leave a space to improve the appearance of the text. It should, of course, be separated by one or more spaces from any following words.

This example illustrates the use of the Editor to create and save a definition of RND, which generates a random number. It is shown exactly as it would appear

on the display, except for the underlining to indicate the computer's output:

```
PROGRAM
1st screen  number? 30
SCR 30  &1E
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
OK

0 P ( RANDOM NUMBER GENERATOR  )
OK
2 P DECIMAL
OK
4 P VARIABLE SEED
OK
6 P : (RND)  ( ... rand )
OK
7 P      SEED @   259 * 3 +
OK
8 P      32767 AND   DUP SEED ! ;
OK
10 P : RND    ( range ... random )
OK
11 P      (RND)  32767 */ ;
OK
```

```
30 LIST
SCR 30  &1E
  0   ( RANDOM NUMBER GENERATOR  )
  1
  2   DECIMAL
  3
  4   VARIABLE SEED
  5
  6   : (RND)  ( ... rand )
  7         SEED @   259 * 3 +
  8         32767 AND   DUP SEED ! ;
  9
 10  : RND    ( range ... random )
 11        (RND)  32767 */ ;
 12
 13
 14
 15
OK
```

At this point, with the editing of the screen contents complete, the application would normally be tested. The following simple tests show that the application is working as expected. In particular the stack action is shown to be correct in that no items are left on the stack and no extra stack values are used. Putting a few values on the stack and using .S to check the stack contents at various stages is a simple way of verifying the stack action. :

```
30 LOAD
OK

SEED ?
0  OK

1 2 3
OK

.S
1 2 3 OK

(RND)  .
3  OK
```

```
10 RND .
0 OK

.S
1 2 3 OK

: TEST   CR DO   DUP RND . LOOP DROP ;
OK

20 10 TEST
3 16 6 17 1 17 11 6 18 14 OK

.S
1 2 3 OK

. . .
3 2 1 OK
```

When the screen is completed to your satisfaction it can be saved to tape. This is done by typing the word

SAVE

The screen about to be saved to tape will be listed to ensure that it is the one required. The message

OK?

will appear at the bottom of the listing, and if the screen is correct you should press the 'Y' key. Pressing any other key will abort the save.

Once the random number generator has been saved (as screen 30) it may be entered into the dictionary at a later time in the same way as the Editor itself was loaded, by typing

30 LOAD

Screens may be numbered from 0 to 32767 inclusive.

If an application extends to more than one screen the word MORE can be used to save a screen and set up the next blank screen ready for editing.

One problem with the storage of screens on tape is that of modifying a previously written screen, which may be

in the middle of an application. In order to simplify this process the word LOCATE has been provided. It expects a screen number on the stack and will search the tape for the screen immediately before the one specified. It gives an audible signal when this screen is found so that stopping the tape at this point will leave it positioned just before the start of the specified screen. The previous screen is not read into the buffer area, so this provides a simple method of rewriting a single screen in the middle of an application.

8.3.3 The line editor commands

So far the only word we have used from the line editor is P , which puts text onto a given line. A complete list of line editor commands appears below. Each command expects to find the relevant line number on the stack. Most of the commands make use of the scratchpad area, whose starting address is given by PAD , where a line of text can be stored. Text is stored starting at PAD and working towards higher addresses, as opposed to the use of PAD for numeric strings (see section 7.3.4).

Command	Description
P	Put text onto line.
D	Delete the line, moving up the lower lines to close the gap, but hold the deleted line at PAD.
E	Erase the line, leaving it blank. The contents of the line are not saved.
H	Hold the contents of the line at PAD. The line also remains in the screen.
I	Insert the text from PAD at the specified line. The lower lines are moved down to make room for the insertion and line 15 is lost.
R	Replace the contents of the line with the text from PAD.

(cont'd)

S Spread the text by inserting a blank line. Line 15 is lost.

T Type the contents of the line and also copy it to PAD. The text remains in the screen.

In addition, the word TEXT will allow text to be put directly into PAD. Like WORD it expects a delimiter character on the stack. It accepts text from the keyboard up to the first appearance of the delimiter, or until 64 characters are typed, or RETURN is pressed. It is usual to use a delimiter that would not normally appear in the input from the keyboard so that the end of text is marked by RETURN. TEXT is used by all the Editor commands that put text at PAD with ASCII code 01 as a delimiter character, i.e. it is used as:

```
1 TEXT ( ... wait for text input )
```

It is worth setting up a dummy screen to practise using the line editor commands. Once you are familiar with each command, try to:

i) transfer a line to another position
ii) exchange two lines

Make sure that both of these work with line 15.

The following definition, as an example, will invert the order of the lines in a screen:

```
EDITOR DEFINITIONS ( Make sure it is in the Editor )
: INVERT  16  0  DO 15 H ( line 15 to PAD )
       FORTH  I          ( loop index )
       EDITOR  I         ( insert from PAD )
       LOOP L ;
```

It shows how a word can be defined to provide arbitrarily complicated editing features. It also illustrates the way in which the vocabulary structure can be manipulated to use both definitions of the word I . Note that since the vocabulary words are IMMEDIATE they change the CONTEXT vocabulary for the following word(s) but do not themselves appear in the compiled definition for which they are used.

8.3.4 The string editor commands

The string editing facilities allow the location, insertion and deletion of individual character strings within a screen. This is accomplished with the aid of an editing cursor (displayed as||) whose byte offset, from the start of the screen, is stored in the user variable R# . The cursor is set to the beginning of the screen by the word TOP .

The remaining string editing commands are given in the lists below. They are divided into groups according to the type of input they require.

The members of the first group need to be followed by typed text, on which they act (like the line editing command P). There must, of course, be one space between the command and the text, but (again like P) any additional spaces are regarded as part of the text.

Command	Description
C	Insert the given text at the current cursor position.
F	Find the given text and position the cursor immediately after its first occurrence.
TILL	Delete all text, from the current cursor position to the end of the text given. This command will only act on text within a single line of the screen.
X	Find and delete the first occurrence of the given text.

Each of the above commands will also leave the given text at PAD . They give an error message if the text is not found in the screen.

The following example shows their actions. It is

assumed that line 6 of the screen contains:

```
THIS IS A SILLY EXAMPLE
```

and the cursor has been reset by the use of TOP.

```
F  IS A
THIS IS A|| SILLY EXAMPLE 6 OK
C N
THIS IS AN|| SILLY EXAMPLE 6  OK
TILL LY
THIS IS AN|| EXAMPLE 6 OK
X PLE
THIS IS AN EXAM|| 6 OK
```

The words of the following group require no additional input, but expect to find their text at PAD :

Command	Description
N	Find the next occurrence of the text at PAD .
B	Move the cursor back by the number of characters in the text at PAD .

Each command in the next group requires a character count on the stack:

Command	Description
DELETE	(count ...) Delete count characters, backwards from the current cursor position.
M	(count ...) Move the cursor by count characters, either forwards or backwards depending on the sign of count. The text itself is unchanged (0 M is a simple way of displaying the current cursor position).

Finally (we always save the best until last!) there is the word MATCH. This is the string-matching routine used by all the words which search for text. In the Editor it is written, for clarity, in high-level FORTH. To increase the speed of this word it may be replaced by a machine code primitive. The code for such a definition is given in section 8.3.5.

The stack action of MATCH is, however, a bit complicated; using 4 and leaving 2 stack values:

```
MATCH ( addr1\length\addr2\count ... f\offset )
```

The routine attempts to match the string, whose starting address is addr2 and whose length is count bytes, to the contents of memory starting at addr1 and finishing at addr1 + length. It leaves a flag, which will be true (non-zero) if the match succeeds and false (zero) if it fails, beneath the offset, from addr1, to the byte immediately following the matched string. Thus (addr1 + offset - count) will give the address of the start of the matched string.

8.3.5 A machine code MATCH

The listing of the machine code for MATCH is given using the method described in chapter 5 to enter machine code without the use of the assembler.

Like most machine code in FORTH, it is relocatable and so may be copied into your own application if you require a routine to search an area of memory for a particular character string.

```
HEX
CREATE MATCH   HERE  -2 ALLOT ,
   4A9 ,  20  C,  8A +ORIGIN ,
  CACA ,  CACA ,
    94 ,   194 ,
  FFA0 ,   C8 C,
  60C4 ,  2CB0 ,
  62B1 ,  66D1 ,  F5F0 ,
  66E6 ,   2D0 ,
  67E6 ,    F6 ,   2D0 ,
   1F6 ,  64A5 ,   2D0 ,
  65C6 ,  64C6 ,  64A5 ,
  60C5 ,  65A5 ,  61E5 ,
  D5B0 ,
    A9 ,   295 ,   395 ,
  64A4 ,  9818 ,    75 ,
   48 C,    A9 ,   175 ,
   4C C,  65 +ORIGIN ,
DECIMAL
```

This code can be used to replace the entire contents of screens 7 and 8 of the editor application.

8.3.6 A note on UPDATE

The word UPDATE marks the current editing screen as having been modified since it was loaded from mass storage. It does this by setting the most significant bit of the screen number in the buffer to 1. All Editor words which modify the buffer contents include UPDATE and the normal action in FORTH is to save any UPDATEd screen when its buffer is needed again. This can result in screens being saved out of order.

Since screens can only be accessed serially from tape it is essential that they are saved in the correct sequential order. In order to ensure that screens are not saved out of sequence, screen 3 of the Editor contains the command (see section 11.1):

```
ASSIGN UPDATE  TO-DO NOOP
```

This disables the normal action of UPDATE .

If you need, for some reason, to UPDATE a screen during an editing session you may do so by use of the word (UPDATE). This word is also used by SAVE to allow the screen to be written to tape, despite the disabling of UPDATE .

The normal action of UPDATE may be restored by:

```
ASSIGN UPDATE  TO-DO (UPDATE)
```

8.4 Using the Editor with disks

The use of the Editor with a disk-based system is very similar to its use with tape storage. The major difference is that the tape-oriented words, PROGRAM , SAVE , MORE and LOCATE are not necessary. It is also unnecessary to modify the action of UPDATE . You may use them if you wish, but it will probably be more convenient to remove them from the editor application. This can be done very simply by use of the Editor to remove their definitions from screen 3. The most convenient way is to insert the word --> at the

beginning of line 3 of this screen as shown below (make sure there is a space between --> and ASSIGN on the line):

```
3 LOAD
3 LIST
EDITOR
3 T
C -->

SAVE
COLD
3 LOAD
```

Screen n may then be set up for editing with the command

```
n LIST   EDITOR
```

Screens may be edited in any order, each one being made the current editing screen using 'LIST'. Any modified screen will be saved to the disk automatically whenever its buffer is needed for another screen. At the end of the editing session (or at any other time) you should use SAVE-BUFFERS or FLUSH to make sure that no modified screens remain in the buffers.

The number of screens that can be used on a disk-based system is dependent on the disk space reserved for their storage. Space can be allocated by use of the word CREATE-SCREENS. This word is designed to be used only with mass storage on disks. Since it destroys the contents of memory from &5800 to &7BFF inclusive, it should not be used if this region of memory is being used for any other purpose, such as a high-resolution graphics display. To create screen storage, place a formatted disk in the drive and type

```
CREATE-SCREENS
```

The system will respond with:

```
Are you sure  (Y/N)?
```

Pressing Y will cause space to be reserved for 180 screens, numbered 0 to 179; any other key will abort the process. The space is reserved as a number of named

files, each of which may hold several screens. The number of files and the number of screens in each file are determined by the constants MAXFILES and S/FILE respectively. The default values of these constants are 20 and 9 respectively, giving screens numbered from 0-179 on double-density disk drives.

If you are using a single-density drive, the value of MAXFILES should be reduced to 10 by typing:

```
10 ' MAXFILES !
```

In this case screens numbered 0-89 will be available. You can use a smaller value if you wish, with a corresponding reduction in the number of screens available.

You may also change S/FILE if you wish, but it is not recommended since you must then remember to change it whenever you use Acornsoft FORTH with such non-standard disks. In any case, the total length of a file (B/BUF B/SCR S/FILE * *) should not exceed 9K bytes.

The screens created by CREATE-SCREENS cannot be overwritten by other system files, and it is possible to store both screens and other files on the same disk. Any previous screens on a disk are, however, likely to be destroyed by a second use of CREATE-SCREENS, which should be regarded in the same way as the Operating System *FORM40 or *FORM80 commands.

When in use, the file structure of the screens is not apparent to the user but can be examined by using the Operating System *CAT command. This may, of course, be used from within FORTH by typing

```
OS' CAT'
```

9 What CREATE and DOES> doe

9.1 Introduction

FORTH can be considered to operate at a number of different levels. The lowest level is the execution of a word from the dictionary and this can be termed a level 0 operation. The next higher level, which we can call level 1, is the use of a defining word, for example VARIABLE or <:> , to produce a dictionary entry for later (level 0) execution. All levels higher than 0 result in a new entry being made in the dictionary. This chapter is concerned with the next higher level, level 2, in which new defining words are formed. The sequence of operations involved is:

a) generate a new defining word (level 2)

b) use the defining word to produce a new dictionary entry (level 1)

c) execute the new entry (level 0)

One higher level is possible: to produce alternative ways of generating defining words. This level, which is often termed 'meta-FORTH', enables the writing of totally new FORTH-like languages, and is beyond the scope of this manual.

Each defining word in FORTH can be considered as a mini-compiler, dedicated to compiling a particular type of structure into the dictionary. If a new structure is required, for exaqmple an array, a new word is required to allow its compilation. Just as the generation of a new word (level 1) extends the FORTH language, the generation of a new defining word (level 2) extends the FORTH compiler.

The two words CREATE and DOES> are used for this purpose.

9.2 The action of CREATE and DOES>

The two words are used in a definition of the following form (level 2):

: FAMILY CREATE ... DOES> ... ;

where an optional list of words may follow each of the two. This is, in one sense, an ordinary colon-definition and all the words are compiled in the normal way. The use of CREATE and DOES> , however, makes it a level 2 definition.

The words following CREATE are concerned with building the dictionary entry for the new word defined by FAMILY . Those following DOES> determine the action of the new word, and although they are compiled into the definition of FAMILY , they are not executed until the new word is used. It is important to remember that the CREATE words come into effect at compilation time, and the DOES> words at execution time.

The execution of FAMILY takes the form:

FAMILY MEMBER

and may expect one or more values on the stack, depending on the definition of FAMILY . This is a level 1 process and creates a dictionary entry for the word MEMBER .

When MEMBER is executed (level 0) the address of its parameter area is placed on the stack, and then the words following DOES> in the definition of FAMILY are executed. Execution of all words defined by FAMILY begins with this code, so FAMILY produces a group of words with related actions. CREATE and the words following it 'customise' each new word by compiling items unique to it (for example values from the stack, or further words) into the parameter area of its dictionary entry. When the new word is used, the words after DOES> use the address of the parameter area to gain access to these

items, allowing each of the words created by the same defining word to have its own function.

9.3 The use of CREATE and DOES>

Some simple examples may clarify the use of these words.

Let us first have a look at an alternative definition of VARIABLE . This word appears in the nucleus dictionary and its action has been described in section 5.3.2. It creates a dictionary entry with space for a single variable, and initialises it to zero. The following definition of VARIABLE is identical except that the values of the variables it creates are not initialised.

```
: VARIABLE CREATE 2 ALLOT DOES> ;
```

When this is executed by typing:

```
VARIABLE SIZE
```

CREATE creates the dictionary entry for SIZE and 2 ALLOT reserves two bytes in the parameter area. In this case there are no words after DOES> so when SIZE is executed it just leaves the address of the parameter field on the stack. This gives access to the value, which is initally indeterminate, through <!> anc <@> as normal.

The definition of the VARIABLE in the nucleus dictionary would be:

```
: VARIABLE CREATE 0 , DOES> ;
```

Instead of allotting space, the value of zero is compiled into the parameter area by <,> .

The definition of CONSTANT is:

```
: CONSTANT CREATE , DOES> @ ;
```

The compilation stage is identical to that of VARIABLE except that the value is taken from the stack, but when the word defined by CONSTANT is used, @ leaves the value on the stack.

We may also create single byte variables and constants by:

```
: CVARIABLE CREATE 0 C, DOES> ;
    used as CVARIABLE TINY
```

and

```
: CCONSTANT CREATE C, DOES> C@ ;
    used as 10 CCONSTANT TIM
```

The value in both cases is, of course, restricted to the range 0 to 255 inclusive.

9.4 Arrays and tables

The use of CREATE and DOES> to create new types of data structure can be illustrated by the extension of FORTH to handle arrays.

9.4.1 One-dimensional arrays

A simple definition for a one-dimensional array is:

```
: ARRAY
    CREATE 2* ALLOT
    DOES> SWAP 2* +
;
```

A ten-element array of single-precision variables is created by:

```
10 ARRAY VALUES
```

The words after CREATE reserve two bytes for each element. When VALUES is executed, the index is taken from the stack and multiplied by two (to make it a byte offset) and added to the address of the start of the parameter field. It is therefore converted to the address of the corresponding element. The contents of

the array VALUES are not initialised but it may be filled by, for example:

```
5 0 VALUES !
10 1 VALUES !
```

which will put the numbers 5 and 10 into the first two elements of VALUES . The contents of a particular element may be placed on the stack by, for example,

```
1 VALUES @
```

or typed on the display by

```
1 VALUES ?
10 OK
```

Many people replace SWAP 2* + in ARRAY by the equivalent OVER + + since, in some implementations it is faster to execute. In Acornsoft FORTH there is not much difference between the two execution times.

The array index must be on top of the stack before executing VALUES . It must, for the above example, be in the range 0 to 9 inclusive. No checks are made on the range of the index so care must be taken not to over-write other dictionary entries by using an out-of-range index.

The following alternative definition of ARRAY will check the range and give an error message, if needed:

```
: ARRAY
    CREATE DUP 1 - ,  (store maximum index)
    2* ALLOT          (reserve space)
    DOES> 2DUP        (duplicate index & parameter addr)
    @ OVER <          (check if index too large)
    SWAP 0< OR        (or if negative)
    5 ?ERROR          (issue error message if needed)
    2+                (otherwise step over max index value)
    SWAP 2* +         (and convert index to element address)
;
```

If a more specific error message is required, the words 5 ?ERROR may be replaced with, for example,

```
IF DROP CR
    ." RANGE ERROR - ARRAY INDEX = " .
    QUIT
THEN
```

The inclusion of error checks, such as that given above, has the disadvantage that it decreases the speed of execution. A solution to this problem is to develop an application using full error checks until it is working correctly. When it is certain that no errors can occur, the words containing error checks can be replaced by simpler, faster versions. If an application is developed by use of the editing facilities described in chapter 8, it is a simple matter to change these words as the remainder of the application is unchanged.

9.4.2 Two-dimensional arrays

The following definition allows the creation of two-dimensional arrays. The elements are single-precision variables and the array contents are not initialised. No index checking is done but error checks can be added as for one-dimensional arrays.

```
: 2ARRAY
    CREATE DUP ,      ( store second index )
    * 2* ALLOT        ( reserve space )
    DOES> ROT         ( get first index to top )
    OVER @ *          ( multiply by stored index )
    ROT +             ( add second index )
    2*                ( calculate byte offset )
    +                 ( add to base address )
    2+                ( step over stored index )
;
```

It is used, for example, as:

```
10 5 2ARRAY RECTANGLE
```

to create a 10 by 5 array called RECTANGLE . Here, the array indices may range from 0,0 to 9,4 inclusive. The

address of, for example, the 2,3 element is left on the stack by:

```
2 3 RECTANGLE
```

9.4.3 Tables

It may be necessary to create a table of values for which only the starting address is needed. This type of structure can be implemented very simply as follows:

```
: CTABLE
    CREATE ALLOT
    DOES>
;
```

This, when used in the form

```
10 CTABLE DATA
```

will create the word DATA with space for ten single-byte values. When DATA is executed it will leave on the stack the starting address of the data table.

The word ALIST of section 6.3 is an example of a table which leaves both its start address and the number of 16-bit items it contains. It may be created by use of the following definition of the word TABLE :

```
: TABLE
    CREATE DUP ,              ( store number of items )
           2* ALLOT                    ( reserve space )
    DOES> DUP 2+               ( get start address )
           SWAP @             ( max number of items )
;
```

ALIST is then created by

```
n TABLE ALIST
```

where n is the required maximum number of items.

9.5 Strings

There are many ways of implementing string handling in FORTH. Two examples are given in 'BYTE' magazine, in the August 1980 and February 1981 issues.

The following example shows a simple alternative method of handling strings up to 255 characters in length:

```
: STRING     ( max length ... )
    CREATE
    DUP C,   ( keep maximum length )
    0 C,     ( zero length byte = empty )
    ALLOT    ( reserve space )
    DOES>
    1+       ( step over maximum length )
;
```

An empty string is then created by, for example:

```
10 STRING WORDS
```

The string variable WORDS may now hold any character string up to 10 characters in length. A few additional words are required for input and output of strings.

A new definition of $IN (see section 7.1.2.) uses the constant C/L , which gives the number of characters per line in the display (i.e. 64). Remember also that PAD returns the start address of the scratchpad area used for text (section 8.3.3) and for numeric conversion (section 7.3.4).

```
: $IN                                ( ... addr\length )
    WBFR C/L 1+ BLANKS          ( clear memory at WBFR )
    1 WORD                      ( input string to WBFR )
                    ( terminated by carriage return )
    PAD C/L 1+ CMOVE              ( move string to PAD )
    PAD DUP C@ 1+          ( prepare to move string ... )
                                ( including count byte )
;
```

```
: $!                    ( from addr\length\to addr ... )
    2DUP 1 - C@ 1+ >      ( check if space for string )
    IF CR ." STRING OVERFLOW "   ( if not give error )
            2DROP DROP QUIT     ( clear stack & quit )
    THEN
    SWAP CMOVE                ( otherwise store string )
;

: $@                          ( addr1 ... addr2\length )
    COUNT                     ( prepare to type string )
;
```

The following shows how these words are used, assuming that the string variable WORDS has been created as in the earlier example:

```
$IN HELLO
OK
WORDS $!
OK
WORDS $@ TYPE SPACE
HELLO OK
```

If the words LEFT$ and RIGHT$ of section 7.3.2 are also defined, the following examples can be tried:

```
WORDS 2 LEFT$ TYPE SPACE
HE OK
WORDS 3 RIGHT$ TYPE SPACE
LLO OK
```

9.6 A CASE statement

9.6.1 Introduction

The conditional structure of section 6.2 allows a two-way branch using

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

A CASE statement allows a branch to one of many possible word sequences with a return to a common point. There are two basic methods for the selection of the case to be executed. The first is a 'positional' case where the values to be tested are restricted to the first n integers. The second method

is a 'keyed' case where a value is tested against a sequence of explicit values which need not be in numerical order.

9.6.2 A positional CASE

The following simple example of a positional CASE will select the words to be executed by means of an integer value on the stack. The value must be in the range from zero to one less than the number of cases available in the particular example. No error checks are made for a number outside the permitted range. This CASE structure is used in the graphics demonstration, provided with the system and listed in chapter 12.

Here is the definition of the defining word CASE: :

```
: CASE:
    CREATE SMUDGE ]
    DOES>  SWAP 2*
           + @
           EXECUTE
;
```

The word EXECUTE takes the execution (code field) address of a word from the stack and executes the word's definition. Thus:

```
' WARM          ( get parameter field address of WARM )
CFA                   ( convert to code field address )
EXECUTE
```

has the same effect as executing WARM directly from the keyboard.

To use the CASE structure it is first necessary to define each of the possible actions, for example:

```
: NOTHING
    ." CASE 0 DOESN'T DO MUCH " ;

: BELL
    ." CASE 1 RINGS THE BELL "
    7 EMIT ;

: HOME
    ." CASE 2 HOMES THE CURSOR "
    30 EMIT ;
```

These actions are then included in a CASE structure for, say, the word TEST :

```
CASE: TEST
    NOTHING BELL HOME ;
```

When TEST is being created, SMUDGE ensures that the new entry will be found in a dictionary search and] then sets compilation mode, so that the words following TEST will have their addresses compiled into the dictionary entry.

When TEST is executed by:

```
0 TEST
1 TEST
or
2 TEST
```

the words following DOES> convert the case number to a pointer to the address of the correct word in the list, and execute it.

Note that the CASE statements of many high-level languages are based on GOTO-type control transfers, whereas this FORTH CASE has the options compiled into the definition of the case word so that the choice is fixed before execution. Basically, this is because it is not easy to handle forward references, i.e. words that have not yet been defined, in FORTH.

For a further discussion of keyed cases and of a variety of other possible forms of CASE statements in FORTH see FORTH Dimensions, Vol 2, No 3 (1980) (see Appendix E).

10 The FORTH Assembler

10.1 Introduction

The contents of this chapter assume that you have some knowledge of 6502 machine code but previous experience with an assembler is not necessary. There are a number of books available which describe the use of machine code for the 6502, for example, Programming the 6502 by R Zaks (Sybex). The BBC Microcomputer User Guide also includes a comprehensive account of assembly language programming.

When writing long applications there will be occasions when even the high speed of FORTH is not sufficient, and unacceptably long execution times will occur. In addition, certain applications, such as interrupt handlers, cannot be written entirely in a high-level language. In such circumstances it is necessary to resort to machine code.

Short machine code routines can be hand-assembled, placing the appropriate bytes of code directly into the dictionary (as described in chapter 5). This has the advantage of requiring no additional software aids, but has a number of significant drawbacks. For example, it can be a very tedious process for all but the shortest routines, and the resulting source code is virtually unreadable.

The addition of an assembler removes these difficulties, but at the expense of increasing the software overheads. A FORTH assembler, however, occupies only about 1.5K and greatly simplifies the task of producing error-free code. In many cases the application can be developed in high-level FORTH and the time-critical sections subsequently replaced by their machine code equivalents. Such a development will take very little more time than an entirely high-level approach.

The use of a FORTH assembler may appear somewhat strange to anyone accustomed to a 'conventional' assembler. The branch instructions are never used explicitly, the assembly mnemonics and their operands are written in the 'wrong' order, and labels are only rarely required.

It is, however, an extremely sophisticated single-pass assembler with comprehensive error checks. Furthermore, all the power of FORTH itself is available throughout the assembly process. The speed, power and simplicity far outweigh the task of becoming accustomed to the unusual approach.

10.2 An example

We can illustrate some of the features of the FORTH assembler by looking at the creation of a simple machine code definition. Before trying this or any other example, the assembler vocabulary must be loaded. It is provided in seven screens of source code starting at screen 12 and is loaded, in the normal way, by typing

```
12 LOAD
```

It is normal for a number of error messages to appear on the screen during the loading process. These can be ignored as they merely indicate that several words are being redefined to have different meanings in the assembler.

The example we shall look at is DROP , whose action should by now be familiar. The conventional assembly language for the machine code of this word could be written as

```
JMP  POP
```

where POP is the entry point to existing machine code to remove the top stack item and make a valid return to FORTH.

The FORTH assembler allows the word with this code to be created by

```
CODE  DROP
   POP JMP,
END-CODE
```

The two words CODE and END-CODE are used to start and end a machine code definition, rather like <:> and <;> are used for a colon-definition.

CODE generates a name header for the word immediately following it and starts some of the error-checking procedures. Unlike <:>, however, it changes the CONTEXT vocabulary to ASSEMBLER (rather than to the CURRENT vocabulary) and it leaves the system in execution mode. The words in an assembler definition are executed, not compiled. Each section of the code is placed in the dictionary (i.e. compiled) by the execution of an assembler mnemonic (for example JMP,). Every mnemonic acts like a mini-compiler to place its own code in the dictionary entry currently being constructed.

The whole process of assembly is therefore simply a normal interpretation of the text but with the ASSEMBLER as the CONTEXT vocabulary (it is searched first). This means that all of FORTH is also available during assembly, for modifying or manipulating the stack contents, making the whole process extremely flexible.

Two points should be noted here which may help with writing the body of a machine code definition.

Firstly, the assembly mnemonic always ends with a comma, which serves three purposes:

(a) It marks the end of a group of words which would be a single line of conventional assembly code.

(b) The FORTH word <,> compiles values into the dictionary and so, by analogy, the comma indicates the point at which bytes of code are actually compiled.

(c) It ensures that there is no confusion between assembly mnemonics and

i) hexadecimal numbers (e.g. ADC)
or ii) other assembler words (e.g. SEC).

Secondly, the operand (for example POP) is placed before the mnemonic, rather than afterwards as in the conventional assembler. This is always the case whether the operand is a literal numeric value or, as in the above example, a symbolic one. The operand, in either case, causes its value to be placed on the stack ready to be compiled, together with the opcode, by the execution of the assembly mnemonic.

The definition is concluded by END-CODE which, in addition to performing a number of error checks, restores the original CONTEXT vocabulary.

10.3 Machine code labels

All machine code routines must terminate with a jump to existing machine code which will, directly or indirectly, execute the code of NEXT . NEXT is used at the end of every FORTH word and its basic function is to transfer execution to the next word in the sequence of words which make up a dictionary entry. It is described more fully in Appendix B. Valid terminating jumps to the routines are given in the following table:

Routine	Description
NEXT	Transfer execution to the next word in the sequence.
PUSH	Push the accumulator (as high byte) and one byte from the return stack as a new number on the computation stack, and execute NEXT .
PUT	Replace the current top stack item from the accumulator and return stack (as for PUSH) and execute NEXT .
PUSH0A	Push zero (as high byte) and the accumulator (low byte) to the stack and execute NEXT .
POP	Drop the top stack item and execute NEXT .
POPTWO	Drop the top two stack items and execute NEXT .

In the assembler these routines are given labels (as defined constants).

In addition to these terminating routines there is a subroutine which may be used from within a code definition. It is given the label SETUP and acts to transfer up to four items from the stack to a scratch-pad area in page zero. On entry the accumulator should contain the number of items to be transferred. On return from the subroutine the Y-register will contain zero and the value in the accumulator will be doubled; in other words it contains the number of bytes transferred from the stack. The byte immediately preceding the scratchpad area will also contain the number of bytes transferred.

The actual addresses of these routines will be needed if machine code is to be hand-assembled, as described in chapter 5. Since Acornsoft FORTH is provided in relocatable form these addresses cannot be given as absolute values but must be stated relative to points

within the FORTH system. The address values can be written (in hex) as follows:

```
NEXT      6A    +ORIGIN
PUSH      63    +ORIGIN
PUT       65    +ORIGIN
PUSH0A   2F0    +ORIGIN
POP       1DF   +ORIGIN
POPTWO    1DD   +ORIGIN
SETUP     8A    +ORIGIN
```

If any of the allowed Operating System routines are needed they may be included in the assembler as defined constants, as in the following examples:

```
ASSEMBLER DEFINITIONS HEX

FFE0 CONSTANT OSRDCH
FFEE CONSTANT OSWRCH
FFF1 CONSTANT OSWORD
FFF4 CONSTANT OSBYTE

FORTH DEFINITIONS DECIMAL
```

An alternative method is given in section 10.9.

10.4 The registers

Although Acornsoft FORTH uses all the processor registers, they may also be used in a machine code definition, provided that certain conventions are observed. These conventions are implementation-dependent and should not be assumed to apply to all versions of FORTH. The conventions used here, however, are not likely to be too different from those in the majority of other implementations for the 6502.

Whenever a machine code definition is executed, it is entered directly from the routine NEXT . The processor registers will therefore always be subject to the same conditions, which are as follows:

1) The contents of the accumulator are undefined. The accumulator may be used freely in any machine code definition.

2) The Y-register contains zero and may be used without restriction.

3) The X-register points to the low byte of the most accessible number on the computation stack. It may be used provided that its contents are first saved (in XSAVE) and then restored at the end of the routine.

4) The 6502 hardware stack is used as the return stack. The stack pointer contains the address of the first unused byte beyond the end of the stack (normal for the 6502). Its contents should not normally be changed across a code definition, otherwise FORTH return addresses may be lost.

5) The contents of the processor status register, with the exception of the decimal flag, are undefined and may be used freely. The decimal flag is clear (so that the processor is in binary mode) and must be returned in that state.

In addition to all the processor registers, Acornsoft FORTH uses a number of special registers located in page zero. The assembler contains, again as defined constants, the addresses of all the zero page registers used. The addresses of these registers are not affected by relocating FORTH and are given in the following table.

Name	Hex Addr	Size	Format	Comments
N	60	1+8 bytes	xXXXXXXXX	scratchpad
XSAVE	68	2 bytes	XX	temp. for X-register
W	6A	1+2 bytes	xXX	code field pointer
IP	6C	2 bytes	XX	interpretive pointer
UP	6E	2 bytes	XX	user variable pointer

Note that N and W use one extra byte before their stated addresses.

The scratchpad area is used by SETUP , as described earlier, to hold up to four items transferred from the computation stack. In general it is used to hold small amounts of data at a known, fixed location for use by machine code routines. Since this area is used by many of the system words its contents will change frequently and should not be assumed to be retained from one definition to another.

The single byte at XSAVE is reserved for the temporary storage of the contents of the X-register (which normally holds the computation stack pointer) if it is needed within a code definition.

The user variable pointer UP contains the address of the start of the area in which the values of the user variables are stored. The subject of user variables is described in chapter 5.

It now remains to explain the actions of the final two registers, W and IP . These two are fundamental to the operation of FORTH and should be used with great care. If their contents are changed inadvertently the system will 'go away' and require at least a WARM start to recover. They are used, and modified, by NEXT .

The code field pointer W , is used to hold the address of the code field of the currently executing routine. The interpretive pointer IP , always holds the address of the routine to be executed at the conclusion of the current one. The functions of IP , W and NEXT are described in more detail in Appendix B.

10.5 Opcode mnemonics

The assembly mnemonics are divided into two main groups according to their addressing modes. The first group contains the one-byte codes, with only a single (implied) addressing mode. The second group includes all those which have a number of different addressing modes. The mnemonics used are standard for the 6502 except that they all have an additional terminating comma.

10.5.1 Single-mode mnemonics

The single-mode mnemonics are given in the following table:

```
BRK,   CLC,   CLD,   CLI,   CLV,
DEX,   DEY,   INX,   INY,   NOP,
PHA,   PHP,   PLA,   PLP,   RTI,
RTS,   SEC,   SED,   SEI,   TAX,
TAY,   TSX,   TXA,   TXS,   TYA,
```

When one of these is executed it simply compiles the corresponding opcode into the dictionary.

The following example will execute to push a zero to the computation stack:

```
CODE ZERO
   TYA,  PHA,
   PUSH JMP,
END-CODE
```

It makes use of the fact that the Y-register always contains zero on entry to a code routine. The accumulator and the bottom byte of the hardware stack are both set to zero, and it is these two bytes (accumulator first) which are transferred to the computation stack by PUSH .

10.5.2 Multi-mode mnemonics

The multi-mode mnemonics are as follows:

```
ADC,   AND,   ASL,   BIT,   CMP,
CPX,   CPY,   DEC,   EOR,   INC,
JMP,   JSR,   LDA,   LDX,   LDY,
LSR,   ORA,   ROL,   ROR,   SBC,
STA,   STX,   STY,
```

Each of these normally needs an operand which must previously have been placed on the stack. If no address mode is given the operand is assumed to be an address. In this case absolute (16-bit) or zero page (8-bit) modes are chosen, depending on both the magnitude of the given address and on which modes are legal for the particular opcode.

When any of these mnemonics is executed it compiles the appropriate opcode and operand into the dictionary.

10.5.3 Addressing modes

The symbols in the following table are used to specify the addressing mode to be used:

Symbol	Mode	Expected Operand
none	memory	zero page or absolute
.A	accumulator	none
#	immediate	8-bit literal value
,X	indexed X	zero page or absolute
,Y	indexed Y	zero page or absolute
X)	indexed indirect,X	zero page
)Y	indirect indexed,Y	zero page
)	indirect	absolute

The way in which the addressing modes are used is illustrated below. The FORTH version is compared with the corresponding form for a conventional assembler:

FORTH	Conventional assembler
ADDR JSR,	JSR ADDR
.A ASL,	ASL A
4 # CMP,	CMP #4
ADDR ,X LDA,	LDA ADDR,X
ADDR ,Y STA,	STA ADDR,Y
ADDR X) ADC,	ADC (ADDR,X)
ADDR)Y LDA,	LDA (ADDR),Y
ADDR) JMP,	JMP (ADDR)

The following two examples make use of the OSWRCH routine, at address &FFEE, of the BBC Microcomputer Operating System. This displays the contents of the accumulator as an ASCII character. Since this routine does not change the contents of any other register (except the C, N, V and Z flags), no special precautions need be taken to restore them.

The word CHAR displays the contents of the byte at address 128 (&80) as an ASCII character.

```
HEX

CODE CHAR  ( ... )
       80 LDA,
       FFEE JSR,
       NEXT JMP,
END-CODE

DECIMAL
```

The following sequence typed at the keyboard will then display an 'A' on the VDU (note that we are back in decimal mode):

```
65 128 C!
CHAR
```

Modifying the routine to use the 'indirect indexed,Y' addressing mode will allow the display of a character whose address is given in the two bytes at address &80:

```
HEX

CODE (CHAR)
       80 )Y LDA,
       FFEE JSR,
       NEXT JMP,
END-CODE

DECIMAL
```

where we have again made use of the fact that the Y-register contains zero on entry to the code.

If you type the following, you should find that this time a 'B' will be displayed. (The character displayed is, in fact, the first letter of the name of BASE .)

```
' BASE NFA 1+   128 !
(CHAR)
```

10.6 Accessing the stacks

Most machine code routines will need to access the computation stack, the return stack or both. A separate technique is needed for each stack.

You may recall that in section 4.1 we mentioned that the stacks grow towards lower addresses. Throughout the rest of this manual we have been able to ignore this detail and have referred to the most accessible item as being at the 'top' of the stack. In the case of machine code we can not afford to forget the actual structure of the stack. The FORTH assembler therefore refers to the most accessible stack item as the 'bottom' value. In both stacks the more significant byte is found at the higher address.

10.6.1 The computation stack

This stack is located in page zero and is usually addressed in 'zero page ,X' mode with the X-register as the stack pointer. The X-register normally contains the address of the low order byte of the bottom stack item. The bottom two stack items are accessed so frequently that the special words BOT and SEC are provided. Their meaning and the corresponding stack addresses are illustrated in the following table:

Stack	Addressing
Second, high byte	3 ,X or SEC 1+
Second, low byte	2 ,X or SEC
Bottom, high byte	1 ,X or BOT 1+
Bottom, low byte	0 ,X or BOT

As an example we can consider the definition of 4* which performs a fast multiply by four. It uses the same method as for the word 2* which is provided in the system and is considerably faster than either of the possible high-level definitions:

```
: 4*  4  *  ;     or     : 4*  2* 2* ;
```

The definition is

```
CODE 4* ( n1 ... n2 )
       BOT  ASL,    BOT 1+ ROL,
       BOT  ASL,    BOT 1+ ROL,
       NEXT JMP,
END-CODE
```

10.6.2 The return stack

The return stack is located in the 6502 hardware stack, in page one. Unlike the computation stack pointer, the hardware stack pointer contains the address of the first unused byte below the bottom of the stack. Since the hardware stack pointer is only an eight-bit register it can only contain the low byte of the stack address. When machine code 'push' or 'pull' instructions are used the processor automatically adds a high byte of 01 to access the correct page, and allows for the stack pointer indicating one byte below the bottom item. The sequence

```
PLA, PLA,
```

will, therefore, simply remove the bottom item (low byte first).

If it is necessary to manipulate the return stack contents, the hardware stack pointer can be transferred to the X-register. This normally contains the computation stack pointer so its contents must first be saved and later restored, using the temporary storage location XSAVE. The adjustments made to the stack pointer by the processor are not automatically added to any other register and must be included explicitly. The special address modifier RP) is provided to access the bottom byte of the return stack. It is equivalent to 101 ,X so that

```
RP) LDA,
```

will load the accumulator with the bottom byte of the return stack.

The following code will, for example, load the accumulator with the low byte of the second return stack item (i.e. the third byte from the bottom of the hardware stack):

```
XSAVE STX,  TSX,
RP) 2+ LDA,
XSAVE LDX,
```

The first line of this code saves the contents of the X-register (in XSAVE) and transfers to it the contents

of the hardware stack pointer. The third line restores the original contents of the X-register. These two lines will normally enclose any reference, or group of references, to the return stack.

10.7 Conditional structures

The conditional branching structures provided with the assembler are similar to those used in high-level code. They are distinguised by having terminating commas:

```
IF, ... ELSE, ... THEN,
BEGIN, ... AGAIN,
BEGIN, ... UNTIL,
BEGIN, ... WHILE, ... REPEAT,
```

The main difference is that the assembler versions test various bits in the status register, rather than stack values. The tests provided are as follows:

Test	Meaning	True for
CS	Carry set	C=1
VS	Overflow set	V=1
0<	Less than zero	N=1
0=	Equals zero	Z=1
CS NOT	Carry clear	C=0
VS NOT	Overflow clear	V=0
0< NOT	Not less than zero	N=0
0= NOT	Not equal to zero	Z=0

The following examples illustrate some uses of the conditional branches. The first simply adds 3 to the bottom stack item.

```
CODE 3+  ( n1 ... n2 )
   BOT LDA,  CLC,  3 # ADC,
   0= IF,   BOT 1+ INC,   THEN,
   NEXT JMP,
END-CODE
```

The second example illustrates the use of IF, ... ELSE, ... THEN, and the nesting of conditionals. It increments or decrements the second stack item, depending on the sign of the number on the bottom of the stack.

```
CODE INC/DEC  ( n1\n2 ... n3 )
   BOT 1+ LDA,
   0< IF,    SEC LDA,
             0= IF,   SEC 1+ DEC,   THEN,
             SEC DEC,
      ELSE, SEC INC,
             0= IF,   SEC 1+ INC,   THEN,
      THEN,
   POP JMP,
END-CODE
```

The next two examples implement double-precision fetches and stores. They work in the same way as <@> and <!> except that the value occupies four bytes starting at addr.

```
CODE 2@  ( addr ... nd )
   1 # LDA,  SETUP JSR,  3 # LDY,
   BEGIN,  N )Y LDA,
          DEX,  BOT STA,
          DEY,  0=
   UNTIL,
   NEXT JMP,
END-CODE

CODE 2! ( nd\addr ... )
   1 # LDA,  SETUP JSR,
   BEGIN,  BOT LDA,  N )Y STA,  INX,
           INY,  4 # CPY,  0=
   UNTIL,
   NEXT JMP,
END-CODE
```

10.8 Use of ;CODE

Chapter 9 showed how new data structures can be created with the aid of CREATE and DOES> . The high-level code following DOES> defines the action of all members of the family of words created by a particular defining word. One disadvantage, particularly for data structures with complicated actions, is the length of time taken to perform these actions when written in high-level code. It would be useful to be able to define the actions in terms of machine code for cases where speed is important. Needless to say, FORTH

provides the method in the form of ;CODE . It is used as

```
: NAME  CREATE ... ;CODE ... END-CODE
```

The general principles are the same as in the use of CREATE and DOES> . The words following CREATE compile the parameters peculiar to the family member, and all members execute the code following ;CODE in the defining word.

During compilation ;CODE terminates the compilation of the colon-definition and also starts the assembly section. It therefore combines many of the actions of the two words <;> and CODE . When a defining word (such as NAME) is executed to create a new family member a third aspect of ;CODE comes into action. The contents of the code field of the new member are changed to contain the address of the machine code following ;CODE .

A couple of examples should help to clarify the use of ;CODE. The first of these allows the creation of double-precision variables:

```
: 2VARIABLE  ( ... addr )
       CREATE
       0 , 0 ,
;CODE          ( to push the PFA to the stack )
       CLC,  W LDA,  2 # ADC,  PHA,
       TYA,  W 1+ ADC,
       PUSH JMP,
END-CODE
```

This defining word may be used as

```
2VARIABLE DOUBLE
```

to create a double-precision variable, named DOUBLE , which is initialised to zero. When DOUBLE is executed it will use the machine code following ;CODE in 2VARIABLE to leave on the stack the address of the first byte of the parameter field of DOUBLE . A value may be fetched from or stored to DOUBLE using 2@ or 2! (these were defined earlier).

The code used by 2VARIABLE is identical to that of

VARIABLE since both return the same address. The only difference is in the size of the parameter area reserved. Memory usage can be reduced without any loss of execution speed by using the following alternative definition:

```
: 2VARIABLE
    VARIABLE  0 ,  ;
```

However, this is not a very good illustration of how ;CODE works!

As a second example we can look at a possible definition of 2CONSTANT which, not surprisingly, creates double-precision constants. Since such a word must copy four bytes to the stack it cannot use the machine code of CONSTANT . The following version is designed for clarity and to minimise memory usage rather than for maximum execution speed, though in fact it is still quite fast. It requires the prior machine code definition of 2@ :

```
: 2CONSTANT  ( ... nd )
    CREATE  , ,
    ;CODE
       DEX, DEX,
       CLC,  W LDA,  2 # ADC,  BOT STA,
       TYA,  W 1+ ADC,  BOT 1+ STA,
       ' 2@ JMP,
    END-CODE
```

The code is very similar to that of 2VARIABLE , except that the PFA is placed directly on the stack rather than by the use of PUSH . The code is terminated by a jump to the code of 2@ which, after replacing the address on the stack by the double-precision value stored there, leads back to NEXT .

10.9 Macro assembly

A macro is a section of machine code which is used several times. It is used rather like a subroutine. Unlike a subroutine, whose code appears only once, the code of a macro is inserted into the body of the machine code at each point where it is needed. The advantage of a macro is the increase in execution

speed, since a subroutine call and the corresponding return are not needed. This speed increase is gained at the cost of increasing the amount of memory used (unless the assembled code is three bytes or less in length).

Creating a macro in FORTH is very similar to compiling a colon-definition. The assembly instructions are compiled and are not executed until the macro is called at a later time. The main difference is that a macro must be created in the assembler vocabulary, as in the following examples:

```
ASSEMBLER DEFINITIONS HEX

MACRO NEWL  ( ... )
   FFE7 JSR,  ;

MACRO WRCH  ( ... )
   FFEE JSR,  ;
```

These two macros assemble subroutine calls to the operating system's OSNEWL and OSWRCH respectively. Since all user-accessible operating system routines are only used as subroutines, this method is a good alternative to the definition of their addresses as assembler constants.

The following pair of macros allow the assembly of a definite loop similar to the high-level DO ... LOOP . They are used in the form

```
n TIMES, ... LOOP,
```

The loop index is stored in the Y-register which is therefore not available for other purposes within the loop body. The index is limited to the range 0 to 255 and the loop terminates when the index reaches zero.

```
MACRO TIMES,
   # LDY,  BEGIN,  ;

MACRO LOOP,
   DEY,  0= UNTIL,  ;
```

At the conclusion of the definition of one or more macros you should remember to change the CURRENT

vocabulary back from assembler by, for example,

```
FORTH DEFINITIONS  DECIMAL
```

The example below uses all four of the macros we have just defined to present the alphabet on the display.

```
CODE ALPHABET ( ... )
   NEWL
   64 # LDA,      ( ASCII code of @ )
   26 TIMES,
       CLC,  1 # ADC,  WRCH
   LOOP,
   NEWL
   NEXT JMP,
END-CODE
```

10.10 Errors

A comprehensive set of error checks is included with the assembler. They will not, of course, guarantee that the code definition will perform correctly but they do prevent errors of syntax.

The error checks may be divided into five main groups:

1) Each mnemonic, when executed, ensures that a valid addressing mode is used. An error message will be generated, for example, by

   ```
   BOT )Y LDY,
   ```

2) The conditional structures include checks that they are correctly paired and nested. Sequences such as the following are not allowed.

   ```
   ... IF,   ...  UNTIL,
   ```

3) Within the conditionals a check is made that any branch is within range. An error message is given, for example, by

   ```
   ... IF,   256 ALLOT   THEN,
   ```

4) The number of stack items must not change across a code definition. This can be caused, for

example, by an address being left on the stack or by a conditional structure not being completed.

5) A macro definition must be created in the assembler vocabulary. If it is not, an error message is given and the definition will not be accepted.

In all cases except the last the definition will be left in an incomplete (and non-executable) form, with the CONTEXT vocabulary set to ASSEMBLER.

11 Execution vectors and recursion

11.1 Execution vectors

The action of a dictionary entry is normally fixed at the time of its definition. The word may be redefined at a later stage and from that point onwards all references to the word will use its new definition. All earlier references, however, will still use the old meaning. It would be useful to have a simple method of changing a definition in such a way that all previously compiled references were also switched to using the new version. One method of doing this is by the use of execution vectors.

The idea of an execution vector can best be described by means of an example. Suppose we define a few words like this:

```
: ICAN
   ." I CAN DO "  ;

: 1STWORD
   ICAN  ." THIS"  ;

: 2NDWORD
   ." OR " ICAN ." SOMETHING ELSE"  ;
```

We now define a word DEMO which executes 1STWORD . We do not write this as a direct colon-definition, but make the execution work in a more indirect way, via the contents of a variable:

```
VARIABLE ACTION
' 1STWORD CFA   ACTION !
```

ACTION now contains the execution address of 1STWORD , so DEMO can be written as

```
: DEMO
   ACTION @  EXECUTE ;
```

Executing DEMO will type the message

I CAN DO THIS

as expected. The point of working in such a roundabout manner is that we can now change what DEMO does simply by changing the contents of the variable ACTION . For example,

```
' 2NDWORD CFA  ACTION !
```

will cause DEMO to type

OR I CAN DO SOMETHING ELSE

The significance of this is that we have been able to change the action of DEMO after it has been defined. Furthermore all compiled references to DEMO will also change their action. DEMO can be made to execute any other word whose execution address is placed in the variable ACTION . This variable is known as the execution vector for DEMO .

This facility is very useful but there are a number of problems associated with the simple method described above. Firstly it is rather inefficient in its use of memory since three dictionary entries are needed for each vectored routine. These are the vectored word itself, the routine that it is to execute and the variable used to contain the execution vector. It can be made more efficient by removing the need to have a separate dictionary entry for the variable. Secondly, and more importantly, it is possible to forget to put a sensible address in the execution vector. This can cause quite spectacular crashes!

In Acornsoft FORTH these problems are resolved by the inclusion of a new defining word, EXVEC: to create an execution-vectored word without the need to use an additional variable. All words created by EXVEC: are automatically given a default vector to a routine which

prints an error message. No harm can be done if you forget to specify the action of the vectored word.

Specifying the routine to be executed is known as assignment and the words ASSIGN and TO-DO are provided to simplify the process.

The previous example can now be expressed more simply, as follows. First

```
FORGET ACTION
```

since we shall not be needing it again. Then create the vectored word DEMO by

```
EXVEC: DEMO
```

Try executing DEMO at this stage to see the default error message, and then

```
ASSIGN DEMO  TO-DO 1STWORD
```

At any time you may reassign DEMO to execute another word, for example,

```
ASSIGN DEMO  TO-DO 2NDWORD
```

A number of words in the nucleus dictionary are already vectored and these are given in the following list, together with the words they execute after a cold start:

Word	Executes
KEY	(KEY)
EMIT	(EMIT)
NUM	(NUM)
MESSAGE	MSG#
R/W	TR/W or DR/W
UPDATE	(UPDATE)
ABORT	(ABORT)
CREATE	(CREATE)

As an example we can redefine EMIT , which sends characters to the display, so that it will display control codes as well as acting on them:

```
: ^EMIT
   DUP   32 <
   IF    DUP >VDU
         64 +   ." ^"
   THEN
   (EMIT)  ;

ASSIGN EMIT  TO-DO ^EMIT
```

All control codes (less than ASCII 32) are then displayed with a preceding ^ e.g. a 'bell' (control G) is displayed as ^G .

The normal action can be restored either by a cold start, when all vectored words are initialised to their default actions, or less drastically by:

```
ASSIGN EMIT  TO-DO (EMIT)
```

The ability to redefine the action of the error message routine MESSAGE gives a simple method of implementing full error message reporting.

Disk-based systems are provided with all the error messages in screens one and two of the system disk. The error message number corresponds to the screen line in which the appropriate message is found, counting from line zero of screen 1. Error message 17, for example, is found on the second line of screen 2. Producing full error messages is particularly simple in this case, as shown below.

```
: ERRMESS
   ?DUP IF                                ( non-zero )
           16 /MOD  1+              ( convert msg no. )
                               ( to  line  & screen )
           .LINE
        THEN  ;

ASSIGN MESSAGE  TO-DO ERRMESS
```

This approach is not suitable for tape-based systems and the following method would be preferred. Instead of reading the error messages from mass storage screens, the messages are kept in the dictionary. The correct message is selected by the use of a CASE structure as described in chapter 9.

```
CASE:  CREATE SMUDGE ]
       DOES>  OVER + + @EXECUTE ;
```

Each error message is written as a colon-definition:

```
: ESTK ." Stack Empty" ;
: DFUL ." Dictionary Full" ;
: AMOD ." Address Mode" ;
: REDF ." Isn't Unique" ;
: PARA ." Parameter? " ;
: SCR# ." Screen Number?" ;
: FSTK ." Stack Full" ;
: EFIL ." File Error" ;
: R/WE ." Read/Write Error" ;
: EOLN ." End-of-Line?" ;
: /ZER ." Division by 0" ;
: EVEC ." Undefined Vector" ;
: EBRN ." Branch too Long" ;
: CURV ." CURRENT Vocabulary?" ;
: COMP ." Compilation Only" ;
: EXEC ." Execution Only" ;
: COND ." Conditionals?" ;
: DEFN ." Unfinished" ;
: PROT ." Protected" ;
: LODG ." LOADing Only" ;
: EDSC ." Off Editing Screen" ;
: CRNT ." Not in CURRENT" ;
: EMEM ." Memory Clash" ;
```

These messages are shortened versions of those appearing in chapter 13. You may of course replace them with the full versions, or with any other wording that you prefer.

```
CASE: ERRMESS
       NOOP  ( do nothing for error zero )
       ESTK DFUL AMOD REDF PARA SCR# FSTK FILE
       R/WE EOLN /ZER EVEC EBRN CURV NOOP NOOP
       COMP EXEC COND DEFN PROT LODG EDSC CRNT
       EMEM       ;

ASSIGN MESSAGE  TO-DO ERRMESS
```

A useful variation on a system ABORT is one which also

prints out the stack contents, as in the following example:

```
: .SABORT
   CR ." STACK IS" .S   (ABORT)  ;

ASSIGN ABORT  TO-DO .SABORT
```

Further examples of the use of vectored routines include the redefining of NUM (the numeric interpreter used by INTERPRET) so that other number formats may be accepted by the system and the vectoring of UPDATE, as mentioned in section 8.3.6. A particularly useful application is in the handling of forward references in recursive definitions, which are the subject of the next section.

11.2 Recursion

A recursive routine is one which uses itself as part of its own definition. As a consequence of the security built into the compiler it is not normally possible for a FORTH word to contain a reference to itself. The name header of the word currently being defined is made unrecognisable in a dictionary search, using SMUDGE . It is restored by a second use of SMUDGE at the successful conclusion of the definition. This is a very effective method of preventing execution of an incomplete or erroneous definition but has the side effect that no word may contain a direct reference to itself.

A popular solution is to define the word MYSELF before writing any recursive definitions. This word calculates the execution address of the word currently being defined and compiles it into the definition. It by-passes the dictionary search normally used to find the address of a word and is defined as

```
: MYSELF
   LAST      ( addr of lst byte of last word )
   PFA CFA   ( convert to execution addr )
   ,   ;     ( compile the addr )
IMMEDIATE    ( must execute during compilation )
```

Using this method we can define a recursive counting routine as follows:

```
: COUNTS ( n ... )
   ?DUP IF DUP 1-
          MYSELF
          CR  .
        THEN  ;
```

In Acornsoft FORTH a special form of colon-definition is provided to aid the writing of recursive routines. If <:> and <;> are replaced by their recursive forms, R: and R; the previous definition may be written as

```
R: COUNTS ( n ... )
    ?DUP IF
            DUP 1-
            COUNTS
            CR  .
         THEN  R;
```

The definitions of R: and R; could be written as

```
: R:
   : SMUDGE ;

: R;
   [COMPILE] ;
   SMUDGE  ;
IMMEDIATE
```

They allow references by name to the definition currently being compiled at the expense of a certain amount of compiler security. If an error occurs during compilation, the incomplete definition will be left in an executable form.

The next example uses a recursive definition to calculate factorials:

```
R: (FACT)  ( n1\n2 ... n3 )
    ?DUP IF
           DUP ROT * SWAP 1-
           (FACT)
         THEN ;
```

```
: FACT ( n ... factn )
   DUP 0<  OVER 7 >
   OR  5 ?ERROR
   1 SWAP  (FACT)
   .  ;
```

The calculation of the factorial is performed by (FACT) which leaves the result on the stack. Error checking is confined to FACT . Factorials are not defined for negative numbers and an attempt to calculate the factorial of a number greater than 7 will result in an arithmetic overflow when single-precision numbers are used.

The following alternative definition makes use of MD* , which was defined in section 4.4, to extend the calculation to give a double-precision answer. This will allow the calculation of factorials of numbers up to and including 12:

```
R: (FACT)
    ?DUP IF
            DUP 1-
            >R  MD*  R>
            (FACT)
         THEN  ;

: FACT  ( n ... )
   DUP 0<  OVER 12 >
   OR  5 ?ERROR
   1 0 ROT  (FACT)
   D.  ;
```

11.3 Forward references

The normal compiling process can only compile the address of a word that has previously been defined and can be found in the dictionary. The use of execution vectors provides a simple solution to the problem of including in a definition a reference to a word which has not yet been written. Normally this type of difficulty can be avoided by defining the words of an application in the correct sequence. There are cases, however, when this is not possible. An example would be where two words are required to call each other in a form of mutual recursion - whichever word is defined

first, it must still contain a reference to the other.

One easy solution is to define the 'second' word as a vectored routine before writing the definition of the 'first', as in the following example:

```
EXVEC: PART-TWO

: PART-ONE
   CR CR ." I am part one"
   CR  ." I call part two"
   PART-TWO ;
```

The full definition of the second part can then be written:

```
: PARTWO
   CR CR ." I am part two"  CR
   ." Do you want to go to part one? "
   KEY 89 =                    ( Y key pressed? )
   IF   PART-ONE   THEN  ;
```

Finally PART-TWO is assigned to execute PARTWO :

```
ASSIGN PART-TWO  TO-DO PARTWO
```

A more useful example is in the graphics routine provided as a demonstration and listed in chapter 12.

12 Graphics and sound

The graphics and sound facilities provided on the BBC Microcomputer are extremely powerful and it would be impractical to cover all possibilities in this manual. What we can do, however, is to look at a range of simple examples which illustrate some of the basic techniques that are available, and these will be the subject of this chapter. In addition, the examples will develop some of the ideas introduced in earlier chapters.

12.1 Graphics

Since many applications do not require advanced graphics, the only graphics-oriented word provided in the nucleus dictionary is PLOT . The extensible nature of FORTH means that it is very easy to add the specific words required to tailor the system to your exact requirements.

The versions of Acornsoft FORTH provided on tape or disk are loaded into RAM, part of which would be used by the higher resolution graphics modes. These modes cannot, therefore, be used and the highest resolution available from FORTH is mode 4. Most of the examples will be given in mode 5, in which four-colour graphics are possible. Apart from this limitation the full facilities of the computer are available to you.

12.1.1 PLOT

The word PLOT can be used, in any available graphics mode, to plot points, draw lines, move the graphics cursor or fill triangles with a selected colour. It behaves similarly to the PLOT command as described in the BBC Microcomputer User Guide. It is used in the form:

```
K  X  Y  PLOT
```

where X and Y are the coordinates of a point, and the value of K determines the type of plotting action. Thus,

```
5 MODE
4 200 200 PLOT
```

will move to the point (200,200) without plotting anything,

```
5 700 500 PLOT
```

will draw a line to (700,500) in the current graphics foreground colour, and

```
21 200 500 PLOT
```

will draw a dotted line to (200,500).

We can write definitions to draw simple shapes: the example that follows plots a triangle from the coordinates on the stack:

```
: TRIANGLE (x0\y0\x1\y1\x2\y2 ... )
    2DUP >R >R          ( save coordinates of last point)
   4 ROT ROT PLOT       ( move to first vertex )
   6 ROT ROT PLOT       ( plot 1st side )
   6 ROT ROT PLOT       ( plot 2nd side )
   6  R>  R>  PLOT ;    ( and back )
```

We can use this definition as, for example:

```
5 MODE
50 50  500 300  200 800  TRIANGLE
```

The triangle is plotted using K=6, i.e. the lines are drawn in 'logical inverse colour'. This means that plotting the figure a second time in the same place will cause it to disappear. This technique can be used to display simple animated graphics:

```
5 CONSTANT DISPX  5 CONSTANT DISPY
                            ( X and Y displacements )
VARIABLE X0       VARIABLE Y0
VARIABLE X1       VARIABLE Y1
VARIABLE X2       VARIABLE Y2

: 1STPOS ( ... )
   20 X0 !   0 Y0 !
    0 X1 !  20 Y1 !
   50 X2 !  50 Y2 ! ;

: THISPOS ( ... x0\y0\x1\y1\x2\y2 )
   X0 @ Y0 @  X1 @ Y1 @  X2 @ Y2 @ ;

: NEXTPOS ( ... )
   DISPX X0 +!  DISPY Y0 +!
   DISPX X1 +!  DISPY Y1 +!
   DISPX X2 +!  DISPY Y2 +! ;

: DELAY ( ... )
   1000 0 DO LOOP ;

: DISPLAY ( ... )
   1STPOS
   BEGIN  THISPOS TRIANGLE     ( plot triangle )
          THISPOS            ( keep its coords )
          NEXTPOS DELAY        ( do calcs etc. )
          TRIANGLE           ( unplot triangle )
          X2  @   1000 >   ( at top of screen? )
   UNTIL ;
```

The application may be used in mode 4 or 5; for example

```
4 MODE  DISPLAY
```

The next example is a listing of the graphics demonstration provided with the system. In addition to using PLOT, it shows an application of mutually recursive routines and the use of execution vectors to allow forward references. Recursion is a very useful technique for the generation of complex displays:

```
FORTH DEFINITIONS DECIMAL

: CASE:
   CREATE  SMUDGE ]
   DOES>   OVER + + @EXECUTE ;

6    CONSTANT N      ( number of steps )
1024 CONSTANT H0     ( size of picture )

VARIABLE H           ( scale for current step )
VARIABLE X0          ( initial plot coords )
VARIABLE Y0
VARIABLE X           ( coords for next plot )
VARIABLE Y

: XYLINE ( ... )     ( draw line to X,Y )
   5  X @  Y @  PLOT ;

: X+  (  ...  )      ( line in +ve X-dir )
   H @          X +!  XYLINE ;
: Y+  (  ...  )      ( line in +ve Y-dir )
   H @          Y +!  XYLINE ;
: X-  (   ...)       ( line in -ve X-dir )
   H @ NEGATE   X +!  XYLINE ;
: Y-  (  ...  )      ( line in -ve Y-dir )
   H @ NEGATE   Y +!  XYLINE ;

EXVEC: DRAW-SIDES
: 3SIDES ( n1\n2...n1) ( draw 3 sides of square)
   OVER DUP
   IF    1- SWAP DRAW-SIDES
   ELSE  2DROP
   THEN ;

: OR1 ( n... ) ( these are the 4 possible
                 3-sided figures )
   3 3SIDES X-   0 3SIDES Y-
   0 3SIDES X+   1 3SIDES DROP ;
: OR2 ( n... )
   2 3SIDES Y+   1 3SIDES X+
   1 3SIDES Y-   0 3SIDES DROP ;
: OR3 ( n... )
   1 3SIDES X+   2 3SIDES Y+
   2 3SIDES X-   3 3SIDES DROP ;
: OR4 ( n... )
   0 3SIDES Y-   3 3SIDES X-
   3 3SIDES Y+   2 3SIDES DROP ;
```

```
CASE: (SIDES) ( n1\n2... ) ( draw one of the 4 )
   OR1  OR2  OR3  OR4  ;
ASSIGN DRAW-SIDES   TO-DO  (sides)

: INITIALISE ( ... )
   H0   DUP  H !
   2/ DUP  X0 !  Y0 !
   4 MODE ;

: XYSET ( ... )
   H @  2/ DUP  H !
   2/ DUP  X0 +!  Y0 +!
   4  X0 @  Y0 @
   2DUP  Y !  X !
   PLOT ;

: PLOT-IT ( ... )
   INITIALISE
   0
     BEGIN  1+
            XYSET
            0 3SIDES
            DUP  N =
     UNTIL  DROP
   KEY DROP  12 EMIT ;
```

When PLOT-IT is executed, INITIALISE sets the initial line lengths (in H) and the starting coordinates for the plot (in X0 and Y0). It also sets mode 4 graphics. The main work is done in the BEGIN ... UNTIL loop. A step counter, initially zero, is kept on the stack and incremented by 1 each time round the loop. This process continues until the loop counter reaches the value of the constant N when the loop is left and the counter DROPped. The routine then waits for any key to be pressed (so that you can admire the result) before clearing the screen.

Within the loop XYSET sets up the correct initial conditions for each step. The contents of H are halved (since at each step all line lengths are half the length used in the previous step) and the contents of X0 and Y0 are adjusted to the correct starting position. These coordinates are also stored in X and Y and the graphics cursor is moved to this point, by a PLOT with K=4.

The actual job of drawing each section of the picture is done by 3SIDES and this is where things start to get interesting (much better than saying 'complicated', don't you think?). 3SIDES expects two values on the stack, the step counter beneath a number in the range 0 to 3, which indicates the orientation of the 3-sided figure to be plotted. It leaves the step counter on the stack for later use. Strictly speaking, this is bad practice in FORTH since a word should normally destroy all stack values that it uses. However, this would involve more involved manipulations of the stack and it is simpler in this case to leave the stack 'dirty'.

The recursive nature of this example can be seen by examining the action of 3SIDES. It calls DRAW-SIDES which has been assigned to execute one of the cases of (SIDES). These are the four possible orientations, OR1 to OR4 and each of these makes calls back to 3SIDES to complete the recursion.

Each time that 3SIDES is executed a copy of the step count is reduced by 1 and further calls to 3SIDES are made, until a depth of call is reached where the count has been reduced to zero. Thus the first time that 3SIDES is called is with a step count of 1 and only one level of recursion is reached. The figure plotted is therefore simply three sides of a square. The next time it is called is with a count of 2, so that two levels of recursion are used, and a 3-sided figure is drawn on each of the 3 sides of the first shape (with each side unit half the length of those in the previous figure).

The various stages can be examined by use of the following definition.

```
: STEP ( n ... )
   INITIALISE
   DUP 0 DO XYSET LOOP
   0 3SIDES DROP ;
```

Typing 1 STEP, 2 STEP, etc. will show the various stages separately.

The most complex display that can be shown in mode 4 is produced by typing 7 STEP . If you try 8 STEP , you will find an extremely slow (but recursive!) method of drawing a large white square.

12.1.2 The VDU drivers

In Acornsoft FORTH the word >VDU is used to send a byte to the VDU drivers. It sends the less significant byte of the top stack item, and is therefore rather similar to EMIT . There are, however, two main differences between these words. Whereas EMIT sends only a true 7-bit ASCII code, >VDU sends the full eight bits. Also, each character sent by EMIT adds 1 to the value of the user variable OUT , so that the total number of characters sent to the display is known. Sending bytes with >VDU does not affect this character count.

Changing colours

So far the graphics have been plotted in the default colours of the graphics mode being used. The options to change colours are divided into 3 types which are:

change text colour
change graphics colour
change logical colour

These options are all available on the BBC Microcomputer by sending the appropriate code sequence to the VDU drivers.

For compatibility with other languages we can define the word COLOUR to change the text foreground and background colours:

```
: COLOUR ( n... )
   17 >VDU   >VDU ;
```

This expects the 'colour number' on the stack and in mode 5, for example, we may use

```
0 COLOUR    1 COLOUR    2 COLOUR    or 3 COLOUR
```

Numbers greater than 127 cause the background colour to be changed and in mode 5 the valid numbers are 128, 129, 130 and 131.

The colour used by all following graphics instructions can be redefined using GCOL. This is defined as follows:

```
: GCOL ( n1\n2... )
   18 >VDU  SWAP >VDU  >VDU ;
```

The definition is arranged so that the order in which the two numbers are typed is the same as in BASIC and other languages. The colour is specified by the top stack item and its meaning is the same as for COLOUR. The value beneath it specifies the mode of action, i.e.

0 plot in the specified colour
1 OR the specified colour with the one already there
2 AND the specified colour with the one already there
3 exclusive-OR the colour with the one already there
4 invert the colour already there.

If you do not want to change the type of plotting action you may define a word that will perform only one of these functions. For example

```
: PCOL ( n... )
   18 >VDU  0 >VDU  >VDU ;
```

needs only the colour value and will always plot in the specified colour.

The colours referred to so far are termed 'logical colours' in that the number need not always refer to the same 'actual colour'. The colours which are displayed in two or four colour modes are selected from the sixteen colours in the total palette of the

computer. The actual colour numbers are:

0	black
1	red
2	green
3	yellow
4	blue
5	magenta
6	cyan
7	white
8	flashing black/white
9	flashing red/cyan
10	flashing green/magenta
11	flashing yellow/blue
12	flashing blue/yellow
13	flashing magenta/green
14	flashing cyan/red
15	flashing white/black

The default colours in, say, a four-colour mode are:

Logical	Actual	Colour
0	0	black
1	1	red
2	3	yellow
3	7	white

Thus using, for example, 2 PCOL will cause all future graphics to be plotted in logical colour 2, which is actual colour 3 (yellow).

If we were to change the relationship between the logical colour and the actual colour so that logical colour 2 used actual colour 4 (blue) then 2 PCOL would cause plotting to be in blue instead of yellow. Furthermore, all areas on the screen which had previously been plotted in logical colour 2 (and therefore appeared as yellow) would also change to blue since we would simply have changed the definition of which colour is represented by the number 2.

The logical colour can be redefined using VDU 19, as in this example:

```
: LCOL ( lcol\acol... )
   19 >VDU  SWAP >VDU    >VDU
   0  >VDU    0  >VDU  0 >VDU ;
```

The earlier example of changing logical colour 2 to represent actual colour 4 (blue) can be done by

```
2 4 LCOL
```

In the same way logical colour 1 (originally red) can be set to actual colour 6 (cyan) by

```
1 6 LCOL
```

The ability to change large areas of colour on the screen with one simple command opens up a whole range of possibilities for graphics, as the next two examples show. The first plots a rapidly changing circle of colour, made up of a number of differently coloured triangles. It makes use of PCOL and LCOL as defined above. The definition of TRIANGLE is different from the one given earlier in that it plots a solid triangle in the colour whose (logical) number is on top of the stack:

```
VARIABLE COUNTER                    ( step counter )

: ARRAY ( lots... ) ( this version fills array )
   CREATE 0 DO , LOOP ( with values from stack )
   DOES>   OVER + + ;

    900  790  640  490  380  340 ( triangle X-coords )
    380  490  640  790  900  940
    12 ARRAY XX

    350  240  200  240  350  500 ( triangle Y-coords )
    650  760  800  760  650  500
    12 ARRAY YY
```

```
: TRIANGLE ( x0\y0\x1\y1\x2\y2\col... )
   PCOL
   4  ROT ROT PLOT
   4  ROT ROT PLOT
   85 ROT ROT PLOT ;

: ONE-STEP ( n ... x\y )  ( convert to coords )
   COUNTER @ +  12 MOD  ( get step, modulo 12 )
   DUP   XX @           ( look up coordinates )
   SWAP  YY @ ;

: WHEEL ( ... )
   0 COUNTER !   5 MODE
   BEGIN  1 COUNTER +!
         COUNTER @  DUP 1+ LCOL
                    ( set LCOL J to ACOL J+1 )
         12 0 DO
                 640 500  ( centre of circle )
                 I STEP ( coords of a vertex )
                 I 1+ STEP     ( next vertex )
                 I 4 MOD     ( pick a colour )
                 TRIANGLE
              LOOP
         COUNTER @  100 > ( plot 100 circles )
   UNTIL
   7 MODE ;         ( get rid of funny colours )
```

In WHEEL the same sequence of logical colours is used every time round the circle. Between each plot of the circle, however, one logical colour is changed to display a different actual colour so that a constantly varying effect is produced.

The next example shows how the redefinition of logical colours can be used to produce extremely fast animation effects. It displays a moving row of rectangles across the bottom of the screen. It uses PCOL and LCOL as defined earlier.

The first part of the application is concerned with setting up the display.

```
VARIABLE  LX    ( left X-coord for rectangle )
VARIABLE  RX    ( right X )
VARIABLE  BY    ( bottom Y )
VARIABLE  TY    ( top Y )
```

```
: RECTANGLE ( ... )
    4  RX @  BY @  PLOT
    4  RX @  TY @  PLOT
   85  LX @  BY @  PLOT
   85  LX @  TY @  PLOT ;

: ROW ( ... )
    50 LX !  100 RX !
   100 BY !  200 TY !
   8 0 DO
         4 1 DO  I PCOL
                 RECTANGLE
                 50 LX +!
                 50 RX +!
             LOOP
       LOOP ;
```

Executing ROW in mode 5 graphics will display a row of rectangles in 3 colours (logical colours 1, 2 and 3):

3	1	2	3	1	2	3

The next section is concerned with changing the actual colours displayed to allow the illusion of movement to be created. The trick is to make two logical colours display the same actual colour so that the display appears as shown in the following diagram:

1	1	2	1	1	2	1

First we create three 'colour registers' to hold the actual colours to be displayed for logical colours 1, 2 and 3. Then we define SETCOLS to change the display colours to those in the three registers:

```
VARIABLE 1COL  1 1COL !
VARIABLE 2COL  2 2COL !
VARIABLE 3COL  1 3COL !
```

```
: SETCOLS ( ... )
   1  1COL @   LCOL
   2  2COL @   LCOL
   3  3COL @   LCOL ;
```

Executing

```
5 MODE  ROW   SETCOLS
```

should now give a display like that shown in the last diagram.

We now write a word ROTCOLS which exchanges the contents of the three colour registers in a circular fashion, so that 1->2, 2->3 and 3->1.

```
: ROTCOLS ( ... )
   3COL @
   2COL @  3COL !
   1COL @  2COL !
   1COL ! ;
```

Typing

```
ROTCOLS  SETCOLS
```

should change the display from

1	1	2	1	1	2	1

to

2	1	1	2	1	1	2

so that the second colour appears to have moved one rectangle to the right.

The final part of the application automates the process of changing the colours. It introduces a variable delay between each change so that the speed of motion will change:

```
: DELAYS ( n ... )
   0 DO LOOP ;

: SHIFT ( ... )
   2000                          ( initial delay )
   BEGIN  DUP DELAYS
          10 -                    ( reduce delay )
          ROTCOLS
          SETCOLS
          ?DUP 0=           ( stop when delay = 0 )
   UNTIL ;

: GO ( ... )                  ( do the whole thing )
   5 MODE
   ROW  SHIFT
   KEY DROP
   7 MODE ;
```

Single byte VDU commands

A number of VDU commands need only a single byte to be sent to the VDU drivers. These are very simple to define if you wish to do so. In many circumstances you may find it simpler just to send the byte by direct use of >VDU although this, unless commented, makes it less easy to understand what is going on. For example,

```
12 >VDU
```

is the command to clear the text area of the screen, but you may prefer to use the command CLS , defined as

```
: CLS   12 >VDU ;
```

Similarly

```
: CLG   16 >VDU ;
```

will clear the graphics area.

The cursor controls may be written as:

```
: LEFT      8  >VDU ;
: RIGHT     9  >VDU ;
: DOWN     10  >VDU ;
: UP       11  >VDU ;       and
: RUBOUT  127  >VDU ;
```

To join and separate the text and graphics cursors we can use

```
: GTEXT     5    >VDU ;   and
: TTEXT     4    >VDU ;
```

respectively.

All other single byte commands can be defined in the same way.

Redefining graphics characters

Graphics characters in the range 224 to 255 may readily be reprogrammed by the use of VDU 23 . This requires another nine bytes of data; the character number and the bit pattern for each of the eight rows of the character. Clearly, the order in which the bytes are presented to the VDU drivers is rather important, and this is one case where the use of stacks to store numbers can cause problems. The simplest method (for the computer) would be to enter the nine required values in reverse order to that in which they are needed. The bit patterns would be entered first, starting with the bottom row and working towards the top, and the character number last of all. The bytes, taken one by one from the stack, are then in the correct order for the VDU drivers. A definition that works this way could be:

```
: DEFCHAR1 ( r7\r6\r5\r4\r3\r2\r1\r0\nchar ... )
   23  >VDU
   9 0 DO  >VDU  LOOP ;
```

A second alternative would be to preserve the order of entry exactly as is used in, say, BASIC i.e. the character number first and then the bit patterns (with the top row first and the bottom row last). The numbers are then completely in the wrong order to be

sent to the VDU drivers and some fancy stack-shuffling is required. A definition which worked this way would be

```
: DEFCHAR2 ( nchar\r0\r1\r2\r3\r4\r5\r6\r7 ... )
   23  >VDU
   1 9 DO  I ROLL >VDU  -1 +LOOP ;
```

A third possibility, which feels more FORTH-like, uses a mixture of these two techniques. The bit patterns are still entered in the 'normal' order, top row first (so that it is easy to use numbers worked out whilst working with other languages), but with the character number entered last, so that it is on top of the stack.

The definition would then look like this:

```
: DEFCHAR ( r0\r1\r2\r3\r4\r5\r6\r7\nchar ...)
   23  >VDU
   >VDU
   1 8 DO  I ROLL >VDU  -1 +LOOP ;
```

The following example defines character number 240 to be a sp*c* *nv*d*r-type character, using this last version of DEFCHAR.

```
36  126 90  126 126 66  102 36
240  DEFCHAR

5 MODE  240 >VDU
```

Looks OK, don't you think?

Such characters usually 'wiggle' so we had better define another, slightly different, shape as character 241:

```
36  126 90 126 126 66  195 129
241 DEFCHAR
```

Now we have to display them alternately in the same place. The following word will display either of the two characters, depending on whether there is a zero or a one on the stack:

```
: PIC  ( n ... )
   240 +  >VDU ;

5 MODE  0 PIC  1 PIC
```

We shall want to rub out one version and plot the other in the same place. Here is a definition which will do that, using a couple of words from the previous section:

```
: SHOW  ( n ... )
   RIGHT
   RUBOUT
   PIC ;
```

It will be necessary to specify the position on the screen where the character is displayed, using a word like the (BASIC) MOVE. Since MOVE is a word in FORTH-79 we must use something different, e.g. GMOVE :

```
: GMOVE ( x\y ... )
   4 ROT ROT  PLOT ;
```

Then

```
: 1ROW  ( n\x\y ... )
   GMOVE
   0 DO  0 SHOW  LOOP ;
```

gives us a whole row of n character 240s, starting at (x,y). A row of the second character can be produced by

```
: 2ROW  ( n\x\y ... )
   GMOVE
   0 DO  1 SHOW  LOOP ;
```

All we need now is a word to put it all together, with a delay between the two sets of characters. Here is another version of DELAY that takes its delay time from the variable SPEED :

```
VARIABLE  SPEED
10000 SPEED !
: DELAY ( ... )
   SPEED @  0  DO LOOP ;
```

```
: MONSTERS ( ... )
   5 MODE  GTEXT
   BEGIN   10
   100 500
   1ROW  DELAY
   10
   100 500
   2ROW  DELAY
   ?TAB
   UNTIL
   TTEXT ;
```

A bit of sound would improve matters, but that will have to wait until section 12.3.

So far the redefined graphics characters have had to be sent to the screen by use of >VDU . It would be useful if we could also send them via EMIT , which is the normal word to send codes to the display. The problem is that EMIT sends only the lower 7 bits of a byte and codes above 127 need the full 8 bits. EMIT is, however, a vectored word so we can make it do whatever we like.

Let us assume that you do not like the appearance of the '9' on the display and wish to change it to look more like the version that appears on the bottom of a cheque. If we define character 250 to be that shape,

```
124  68  68  124  12  12  12  0
250 DEFCHAR
```

we can then make EMIT use that character whenever it meets a '9' (whose ASCII code is 57, or &39).

```
: "9"EMIT ( n ... )
   DUP 127 AND                          ( mask to 7 bits )
   57 =                                  ( is it a '9' ? )
         IF    DROP 250 >VDU ( send char 250 instead )
               1 OUT +!          ( and increment 'OUT' )
   ELSE  (EMIT)              ( normal action for others )
   THEN ;

ASSIGN EMIT TO-DO "9"EMIT
```

The reason for converting to a 7-bit code before testing the character is that the last byte of the name

of every word in the dictionary has its most significant bit set, to indicate the end of the name. If this last character were a '9' it would slip through the test and be displayed by (EMIT) in its old form.

After typing in the above code, all occurrences of '9' displayed by EMIT will be in the new format. Note that this will not apply to characters typed in at the keyboard since these are not displayed by EMIT .

12.2 Sound

Sound generation on the BBC Microcomputer requires, in any language, the equivalent of the BASIC keywords SOUND and ENVELOPE. These are not provided in the nucleus of Acornsoft FORTH but are quite simple to define. They do, however, require short sections of machine code to make the relevant calls to the OSWORD subroutine in the Operating System.

12.2.1 Machine code for SOUND and ENVELOPE

The routines are given here in two forms: in standard FORTH assembler (which will require the assembler vocabulary to be loaded first) and in hand-assembly form, as described in chapter 5.

OSWORD requires a table of values to be present in memory, and on entry to the routine, the processor's X and Y registers must point to (contain the address of) the first byte of this table. The high order byte of the address must be in the Y register and the low order byte in the X register.

If we constructed the table at some arbitrary place in memory we would have to load the X and Y registers with the correct values to point to this area. However, we can do much better than that.

Remember that, on entry to a machine code routine in Acornsoft FORTH, the X register contains the address of the most accessible item on the stack and the Y register contains zero. Since the stack is in page zero (i.e. all addresses are &00xx) the X and Y registers point to the stack contents in exactly the

way required by OSWORD. All we have to do is to make the data on the stack conform to that expected by OSWORD and let the registers take care of themselves. The only precaution we have to take is to save the contents of the X register and restore them after the call since OSWORD will, in general, leave the contents undefined (i.e. probably changed).

A sound may be generated by making a call to OSWORD with the accumulator containing 7 and eight bytes of data on the stack. Assuming that this data is present we can use the assembler to make the following machine code primitive:

```
HEX
CODE (SOUND)  ( n1\n2\n3\n4 ... )
   XSAVE STX,
   7 # LDA,    FFF1  JSR,      ( call to OSWORD )
   XSAVE LDA,  CLC,  8 # ADC,    TAX,
                        ( quick way to drop 4 items )
   NEXT JMP,
END-CODE
DECIMAL
```

In hand-assembly form this is:

```
HEX
CREATE (SOUND)   HERE -2 ALLOT ,
   6886 ,
    7A9 ,  20 C, FFF1 ,
   68A5 ,  18 C,  869 ,  AA C,
    4C C,  6A +ORIGIN ,
DECIMAL
```

This will require an additional high-level definition to ensure that the data on the stack is arranged correctly and this is described in the next section.

A call to OSWORD with the accumulator containing 8 is required for ENVELOPE . For this call 14 bytes (7 items) of data are expected and the code definition is very similar to that for (SOUND) .

```
HEX
CODE (ENVELOPE) ( n1\n2\n3\n4\n5\n6\n7 ... )
   XSAVE STX,
   8 # LDA,  FFF1 JSR,
   XSAVE LDA,  CLC,  0E # ADC,  TAX,
   NEXT JMP,
END-CODE
DECIMAL
```

For hand-assembly this is entered as

```
CREATE (ENVELOPE)   HERE -2 ALLOT ,
   6886 ,
    8A9 ,  20 C,  FFF1 ,
   68A5 ,  18 C,   E69 ,  AA ,
    4C C,  6A  +ORIGIN ,
DECIMAL
```

Again a high-level definition will be needed to arrange the stack contents in the correct sequence.

12.2.2 Completion of SOUND and ENVELOPE

If the four parameters for SOUND are to be entered in the 'normal' sequence (i.e. channel, amplitude, pitch and then duration) they will be in the wrong order to the one expected by OSWORD . Fortunately each of the four values is expected as a two byte number so we only have to reverse their order before using (SOUND) . This can easily be done as shown below:

```
: SOUND ( c\a\p\d ... )
   SWAP  ROT  4 ROLL      ( change to d\p\a\c )
   (SOUND) ;
```

This may then be used to produce sounds; for example

```
1 -15 100 20 SOUND
```

will produce a loud one-second note of pitch C on channel 1.

The high-level definition for ENVELOPE is (of course!) more complicated. The 14 numbers, when entered, occupy two bytes each but OSWORD requires the data in a table of only 14 bytes. In addition to reordering the values it is also necessary to 'pack' them so that all 14

numbers occupy only 7 stack values.

The basic method of doing this is simple, provided that all the numbers are less than 256. Suppose we have two such values on the stack as shown below (in hex) where the top of the stack is to the right, as usual.

0034 0012

If we multiply the top number by 256 (&100) the stack will now appear as

0034 1200

Simply adding the numbers then gives

1234

where we have not only packed the two values from four bytes to two, but have also reversed their byte order.

Multiplication is usually a slow process (second only to division) and should be avoided where possible. With the numbers as shown above, multiplying by 256 has simply had the effect of exchanging the low and high bytes of the number. There happens to be a word in Acornsoft FORTH to do just this - its name is >< (pronounced 'swap-bytes'). To pack the numbers as shown above we could therefore use

```
     256 *  +
or
     ><  +
```

The second of these is clearly preferable since it is both shorter and much faster.

An additional problem arises if either number is negative, in which case the high byte will not be zero. In this case the two methods shown above will not be equivalent and both will fail to work as required. However, since only the low byte of each number is used, we can simply force the high byte to zero by performing a logical AND with 255 (&FF). An order-

changing packing routine can therefore be defined as follows:

```
: EXPACK  ( n1\n2 ... n3 )
   >R  255 AND
   R>  255 AND
   ><  +  ;
```

and a 'straight' pack by

```
: PACK   ( n1\n2 ... n3 )
   255 AND
   SWAP 255 AND
   ><  +  ;
```

We can now complete the definition of ENVELOPE .

```
: ENVELOPE ( n1\n2---n13\n14 ... )
   EXPACK           ( top 2 items )
   ROT ROT EXPACK   ( next 2 )
   9 4 DO  I ROLL   ( dig deeper )
           I ROLL
           EXPACK
       LOOP ;
```

This may be used in exactly the same way as described in the BBC Microcomputer User Guide except, of course, that the parameters should be entered before the word ENVELOPE .

13 Errors

13.1 Error handling

In general FORTH perfoms error checks only where absolutely essential. The errors detected are those which are the most likely causes of a system crash if allowed to pass undetected.

Since FORTH gives such complete control over all the facilities of the computer, it would be prohibitively slow if all possible sources of error were checked (assuming that this were possible). For example, there is nothing to prevent you attempting to fill the entire dictionary with nulls.

It is, of course, possible to include full error checks in your own application, as mentioned in chapter 10. Use of full error checking during the development of an application is recommended - many of the checks can be removed later on when the application has been debugged.

Since FORTH applications can be developed a word at a time, and each word can be tested as it is written, serious errors are fortunately rare.

Many non-fatal errors (and quite a few fatal ones!) are not detected. As an example, we can consider the arithmetic operators. It is quite possible to perform a calculation which gives a result too large to be represented as a two's complement number in 16 bits (or 32 bits for double-precision). The overflow condition is ignored and, in general, the value left on the stack will be the least significant 16 (or 32) bits of the result. What happens afterwards will depend on the way the result is used. It is your responsibility to make sure that this sort of situation does not occur in an application.

The case of division by zero is, however, always fatal and therefore this error is detected by the system. Since this error check slightly slows down the execution of divisions one word, U/, has been left unprotected. It is available for use in situations where speed of execution is important, provided you make sure that division by zero will not occur.

13.2 Detected errors

The errors which are detected fall into two main classes:

i) errors detected by the Operating System
ii) errors detected by FORTH's internal checks

13.2.1 Operating system errors

The response to an error detected by the Operating System is to display the error number and a brief message, for example:

O.S. Error 194 File Open

These error messages will only appear when accessing operating system facilities as when reading or writing tape or disk files, or when using OS' to pass commands to the O.S. command line interpreter.

13.2.2 FORTH errors

Most detected errors in Acornsoft FORTH result in an error message of the form

? cccc MSG # n

Here cccc is the name of the word from the input stream which resulted in the error and n is the error number from the list below. This number is always displayed in decimal, regardless of the numeric base currently in use.

Error	Message
0	(reserved)
1	Stack Empty
2	Dictionary Full
3	Has Incorrect Address Mode (Assembler)
4	Isn't unique
5	Parameter Outside Valid Range
6	Screen Number Out of Range
7	Stack Full
8	Can't Open or Extend File
9	Read/Write not Completed
10	Can't Redefine End-of-Line
11	Can't Divide by Zero
12	Undefined Execution Vector
13	Branch Too Long (Assembler)
14	Incorrect CURRENT Vocabulary
15	(reserved)
16	(reserved)
17	Compilation Only
18	Execution Only
19	Conditionals not Paired
20	Definition not Finished
21	In Protected Dictionary
22	Use Only When LOADing
23	Off Current Editing Screen
24	Not in CURRENT Vocabulary
25	System Memory Clash

The exception to this format occurs when a word from the input stream cannot be found in a dictionary search, nor converted to a valid number. In this case the name of the word is displayed, followed by a question mark.

In nearly all cases the error causes a return to the keyboard interpreter with both the return and computation stacks cleared. The one exception to this rule is error message 4, indicating the redefinition of an existing word. In this case the message is a warning only and the current task continues after the warning has been given.

Error numbers greater than 25 are available for user-definable error checks.

Glossary

This glossary presents all words in Acornsoft FORTH, listed in ASCII order, with the following information:

Word: The name of the word

Pronunciation: Unless it is obvious, the pronunciation is given.

Stack action: The computation (parameter) stack action is shown, where appropriate, as a list of the values and their types before and after the execution of the word, in the form:

(stack contents before ... stack contents after)

In all references to the stack, numbers to the right are at the top of the stack. The notation n1\n2 is read as 'n1 is beneath n2'. The symbols used to represent the different stack value types include:

n	16-bit (single precision) signed number
u	16-bit (single precision) unsigned number
addr	16-bit address (unsigned)
nd	32-bit (double precision) signed number
ud	32-bit (double precision) unsigned number
b	8-bit one-byte number (unsigned)
c	7-bit ASCII character
count	6-bit string length count
f	boolean flag: 0 = false, non-zero = true
ff	boolean false flag = 0
tf	boolean true flag = non-zero

The number of stack values the word uses and leaves: For example, 2 0

Status: Some words have an additional letter indicating their status:

C may only be used in a colon-definition
E intended for execution only
I is IMMEDIATE; will execute even when in compile mode

Description: A description of the FORTH word.

List of FORTH words in Ascii order

```
!
!CSP
#
#>
#BUF
#S
'
(
($+)
(+LOOP)
(.")
(;CODE)
(ABORT)
(CLI)
(CREATE)
(DO)
(EMIT)
(FIND)
(KEY)
(LINE)
(LOOP)
(NUM)
(OPEN)
(R/W)
(ULOOP)
(UPDATE)
(WARM)
(WORD)
*
*/
*/MOD
+
+!
+-
+BUF
+LOOP
+ORIGIN
,
-
-->
-FIND
-TRAILING
.
."
.LINE
.R
.S
/
/MOD
0
1
2
-1
-2
0<
0=
0>
0BRANCH
1+
1-
1WORD
2*
2+
2-
2/
2DROP
2DUP
2OVER
2SWAP
4HEX
79-STANDARD
:
;
;CODE
<
<#
=
>
><
>CLI
>IN
>R
>VDU
?
?COMP
?CSP
?DUP
?ERROR
?EXEC
?KEY
?LOADING
?PAIRS
?STACK
?TAB
@
@EXECUTE
ABORT
ABS
AGAIN
ALLOT
AND
ASSIGN
B/BUF
B/SCR
BACK
BASE
BEGIN
BL
BLANKS
BLK
BLOCK
BRANCH
BUFFER
BUFSZ
C!
C,
C/L
C@
CFA
CHANNEL
CLOSE
CMOVE
COLD
COMPILE
CONSTANT
CONTEXT
CONVERT
COUNT
CR
CREATE
CREATE-SCREENS
CSP
CURRENT
D+
D+-
D.
D.R
D<
DABS
DEC.
DECIMAL
DEFINITIONS
DEPTH
DIGIT
DISK
DLITERAL
DNEGATE
DO
DOES>
DOVEC
DP
DPL
DR/W
DROP
DUP
ELSE
EMIT
EMPTY-BUFFERS
ENCLOSE
ERASE
ERROR
ESCAPE
EXECUTE
EXIT
EXPECT
EXVEC:
FENCE
FILL
FIND
FIRST
FLUSH
FNAME
FORGET
FORTH
H.
HERE
HEX
HLD
HOLD
I
ID.
IF
IMMEDIATE
INDEX
INITBUF
INITVECS
INTERPRET
J
KEY
KEY'
LAST
LEAVE
LFA
LIMIT
LIST
LIT
LITERAL
LOAD
LOOP
M*
M/
M/MOD
MAX
MAXFILES
MESSAGE
MIN
MINBUF
MOD
MODE
MOVE
MSG#
NEGATE
NFA
NOOP
NOT
NOVEC
NUM
NUMBER
OFFSET
OPEN
OR
OS'
OSCLI
OSERROR
OUT
OVER
PAD
PFA
PICK
PLOT
PREV
PRUNE
QUERY
QUIT
R#
R/W
R0
R:
R;
R>
R@
REPEAT
ROLL
ROT
RP!
RP@
S->D
S/FILE
S0
SAVE-BUFFERS
SCR
SETBUF
SIGN
SMUDGE
SP!
SP@
SPACE
SPACES
ST-ADDR
START
START-KEYS
STATE
STRING
SWAP
TAPE
TEXT
THEN
TIB
TLD
TO-DO
TOGGLE
TR
TR/W
TRAVERSE
TRIAD
TSV
TW
TYPE
U*
U.
U/
U/MOD
U<
UNTIL
UPDATE
USE
USER
VARIABLE
VLIST
VOC-LINK
VOCABULARY
WARM
WARNING
WBFR
WDSZ
WHILE
WIDTH
WORD
X
XOR
[
[COMPILE]
]
```

Glossary

!

Pronounced: store
Stack Action: (n\addr ...)
Uses/Leaves: 2 0
Status:
Description: Stores the value n at the address addr.

!CSP

Pronounced: store c-s-p
Stack Action:
Uses/Leaves: 0 0
Status:
Description: Stores the stack pointer value in user variable CSP. Used as part of the compiler security.

#

Pronounced: sharp
Stack Action: (nd1 ... nd2)
Uses/Leaves: 2 2
Status:
Description: Converts the least-significant digit (in the current base) of the double-precision number nd1 to the corresponding ASCII character, which it then stores at PAD . The remaining part of the number is left as nd2 for further conversions. # is used between <# and #> .

Glossary

#>

Pronounced: sharp-greater
Stack Action: (nd ... addr\count)
Uses/Leaves: 2 2
Status:
Description: Terminates numeric output conversion by dropping the double number nd and leaving the address and character count of the converted string in a form suitable for TYPE .

#BUF

Pronounced: sharp-buff
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A variable containing the number of mass-storage buffers. Initialised to 2 on a cold start.

#S

Pronounced: sharp-s
Stack Action: (nd1 ... nd2)
Uses/Leaves: 2 2
Status:
Description: Converts the double-precision number nd1 into ASCII text by repeated use of # , and stores the text at PAD . The double-precision number nd2 is left on the stack, and has the value zero. #S is used between <# and #> .

Glossary

Pronounced: tick
Stack Action: (... addr)
Status: I (during execution)
(during compilation)
Uses/Leaves: 0 0
Description: Used in the form ' nnnn and leaves the parameter field address of dictionary word nnnn if in execution mode.

If used within a colon-definition it will execute to compile the address as a literal numerical value (preceded by the address of the literal handler routine, LIT) in the definition.

Pronounced: paren
Stack Action:
Uses/Leaves:
Status: I
Description: Used in the form (nnnn) to insert a comment. All text nnnn up to a right parenthesis on the same line is ignored. Since (is a FORTH word it must be followed by a space. A space is not necessary before) since it is only used as a delimiter for the text.

) is pronounced 'close-paren'.

Glossary

($+)

Pronounced: bracket-dollar-plus
Stack Action: (addr1\count\addr2 ...)
Uses/Leaves: 3 0
Status:
Description: The string of length 'count' whose first character is at addr1 is added to the end of the string whose count byte is at addr2 (i.e. whose first character is at addr2+1). The count byte at addr2 is incremented to be the new length of the concatenated string.

(+LOOP)

Pronounced: bracket-plus-loop
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: The run-time procedure compiled by +LOOP . It increments the loop index by the signed quantity n and tests for loop completion. See +LOOP .

(."")

Pronounced: bracket-dot-quote
Stack Action:
Uses/Leaves:
Status:
Description: The run-time procedure compiled by <."">. It transmits the following in-line text to the output device. See <."> .

(;CODE)

Pronounced: bracket-semi-colon-code
Stack Action:
Uses/Leaves:
Status: C
Description: The run-time procedure compiled by ;CODE that rewrites the code field address of the most recently defined word to point to the machine code following (;CODE) . It is used by the system defining words (<:>, CONSTANT etc.) to define the machine code actions of dictionary entries using them. This is, in a sense, a machine-code version of DOES> .

(ABORT)

Pronounced: bracket-abort
Stack Action:
Uses/Leaves:
Status:
Description: Clears the data and return stacks and sets execution mode. Control is returned to the keyboard interpreter. See ABORT .

(CLI)

Pronounced: bracket-c-l-i
Stack Action:
Uses/Leaves:
Status:
Description: The run-time procedure, compiled by >CLI . It transmits the following in-line text to the operating system command line interpreter.

(CREATE)

Pronounced: bracket-create
Stack Action:
Uses/Leaves:
Status:
See CREATE .

(DO)

Pronounced: bracket-do
Stack Action:
Uses/Leaves:
Status: C
Description: The run-time procedure compiled by DO . It moves the loop control parameters to the return stack. See DO .

(EMIT)

Pronounced: bracket-emit
Stack Action: (c ...)
Uses/Leaves: 1 0
Status:
See EMIT .

(FIND)

Pronounced: bracket-find
Stack Action: (addr1\addr2... cfa\b\tf)[found]
Uses/Leaves: 2 3
Stack Action: (addr1\addr2 ... ff)[not found]
Uses/Leaves: 2 1
Status:
Description: Searches the dictionary starting at the name field address addr2 for a match with the text starting at addr1. For a successful match the code field (execution) address and length byte of the name

field plus a true flag are left. If no match is found only a false flag is left.

(KEY)

Pronounced: bracket-key
Stack Action: (... c)
Uses/Leaves: 0 1
Status:
See KEY .

(LINE)

Pronounced: bracket-line
Stack Action: (n1\n2 ...addr\count)
Uses/Leaves: 2 2
Status:
Description: Returns the start address in the mass storage buffers of the data in line n1 of screen n2, and also the count of characters in the line. A count of 64 is returned so that a full line of text can be displayed.

(LOOP)

Pronounced: bracket-loop
Stack Action:
Uses/Leaves:
Status:
Description: The run-time procedure compiled by LOOP . It increments the loop index by one and tests for loop completion. See LOOP .

Glossary

(NUM)

Pronounced: bracket-num
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
See NUM .

(OPEN)

Pronounced: bracket-open
Stack Action: (addr ... b)
Uses/Leaves: 1 1
Status:
Description: Opens a named file for reading and writing. The name of the file is given by the string whose first character is at addr and whose last character is &0D . The eight-bit channel number b is left on the stack. See CHANNEL , OPEN .

(R/W)

Pronounced: bracket-read-write
Stack Action:
Uses/Leaves: 8 4
Status:
Description: A system-dependent routine to read or write a screen in the disk-filing system. It calls the operating system routine OSGBPB which is not available on cassette systems. It is used by R/W in a disk-based system and is not intended to be executed in any other circumstance.

(ULOOP)

Pronounced: bracket-u-loop
Stack Action:
Uses/Leaves:
Status:
Description: The run-time procedure for an unsigned version of LOOP . It is used by the system for loops involving unsigned indices, for example, addresses. It is not available to the user in the system as provided but can be made so by entering the following definition:

```
: ULOOP              3 ?PAIRS COMPILE (ULOOP) BACK ;
IMMEDIATE
```

It may then be used in a colon-definition as:

```
... DO .... ULOOP ....
```

(UPDATE)

Pronounced: bracket-update
Stack Action:
Uses/Leaves:
Status:
Description: See UPDATE .

(WARM)

Pronounced: bracket-warm
Stack Action:
Uses/Leaves:
Status:
Description: A routine which returns control to the keyboard interpreter. It is used by COLD , WARM and the error-handling procedures. The numeric base is set to decimal and FORTH becomes both the current and context vocabularies. The return stack (but not the computation stack) is cleared.

(WORD)

Pronounced: bracket-word
Stack Action: (c ... addr\count)
Uses/Leaves: 1 2
Status:
Description: Scans the input buffer, ignoring leading occurrences of the delimiter character c, for the next word. The start address and length of the text up to the terminating delimiter are left. No text is moved. See WORD .

---

*

Pronounced: times
Stack Action: (n1\n2 ... n3)
Uses/Leaves: 2 1
Status:
Description: Leaves as n3 the product of the two signed numbers n1 and n2.

---

*/

Pronounced: times-divide
Stack Action: (n1\n2\n3 ... n4)
Uses/Leaves: 3 1
Status:
Description: Leaves as n4 the value n1 n2 * n3/. The product n1 n2 * is here kept as a double-precision intermediate value, resulting in a more accurate result than can be obtained if the operations are performed independently.

Glossary

*/MOD

Pronounced: times-divide-mod
Stack Action: (n1\n2\n3 ... n4\n5)
Uses/Leaves: 3 2
Status:
Description: Leaves, as n4 and n5 respectively, the remainder and the integer value of the result of n1 n2 * n3 /MOD . The product n1 n2 * is here kept as a double-precision intermediate value, resulting in a more accurate result than can be obtained if the operations are performed independently.

+

Pronounced: plus
Stack Action: (n1\n2 ... n3)
Uses/Leaves: 2 1
Status:
Description: Leaves as n3 the sum of n1 and n2.

+!

Pronounced: plus-store
Stack Action: (n\addr ...)
Uses/Leaves: 2 0
Status:
Description: Adds n to the value at the address addr.

+-

Pronounced: plus-minus
Stack Action: (n1\n2 ... n3)
Uses/Leaves: 2 1
Status:
Description: Leaves as n3 the result of applying the sign of n2 to n1.

Glossary

+BUF

Pronounced: plus-buff
Stack Action: (addr1 ... addr2\f)
Uses/Leaves: 1 2
Status:
Description: Advances the buffer address addr1 to the start of the next buffer at addr2. The flag is false when addr2 is the address of the most recently referenced buffer.

+LOOP

Pronounced: plus-loop
Stack Action: (n ...)
Uses/Leaves: 1 0
Status: I,C
Description: Used in colon-definition in the form:

DO ... +LOOP

During execution +LOOP controls branching back to the corresponding DO , dependent on the loop index and loop limit. The loop index is incremented by n, which may be positive or negative. Branching to DO will occur until

a) for positive n, the loop index is greater than or equal to the loop limit, or

b) for negative n, the loop index is less than the loop limit.

Execution then continues with the word following +LOOP .

+ORIGIN

Pronounced: plus-origin
Stack Action: (n ... addr)
Uses/Leaves: 1 1
Status:
Description: Leaves the address of the nth byte after the start of the boot-up parameter area. Used to access or modify the boot-up parameters.

,

Pronounced: comma
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Stores (compiles) n in the first two available bytes at the top of the dictionary and increments the dictionary pointer by two.

-

Pronounced: subtract
Stack Action: (n1\n2 ... n3)
Uses/Leaves: 2 1
Status:
Description: Leaves as n3 the difference n1 - n2.

-->

Pronounced: next screen
Stack Action:
Uses/Leaves:
Status: I
Description: Continues interpretation with the next screen of source code from mass storage.

-FIND

Pronounced: dash-find
Stack Action: (addr ... cfa\b\tf)[if found]
Uses/Leaves: 1 3
Stack Action: (addr ... ff)[if not found]
Uses/Leaves: 1 1
Status:
Description: Used as, for example,

```
CONTEXT @ @ -FIND nnnn
```

The CONTEXT and FORTH vocabularies are searched for the word nnnn. If found, the entry's code field (execution) address, name length byte and a true flag are left; otherwise just a false flag is left.

-TRAILING

Pronounced: dash-trailing
Stack Action: (addr\n1 ... addr\n2)
Uses/Leaves: 2 2
Status:
Description: Changes the character count n1 of the text string at the address addr so as not to include any trailing blanks, and leaves the result as n2.

.

Pronounced: dot
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Prints the number n on the terminal device in the current numeric base. The number is followed by one blank space.

Glossary

."

Pronounced: dot-quote
Stack Action:
Uses/Leaves:
Status: I
Description: Used as:

." cccc"

In a colon-definition the literal string cccc is compiled together with the execution address of a routine to transmit the text to the terminal device.

In the execution mode the text up to the second " will be printed immediately.

.LINE

Pronounced: dot-line
Stack Action: (n1\n2 ...)
Uses/Leaves: 2 0
Status:
Description: Print, on the terminal, line n1 of screen n2 from mass storage. Trailing blanks are suppressed.

.R

Pronounced: dot-r
Stack Action: (n1\n2 ...)
Uses/Leaves: 2 0
Status:
Description: Print the number n1 at the right-hand end of a field of n2 spaces. Unlike <.> no following space is printed.

Glossary

.S

Pronounced: dot-s
Stack Action:
Uses/Leaves:
Status:
Description: Display (without changing) the contents of the computation stack. The most accessible stack item is to the right.

/

Pronounced: divide
Stack Action: (n1\n2 ... n3)
Uses/Leaves: 2 0
Status:
Description: Leaves the value n3 = n1 n2 /.

/MOD

Pronounced: divide-mod
Stack Action: (n1\n2 ... n3\n4)
Uses/Leaves: 2 2
Status:
Description: Leaves the remainder n3 and quotient n4 of n1 n2 /. The remainder has the sign of the dividend.

0, 1, 2, -1, -2

Pronounced:
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: These often-used numerical values are defined as constants in the dictionary to save both time and dictionary space.

0<

Pronounced: zero-less
Stack Action: (n ... f)
Uses/Leaves: 1 1
Status:
Description: Leaves a true flag if n is less than zero, otherwise leaves a false flag.

0=

Pronounced: zero-equals
Stack Action: (n ... f)
Uses/Leaves: 1 1
Status:
Description: Leaves a true flag if n is equal to zero, otherwise leaves a false flag.

0>

Pronounced: zero-greater
Stack Action: (n ... f)
Uses/Leaves: 1 1
Status:
Description: Leaves a true flag if n is greater than zero, otherwise leaves a false flag.

0BRANCH

Pronounced: zero-branch
Stack Action: (f ...)
Uses/Leaves: 1 0
Status:
Description: The run-time procedure to cause a conditional branch. If f is false the following in-line number is added to the interpretive pointer to cause a forward or backward branch. It is compiled by IF , UNTIL and WHILE .

Glossary

1+

Pronounced: one-plus
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Increments n1 by one to give n2.

1-

Pronounced: one-minus
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Decrements n1 by one to give n2.

1WORD

Pronounced: one-word
Stack Action: (c ... addr)
Uses/Leaves: 1 1
Status:
Description: Similar to WORD , except that the character count at the beginning of the string at address addr has a minimum value of 1, even if the input stream is exhausted when 1WORD is called. See WORD .

2*

Pronounced: two-times
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Fast multiply by two. Equivalent to 2 * .

Glossary

2+

Pronounced: two-plus
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Increments n1 by two to give n2.

2-

Pronounced: two-minus
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Decrements n1 by two to give n2.

2/

Pronounced: two-divide
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Fast divide by two. Equivalent to 2 / .

2DROP

Pronounced: two-drop
Stack Action: (nd ...)
Uses/Leaves: 2 0
Status:
Description: Drops the double-precision number nd (or two single-precision numbers) from the stack.

Glossary

2DUP

Pronounced: two-dup
Stack Action: (nd ... nd\nd)
Uses/Leaves: 2 4
Status:
Description: Duplicates the top double-precision number (or the top two single-precision numbers) on the stack.

2OVER

Pronounced: two-over
Stack Action: (nd1\nd2 ... nd1\nd2\nd1)
Uses/Leaves: 4 6
Status:
Description: Copies the second double stack value over the top double stack value.

2SWAP

Pronounced: two-swap
Stack Action: (nd1\nd2 ... nd2\nd1)
Uses/Leaves: 4 4
Status:
Description: Exchanges the top two double stack items.

4HEX

Pronounced: four-hex
Stack Action: (n ... addr\count)
Uses/Leaves: 1 2
Status:
Description: Converts the number n to a four-character hexadecimal string whose first character is at address addr. The value of 'count' is 4.

79-STANDARD

Pronounced: 79-standard
Stack Action:
Uses/Leaves:
Status:
Description: Can be executed to assure the user that a FORTH-79 standard system is available. It is the last word of the FORTH-79 standard required vocabulary in the dictionary. FORGETing this or any earlier word will result in a non-standard system.

:

Pronounced: colon
Stack Action:
Uses/Leaves:
Status: E
Description: Used to create a colon-definition in the form

```
: CCCC .... ;
```

Creates a dictionary entry for the word CCCC as being equivalent to the sequence of FORTH words until the next <;>. Each word in the sequence is compiled into the dictionary entry, unless it is in the immediate execution mode.

;

Pronounced: semi-colon
Stack Action:
Uses/Leaves:
Status: I,C
Description: Terminates a colon-definition and stops further compilation.

;CODE

Pronounced: semi-colon-code
Stack Action:
Uses/Leaves:
Status: I,C
Description: The use of this word requires the ASSEMBLER vocabulary to be loaded. Use in the form

```
: NNNN ... ;CODE ... (assembler words) ... END-CODE
```

Compilation of the definition NNNN is terminated and ASSEMBLER becomes the CONTEXT vocabulary. A defining word NNNN is created which when executed in the form

```
NNNN  CCCC
```

will create a new word CCCC. When CCCC is itself executed its action will be determined by the machine code following ;CODE in NNNN .

<

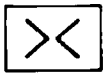
Pronounced: less-than
Stack Action: (n1\n2 ... f)
Uses/Leaves: 2 1
Status:
Description: Leaves a true flag if n1 is less than n2, otherwise leaves a false flag.

<#

Pronounced: less-sharp
Stack Action:
Uses/Leaves:
Status:
Description: Sets up for numeric output formatting. The conversion is performed on a double number to produce text at PAD . See also #, #>, #S , SIGN .

Pronounced: equals
Stack Action: (n1\n2 ... f)
Uses/Leaves: 2 1
Status:
Description: Leaves a true flag if n1 is equal to n2, otherwise leaves a false flag.

Pronounced: greater-than
Stack Action: (n1\n2 ... f)
Uses/Leaves: 2 1
Status:
Description: Leaves a true flag if n1 is greater than n2, otherwise leaves a false flag.



Pronounced: swap-bytes
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Exchanges the high and low order bytes of n1 to give n2.

Pronounced: to-c-1-i
Stack Action: (addr\count ...)
Uses/Leaves: 2 0
Status: I
Description: In a colon-definition the text starting at the address addr and of the given length is compiled, together with the execution address of a routine to transmit the text to the operating system command line interpreter. In the execution mode the

text is transmitted immediately. In both cases the character string must have a terminating &0D.

>IN

Pronounced: to-in
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the byte offset to the present position in the input buffer (terminal or mass storage) from where the next text will be accepted.

>R

Pronounced: to-r
Stack Action: (n ...)
Uses/Leaves: 1 0
Status: C
Description: Removes a number from the computation stack and places it on the return stack. Its use must be balanced with R> in the same definition. It is used to remove a number temporarily from the stack to access a lower number. See R> .

>VDU

Pronounced: to-v-d-u
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Transmits the low byte of n to the VDU driver, without incrementing OUT . See EMIT .

?

Pronounced: question-mark
Stack Action: (addr ...)
Uses/Leaves: 1 0

Status:
Description: Prints the value contained in the two bytes starting at the address addr. Equivalent to <@ .> .

?COMP

Pronounced: query-comp
Stack Action:
Uses/Leaves:
Status:
Description: Issues an error message if not compiling.

?CSP

Pronounced: query-c-s-p
Stack Action:
Uses/Leaves:
Status:
Description: Issues an error message if stack position differs from that saved in CSP . Used as part of the compiler security.

?DUP

Pronounced: query-dup
Stack Action: (ff ... ff)
Uses/Leaves: 1 1
or
Stack Action: (tf ... tf\tf)
Uses/Leaves: 1 2
Status:
Description: Duplicates the top stack item if it is true (non-zero).

Glossary

?ERROR

Pronounced: query-error
Stack Action: (f\n ...)
Uses/Leaves: 2 0
Status:
Description: Issues error message number n if the boolean flag f is true. Uses ERROR . The stack is always empty after an error message.

?EXEC

Pronounced: query-exec
Stack Action:
Uses/Leaves:
Status:
Description: Issues an error message if not executing.

?KEY

Pronounced: query-key
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Flushes the keyboard buffer of all characters and tests if a key is being pressed. There are two cases:

1) If n1 is positive, i.e. in the range 0 to 32767 inclusive, ?KEY will wait for up to n1 hundredths of a second, constantly testing to see if a key has been pressed. If a key is pressed within the time limit its ASCII value will be returned as n2. If the time limit expires before a key is pressed a negative number will be returned as n2.

2) If n1 is negative a test will be made to see if a particular key is pressed at the instant ?KEY is called. The value of n1 determines which key is to be tested according to the table given in the description of INKEY in the BBC Microcomputer User Guide (page 275). If the key is pressed n2 will be returned as -1, otherwise n2 will be zero. These may be treated as true and false flags respectively.

?LOADING

Pronounced: query-loading
Stack Action:
Uses/Leaves:
Status:
Description: Issues an error message if not loading from mass storage.

?PAIRS

Pronounced: query-pairs
Stack Action: (n1\n2 ...)
Uses/Leaves: 2 0
Status:
Description: Issues an error message if n1 does not equal n2. The message indicates that compiled conditionals (IF ... ELSE ... THEN or BEGIN ... UNTIL etc.) do not match. It is part of the compiler security. The error message is given if, for example, the sequence IF ... UNTIL is found during compilation of a dictionary entry.

?STACK

Pronounced: query-stack
Stack Action:
Uses/Leaves:
Status:
Description: Issues an error message if the stack is out of bounds.

Glossary

?TAB

Pronounced: query-tab
Stack Action: (... f)
Uses/Leaves: 0 1
Status:
Description: Tests if the TAB key is pressed at the instant ?TAB is called. A true flag indicates that the key was pressed, otherwise a false flag is left.

@

Pronounced: fetch
Stack Action: (addr ... n)
Uses/Leaves: 1 1
Status:
Description: Leaves on the stack the 16-bit value n found at the address addr.

@EXECUTE

Pronounced: fetch-execute
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
Description: Executes the definition whose code field (execution) address is contained in the two bytes at the address addr.

ABORT

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: A vectored routine initialised by a cold start to execute (ABORT) which clears the data and return stacks, and returns control to the terminal. The

action may be changed by the user with the aid of ASSIGN .

ABS

Pronounced:
Stack Action: (n ... u)
Uses/Leaves: 1 1
Status:
Description: Leaves u as the absolute value of n.

AGAIN

Pronounced:
Stack Action:
Uses/Leaves:
Status: I,C
Description: Used in a colon-definition in the form:

BEGIN ... AGAIN

During execution of a word containing this sequence, AGAIN forces a branch back to the corresponding BEGIN to create an endless loop.

ALLOT

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: The value of n is added to the dictionary pointer to reserve n bytes of dictionary space. The dictionary pointer may be moved backwards by use of a negative n but this should be used with caution to avoid losing essential dictionary content.

Glossary

AND

Pronounced:
Stack Action: (n1\n2 ... n3)
Uses/Leaves: 2 1
Status:
Description: Leaves as n3 the bit-by-bit logical AND of n1 and n2.

ASSIGN

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Used in the form:

ASSIGN NNNN TO-DO CCCC

It causes all uses of the vectored word NNNN to execute CCCC. The word NNNN must previously have been defined using EXVEC .

B/BUF

Pronounced: b-slash-buf
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: A constant returning the number of bytes per mass-storage buffer. In all FORTH-79 standard systems this number must be 1024.

B/SCR

Pronounced: b-slash-screen
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: A constant returning the number of buffers per 1024 byte screen. In this system the value is 1 but is included for compatibility with other systems, and to allow reconfiguring of the mass-storage interface.

BACK

Pronounced:
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
Description: Calculates the backward branch offset from HERE to addr and compiles into the next available dictionary memory address. Used in the compilation of conditionals (AGAIN, UNTIL etc).

BASE

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the current number base used for input and output conversion.

BEGIN

Pronounced:
Stack Action:
Uses/Leaves: 0 1
Status: I,C
Description: Used in a colon definition in the forms:

```
BEGIN ... AGAIN
BEGIN ... UNTIL
BEGIN ... WHILE ... REPEAT
```

BEGIN marks the start of a sequence that may be executed repeatedly. It acts as a return point from the corresponding AGAIN , UNTIL or REPEAT .

BL

Pronounced: b-l
Stack Action: (... c)
Uses/Leaves: 0 1
Status:
Description: A constant that leaves the ASCII value for 'blank' or 'space' (hex 20).

BLANKS

Pronounced:
Stack Action: (addr\n ...)
Uses/Leaves: 2 0
Status:
Description: Fills n bytes of memory starting at the address addr with blanks.

Glossary

BLK

Pronounced: b-l-k
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the number of the mass storage block from which input is being taken. If BLK contains zero input is taken from the keyboard.

BLOCK

Pronounced:
Stack Action: (n ... addr)
Uses/Leaves: 1 1
Status:
Description: Leaves the address of the first byte of data in block (screen) n. If the block is not already in memory it is transferred from mass storage into whichever memory buffer has least-recently been accessed. If the block occupying that buffer has been UPDATEd it is written to mass storage before block n is read into the buffer. Only the data in the latest block referenced by BLOCK is guaranteed not to have been overwritten.

BRANCH

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: The run-time procedure to cause an unconditional branch. The following in-line value is added to the interpretive pointer to cause a forward or backward branch. It is compiled by ELSE , AGAIN and REPEAT .

Glossary

BUFFER

Pronounced:
Stack Action: (n ... addr)
Uses/Leaves: 1 1
Status:
Description: Obtain the next block buffer, assigning it to block (screen) n. The screen is not read from mass storage but if the previous contents are marked as UPDATEd they are written to mass storage, freeing the buffer for use. The address left is the first byte in the buffer available for data storage.

BUFSZ

Pronounced: buf-size
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: A constant giving the total length of a buffer. Its value is B/BUF +4.

C!

Pronounced: c-store
Stack Action: (b\addr ...)
Uses/Leaves: 2 0
Status:
Description: Stores byte b (8 bits) at the address addr.

C,

Pronounced: c-comma
Stack Action: (b ...)
Uses/Leaves: 1 0
Status:
Description: Stores (compiles) b in the next available dictionary byte advancing the dictionary pointer by one.

Glossary

C/L

Pronounced: c-slash-l
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: A constant containing the number of characters per line. This is normally 64, so a full FORTH 'line' will, in Mode 7, occupy about 1½ lines of the VDU display.

C@

Pronounced: c-fetch
Stack Action: (addr ... b)
Uses/Leaves: 1 1
Status:
Description: Leaves b as the 8-bit contents of the address addr.

CFA

Pronounced: c-f-a
Stack Action: (pfa ... cfa)
Uses/Leaves: 1 1
Status:
Description: Converts the parameter field address of a word to its code field (execution) address.

CHANNEL

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A variable used to store the channel number of the currently open file when accessing screens on disk.

Glossary

CLOSE

Pronounced:
Stack Action: (b ...)
Uses/Leaves: 1 0
Status:
Description: Closes the file whose channel number is b. If b is zero all open files are closed.

CMOVE

Pronounced: c-move
Stack Action: (from\to\count ...)
Uses/Leaves: -3 0
Status:
Description: Moves 'count' bytes, starting at 'from' to the block of memory starting at 'to'. The byte at 'from' is moved first and the transfer proceeds towards high memory. No check is made as to whether the destination area overlaps the source area. Nothing is moved if 'count' is zero or negative.

COLD

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: The cold start procedure used on first entry to the system. The dictionary pointer and user variables are initialised from the boot-up parameters and the system re-started via (ABORT) . The mass storage buffers are cleared, function keys 8 and 9 are initialised, and printer output is disabled. All vectored words are set to their default actions. It may be called from the keyboard to remove all application programs and restart with the nucleus dictionary alone.

COMPILE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: COMPILE acts during the execution of the word containing it. The code field (execution) address of the word following COMPILE is compiled into the dictionary instead of executing, cf. [COMPILE] .

CONSTANT

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: A defining word used in the form:

n CONSTANT CCCC

It creates a constant CCCC with the value n contained in its parameter field. When CCCC is executed the value n will be left on the stack.

CONTEXT

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable leaving the address of a pointer to the VOCABULARY in which a dictionary search will start.

CONVERT

Pronounced:
Stack Action: (nd1\addr1 ... nd2\addr2)
Uses/Leaves: 3 3
Status:
Description: Converts the text beginning at the address addr1 to the equivalent stack number. The value is accumulated into double number nd1, with regard to the current numeric base, being left as nd2. The address of the first non-convertible character is left in addr2.

COUNT

Pronounced:
Stack Action: (addr1 ... addr2\n)
Uses/Leaves: 1 2
Status:
Description: Leaves the address addr2 and byte count n of a text string starting at addr1, in a form suitable for use by TYPE . It is assumed that the text string has its count byte at addr1 and that the actual character string starts at addr1 + 1.

CR

Pronounced: c-r
Stack Action:
Uses/Leaves:
Status:
Description: Transmits a carriage return and line feed to the terminal output device.

CREATE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: A vectored routine initialised on a cold start to execute (CREATE) which creates a new dictionary header. Used as

```
CREATE CCCC
```

to create a dictionary header for the word CCCC with the code pointer of VARIABLE . Later execution of CCCC will therefore leave the address of the first byte of its parameter field. See also DOES> .

CREATE-SCREENS

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Intended only for use with disk-based systems. Creates, on the currently selected drive, a number of files for the storage of source screens. The disk must previously have been formatted with the *FORM40 and *FORM80 utilities. The number of files created is given by MAXFILES and each file contains S/FILE screens. A total of S/FILE * MAXFILES screens are created and they are numbered from 0.

When the routine is called a message 'Are you sure? (Y/N)?' is first given and no screens will be created unless the Y key is pressed in response. The routine will delete the contents of all screens previously stored on the disk. It also destroys the memory between &5800 and &7BFF inclusive. On exit the display will be in mode 7.

If memory above &5800 is being used by the system the routine will abort with an error message.

CSP

Pronounced: c-s-p
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable used for temporary storage of the stack pointer in checking of compilation errors.

CURRENT

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing a pointer to the vocabulary into which new definitions will be placed. As soon as a definition is made in the CURRENT vocabulary, it automatically becomes also the CONTEXT vocabulary.

D+

Pronounced: d-plus
Stack Action: (nd1\nd2 ... nd3)
Uses/Leaves: 4 2
Status:
Description: Leaves as nd3 the double number sum of double numbers nd1 and nd2.

Glossary

D+-

Pronounced: d-plus-minus
Stack Action: (nd1\n ... nd2)
Uses/Leaves: 3 2
Status:
Description: Applies the sign of single number n to the double number nd1, leaving the result nd2. See +-.

D.

Pronounced: d-dot
Stack Action: (nd ...)
Uses/Leaves: 2 0
Status:
Description: Prints the signed double number nd according to the current numeric base. One blank is printed after the number. See <.>.

D.R

Pronounced: d-dot-r
Stack Action: (nd\n ...)
Uses/Leaves: 3 0
Status:
Description: Prints a signed double number nd on the right of a field n characters wide. See<.R>. No trailing blank is printed.

D<

Pronounced: d-less-than
Stack Action: (nd1\nd2 ... f)
Uses/Leaves: 4 1
Status:
Description: Leaves a true flag if nd1 is less than nd2, and a false flag otherwise.

DABS

Pronounced:
Stack Action: (nd ... ud)
Uses/Leaves: 2 2
Status:
Description: Leaves the absolute value ud of a signed double number nd. See ABS .

DEC.

Pronounced: dec-dot
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Displays n in DECIMAL base, using the format of <.>, regardless of the current value of BASE .

DECIMAL

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Sets BASE to decimal numeric conversion for input and output.

DEFINITIONS

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Sets the CURRENT vocabulary to the CONTEXT vocabulary. If used in the form:

CCCC DEFINITIONS

where CCCC is a VOCABULARY word, all subsequent definitions will be placed in the vocabulary CCCC .

DEPTH

Pronounced:
Stack Action: (... n)
Uses/Leaves: 0 1
Status:

Description: Leaves the number of single-precision values on the stack, not counting n itself.

DIGIT

Pronounced:
Stack Action: (c\nl ... n2\tf)
Uses/Leaves: 2 2 [valid]
Stack Action: (c\nl ... ff)
Uses/Leaves: 2 1 [invalid]
Status:
Description: Converts ASCII character c, with base nl, to its binary equivalent n2 and a true flag. If c is not a valid character in base nl, then only a false flag is left.

DISK

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Switches to the Disk Operating System for mass storage. An error message is given if the Disk Operating System requires the use of memory allocated to FORTH . This is the default system in the version provided on disk.

DLITERAL

Pronounced: d-literal
Stack Action: (nd ...)
Uses/Leaves: 2 0 [compiling]
Status: I
Description: In the compiling state a double number nd is compiled as a double literal number in the dictionary. Later execution of the word including this literal number will replace nd on the stack.

In the execution mode DLITERAL has no effect.

DNEGATE

Pronounced: d-negate
Stack Action: (nd1 ... nd2)
Uses/Leaves: 2 2
Status:
Description: Change the sign of the double number nd1, leaving it as nd2.

DO

Pronounced:
Stack Action: (n1\n2 ...)
Uses/Leaves: 2 0
Status: I,C
Description: May only be used within a colon-definition in the forms

```
n1 n2 DO ... LOOP
n1 n2 DO ... +LOOP
```

This is the equivalent of a FOR ... NEXT loop in BASIC, repeating a sequence of operations a fixed number of times. The value of n1 is the loop limit and n2 is the initial value of the loop index. The loop terminates when the loop index equals or exceeds the limit. The sequence of operations in the loop will always be executed at least once. See I , LOOP , +LOOP , LEAVE .

DOES>

Pronounced: does-greater
Stack Action:
Uses/Leaves:
Status:
Description: Used with CREATE in the form:

```
 : NNNN CREATE ...  DOES> ... ;
```

It creates a new defining word NNNN . Executing NNNN in the form

```
NNNN CCCC
```

creates a new word CCCC whose parameter area is allocated by the words following CREATE and whose action is governed by the words following DOES> in NNNN .

DOVEC

Pronounced: do-vec
Stack Action: (addr\pfa ...)
Uses/Leaves: 2 0
Status:
Description: Converts the parameter field address pfa to its code field (execution) address and stores the result at the address addr. Used in the reassignment of execution vectors.

DP

Pronounced: d-p
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: The dictionary pointer, a user variable which leaves the address addr, whose contents point to the first free byte at the top of the dictionary.

DPL

Pronounced: d-p-l
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable which may be used to contain information about the position of the decimal point in a number. Not used in the system provided.

DR/W

Pronounced: disk-read-write
Stack Action: (addr\n\f ...)
Uses/Leaves: 3 0
Status:
Description: See R/W .

DROP

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Drops the top number on the stack.

DUP

Pronounced:
Stack Action: (n ... n\n)
Uses/Leaves: 1 2
Status:
Description: Duplicates the top number on the stack.

ELSE

Pronounced:
Stack Action:
Uses/Leaves:
Status: I,C
Description: Used in a colon-definition in the form:

IF ... ELSE ... THEN

During execution ELSE causes a branch to the words after THEN if the flag tested by IF was true, and is the destination of the branch taken at IF if the flag was false. See IF .

EMIT

Pronounced:
Stack Action: (c ...)
Uses/Leaves: 1 0
Status:
Description: A vectored routine, initialised on a COLD start to execute (EMIT), which transmits ASCII

character c to the output device. The contents of OUT are incremented for each character output. The stack value is masked to a 7-bit value before transmission.

EMPTY-BUFFERS

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Fill the whole of the mass storage buffer area with zeros. No data is written to mass storage.

ENCLOSE

Pronounced:
Stack Action: (addr\c ... addr\n1\n2\n3)
Uses/Leaves: 2 4
Status:
Description: The text-scanning primitive used by WORD . The text starting at the address addr is searched, ignoring leading occurrences of the delimiter c, until the first non-delimiter character is found. The offset from addr to this character is left as n1. The search continues from this point until the first delimiter after the text is found. The offsets from addr to this delimiter and to the first character not included in the scan are left as n2 and n3 respectively. The search will, regardless of the value of c, stop on encountering an ASCII null (0) which is regarded as an unconditional delimiter. The null is never included in the scan.
Examples:

Text at addr	n1	n2	n3
ccABCDcc	2	6	7
ABCDcc	0	4	5
ABC0cc	0	3	3
0ccc	0	1	0

ERASE

Pronounced:
Stack Action: (addr\n ...)
Uses/Leaves: 2 0
Status:
Description: Sets n bytes of memory starting at the address addr to contain zeroes.

ERROR

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Gives a notification of error number n. After a cold start this is in the form of an error number followed by a system ABORT . The vectored routines MESSAGE and ABORT are used and may therefore be reassigned by the user to modify the error response.

ESCAPE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Prints the message 'Escape' and re-enters the system via QUIT . This is the routine executed when the ESCAPE key is pressed.

EXECUTE

Pronounced:
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
Description: Executes the definition whose code field (execution) address is on the stack.

EXIT

Pronounced:
Stack Action:
Uses/Leaves: 0 0
Status:
Description: When compiled within a colon-definition, terminates execution of the definition at that point. It may not be used within a DO ... LOOP . It is also used to terminate the interpretation of mass storage.

EXPECT

Pronounced:
Stack Action: (addr\count ...)
Uses/Leaves: 2 0
Status:
Description: Transfers characters from the keyboard to the memory starting at the address addr until a <RETURN> (&0D) is found, or until the maximum count of characters has been received. Backspace deletes characters from both the display and the memory area but will not move past the starting point at the address addr. One or more nulls are added at the end of the text. Control characters are passed to the VDU but are not transferred to memory at the address addr.

EXVEC:

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Used in the form:

EXVEC: NNNN

It creates an execution-vectored word NNNN , initially assigned to execute NOVEC which gives an error message. The action of NNNN should then be assigned to execute some other word CCCC by the use of

ASSIGN NNNN TO-DO CCCC

The action of NNNN may be reassigned at any time, when all previously compiled uses of NNNN will be changed to the new assignment.

FENCE

Pronounced:
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
Description: A user variable containing an address below which the user is not allowed to FORGET . In order to use FORGET on an entry below this point it is necessary to alter the contents of FENCE .

FILL

Pronounced:
Stack Action: (addr\n\b ...)
Uses/Leaves: 3 0
Status:
Description: Fills n bytes of memory starting at the address addr with the value b.

FIND

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: Used as

FIND NNNN

and leaves the code field (execution) address of the next word name NNNN found in the input stream. If that word cannot be found in a search of CONTEXT and then FORTH , zero is left.

FIRST

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A constant that leaves the address addr of the first byte of the mass storage buffer area.

FLUSH

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Transfers all UPDATEd screens from the buffers to mass storage, marking all buffers as empty.

FNAME

Pronounced: file-name
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: Gives the address of the start of an 8-byte region of memory containing the name of the last-used disk file. The contents of this memory should not be changed by the user.

FORGET

Pronounced:
Stack Action:
Uses/Leaves:
Status: E
Description: Used in the form:

FORGET CCCC

It deletes the definition with name CCCC and all dictionary entries following it. An error message is given if the word cannot be found in the CURRENT or FORTH vocabularies. An error message is also given if CCCC is in the protected area of the dictionary, below FENCE .

FORTH

Pronounced:
Stack Action:
Uses/Leaves:
Status: I
Description: The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. It is IMMEDIATE so it will execute if used during the creation of a colon-definition. Until other vocabularies are defined, all new words become a part of FORTH. All other vocabularies ultimately link to the FORTH vocabulary.

H.

Pronounced: hex-dot
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Displays n in HEXADECIMAL base, using the the format of <.>, regardless of the current value of BASE .

HERE

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: Leaves the contents of DP , i.e. the address of the first unused byte in the dictionary.

HEX

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Sets the numeric conversion BASE to sixteen (hexadecimal).

HLD

Pronounced: h-l-d
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the address of the latest character of text produced during numeric output conversion (by #).

HOLD

Pronounced:
Stack Action: (c ...)
Uses/Leaves: 1 0
Status:
Description: Used between <# and #> to insert an ASCII character c into a converted numeric string. 2E (hex) HOLD will place a decimal point in the string.

I

Pronounced:
Stack Action: (... n)
Uses/Leaves: 0 1
Status: C
Description: Used in a DO ... LOOP to place the current value of the loop index on the stack. It must be used at the same level of nesting as the DO ... LOOP i.e. it will not operate correctly if included in a colon-definition word between DO and LOOP .

ID.

Pronounced: i-d-dot
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
Description: Prints the name of a word from its name field address on the stack.

IF

Pronounced:
Stack Action: (f ...)
Uses/Leaves: 1 0
Status: P,C
Description: Used in a colon-definition in the forms

a) IF (true) ... THEN
b) IF (true) ... ELSE (false) ... THEN

If the flag f is true, the sequence of words after IF is executed and execution is then transferred to the word immediately following THEN . If f is false, execution transfers

a) to the word following THEN , or
b) to the sequence of words following ELSE and subsequently to the first word after THEN .

IMMEDIATE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Sets the precedence bit of the most recently defined word so that it will execute rather than being compiled during the compilation of a word definition. See [COMPILE] .

INDEX

Pronounced:
Stack Action: (n1\n2 ...)
Uses/Leaves: 2 0
Status:
Description: Lists the first lines of screens n1 to n2 inclusive from mass storage.

INITBUF

Pronounced: init-buf
Stack Action:
Uses/Leaves:
Status:
Description: Initialises the mass storage buffer area to hold the number of screens contained in the constant MINBUF and marks all buffers as empty. The number of buffers is also stored in the variable #BUF .

INITVECS

Pronounced: init-vecs
Stack Action:
Uses/Leaves:
Status:
Description: Initialises all vectored words in the nucleus dictionary to their default assignments.

INTERPRET

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: The outer text interpreter which either executes or compiles a text sequence, depending on STATE , from the current input buffer (terminal or tape). If the word name cannot be found after a search of the CONTEXT and then the FORTH vocabularies, it is converted to a number using the current base. If this conversion also fails an error message is given.

If a decimal point is found as the last character of a number a double number will be left on the stack. The number itself will not contain any reference to the decimal point.

J

Pronounced:
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: Returns the index of the next outer loop. It may only be used within a nested DO-LOOP of the form

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

KEY

Pronounced:
Stack Action: (... c)
Uses/Leaves: 0 1
Status:
Description: A vectored routine initialised on a cold start to execute (KEY) which leaves the ASCII

value of the next available character from the current input device.

KEY'

Pronounced: key-quote
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Used in the form:

n KEY' text '

It programs user-defined key n to execute the following text, up to the terminating single quote. A <RETURN> may be embedded in the text by including |M, as described in chapter 25 of the BBC Microcomputer User Guide. When used in a colon-definition n must be either a literal numeric value or a constant appearing immediately before KEY' .

LAST

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: Leaves the name field address of the most recently defined word in the CURRENT vocabulary.

LEAVE

Pronounced:
Stack Action:
Uses/Leaves:
Status: C
Description: Forces the termination of a DO ... LOOP at the first following time that LOOP or +LOOP is reached. This is done by setting the loop limit equal to the current value of the loop index, which is not changed. Execution will continue normally until

reaching LOOP or +LOOP .

LFA

Pronounced: l-f-a
Stack Action: (pfa ... lfa)
Uses/Leaves: 1 1
Status:
Description: Converts the parameter field address pfa to its link field address lfa.

LIMIT

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A constant leaving the address of the first byte after the highest memory available for the tape I/O buffer.

LIST

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Lists screen n. If screen n is not in the buffers it is read from mass storage. The numeric conversion base is set to DECIMAL.

LIT

Pronounced:
Stack Action: (... n)
Uses/Leaves: 0 1
Status: C
Description: Within a colon-definition, LIT is automatically compiled before each 16-bit literal number encountered in the input text. Later execution of LIT causes the contents of the following two bytes to be pushed onto the stack.

LITERAL

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status: I,C
Description: During compilation the stack value n is compiled into the dictionary entry as a 16-bit (single-precision) number.

A possible use is:

```
: NNNN ... [ calculate a value ] LITERAL ... ;
```

Compilation is suspended (by [ ) for a value to be calculated and then resumed (by]) for LITERAL to compile the value into the definition of NNNN .

LOAD

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Makes screen n the input stream, preserving the position in the present input stream, and begins interpretation of screen n. When the new input stream is terminated or exhausted, control

returns to the original input stream. Screen 0 may not be loaded.

LOOP

Pronounced:
Stack Action:
Uses/Leaves:
Status: I,C
Description: Used in a colon-definition in the form:

DO ... LOOP

During execution LOOP controls branching back to the corresponding DO , dependent on the loop index and loop limit. The loop index is incremented by one and tested against the loop limit. Branching to DO continues until the index is equal to or greater than the limit when execution continues with the word following LOOP .

M*

Pronounced: m-times
Stack Action: (n1\n2 ... nd)
Uses/Leaves: 2 2
Status:
Description: Leaves as the double-precision number nd the signed product of the two signed single-precision numbers n1 and n2.

M/

Pronounced: m-divide
Stack Action: (nd\n1 ... n2\n3)
Uses/Leaves: 3 2
Status:
Description: Leaves, as the single numbers n2 and n3 respectively, the signed remainder and signed quotient from the division of the double number dividend nd by the single number divisor n1. The sign of the

remainder is that of the dividend.

M/MOD

Pronounced: m-divide-mod
Stack Action: (ud1\u2 ... u3\ud4)
Uses/Leaves: 3 3
Status:
Description: Leaves, as the double number ud4 and the single number u3 respectively, the quotient and remainder from the division of the double number dividend ud1 by the single number divisor u2. All are unsigned integers.

MAX

Pronounced:
Stack Action: (n1\n2 ... max)
Uses/Leaves: 2 1
Status:
Description: Leaves as max the larger of n1 and n2.

MAXFILES

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: A constant returning the maximum number of mass-storage files used to store screens in a disk-based system. It is initally set to 20 to allow the use of double-density drives. See CREATE-SCREENS .

MESSAGE

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: A vectored error routine initialised on a cold start to execute MSG# which displays n as an error message number.

MIN

Pronounced:
Stack Action: (n1\n2 ... min)
Uses/Leaves: 2 1
Status:
Description: Leaves as min the smaller of n1 and n2.

MINBUF

Pronounced:
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: A constant which leaves the minimum number of mass storage buffers to be used by the system. It is initially set to two but may be increased by the user. It should not be reduced since at least two buffers are required for the correct operation of the system.

MOD

Pronounced:
Stack Action: (n1\n2 ... mod)
Uses/Leaves: 2 1
Status:
Description: Leaves as mod the remainder from the division of n1 by n2, with the sign of n1.

MODE

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Sets the VDU display to mode n. Modes 4-7 inclusive are allowed.

MOVE

Pronounced:
Stack Action: (from\to\count ...)
Uses/Leaves: 3 0
Status:
Description: Moves 'count' words (16-bit values) starting at 'from' to the block of memory starting at 'to'. The 16-bit value at 'from' is moved first and the transfer proceeds towards high memory. No check is made as to whether the destination and source areas overlap. If 'count' is zero or negative nothing is moved.

MSG#

Pronounced: m-s-g-hash
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: See MESSAGE .

NEGATE

Pronounced:
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Changes the sign of n1 and leaves the result as n2. The sign is changed by forming the two's complement.

NFA

Pronounced: n-f-a
Stack Action: (pfa ... nfa)
Uses/Leaves: 1 1
Status:
Description: Converts the parameter field address of a definition to its name field address.

NOOP

Pronounced: no-op
Stack Action:
Uses/Leaves:
Status:
Description: A no-operation in FORTH. One possible use is to reserve address space in a colon-definition for later over-writing by the execution address of a subsequent definition.

NOT

Pronounced:
Stack Action: (f1 ... f2)
Uses/Leaves: 1 1
Status:
Description: Reverse the boolean value of f1 leaving the result as f2. It is identical to 0= .

NOVEC

Pronounced: no-vec
Stack Action:
Uses/Leaves:
Status:
Description: The action assigned to a newly-created execution vector. See EXVEC: .

NUM

Pronounced:
Stack Action:
a) (addr ...) (compiling)
b) (addr ... n) (executing)
c) (addr ... nd)
Uses/Leaves:
a) 1 0
b) 1 1
c) 1 2
Status:
Description: A vectored routine initialised on a cold start to execute (NUM) . It is used in INTERPRET for number conversion. In the compiling state the number is compiled into the dictionary but in execution mode the number is left in the stack.

In both cases addr is the address of the count byte of a string to be converted to a number. The string may contain a leading minus sign and a trailing decimal point. If the decimal point is present a double-precision number results otherwise a single-precision number is produced.

The routine may be reassigned by the user to allow INTERPRET to handle different numeric formats.

NUMBER

Pronounced:
Stack Action: (addr ... nd)
Uses/Leaves: 1 2
Status:
Description: Converts the character string starting with a character count byte at the address addr to the signed double number nd using the current numeric base. If a valid numeric conversion is not possible an error message will be given. The string may contain a leading negative sign and a trailing decimal point. The decimal point is ignored by NUMBER .

OFFSET

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable which may contain a block (screen) offset for mass storage. The contents of OFFSET are added to the screen number on the stack by BLOCK . It is initialised to zero on a cold start.

OPEN

Pronounced:
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
Description: Opens a named file for reading and writing. The name of the file starts at the address addr, as described in (OPEN) . If the file is opened successfully its channel number is stored in the variable CHANNEL . If the attempt to open the file does not succeed an error message is given and the previous contents of CHANNEL are maintained.

OR

Pronounced:
Stack Action: (n1\n2 ... or)
Uses/Leaves: 2 1
Status:
Description: Leaves as 'or' the bit-by-bit logical OR of n1 and n2.

OS'

Pronounced: o-s-quote
Stack Action:
Uses/Leaves:
Status: I
Description: Used as

```
OS' text '
```

to transmit text to the operating system command line interpreter. In compile mode the text is compiled together with the address of a routine to transmit the text to the command line interpreter. In execution mode the text is transmitted directly. The required closing &0D is supplied automatically.

OSCLI

Pronounced:
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
Description: The machine code routine used to transfer the text string, whose first character is at the address addr and which has a terminating &0D, to the operating system command line interpreter.

OSERROR

Pronounced: o-s-error
Stack Action:
Uses/Leaves:
Status:
Description: The routine executed when an operating system error is detected. The error message number is given in decimal base and the relevant operating system error message is displayed. Control is returned to the keyboard via (WARM) .

OUT

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing a value that is incremented by EMIT . It may be examined and changed by the user to control display formats.

OVER

Pronounced:
Stack Action: (n1\n2 ... n1\n2\n1)
Uses/Leaves: 2 3
Status:
Description: Copies the second stack item over the top item.

PAD

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A constant which leaves the address of the start of the text scratchpad buffer. Numeric output characters are stored downwards from PAD , character text is stored upwards.

PFA

Pronounced: p-f-a
Stack Action: (nfa ... pfa)
Uses/Leaves: 1 0
Status:
Description: Converts the name field address nfa of a dictionary entry to its parameter field address pfa.

PICK

Pronounced:
Stack Action: (n1 ... n2)
Uses/Leaves: 1 1
Status:
Description: Returns the contents of the 'n1-th' stack value, not counting n1 itself. Nothing is done if n1 is less than one. The value returned will be meaningless if n1 is greater than DEPTH .

1 PICK is equivalent to DUP
2 PICK is equivalent to OVER

PLOT

Pronounced:
Stack Action: (n\x\y ...)
Uses/Leaves: 3 0
Status:
Description: Used to plot points, lines and triangles. The value of n determines the type of plotting action and x and y are the coordinates. The action is similar to that of the PLOT keyword in the BBC Microcomputer User Guide. For example,

```
4       100       50       PLOT
```

will move to the absolute position (100,50) without drawing a line.

PREV

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A variable containing a pointer to the start of the most recently used mass storage buffer.

PRUNE

Pronounced:
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status:
Description: Scans through the dictionary, starting at the most recently defined word, cutting off any vocabulary links that extend beyond the address addr. Used by FORGET to ensure that no links in any VOCABULARY word are left pointing into the discarded area of the dictionary.

QUERY

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Inputs up to 80 characters terminated by <RETURN> (&0D) from the keyboard. The text is stored in the terminal input buffer whose address is given by TIB . The value of >IN is set to zero (in preparation for interpretation by INTERPRET).

QUIT

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Clears the return stack, stops and returns control to the keyboard. No message is given.

R#

Pronounced: r-sharp
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable which contains the location of the editing cursor for the Editor.

R/W

Pronounced: read-write
Stack Action: (addr\n\f ...)
Uses/Leaves: 3 0
Status:
Description: A vectored routine to perform mass storage read/write operations. It is initialised on a COLD start to execute either TR/W (tape version) or DR/W (disk version). The flag f indicates read (true) or write (false). Screen n is transferred to or from the mass storage buffer whose first byte of data is at the address addr.

R0

Pronounced: r-zero
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the initial address of the top of the return stack.

R:

Pronounced: r-colon
Stack Action:
Uses/Leaves:
Status:
Description: A recursive version of <:> used as

```
R: NNNN .... R;
```

With this form of colon-definition references may be made from within the definition to the name NNNN itself. It should be used with care since any error during compilation will leave the incomplete definition in an executable form.

R;

Pronounced: r-semi-colon
Stack Action:
Uses/Leaves:
Status:
Description: The form of <;> used to terminate a recursive colon-definition.

R>

Pronounced: r-from
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: Removes the top value from the return stack and leaves it on the computation stack. See >R .

R@

Pronounced: r-fetch
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: Copy the top of the return stack to the computation stack. The action is identical to that of I .

REPEAT

Pronounced:
Stack Action:
Uses/Leaves:
Status: I
Description: Used in a colon-definition in the form:

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

In execution REPEAT forces an unconditional branch back to BEGIN .

ROLL

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 1
Status:
Description: Rotates the top n stack items so that the nth item is moved to the top.

1 ROLL has no effect
2 ROLL is equivalent to SWAP
3 ROLL is equivalent to ROT

No action is taken if n is less than 1.

ROT

Pronounced:
Stack Action: (n1,n2,n3 ... n2,n3,n1)
Uses/Leaves: 3 3
Status:
Description: Rotates the top three items on the stack, bringing the third item to the top.

RP !

Pronounced: r-p-store
Stack Action:
Uses/Leaves:
Status:
Description: Initialises the return stack pointer.

RP @

Pronounced: r-p-fetch
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: Leaves the address of the return stack pointer. Note that this points one byte below the last return stack value.

S->D

Pronounced: s-to-d
Stack Action: (n ... nd)
Uses/Leaves: 1 2
Status:
Description: Leaves as nd the signed single-precision number n converted to the form of a signed double-precision number (with unchanged value).

S/FILE

Pronounced: s-per-file
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: A constant returning the number of screens per disk file.

S0

Pronounced: s-zero
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the address which marks the initial top of the computation stack.

SAVE-BUFFERS

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Transfers all UPDATEd screens from the buffers to mass storage. The contents of the buffers are not changed.

SCR

Pronounced: s-c-r
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the number of the most recently listed source text screen.

SETBUF

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Allocates memory for mass storage buffers without erasing the current contents. The number of buffers to be used is found from the contents of #BUF and must be greater than or equal to 2.

SIGN

Pronounced:
Stack Action: (n\nd ... nd)
Uses/Leaves: 3 2
Status:
Description: Stores an ASCII '-' sign in the converted numeric output string at PAD if n is negative. The sign of n is usually that of the double number to be converted. Although n is discarded the double number nd is kept either for further conversion or to be dropped by #> . SIGN may only be used between <# and #> .

SMUDGE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: TOGGLEs the 'smudge bit' in the name header of the most recently created definition in the CURRENT vocabulary. This switches between enabling and disabling the finding of the entry during a dictionary search.

The name field is smudged during the definition of a word to prevent the incomplete definition from being found, and then smudged again on completion.

SP!

Pronounced: s-p-store
Stack Action:
Uses/Leaves:
Status:
Description: Initialises the computation stack pointer (i.e. clears the stack).

SP@

Pronounced: s-p-fetch
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: Leaves the value of the stack pointer on the stack. The value corresponds to the state of the stack before the operation.

SPACE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Transmits an ASCII blank to the output device.

SPACES

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Transmits n ASCII blanks to the output device.

ST-ADDR

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Calculates the correct COLD and WARM entry addresses and passes them to the user-defined key initialisation routine START-KEYS .

START

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: The high-level entry point to FORTH on a cold start. The computation and return stacks are cleared. Any applications dictionary is discarded and all vectored words are initialised to their default values. The mass storage buffers are initialised to the number of buffers given by MINBUF and marked as being empty; OFFSET is set to zero. User-defined keys 8 and 9 are programmed for the correct WARM and COLD entry points respectively and printer output is disabled. Control is passed to the keyboard interpreter via (ABORT) .

START-KEYS

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Initialises the user-defined keys 8 and 9 to perform a warm and cold start respectively. These functions are best used only after the BREAK key has been pressed.

STATE

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable indicating the state of compilation. A zero value indicates execution and a non-zero value indicates compilation.

STRING

Pronounced:
Stack Action: (c ... addr\count)
Uses/Leaves: 1 2
Status:
Description: Uses the delimiter character with ASCII code c to accept text from the input stream up to the first appearance of the delimiter. The address of the first character of the string and its length are left. The delimiter may be any character except a space; a null (00) is regarded as an unconditional delimiter.

SWAP

Pronounced:
Stack Action: (n1\n2 ... n2\n1)
Uses/Leaves: 2 2
Status:
Description: Exchanges the top two items on the stack.

TAPE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Selects the tape operating system. This is the default system in the version provided on tape.

TEXT

Pronounced:
Stack Action: (addr\count ...)
Uses/Leaves: 2 0
Status:
Description: Compiles the text of the string, whose first character is at the address addr and of length 'count', into the dictionary with a preceding length byte.

THEN

Pronounced:
Stack Action:
Uses/Leaves:
Status: I,C
Description: Used in a colon-definition in the forms:

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

Marks the destination of forward branches from IF or ELSE as the conclusion of the conditional structure. See IF .

TIB

Pronounced: t-i-b
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the address of the terminal input buffer.

TLD

Pronounced: tape-load
Stack Action:
Uses/Leaves:
Status:
Description: Used by TR to load a screen from tape. It is not designed for use independently of TR .

TO-DO

Pronounced:
Stack Action: (addr ...)
Uses/Leaves: 1 0
Status: I
Description: Accepts a name from the input stream and stores its code field (execution) address at addr. Used in the assignment of execution vectors. See ASSIGN .

TOGGLE

Pronounced:
Stack Action: (addr\b ...)
Uses/Leaves: 2 0
Status:
Description: Complements the contents of the address addr by the bit pattern b.

TR

Pronounced: tape-read
Stack Action: (addr\n ...)
Uses/Leaves: 2 0
Status:
Description: Reads screen n from tape and stores it in the 1024 bytes starting at the address addr.

TR/W

Pronounced: tape-read-write
Stack Action: (addr\n\f ...)
Uses/Leaves: 3 0
Status:
Description: The Cassette Operating System version of R/W. This is the default routine in systems provided on tape. See R/W .

TRAVERSE

Pronounced:
Stack Action: (addr1\n ... addr2)
Uses/Leaves: 2 1
Status:
Description: Moves across the name field of a dictionary entry. If n=1, addr1 should be the address of the name length byte (i.e. the NFA of the word) and the movement is towards high memory. If n= -1, addr1 should be the last letter of the name and the movement is towards low memory. The addr2 that is left is the address of the other end of the name.

TRIAD

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Performs a form-feed and then lists the three screens which start at a multiple of 3 and contain screen n. The format is suitable for a standard page of continuous printer stationery.

TSV

Pronounced: tape-save
Stack Action:
Uses/Leaves:
Status:
Description: Uses the Operating System command line interpreter to save a screen to tape. Used by TW .

TW

Pronounced: tape-write
Stack Action: (addr\n ...)
Uses/Leaves: 2 0
Status:
Description: Saves the 1024 bytes starting at the address addr to tape as screen n. The screen number, as a four-digit hexadecimal number, is used as the file name.

TYPE

Pronounced:
Stack Action: (addr\count ...)
Uses/Leaves: 2 0
Status:
Description: Transmits 'count' characters of a string starting at the address addr to the output device.

U*

Pronounced: u-times
Stack Action: (u1\u2 ... ud)
Uses/Leaves: 2 2
Status:
Description: Leaves the unsigned double-precision product of two unsigned numbers.

U.

Pronounced: u-dot
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: Transmits the 16-bit value n to the output device. n is represented as an unsigned integer in the current numeric conversion base. A trailing space is printed.

U/

Pronounced: u-divide
Stack Action: (ud\ul ... u2\u3)
Uses/Leaves: 3 2
Status:
Description: Leaves the unsigned remainder u2 and unsigned quotient u3 from the division of the unsigned double number dividend ud by the unsigned divisor ul. No protection is given against arithmetical overflow or division by zero.

U/MOD

Pronounced: u-divide-mod
Stack Action: (ud\ul ... u2\u3)
Uses/Leaves: 3 2
Status:
Description: The action is similar to that of U/ except that an error message is given if division by zero is attempted. All other division words use U/MOD as their basis and are therefore protected against division by zero.

U<

Pronounced: u-less-than
Stack Action: (un1\un2 ... f)
Uses/Leaves: 2 1
Status:
Description: Unsigned comparision. Leaves a true flag if un1 is less than un2, otherwise leaves a false flag.

UNTIL

Pronounced:
Stack Action: (f ...)
Uses/Leaves: 1 0
Status: I,C
Description: Used in a colon-definition in the form

BEGIN ... UNTIL

If f is false execution branches back to the corresponding BEGIN .

If f is true execution continues with the next word after UNTIL .

UPDATE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: A vectored routine, initialised on a cold start, to execute (UPDATE) which marks the current editing screen as having been changed. Any UPDATEd screen will be saved automatically before its buffer is reused. In a tape-based system its action is modified to execute NOOP, to simplify the use of the Editor.

USE

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A variable containing the address of the mass storage buffer to use next, as the least recently written.

USER

Pronounced:
Stack Action: (n ...)
Uses/Leaves: 1 0
Status:
Description: A defining word used in the form:

n USER CCCC

It creates a user variable CCCC , execution of which leaves the address, in the user area, of the value of CCCC . The value of n is the offset from the start of the user variable area to the memory location (2 bytes) in which the value is stored. The value is not initialised. Offsets from 0 to &30 inclusive are used by the system.

VARIABLE

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: A defining word used in the form:

VARIABLE CCCC

It creates a variable CCCC with initial value zero. Execution of CCCC leaves the address, in the parameter

area of CCCC , containing the value.

VLIST

Pronounced: v-list
Stack Action:
Uses/Leaves:
Status:
Description: Display, on the output device, a list of the names of all words in the CONTEXT vocabulary and any other vocabulary to which the CONTEXT vocabulary is chained. All VLISTs will therefore include a listing of the words in the FORTH vocabulary. The listing can be interrupted by pressing the TAB key and resumed by pressing the Space Bar. If, after interruption, any key except the Space Bar is pressed, the listing will be aborted.

VOC-LINK

Pronounced: voc-link
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the address of a vocabulary link field in the word which defines the most recently created vocabulary. All vocabularies are linked through these fields in their defining words.

VOCABULARY

Pronounced:
Stack Action:
Uses/Leaves:
Status: E
Description: A defining word used in the form:

VOCABULARY CCCC

It creates a defining word for a vocabulary with name CCCC . Execution of CCCC makes it the CONTEXT vocabulary in which a dictionary search will start. Execution of the sequence

CCCC DEFINITIONS

will make CCCC the CURRENT vocabulary into which new definitions are placed. Vocabulary CCCC is so linked that a dictionary search will also find all words in the vocabulary in which CCCC was originally defined. All vocabularies, therefore, ultimately link to FORTH .

By convention all vocabulary defining words are declared IMMEDIATE .

WARM

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: Performs a warm start. The stacks are cleared. The CURRENT and CONTEXT vocabularies are set to FORTH, and DECIMAL numeric base is selected. No other initialisation takes place. In particular the user's dictionary and the contents of the buffers are preserved. All vectored routines maintain their current assignments.

WARNING

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable whose value can be used to determine the action on detection of non-Operating System errors. A negative value will cause execution of ABORT when an error occurs, otherwise an error message is given. The user may, by reassigning MESSAGE, test for a positive value to control the form of error message given. The value is

initialised to zero on a cold start.

WBFR

Pronounced: word-buffer
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A constant returning the address of the first byte of the buffer used by WORD .

WDSZ

Pronounced: word-size
Stack Action: (... n)
Uses/Leaves: 0 1
Status:
Description: A constant returning the length in bytes of the buffer used by WORD . It is set to a value of 258, allowing strings of up to 256 (WDSZ-2) characters to be handled.

WHILE

Pronounced:
Stack Action: (f ...)
Uses/Leaves: 1 0
Status: I,C
Description: Used in a colon-definition in the form:

BEGIN ... WHILE ... REPEAT

WHILE tests the top value on the stack. If it is true execution continues to REPEAT which forces a branch back to BEGIN . If f is false execution skips to the first word after REPEAT . See BEGIN .

WIDTH

Pronounced:
Stack Action: (... addr)
Uses/Leaves: 0 1
Status:
Description: A user variable containing the maximum number of letters saved during the compilation of a definition's name. It must be a value between 1 and 31 inclusive and has a default value of 31. The value may be changed at any time provided it is kept within the above limits. Use of a value less than 3 is not recommended.

WORD

Pronounced:
Stack Action: (c ... addr)
Uses/Leaves: 1 1
Status:
Description: Accepts characters from the input stream until the non-zero delimiting character c is encountered, or the input stream is exhausted. Leading delimiters are ignored. The characters are stored as a packed string with the character count in the first position. The actual delimiter encountered (c or null) is stored at the end of the text but not included in the count. If the input stream is exhausted when WORD is called then a zero length will result. The address of the count byte of the string is left on the stack.

X

Pronounced:
Stack Action:
Uses/Leaves:
Status:
Description: This is a pseudonym for the dictionary entry whose name is one character of ASCII null (00). It is the procedure to terminate interpretation of text from the input buffer, since the buffer has at least one null character at the end.

XOR

Pronounced: x-or
Stack Action: (n1\n2 ... xor)
Uses/Leaves: 2 1
Status:
Description: Leaves the bit-by-bit logical Exclusive-OR of n1 and n2.

[

Pronounced: left-bracket
Stack Action:
Uses/Leaves:
Status:
Description: Used in the creation of a colon-definition in the form:

```
: NNNN ... [ ... ] ... ;
```

It suspends compilation of the definition and allows the subsequent input to be executed. See] .

[COMPILE]

Pronounced: bracket-compile
Stack Action:
Uses/Leaves:
Status: I,C
Description: Used in the creation of a colon-definition to force the compilation of an IMMEDIATE word which would otherwise execute. The most frequent use is with vocabulary words for example.

```
[COMPILE] FORTH
```

to delay the change of the CONTEXT vocabulary to FORTH until the word containing the above sequence executes.

]

Pronounced: right-bracket
Stack Action:
Uses/Leaves:
Status:
Description: Used during execution mode to force compilation of the subsequent input. See [.

Appendix A

The FORTH-79 Standard

The FORTH-79 Standard document, produced by the FORTH Standards Team and distributed by the FORTH Interest Group, specifies the requirements of a 79-standard system. Copies of this document are available from the FORTH Interest Group UK (see Appendix E).

1 System requirements

The minimum requirements of the host computer are given below. The resources provided in Acornsoft FORTH are given in brackets.

1) 2000 (5000 min) bytes of memory for application dictionary.
2) Data stack of 64 (72) bytes.
3) Return stack of 48 (234) bytes.
4) Mass storage capacity of 32 (90 or more) blocks, numbered consecutively from zero.
5) One ASCII input/output device acting as an operator's terminal (keyboard+VDU).

2 Required word set

A FORTH-79 Standard system must include all words in the Required Word Set. This word set is given below, divided into groups to show like characteristics. The lower case entries refer to the run-time code corresponding to a compiling word and need not be present as a dictionary entry separate from the corresponding compiling word.

Nucleus words

! * */ */MOD + +! +loop - /MOD 0< 0= 0> 1+ 1- 2+ 2- < = > >R ?DUP @ ABS AND BEGIN C! C@ colon CMOVE constant create D+ D< DEPTH DNEGATE do does> DROP DUP else EXECUTE EXIT FILL I if J LEAVE literal loop MAX MIN MOD MOVE NEGATE NOT OR OVER PICK R> R@ repeat ROLL ROT semicolon SWAP then U* U/ until variable while XOR

Interpreter words

£ £> £S ' (-TRAILING 79-STANDARD <£ >IN ? ABORT BASE BLK CONTEXT CONVERT COUNT CR CURRENT DECIMAL EMIT EXPECT FIND FORTH HERE HOLD KEY PAD QUERY QUIT SIGN SPACE SPACES TYPE U. WORD

Compiler words

+LOOP , ." : ; ALLOT BEGIN COMPILE CONSTANT CREATE DEFINITIONS DO DOES> ELSE FORGET IF IMMEDIATE LITERAL LOOP REPEAT STATE THEN UNTIL VARIABLE VOCABULARY WHILE [[COMPILE]]

Device words

BLOCK BUFFER EMPTY-BUFFERS LIST LOAD SAVE-BUFFERS SCR UPDATE

3 Usage requirements

A FORTH Standard program may reference only the definitions of the Required Word Set, and definitions which are subsequently defined in terms of these words. Furthermore, a FORTH Standard program must use the standard words as required by any conventions of the Standard. Equivalent execution on any FORTH-79 Standard system must result from Standard programs.

In a Standard program the user may only operate on data which was stored by the application.

A Standard program may address:

1) parameter fields of variables, constants and DOES> words. A DOES> word's parameter field may only be addressed with respect to the address left by DOES> itself.

2) dictionary space ALLOTed.

3) data in mass storage block buffers.

4) the user area and PAD .

A Standard program may NOT address:

1) directly into the data or return stacks.

2) into a definition's name field, link field or code field.

3) into a definition's parameter field if not stored by the application.

FORTH Standard definitions may not be redefined in a Standard system or Standard program.

***IMPORTANT NOTE**
These restrictions are severe. It must be emphasised that the demonstrations and applications in this manual are not intended to be Standard programs.

4 Error conditions

A number of words in the FORTH-79 Standard have an associated error condition and the system action for each of these must be specified. The following list gives the action in Acornsoft FORTH for each error condition. All other errors are described either in chapter 13 or in the individual glossary entries for the words concerned.

' An error condition exists if the word following <'> can not be found in either the CONTEXT or FORTH vocabularies. The word's name is repeated, followed by a question mark and control is returned to the keyboard via ABORT .

(There is an error condition if the input stream is exhausted before a right parenthesis is found. The comment is simply terminated.

-TRAILING A negative string length count on the stack is an error condition. An error message is given.

." There is an error condition if the input stream is exhausted before a <"> is found. In this case the string is terminated and no other action is taken.

79-STANDARD
This simply marks the point in the dictionary where all the FORTH-79 Required Word Set is included. No checks are made on the system. FORGETing or changing words before 79-STANDARD is an error.

If, during compilation, a word is encountered that cannot be found in the CURRENT or FORTH vocabularies, an attempt is made to convert to a number. If this attempt also fails there is an error condition. The word's name is repeated, followed by a question mark, compilation is terminated and control is returned to the keyboard.

; During compilation, an error message is given if the input stream is exhausted before a <;> is found. Compilation is terminated.

BLOCK If the specified block number is out of range error message 6 is given. If a read or write fails for any other reason an Operating System error message is given.

BUFFER A block number out of range or a failure to write to mass storage results in an error message as for BLOCK .

FORGET Error message 24 is given if the specified word can not be found in a search of the CURRENT or FORTH vocabularies. If the specified word is found in the protected area of the dictionary, error message 21 is given.

PICK An error condition exists if the count on the stack is less than 1. In this case PICK has no action.

ROLL As for PICK.

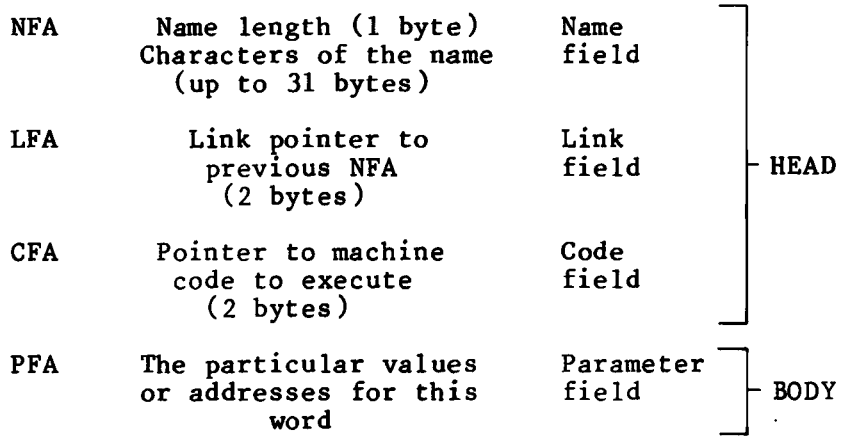
SAVE-BUFFERS
As for BLOCK and BUFFER .

Appendix B

How FORTH works

1 The structure of a dictionary entry

The basic structure of a dictionary entry in FORTH may vary slightly from one implementation to another but all FORTH words generally consist of two main parts, the head and the body. The head of the entry usually contains some information about the name of the word, a pointer to the previous word in the dictionary and a pointer to some actual machine code. The order in which these appear may vary and the following description applies to Acornsoft FORTH. This form is also used by the 'fig-model' and is probably the most common method used:



2 The interpretation and execution of FORTH

FORTH is an interpretive threaded language. This means that the instructions which make up an application are stored as a list of previously defined routines. This list is threaded together during the entry of source

code from either the keyboard or the mass storage buffers.

The process of producing the list is often termed compilation, but this is not strictly accurate since the result of true compilation should be native machine code. Most implementations of FORTH (this one included) store a list of addresses rather than pure machine code and so the process should more accurately be termed interpretation. Since the result is a list of pointers to locations containing the address of machine code (i.e. to the CFA's) rather than the start of the code routines themselves, this type of implementation is known as indirect threaded code. If the pointers do indicate the actual start of machine code the implementation is using direct threaded code.

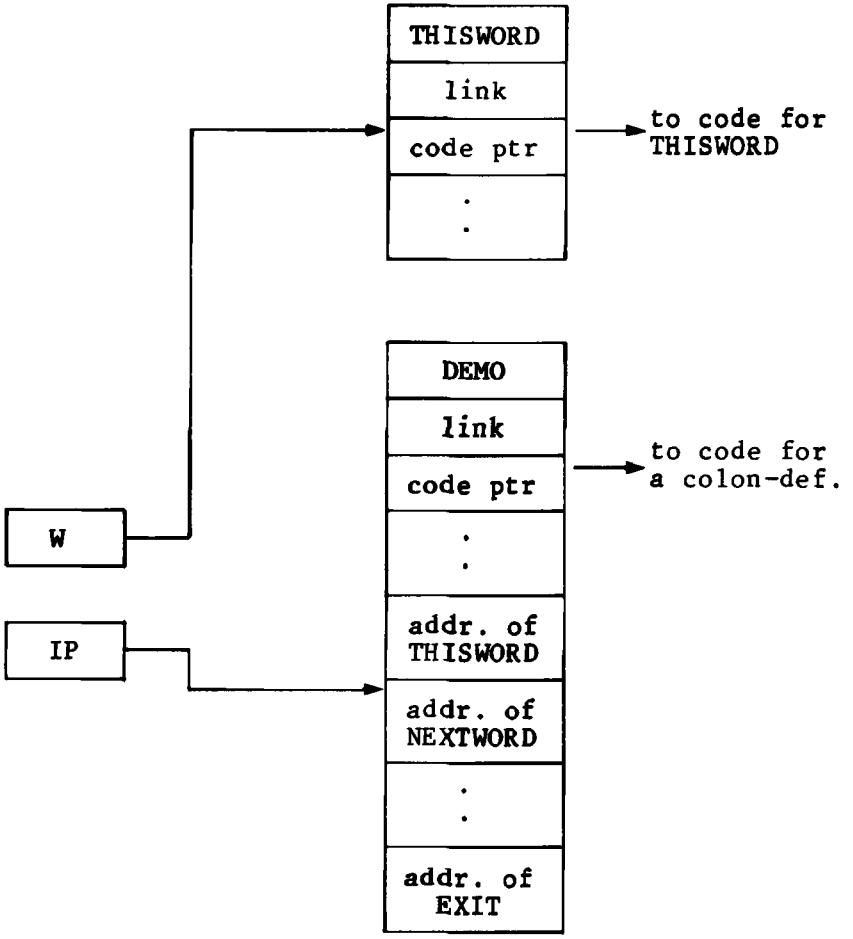
Some implementations produce a list of subroutine calls instead of a list of addresses and are therefore true compilers. This method uses subroutine threaded code. Executing a routine in such an implementation is simply a matter of executing the code of the subroutine calls and this tends to be somewhat faster than the execution of direct or indirect threaded code. The penalty is that each entry in the list occupies three bytes of memory instead of the two bytes needed for direct or indirect threaded code, so the system is usually much larger.

During the execution of direct or indirect threaded code the list of addresses has to be translated into the execution of the routines themselves and this is again a form of interpretation.

An implementation such as Acornsoft FORTH, using indirect threaded code, therefore needs two interpreters, the 'outer', or 'text' interpreter and the inner interpreter. The outer interpreter accepts text from the input stream and either executes or compiles it as appropriate. The 'compiled' text is left as a list of the addresses of other FORTH words. When such a list is to be executed each address must be interpreted, to cause the execution of the correct word, before moving to the next address in the list. Two pointers are used to assist this process. One, the interpretive pointer, holds the current position in the list of addresses being interpreted and the other (W)

is used to store the address of the code field of the word being executed.

Let us assume that a colon-definition with the name DEMO is being executed, as illustrated in the following diagram. Imagine that we are part of the way through the definition and that the word currently being executed is THISWORD .



At this time the code field pointer, W, contains the code field address of THISWORD . The interpretive pointer, IP, contains the address of (points to) the

execution address of NEXTWORD , which is the next routine to be executed. When the execution of THISWORD is completed control will be passed to the machine code of NEXT . This transfers the contents of the location pointed to by IP to W , which now contains the address of the code field of NEXTWORD . The value of IP is then incremented by two so that it again contains the address of the routine following the current one. The last instruction in NEXT is an indirect jump via W to the machine code of NEXTWORD . It is by this method that the routines of which DEMO is composed are executed in their correct sequence. The process continues until the word EXIT , which returns control to the word which called DEMO , is executed.

The two points yet to be described are the initial entry into the sequence and the final exit. These depend on the nature of the word being executed and are discussed in the next section.

3 The different types of dictionary entry

The various classes of word differ only in the contents of their code fields and parameter fields. The code field always contains a pointer to the start of an executable machine code routine and the different possibilities are given in the following list.

a) Machine code primitives

The parameter field of a machine code word contains the actual code to be executed and the code field contains the address of its start. The operation of NEXT causes a jump to the address contained in the code field, and the machine code is executed immediately. The code ends with a jump to NEXT which transfers execution to the next word in the sequence.

b) Constants

The value of the constant is contained in a two-byte parameter field. The code field contains the address of a piece of machine code (called 'constant' in the FORTH-79 Standard) which uses the address in W to locate the parameter field and copy its contents to the stack. Remember that W contains the address of the code

field of the word being executed and that the parameter field is just two bytes further on. The final instruction in 'constant' is a jump to NEXT .

c) Variables

The structure of a variable is similar to that of a constant except that the code field contains a pointer to machine code (called 'variable' in FORTH-79). This code uses the current contents of W to locate the parameter field and place its address on the stack. A jump to NEXT terminates the code of 'constant'. (See section 10.8 for the code used for variables and constants.)

d) User variables

User variables have only a single byte parameter area, containing the offset from the start of the user variable area to the two bytes containing the value. The code field contains a pointer to code which adds the offset to the contents of the user variable pointer UP and places the result on the stack, finally jumping to NEXT .

e) Colon-definitions

As we saw earlier, the parameter field of a colon-definition contains a list of the addresses of other FORTH words and section B.2 showed how execution steps from one address to the next. The code field of a colon-definition contains a pointer to the code ('colon' in FORTH-79) to start interpretation of this list. The colon-definition must have been called from some other word and at this point IP contains the address of a point in the calling word.

The first action of 'colon' is to copy the contents of IP to the return stack for later retrieval. Two is added to the contents of W to give the address of the start of the new word's parameter area and this address is stored in IP . A jump to NEXT then starts the interpretation of the new word. The last address in all colon-definitions is that of the word EXIT . This is a machine code routine whose action is to transfer the top value from the return stack into IP and then jump to NEXT . This value is the old contents of IP, placed

on the return stack by 'colon' and so the effect of EXIT is to restore execution back to the point from which the new word was called.

f) Words constructed using CREATE and DOES>

The parameter area of such a word may contain any combination of values and addresses, depending on the CREATE part of the creating word. On execution the address of the parameter area is left on the stack so that the contents can be used by the sequence of words following DOES> in the creating word. The code field of such a word contains a pointer to code (called 'does>' in FORTH-79) which

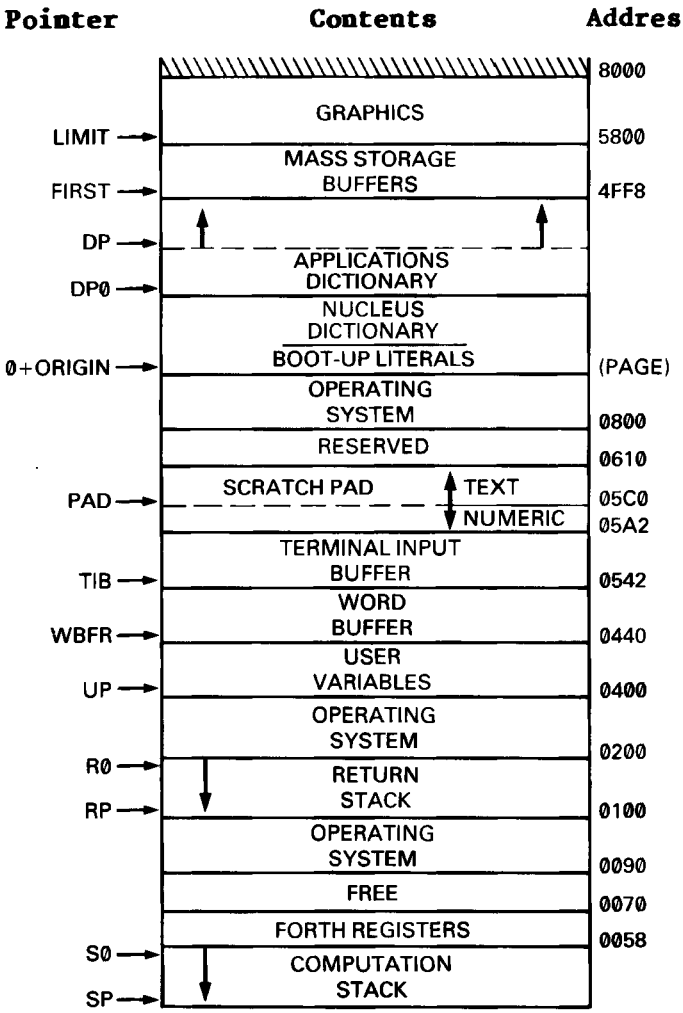
i) places the address of the start of the parameter area on the stack

ii) loads W with an address which is two bytes before the start of the words following DOES> in the creating word, and then jumps to 'colon'.

It will probably take a little time with paper and pencil to see that this really does perform the required action of words generated by the use of CREATE and DOES>.

Appendix C

Memory allocation



Appendix D

Two's-complement arithmetic

In unsigned arithmetic using 16-bit numbers, the lowest value that can be represented is zero, appearing as binary notation as

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ,

and the highest number appears as

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

which represents the decimal value 65535. There are therefore, including zero, 65536 different numbers.

To understand the operations on signed numbers, consider what happens if one is added to the highest unsigned value, 65535. In binary notation this sum appears as

```
    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
+   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
-----------------------------------
(1) 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-----------------------------------
```

In a computer, working to 16-bit accuracy, the one in the 17th place is lost and the value stored as the result will be zero. If we add one to a number and find the result is zero, it is natural to interpret the original number as having a value of -1.

Thus, for signed arithmetic, the number

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

can be used to represent -1.

In general the number -x is represented by the value which gives a zero result when +x is added to it (ignoring any overflow into the 17th place). The signed values -2 , -23 and -32768 are therefore represented by

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1

and

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

respectively.

All negative values are represented by binary numbers whose most significant (16th) bit is a one. Accordingly, the highest positive number that can be represented is:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

or +32767, and the most negative number is -32768, shown earlier.

The range for a signed number is thus from -32768 to +32767 which, including zero, gives a total of 65536 different values (as for unsigned numbers).

Whether a number is interpreted as a signed or an unsigned value is entirely a matter of context; the binary number

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

may represent either +65535 or -1 depending on the conversion routine used.

The above discussion has been confined to 16-bit numbers but similar considerations apply to any precision. In all cases the most significant bit of the number will be zero for a positive value and one for a negative value. It may, therefore, be regarded as a 'sign bit'.

In general the binary representation of a negative number may be found by writing down the binary

representation of the corresponding positive number, inverting all the bits and adding one. This is shown in the following example to find the two's-complement representation of -4 (in 8-bit precision):

0 0 0 0 0 1 0 0 (+4)

invert all bits (form the one's-complement):

1 1 1 1 1 0 1 1

add 1 (form the two's-complement):

1 1 1 1 1 1 0 0 (-4)

Appendix E

Further reading

1) The FORTH Interest Group in the USA supply many documents relating to FORTH, including assembly listings for many different microprocessors, a language model, reprints of 'BYTE' magazine articles and a bi-monthly magazine entitled 'FORTH Dimensions'.

For details of current costs for membership and their publications write (with sae please) to:

FORTH Interest Group
PO Box 1105
San Carlos
Ca 94070

2) The FORTH Interest Group UK is the British branch of the USA group. At present it meets on the first Thursday of every month at 7.00 pm at the Polytechnic of the South Bank in London. Membership includes a bi-monthly newsletter entitled 'FORTHWRITE'. Like its parent group, FIG UK exists to promote interest in and the use of the FORTH language and its members are prepared to help with any difficulties that may be encountered. For further details contact (sae please):

The Honorary Secretary
FIG UK
15, St Albans Mansions
Kensington Court Place
London W8 5QH

3) A number of articles on FORTH have appeared in 'BYTE' magazine:

August 1980, A FORTH language 'special'
February 1981, Stacking Strings in FORTH
March 1981, A Coding Sheet for FORTH
October & November 1981, P.S. - A FORTH-like language

April 1982, A Disk Operating System for FORTH
December 1982, Cosmic conquest - a game in FORTH

The August 1980 issue is now unobtainable, but reprints of the BYTE articles are available from the USA FORTH Interest Group (Ref. 1).

4) 'FORTH for Microcomputers' Dr Dobb's Journal No. 25 (May 1978). A brief review of the external and internal workings, with a variety of examples.

5) 'Starting FORTH' L Brodie.
published by Prentice-Hall (November 1981).

Available from most good bookshops; or

Computer Solutions Ltd
Treway House
Hanworth Lane
Chertsey
Tel: Chertsey (09328) 65292

The author is from FORTH Inc, the company started by Charles Moore, the inventor of FORTH. This is a very good introduction to the language, with lots of examples.

6) 'Threaded Interpretive Languages' R G Loeliger, published by Byte Books (McGraw-Hill) (1981)

A good, clear, description of the internal workings of FORTH-like languages, based on an implementation for the Z-80 microprocessor. Not for the complete beginner, but try it in a couple of months' time!

Index

FORTH

on the BBC Microcomputer

About this book

FORTH on the BBC Microcomputer serves as a general introduction to FORTH, and includes a full description of Acornsoft FORTH with a glossary defining the actions of all standard words.

Acornsoft FORTH is a complete implementation of the FORTH language to the 1979 Standard specification for the BBC Microcomputer Model B. In addition to a comprehensive set of arithmetic and stack operators, control transfer words and defining words, Acornsoft FORTH includes full graphics commands and the more advanced features for defining the actions of defining words themselves. This opens the door to Meta-FORTH, and user-defined FORTH-based languages.

The Acornsoft FORTH system includes the FORTH Assembler, Editor, and graphics demonstration.

About the author

Richard De Grandis-Harrison read Physics at Balliol College, Oxford. He became interested in FORTH during four years of research in Astronomy at the Royal Greenwich Observatory and Imperial College London. He now lectures in Mathematics, Physics and Computing at Hackney College, and is Chairman of the UK FORTH Interest Group.

Acornsoft Limited, 4a Market Hill, Cambridge, CB2 3NJ, England

ISBN: 0 907876 06 4

SBD03