FORTH Theory & Practice

Richard DeGrandis-Harrison

ACORNSOFT

SECOND EDITION

ISBN 0 907876 05 6

Published by:

Acornsoft Limited
4a Market Hill
Cambridge
CB2 3NJ
England

ATOM FORTH on cassette and System FORTH on disc are available from Acornsoft.

FORTH THEORY AND PRACTICE

CONTENTS

To Annette

Acknowledgments

This version of FORTH is based on the public domain publications provided through the courtesy of the

FORTH Interest Group,
P.O. Box 1105,
San Carlos,
CA. 94070

The greatest common divisor routine of Section 6.4 is by R.L. Smith. The random number routine of Section 8.2.2 is by J.E. Rickenbacker. These routines appeared in FORTH DIMENSIONS Vol.2, pages 167 and 34 respectively. The factorial calculations of Section 11.2.1 were provided by David Johnson-Davies of Acornsoft.

I am grateful for much helpful advice, particularly on the Acorn operating systems, given by David Johnson-Davies, and Joe Dunn of Acorn Computers.

I also owe a special debt of gratitude to Chris Jordan of Acornsoft who has had the unenviable task of ensuring that this manual makes sense. He has made many improvements to the text and has, I think, done an excellent job.

1 About this Manual

Although this manual is written to explain the operation of FORTH for the Acorn ATOM, most of the contents are applicable to any version of the FORTH language. The implementation, like most others for home computer use, is based on the model produced by the FORTH Interest Group.

Individual addresses of registers, subroutines etc. and the details of the memory map will, of course, 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 9) 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 a FORTH word 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 (e.g. 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 very great advantage on small systems of using very little memory for program storage. One further advantage, which will become more 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 can not 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 ATOM 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

ATOM FORTH is supplied in pre-compiled form on cassette. All that is needed to load the system is to place the tape in the cassette recorder and use the monitor command:

```
*RUN "FORTH"
```

Loading will take about five minutes, after which the system will respond with the sign-on message:

```
ATOM FORTH
OK
```

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

Once FORTH is loaded and the sign-on message 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 if you have typed the example in 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) which will give a list of all the words present in the dictionary. The listing may be stopped at any time by pressing the ESC 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 ESC key twice) execute the word by typing STARS (don't forget the carriage return) and the system will type an endless display of stars. You will obtain no response from any key except BREAK, since an endless loop is being executed.

Press the BREAK key to get the BASIC prompt '>' and then type:

```
LINK #2804
```

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. The warm start entry point to FORTH is at hexadecimal address 2804.

Repeat the sequence of executing STARS and pressing the BREAK key, but this time re-enter FORTH by typing:

```
LINK #2800
```

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 ? STARS MSG # 0
```

Error message 0 is given whenever a word is not recognised by FORTH. Restarting FORTH at hexadecimal 2800 does 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 and FORTH is entered in the following state:

Numeric conversion base:	DECIMAL
CURRENT vocabulary:	FORTH
CONTEXT vocabulary:	FORTH
Computation stack:	cleared
Return stack:	initialised

4 Stacks of Arithmetic

4.1 Stacks

Most high level languages use one or more stacks for their internal operations, e.g. 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, and (b) is the state after the value 15 has been 'pushed' onto the top of the stack. (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 appears like this:

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

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

In this manual all descriptions will use the conventional wording, 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 may arise.

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, with caution, by the programmer 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 ATOM 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 type '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 A).

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.> .

```
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 '\':

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
MINUS	(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
2+	(n1 ... n2)	Add 2 to the top stack item
2*	(n1 ... n2)	Fast multiply by two

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 number one (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 one 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 in the value of x (2) 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. These words are given in the following list:

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 two items.
ROT	(n1\n2\n3 ... n2\n3\n1)	Rotate the top 3 items, so that the third item moves to the top.

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

a) 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.

b) 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). These are:

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	
1 2	2 1 1	
1 2	1 2 1 1	
1 2 3	2 1 3	
1 2 3	3 2 1	

It is useful practice to work out the solutions for this list. If you are not sure your solution is correct, 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 could be done, and is broken into several sections so that the stack contents can be shown at each stage. Remember that the top stack item is on the right.

	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.

= 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.

The action of U< is to subtract the two numbers and examine the sign of the result. This should be used carefully, since if the difference between the two numbers is greater than 32767 the result will be interpreted as being negative and gives an apparently wrong result. The action of U< is:

Second stack value	Top stack value	Result of U<
0	32767	1
32767	32769	1
32769	62769	1
0	32769	0

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 Section 4.4.

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

```
: SMUDGE LATEST 20 TOGGLE ;
```

LATEST 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. This is discussed more fully in the description of CREATE in Chapter 5.

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 high order part 'above' the low-order part.

A double-precision number may be entered from the keyboard by including a decimal point anywhere in the number.

Thus typing in:

```
12.
583.2478
```

will place the numbers 12 or 5832478 on the stack, in double precision form. Note that the position of the decimal point has no effect on the way in which the number is stored. Thus

```
123456.
1234.56
123.456
.123456
```

will all be stored on the stack as the double-precision integer 123456. The number of digits to the right of the decimal point is, however, stored in the user variable DPL and may be used, for example, to control the format of numeric output. When a single-precision number is input from the keyboard the value in DPL is always set to -1.

There is one purely double-precision arithmetic word and this is:

D+	Double-precision add

In addition there are five mixed-precision operators:

M*	Multiply two signed single-precision numbers to give a signed double-precision product.
U*	As M* , but all numbers are unsigned. This is the multiplication primitive (machine code).

M/ Divide the double-precision number second on the stack by the single-precision number on the top. A single-precision quotient is left and all numbers are signed.

U/ As M/ , but the remainder is left beneath the quotient and all numbers are unsigned. This is the division primitive.

M/MOD As U/ , but leaving a double-precision quotient. Again all quantities are unsigned.

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

DMINUS 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.

Finally, there are two stack operators in double precision:

2DROP Remove the double-precision top stack item.

2DUP Duplicate the double-precision top stack item.

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

There are no relational or logical operations provided for double-precision numbers.

As an example of the use of some of the double-precision words consider the following definition of the word MD*. It 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>       ( modulus 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 to be extended to cope with any situation that may arise.

There are several ways in which new words may 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 are given in Appendix C. It will, however, be useful to give here a general description of the construction of a typical dictionary entry.

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

a) The name of the entry (variable length),

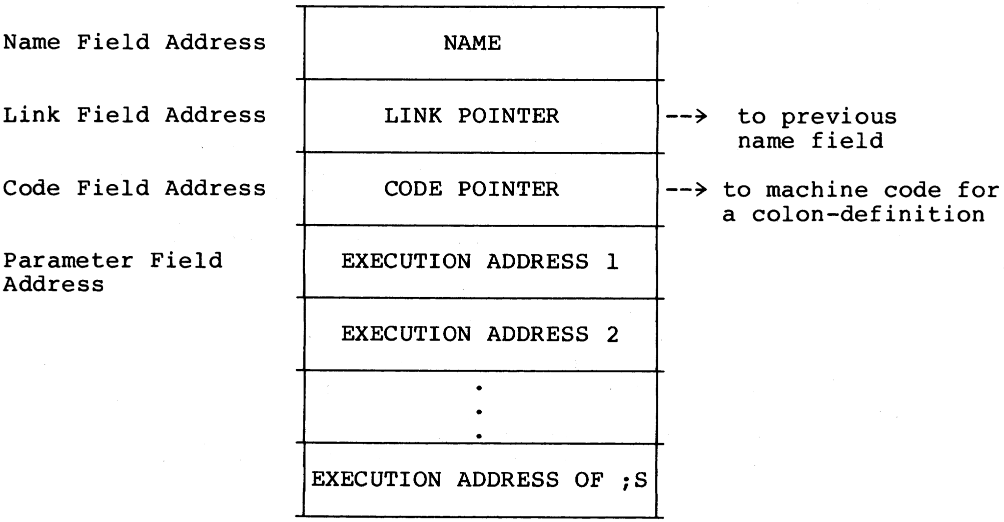
b) a link pointer to the name of the previous entry,

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 which defining word 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 <;S> which causes an exit from the word.

The diagram 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. The words are as follows:

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 return either the parameter field address or the name field address. If such a conversion is required it is, however, very simple 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. 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 first to 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 subequently 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, i.e. 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 <'> and <!> 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:

```
30 VARIABLE XLENGTH
```

This will create a variable with name XLENGTH and initial value 30. 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 <@> 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 !
```

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

The value of a variable can be incremented by the use of +! e.g.

```
1 XLENGTH +!
```

will increment the value of XLENGTH by one. The increment may be of any magnitude (subject to the valid range for single precision numbers) and may be positive or negative.

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

```
: ?   @ . ;
```

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 memory map).

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 97C4 hex with a maximum offset of 3A hex (58 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:

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.

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 tape buffer.

IN   Current offset into the input buffer.

OUT   No. of characters output (not used by system words).

SCR   Current tape/disc screen number.

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 in last number from keyboard.

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.

There are in addition two user variables that are used by the system but do not have name headers. They are:

R0   The address of the start of the return stack
S0   The address of the start of the computation stack

They may be given headers by:

```
8 USER R0 6 USER S0
```

5.4 CREATE

CREATE is a word that will produce a dictionary entry consisting of a name header, a link pointer, and a code field pointing to the start of a blank parameter field. It is used by <:> , CONSTANT , VARIABLE , and USER to generate their name headers.

The main use of CREATE in ATOM FORTH is to allow the definition of new machine-code routines, without the use of a FORTH assembler which would require loading from tape and take up valuable dictionary space. In such a use we are performing a hand compilation of a dictionary entry and make explicit use of the compiling words <,> and <C,>. The word <,> takes a two-byte value from the top of the stack and places it in the first two unused bytes of the dictionary (pointed to by the user variable DP). Usually, as in the following example, this will be in the parameter area of the dictionary entry which is being compiled. The value of DP is incremented by two. The word <C,> has a similar action except that it places only the low byte of the top stack item in the dictionary and increments DP by one.

As an example of the use of CREATE, consider the definition of 2DROP (which is provided in the nucleus dictionary):

```
HEX CREATE  2DROP  4C C, 291E , SMUDGE
```

where

HEX Sets the base to hexadecimal for the machine code.

CREATE 2DROP Generates the header for the word, with a code field pointing to the following code.

4C C, Compiles the single byte 4C into the parameter area.

291E , Compiles the two byte address 291E into the parameter area.

SMUDGE Toggles the 'smudge' bit in the name field to allow the new definition to be found in a dictionary search.

This word will execute the machine code

```
Machine Code:    Assembler Mnemonic:
  4C 1E 29           JMP POPTWO
```

(POPTWO is the label of machine code to erase the top two stack items and then execute NEXT, which moves on to the next word).

The dictionary entry produced by CREATE has the 'smudge' bit set to prevent the possibility of a partially completed definition from being found in a dictionary search, and it must be reset by the use of SMUDGE before the new definition can be either executed or used in a further definition. It will, however, appear in a VLIST in its smudged or unsmudged state. SMUDGE always acts on the last definition in the dictionary.

Note that an error in the use of a colon definition will leave a partially completed entry, in a smudged state, which can be found by VLIST but not used or forgotten. In this case using SMUDGE will then allow you to FORGET the partial definition and start again.

All machine code routines must terminate with a jump to existing machine code, which will, directly or indirectly, execute the code of NEXT. Valid terminating jumps are:

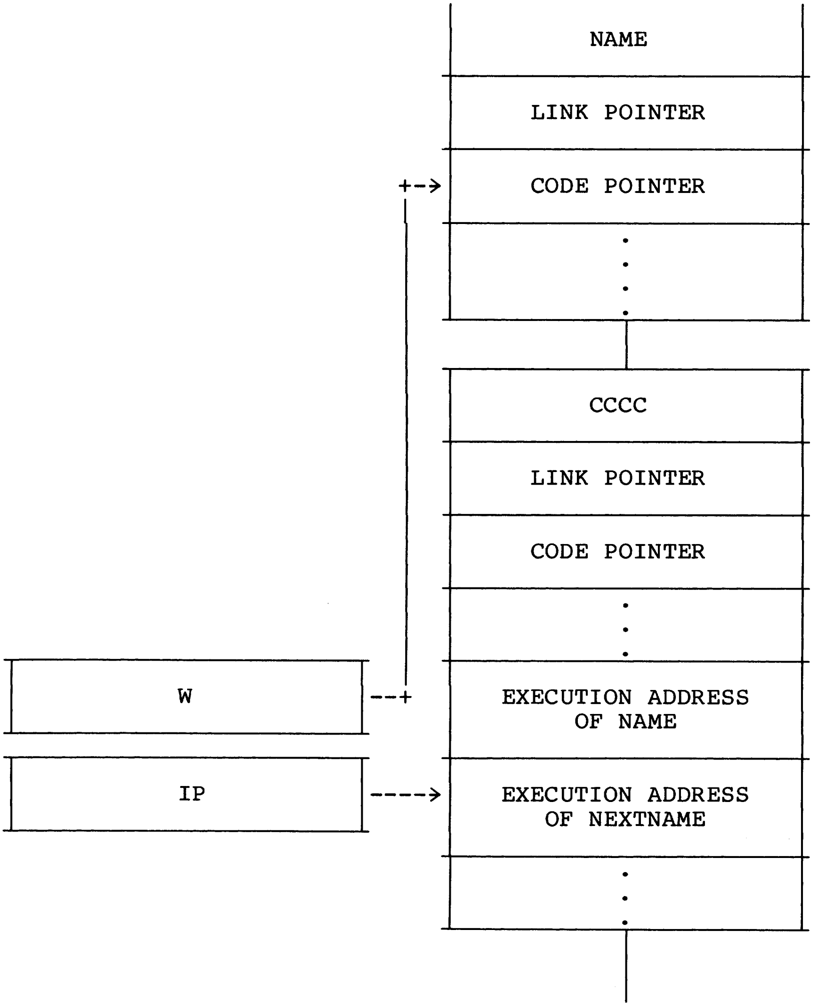
Routine	Hex Address	Description
NEXT	2842	Transfer execution to the next word in the sequence, or, if it is the last word, return to the keyboard.
PUSH	283B	Push the accumulator (as high byte) and one byte from the return stack as a new number on the computation stack and execute NEXT.
PUT	283D	Replace the current top stack item from the accumulator and return stack and execute NEXT.
PUSHOA	2B21	Push zero (high byte) and accumulator (low byte) to the computation stack and execute NEXT.
POP	2920	Drop the top stack item and execute NEXT
POPTWO	291E	Drop the top two stack items and execute NEXT.

The X-register of the 6502 is used by ATOM FORTH as the computation stack pointer. If a machine code routine uses the X-register its contents must be saved before it is used and restored before exit from the machine code by one of the above jumps. One byte of memory at address #8E in zero page is reserved as a temporary store for the X-register, and is known as XSAVE. This and other reserved zero page locations are given in the following table:

Name	Hex address	Format	Comments
W	88	xXx	1+2 bytes, codefield pointer
IP	8A	Xx	2 bytes, interpretive pointer
UP	8C	Xx	2 bytes, user area pointer
XSAVE	8E	X	1 byte, temporary for X-register
N	90	xXxxxxxxx	1+8 bytes, scratch pad

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

The significance of UP (see Section 5.3.3) and XSAVE have been discussed. To explain the functions of IP and W, consider that the word NAME in the definition of CCCC is being executed, as shown:



W is the code field pointer and at this time contains the address of ('points to') the code pointer of NAME . IP is the interpretive pointer and holds the address of the next instruction in CCCC . When the execution of NAME is complete the contents of the location pointed to by IP are transferred to W. This means that W will now contain the address of the code pointer of NEXTNAME . The value in IP is then incremented by two so that it again points to the instruction following the current one. An indirect jump to W will then start the execution of the code for NEXTNAME . This sequence of actions is performed by the machine code of NEXT .

The scratchpad area N is a nine-byte area into which up to four stack values can be transferred by use of the subroutine SETUP at address #2863. This expects to find the number of 16-bit values to be transferred in the accumulator. It uses the Y register and will return with zero in Y and the value in A doubled.

As an example of its use the sequence:

```
A9 03      LDA @3
20 63 28   JSR SETUP
```

will transfer the top three stack items into N. Each stack item occupies two bytes of N with its low-order byte at the lower address. The top stack item is first in the list. On return from the subroutine, Y will contain 0 and A will contain 6. The three items will have been dropped from the stack. The byte immediately preceeding N contains the number of bytes transferred (i.e. 6). This subroutine is useful to place stack values at a known, fixed location for use by machine code routines. Since this scratchpad area is used by many of the system words, its contents will change frequently, so the contents should only be used within the definition that placed them there.

The following examples are for words which exist in the nucleus dictionary. If you enter them at the keyboard you will find error message number 4 given, which is a warning that you are redefining an existing word. This will not, however, affect their operation. The examples are given in two forms, the first being how the routines are entered and the second is a conventional assembly language listing with explanations.

HEX (all the following are in HEX base)

```
CREATE AND (logical AND (section 4.4.4))
B5 C, 00 C, 35 C, 02 C,
48 C, B5 C, 01 C, 35 C,
03 C, E8 C, E8 C, 4C C,
283D , SMUDGE
```

```
B5 00     LDA 0,X        load low byte, top of stack
35 02     AND 2,X        AND with low byte 2nd on stack
48        PHA            save result on return stack
B5 01     LDA 1,X        load high byte, top  of stack
35 03     AND 3,X        AND with high byte 2nd on stack
E8        INX            drop the top
E8        INX            stack value
4C 3D 28 JMP  PUT        replace (old 2nd) by result (as new top),
                         then execute NEXT
```

Note that <,> stores the two bytes of 283D in reverse order. Provided this is remembered the amount of typing can be reduced, as in the next example, particularly as leading zeroes need not be typed.

```
CREATE +  ( single precision add )
18 C, B5 , ( 'High' byte is zero )
275 , 295 , 1B5 ,
375 , 395 , E8E8 ,
4C C, 2842 , SMUDGE
```

```
18        CLC            clear carry for addition
B5 00     LDA 0,X        load low byte top of stack
75 02     ADC 2,X        add low byte 2nd on stack
95 02     STA 2,X        store at low byte 2nd on stack
B5 01     LDA 1,X        load high byte top of stack
75 03     ADC 3,X        add high byte 2nd on stack
95 03     STA 3,X        store at high byte 2nd on stack
E8        INX            drop top of
```

```
E8       INX            stack
4C 42 28 JMP NEXT       execute NEXT
```

The next example uses the X-register which must therefore be saved and restored as described earlier.

```
CREATE RP@  ( leaves the contents of the return stack pointer )
8E86 , 8ABA , 8EA6 ,
48C , 1A9 ,
4C C, 283B , SMUDGE
```

```
86 8E    STX XSAVE      save X-register
BA       TSX            stack pointer (low byte) to X
8A       TXA            and then to accumulator
A6 8E    LDX XSAVE      restore X-register
48       PHA            push accumulator to (machine) stack
A9 01    LDA @1         high byte of stack pointer is 01
4C 3B 28 JMP PUSH       push stack pointer to computation stack
                        and then execute NEXT
```

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. When the new definition of QUOTE is made and used in:

```
: NEWPRINT  QUOTE  CR ;
```

NEWPRINT will type out the new message. PRINT will, however, 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 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. To forget TEST-VOC it is necessary to type:

```
FORTH  FORGET  TEST-VOC
```

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 though it is possible to chain vocabularies this can result in a complicated and confusing search sequence and is therefore not recommended.

One of the main uses of the VOCABULARY structure, in addition to separating the words of one application from those of another, 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 sure that FORTH is 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 R is used to replace a line of edited text. In the FORTH vocabulary the word R is used to copy the top item of the return stack to the computation stack.

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 defintion.

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 10.6.2). Their use in a colon-definition is:

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

[and] may also be used to include in a colon-definition a word which has no name header in the dictionary, provided that its execution address is known. For example, the word (ENTER) in ATOM FORTH is a headerless dictionary entry with execution address #3AF5. Its function is to interpret the contents of the tape buffer. It is given a header in the full tape interface by:

```
HEX
: ENTER [ 3AF5 , ] ;
```

5.6.4 Compiler Security

It is important to ensure that the sequence of words to be executed between [and], or any IMMEDIATE word, does not change the number of items on the computation stack. In the above example the number #3AF5 is added to the stack and then removed by <,>. Part of the compiler security system is to check that the number of items on the stack is unchanged across a colon-defintion. If the actions between [and] have the net effect of adding or removing stack items an error message will be given 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 ie. to compile it. WORD is not executed during the execution of NAME.

5.6.7 An Example

A non-trivial 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, and

, — 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, and
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 post-fix, 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
> THEN continue with the following sequence, in either case.

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. IF will then remove the flag, so =2? leaves the stack unchanged. 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 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 re-written 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 loop index is post-incremented, i.e. 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.

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 .

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 an error message, either message 7 (stack full) or message 1 (stack empty). 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, #2804, 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 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.

```
8000 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
```

or even 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  OK
```

Note that the loop still terminates when the incremented index equals or passes the loop limit.

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 leave the outer loop index and the second I leaves 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 units 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 carriage return 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 LIST 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 10.4.3). 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  LIST  LOOK-UP
```

where n is the value to be found.

The definition may be tested without the need to define LIST 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

```
2345 10240 2048 LOOK-UP
```

will search the first 4K of the dictionary for the value 2345 (and fail to find it).

```
11218 10240 2048 LOOK-UP
```

should find the value with an offset of 1057.

```
: 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            ( val )
    ELSE                 ( found )
       ROT ROT 2DROP 1 ( val and addr )
    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 last address in table )
    0 ROT ROT           ( put 0 under addr and last )
    DO
        OVER I @ =      ( table item = val? )
        IF              ( found )
            DROP        ( DROP the 0 )
            I           ( table address [assumed non-zero] )
            I MINUS     ( 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 4096 bytes of memory, i.e. 2048 comparisons, takes about 3 seconds using the first method and about 2.4 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  0  DO
      3  0  DO  CR  J  .
            10  SPACES  I  .
            LOOP  CR
         LOOP ;
```

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, the word J (and I) will only operate correctly if they are used at the same level of definition. The following sequence, for example, will not have the required effect.

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

: OUTER  CR 3 0 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 can only be terminated by pressing the BREAK 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. The definition:

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

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

If we define

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

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 as part of a longer calculation, where the value is not required to be printed, without needing to re-write 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 of the next chapter.

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, immediately above the top of the dictionary, by use of WORD . This expects to find a delimiter character on the stack and this is usually a space, ASCII code 32. Leading delimiter characters are ignored, and text up to the next delimiter is transferred to the region of memory starting at HERE . The first byte will contain the length of the text string, with two or more space characters added to the end.

The text string at HERE 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  HERE  COUNT  -TRAILING TYPE ;
```

will accept text from the keyboard and type it on the display.

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  HERE  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) i.e. " as its delimiter. Since <."> is a FORTH word it must be separated by a space from the text on which it operates. The closing " 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 typing of the input text.

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 that 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 HERE )
                    ( with space - ASCII 32 - as delimiter )
     HERE NUMBER    ( convert to double number )
     DPL @ 1+ 0=    ( was decimal point included? )
     IF DROP THEN   ( if not, make single number )
;
```

This routine leaves either a double- or single-precision number on the stack, depending on whether a decimal point did or did not appear in the input number. The number of digits to the right of the decimal point is stored in DPL . If no decimal point was present the value of DPL defaults to -1.

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 mis-typed input!

Valid characters are:

a) an optional minus sign as the first character

b) an optional decimal point at any position

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 store its execution address in the parameter field of (ABORT) (at address #347C). Setting the value of the user variable WARNING to -1 before the use of NUMIN would cause the error handler to use the new routine. Don't forget to set WARNING back to its original value of 0 afterwards, to restore the normal error messages.

An alternative solution is to use the word (NUMBER) which, in Atom FORTH, is a headerless routine with execution address #33B7.

There are two main differences between NUMBER and (NUMBER).

Firstly, (NUMBER) does not generate an error message on detecting a non-valid character, but simply leaves the address of the first unconvertable 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, (NUMBER) 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 HERE )
        0  0                            ( for building the number )
        HERE                            ( text start address )
        DUP  1+  C@  2D  =              ( test for minus sign )
        DUP  R>  +                      ( save result and skip sign )
                                        ( if present )
        -1                              ( default DPL value )
        BEGIN
            DPL !
            [ 33B7 , ]                  ( (NUMBER) )
            DUP C@                      ( unconvertable character )
            DUP BL = 0=                 ( leaves 1 if not a space )
            SWAP 2E =                   ( and a 1, if decimal point )
        WHILE                           ( decimal point )
            0=                          ( leave 0 to reset DPL )
        REPEAT
    WHILE                               ( invalid character )
        R>  2DROP  2DROP                ( clear the stacks )
        ." INVALID"
    REPEAT                              ( and try again )
    DROP                                ( address of space )
    R> IF DMINUS THEN                   ( apply sign to number )
    DPL @ 1+ 0=                         ( no decimal point? [DPL= -1] )
    IF DROP THEN                        ( if not, make single number )
;
```

The numeric conversion takes place in the inner loop, where DPL is initially set to -1. During the conversion by (NUMBER) , DPL is incremented for each converted digit, provided it is not equal to -1. There are three cases which cause an exit from (NUMBER):

i) finding a decimal point (1\1)

ii) finding a space (0\0)

iii) finding any other non-numeric character (1\0).

The bracketed values in each case represent the top two items on the stack after the two tests in the loop (note that the sequence <= 0=> in the inner loop could be replaced by <-> since any non-zero value is interpreted as true).

In case (i), WHILE finds a 1 on the stack so the loop is repeated, after 0= has changed the second 1 to 0, to reset DPL . Since DPL is no longer -1 it will now be incremented by (NUMBER) to give the number of converted digits since the last decimal point.

In case (ii), the first 0 causes the termination of the loop, leaving the second 0 to terminate the outer loop also.

In case (iii), the inner loop is again terminated, but the remaining 1 causes the outer loop to be repeated, generating an error message and returning for a new value to be typed.

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 NUMIN or to entering numbers via 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 region of memory just above the dictionary. 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 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 problem the word <CMOVE is also provided. Its overall action is the same as that of CMOVE execpt that the byte with the highest address is moved first and the transfer proceeds in order of decreasing address. 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 that are 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 i.e. the ASCII code for a space (or blank).

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-numeric or alphabetic characters.

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:

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

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

Finally, you may like to try executing the following, having previously defined BASE-36 as above.

```
505030.  38210.  676  1375732.
BASE-36  CR  D.  .  D.  D.  CR  DECIMAL
```

7.3 Output

7.3.1 Character Output

To output a single character, the word EMIT can be 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 "Atomic Theory and Practice", page 131) from within a definition, e.g.

```
: BELL  7  EMIT  ;
```

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

```
HERE  COUNT  TYPE
```

will display the string starting with its count byte at the address given by HERE. 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:

```
HERE  COUNT  -TRAILING  TYPE
```

Remember that the use of strings stored at HERE 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 two definitions provide left and right string extraction. They assume that the address of the count byte of the string is initially on the stack under the number of characters to be extracted.

```
: LEFT$  ( addr\nl ... addr2\n2 )
     SWAP  COUNT  ROT  MIN  ;

: RIGHT$
     SWAP  COUNT  ROT  2DUP  >
     IF  DUP  >R  -  +  R>  ELSE  DROP  THEN  ;
```

In both cases the stack is left in a state ready to TYPE the selected character string. If the number of characters exceeds the length of the string, the entire string will be displayed by TYPE .

As an example, type in the following:

```
: $IN     ( STRING input to HERE with ' as delimiter )
     CR  ."  $'"  QUERY  39  WORD ;

: STRINGS
     $IN  HERE  DUP
     CR  COUNT  TYPE  CR
     DUP  10  LEFT$  TYPE
         6  RIGHT$  TYPE  CR  ;
```

Then execute STRINGS as follows:

```
STRINGS  <RETURN>
$'THIS IS A LONG STRING' <RETURN>
THIS IS A LONG STRING
THIS IS A STRING
OK
```

Further discussion of strings is deferred to Chapter 10.

7.2.3 Numeric Output

The numeric output operators provided in FORTH are:

.	(n ...)	Display the signed number n followed by one space
.R	(nl\n2 ...)	Display the signed number nl 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 <.>.

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 a signed or an unsigned number 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.

7.2.4 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! One problem is that the Acorn ATOM does not have a pound sign in its character set. In this example the character # is used since the ASCII code for this symbol is often shown as a pound sign on a printer.

```
: .POUNDS              ( nd ... )
     DUP  ROT  ROT     ( KEEP HIGH PART, INCLUDING SIGN )
     DABS              ( MAKE POSITIVE )
     <#                ( START CONVERSION )
     # #               ( CONVERT 2 DIGITS - PENCE )
     46  HOLD          ( INSERT DECIMAL POINT )
     #S                ( CONVERT REMAINING DIGITS )
     SIGN              ( INSERT SIGN IF NEEDED )
     35 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
;

   1234.5  .REAL  1234.5  OK
   .12345  .REAL  0.12345 OK
   -12.345  .REAL  -12.345  OK
   123  0  5  DPL !  .REAL  0.00123  OK
```

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 Tape Interface and Editor

8.1 Introduction

When the nucleus dictionary of FORTH is loaded it will allow the use of the word LOAD to load an application from tape. The source code on tape is divided into 'screens'. Each screen is 512 (#200) bytes long and is divided into eight 'lines' of 64 characters. On a printer it will be displayed in this format, but on a VDU screen it will appear as sixteen lines of 32 characters.

When an application is loaded from tape each screen is first loaded into a buffer area, starting at address #95C0, and then interpreted in the same way as if it had been typed at the keyboard. Almost anything that can be typed at the keyboard can be used in a tape screen, but there are two major exceptions.

1. A loading screen will not prompt for input from the keyboard. This is sensible since it is impossible to predict how long it will take to complete the keyboard action and loss of synchronisation with the tape will almost certainly result. A running tape, like time and tide, waits for no man.
2. A screen cannot be used as a 'load screen' to load a number of other screens. Again, this is not a disadvantage in a primarily tape-based system since the order of loading screens from tape cannot be changed. The loading of a series of consecutively numbered screens is done by the use of the word --> (see Section 8.2.2).

If either of these two is attempted the system will 'go away', and a BREAK followed by a COLD or WARM start will be necessary.

In order to view the contents of a screen, or to write or modify screens, it is first necessary to load the full tape interface and screen editor. These are provided as an application on the ATOM FORTH tape, in screens 6 to 14 inclusive. Before continuing with this chapter, load these screens. Put the tape in the cassette recorder and type:

```
6 LOAD
```

when the machine will respond with:

```
>6 PLAY TAPE
```

Start the tape and press the space bar. After a few seconds you will notice some slight interference on the screen, indicating that screen 6 is being loaded. After a short pause the message

```
>7
```

will appear, indicating that screen 6 has been loaded successfully and a search is now being made for screen 7. There is no need to press any keys; the loading will take place automatically until all the required screens have been loaded when the response

```
OK
```

will be typed.

During the loading of screen 11 the messages

```
R MSG #4   I MSG #4
```

will appear, but don't worry. All that this means is that the words R and I are redefined in this screen (as part of the EDITOR vocabulary).

8.1 The Tape Interface

When the OK prompt has appeared, at the end of the loading of screen 14, the full tape interface and screen editor are loaded and ready for use. When this happens type:

```
14 LIST
```

This will list on the display the contents of the last screen to be loaded (screen 14) which contains definitions of the words TILL and C. Now type:

```
6 LIST
```

This time the response will be:

```
>6 PLAY TAPE
```

Rewind the tape, start it playing and press the space bar. After a short pause the contents of screen 6 will be displayed, containing the definitions of .LINE , LIST itself, and ENTER . As soon as the listing has started the tape recorder can be switched off. Typing

```
6 LIST
```

again will give an immediate listing of screen 6. The word LIST expects to find a screen number on the stack. If this is the number of the screen present in the tape buffer area it will be listed immediately. Otherwise the loading prompt will be given and the screen loaded from tape before being listed. Now type:

```
14 LIST
```

and start the tape in the usual way. As the tape is searched for screen 14, the number of each screen will be displayed as it is encountered. When screen 14 is found it will be loaded into the tape buffer and listed as before. The search may be stopped at any time during the period before the specified screen is found by pressing the CTRL key. This will cause a re-entry of FORTH via a WARM start. The only point to remember is that the current screen number will have been set to the value used with LIST (eg. 14 in the above case) so that typing 14 LIST again will list the current contents of the tape buffer as screen 14. This is not really a disadvantage since the main reason for terminating a search is if it is realised that the wrong screen number is being used.

This method of terminating a tape search can also be used with LOAD . In both cases the process cannot be terminated by use of the CTRL key once the process of loading the screen into the tape buffer has started.

If you tried stopping the search by the above method, change the stored screen number by typing:

```
13 SCR !
```

and then type 14 LIST again. When you have a listing of screen 14 type:

```
: TASK ;
```

(to mark this point in the FORTH vocabulary) and then:

```
ENTER
```

This is the word which causes the contents of the tape buffer to be interpreted, as though it had been typed at the keyboard. In this case it will cause an immediate error and the message:

```
#LEAD ? MSG  0
```

will appear. This has occurred because the system is in the FORTH vocabulary, but the word #LEAD is in the EDITOR vocabulary and therefore has not been found. It does allow the demonstration of a very useful feature of the system. Immediately after the error has been notified, type:

```
WHERE
```

This will give the screen and line number, and a listing of the offending line, where the FORTH system thinks the error has occurred.

We know about this error so now we can simply type:

```
FORTH FORGET TASK
```

to clear the dictionary and continue (WHERE enters the EDITOR vocabulary, so FORTH must be used to return).

Remove the applications tape from the cassette and try using the word:

```
SAVE
```

This is used to save the current contents of the tape buffer to tape. No screen number is needed; the value of the user variable SCR is assumed. The system will respond with:

```
RECORD TAPE
```

Do not switch the tape to record on this occasion (since there is no tape in the recorder). Normally you would start the tape recording and then press the space bar. Just press the space bar this time, and wait. After a short pause the reminder

```
STOP TAPE
```

will appear and the system will wait until you press the space bar again to indicate that you have done so. This reminder can save many feet of blank tape! Now type 14 LIST again, to make sure that the contents of the screen are still there, and then try typing:

```
CR 0 .LINE
```

which should type the contents of line 0 of the tape buffer. Now try:

```
EMPTY-BUFFERS 14 LIST
```

when you should find that an empty screen 14 is listed. The word EMPTY-BUFFERS clears the contents of the tape buffer.

After this brief survey of the facilities of the tape interface we are ready to try the screen editor.

8.2 The Screen Editor

8.2.1 Introduction

When a BASIC program has been written and is working satisfactorily it may be saved as a named file. This is possible because a BASIC program is stored in source form - very similar, if not identical, to the way it is typed at the keyboard. Since FORTH is a compiled language it is not stored in source form , but as a list of addresses of routines. It is, therefore, to be expected that the writing of a FORTH application which is to be saved on tape will be somewhat different from the corresponding method for BASIC.

In FORTH the source text is edited into a screen, whose contents can be compiled, tested and, if necessary, modified until its action is correct. The screen can then be saved and any further screens written in the same manner.

It is a good idea to develop all but the very simplest of applications by use of the editor since it allows the modification of a definition without the need for excessive retyping. The editing functions are placed in a separate EDITOR vocabulary. When the tape interface and editor screens are loaded, note the difference between the commands:

```
FORTH VLIST
```

and

```
EDITOR VLIST
```

8.2.2 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 screen must be set up and the EDITOR vocabulary declared. To do this the following word sequence can be used:

```
DECIMAL
EMPTY-BUFFERS
300 SCR  !      ( we shall use screen 300 )
SCR @ LIST      ( list this screen )
EDITOR
```

To simplify the process the word PROGRAM has been included in the tape/editor application. This asks for the starting screen number and automatically sets up the system for editing a screen. This example saves a definition of RND , which generates a random number, and tests its action. It is shown exactly as it would appear on the display.

```
PROGRAM
FIRST SCREEN NUMBER ?  300
SCR # 300
0
1
2
3
4
5
6
7
OK
0 P ( RANDOM NUMBER GENERATOR )  OK
1 P DECIMAL  OK
2 P 0  VARIABLE SEED OK
3 P : (RND)  ( ...RAND )  OK
4 P      SEED @ 259 * 3 +  OK
5 P      32767 AND DUP SEED !  ;  OK
6 P : RND  ( RANGE...RANDOM )  OK
7 P      (RND)  32767  */  ;  OK
L
SCR # 300
0 (RANDOM NUMBER GENERATOR)
1 DECIMAL
2 0 VARIABLE SEED
3 : (RND) ( ...RAND )
4       SEED @ 259 * 3 +
5       32767 AND DUP SEED ! ;
6 : RND ( RANGE...RANDOM )
7       (RND) 32767 */ ;
OK
ENTER OK
SEED ? 0 OK
. 0 . ? MSG # 1
(RND) . 3 OK
. 0 . ? MSG # 1
10 RND .  0 OK
10 RND .  1 OK
10 RND .  8 OK
. 0 . ? MSG # 1
1 2 3 OK
: TEST CR 0 DO DUP RND .  LOOP DROP ; OK
20 10 TEST
6  17  1  17  11  6  18  14  19  7  OK
. . . 3 2 1 OK
SAVE
RECORD TAPE
```

(At this point, place a tape in the recorder, set it recording, and press the space bar)

```
STOP TAPE
OK
```

The random number generator is saved as screen 300 and may be entered into the dictionary at a later time by:

```
300 LOAD
```

Screens may be numbered from 0 to 999 inclusive. Any attempt to save a screen with a screen number outside this range will cause an error message (6) to be given.

The EDITOR word P will put the following text onto a line of the screen. It expects to find the line number on the stack and is used as:

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

Since P is a FORTH word it must be separated from the text by a space. It is good practice to leave a space after P, even if you then just press RETURN (to clear a line of text) - otherwise an ASCII null will be placed in the line and will stop interpretation of the screen at this 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 other 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 must, of course, be separated by one or more spaces from any following words.

A FORTH application can be rather difficult to follow, particularly if it makes great use of the stack so that variable names do not appear. For this reason it is a good idea to use plenty of comments when you write a screen. Since they are ignored when the screen is interpreted they cost nothing in terms of dictionary space or execution time. The only cost is in the use of cassette tape and loading time.

The applications provided on tape do not follow this rule and are not commented. This is to reduce the loading time, since they are likely to be used frequently. They are, in any case, fully documented in this manual.

The word L is an EDITOR command to list the current screen and saves typing

```
SCR @ LIST
```

every time.

When the screen is completed it should be fully tested before being saved. In the example the words SEED (RND) and RND are tested in turn. The word <.> is used frequently to check that nothing is left on the stack.

As a final check the word TEST was defined and used to generate ten random numbers in the range 0 to 19. The numbers 1, 2 and 3 were placed on the stack before TEST was used and printed again at the end. This is a simple way to check that the words do not affect values lower down on the stack.

When it is apparent that the words are operating correctly the screen may be saved.

If an application extends to further screens the word --> should be put at the end of the screen. This is an instruction to continue interpretation with the next screen on the tape (if the current screen is screen n the next screen must be numbered n+1). The word --> is IMMEDIATE and so will execute even if the text is being compiled. This means that it is possible for a single definition to extend over a screen boundary. It is preferable not to do this as definitions should, if possible, be kept short. It is, however, sometimes necessary to write a definition (e.g. a machine code primitive) that cannot be fitted into a single screen, particularly as the screens are relatively short. A standard FORTH screen is sixteen lines of 64 characters, i.e. 1024 (1k) bytes. On the Acorn ATOM the screens are half this size to enable the whole screen to be shown on the display at once.

If --> is not used at the end of a screen, interpretation will stop at the end of the screen or at the word <;S> . This word terminates the interpretation and may, therefore, be followed by further comments or instructions, which do not need to be enclosed in parentheses.

In the writing of an application extending to several screens the word MORE can be used to save the current screen and set up the next blank screen (it does not insert --> at the end of the screen).

8.2.3 The Line Editor Commands

So far the only word from the line editor we have used 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 7.2.4).

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 (Does not work for line 7).
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 7 is lost.
R	Replace the contents of the line with the text from PAD.
S	Spread the text by inserting a blank line. Line 7 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 7.

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 vocabulary )
: INVERT
     8 0 DO
         7 H          ( LINE 7 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.

8.2.4 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. It is written, for speed, as a machine code primitive. Like most machine code in FORTH, it is relocatable and so may be copied from the tape/editor screen into your own application if you require a routine to search an area of memory for a particular character string. Its action on the stack is, however, a bit complicated.
    The stack action is:

```
MATCH ( addr1\length\addr2\count ... f\offset )
      using 4 and leaving 2 stack values.
```

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 (addr1 + offset - count gives the address of the start of the matched string).

8.3 A Note on Screen Numbering

The screen numbers, which may be in the range 0 to 999 inclusive, are handled by the system in DECIMAL base, regardless of the current numeric conversion base. If you are working, for example, in base 16 and you type in:

```
30 LOAD
```

the response will be:

```
>48 PLAY TAPE
```

since hexadecimal 30 is decimal 48. If you actually wanted screen 48 (decimal) then proceed as normal. Otherwise the load can be aborted by pressing the space bar and then the CTRL key.

The sign-on message will appear and you can try again. Note that the abort will automatically reset the numeric conversion base to DECIMAL, so

```
30 LOAD
```

should now work as expected. You could have avoided this by typing

```
DECIMAL 30 LOAD
```

8.4 Using Discs

Although the FORTH system for the Acorn ATOM is designed for use with the cassette operating system, it is fully compatible with floppy disc drives using the Acorn DOS. The only change that you may like to make is to remove the STOP TAPE prompt from the SAVE routine (tape interface screen 7), since writing to the disc is controlled by the system.

This can be done by loading the tape interface and editor, listing screen 7 and using the EDITOR commands to delete the sequence

```
." STOP TAPE" KEY DROP CR
```

from line 7.

Screen 7 may then be re-saved, replacing the old version. A COLD start, followed by a re-load of the tape interface and editor, will allow the new version to be used.

8.5 Econet

ATOM FORTH is fully compatible with the Acorn Econet networking system. This system is, however, designed mainly for use with discs under the Acorn DOS. To load files or screens from tape it is therefore necessary to activate the cassette operating system. This can be done by using the *COS command before loading FORTH, or from within FORTH by use of the MONITOR routine of Section 11.4. Remember that in this case the '*' is not needed and the correct command is

```
MONITOR COS
```

9 Graphics

9.1 Introduction

You will probably have noticed that, when FORTH is executing, there is a greater or lesser amount of interference on the display. This is an unavoidable consequence of the use of the top 3K (#8C00 to #97FF) of the graphics memory for the user's dictionary and tape buffer. It does, however, have one advantage in that it is possible to see when the tape screens are loading and compiling! Because of this use, only the lower half of the upper memory area is available for graphics and the maximum resolution available is 128x192.

If you do not need to use the higher resolution graphics, the whole of the upper memory (with the exception of the VDU memory) can be used as extra dictionary space. All that is necessary is to change the value of the dictionary pointer before any applications are loaded. When you have loaded the nucleus dictionary from tape, enter:

```
      HEX 8200 DP !
and   8200 FENCE !
```

New definitions will then start loading from #8200 upwards, giving an extra 2.5K of dictionary space. The lowest graphics mode, with a resolution of 64x48, is still available.

It is also possible to compromise, e.g. setting the initial value of the dictionary pointer and FENCE to #8600 will give an extra 1.5K of dictionary space and allow graphics up to a resolution of 128x96. Note that a COLD start will reset the upper dictionary space to start at #8C00.

The graphics package, provided in screens 18 to 21 of the applications, assumes that the upper dictionary area is set to start at #8C00 and therefore requires no alterations to the system. The graphics package can be loaded, with or without the presence of the EDITOR vocabulary, by typing:

```
18 LOAD
```

in the usual way.

9.2 Graphics Modes

Four modes of graphics are provided, similar to the ATOM BASIC graphics modes 0, 1, 2 and 3. The resolutions and the memory used are given below.

Mode:	Resolution X:	Y:	Memory: #8000 to:-
0	64	48	#81FF (0.5K)
1	128	64	#83FF (1K)
2	128	96	#85FF (1.5K)
3	128	192	#8BFF (3K)

The graphics modes are set by typing:

```
n CLEAR
```

(after having declared the GRAPHICS vocabulary) where n may be 0, 1, 2 or 3. The word CLEAR expects to find this number on the stack and it may be either typed in, as in the above example, or left on the stack as the result of a calculation performed by some other word.

9.3 Point-Plotting

The five words connected with point-plotting are:

Word	Description
BLACK	All subsequent plots are in black.
WHITE	All subsequent plots are in white.
INVERT	All subsequent plots invert the point.
PLOT (X\Y ...)	Plot a point at the position with co-ordinates (X,Y). The co-ordinates of the point are kept in the GRAPHICS variables X and Y.
(PLOT) (X\Y ...)	As PLOT but X and Y are not updated.

One of the words BLACK, WHITE or INVERT must be executed before any use of PLOT or (PLOT). The 'colour' will remain in effect for all subsequent uses.

The following example illustrates the use of these procedures.

```
GRAPHICS DEFINITIONS (definitions to be in the GRAPHICS vocabulary)
: DIAGONAL
      0 DO I I PLOT LOOP ;
```

Then execute

```
0 CLEAR 48 WHITE DIAGONAL
```

```
3 CLEAR 192 DIAGONAL
```

and

```
12 EMIT (to return to text mode)
```

The next example shows the possibilities for animated graphics.

```
: FLASH
    3 CLEAR INVERT
    BEGIN
       192 DIAGONAL  ?ESC
    UNTIL  12 EMIT
;
```

Pressing the ESC key will end the display and clear the screen. The use of animated graphics is restricted by the interference appearing on the screen. This can only be eliminated by adding extra memory from #3C00 upwards (minimum of 2K) so that the dictionary, user variables and tape buffer does not need to use the graphics memory (see Section 9.7 and Appendix D).

9.4 Line-Drawing

A line may be drawn from the last plotted position to the point with coordinates (x,y) by use of the LINE (x\y ...). The points on the line may be plotted, erased or inverted by previous use of WHITE , BLACK or INVERT as in the case of PLOT .

To draw a line from the point (0,0) to the point (15,25), for example, the following sequence can be used.

```
0 X ! 0 Y ! WHITE 15 25 LINE
```

A triangle with vertices at (X0,Y0), (X1,Y1), (X2,Y2) may be drawn by the definition:

```
GRAPHICS DEFINITIONS
: TRIANGLE (X0\Y0\X1\Y1\X2\Y2 ...)
    DUP >R Y !  ( SET X & Y TO X2 & Y2 )
    DUP >R X !  ( AND ALSO SAVE THEM FOR LATER )
    LINE LINE   ( DRAW 2 SIDES )
    R> R> LINE  ( RECOVER COORDINATES & DRAW THIRD SIDE )
;
```

It leaves the coordinates of the final point i.e. X2 and Y2 in the graphics variables X and Y, and may be used, for example, as

```
3 CLEAR 10 15 73 20 100 180 WHITE TRIANGLE
```

The variables XDIR , YDIR , DELTAX , DELTAY , ERR and the words SETXY and (LINE) are used internally by LINE and are not intended to be executed directly by the user.

The word MOVE (X\Y ...) will change the current plot position to the point with coordinates (X,Y) without any plotting action. We could, for example, redefine TRIANGLE as:

```
: TRIANGLE
    2DUP >R >R
    MOVE        ( TO X2,Y2 )
    LINE LINE
    R> R> LINE
;
```

which has the same action as the earlier definition.

9.5 Relative Plotting

The words RPLOT , RLINE and RMOVE have similar actions to PLOT , LINE and MOVE , except that the x and y values on the stack are interpreted as being relative to the last point plotted. The word REL is used to convert the relative coordinates to absolute values.

9.6 Clearing the Screen

The screen may be cleared and returned to text mode by:

```
: CLS   12 EMIT ;
```

9.7 Use of Mode 4 Graphics

Very few changes need to be made to the Graphics Package to enable the use of mode 4 graphics with the applications dictionary moved to extension memory below #8000. They will all fit into the existing screens 18 - 21, with a little editing.

Screen 18:

Insert the following as line 6:

```
: 4MODE F7AA F0 00 1800 ;
```

Modify the new line 7 to read:

```
CASE: NMODE 0MODE 1MODE 2MODE 3MODE 4MODE ; -->
```

Screen 19:

On line 1, replace the words:

```
FFFC AND
```

with:

```
ABS 4 >
```

No further changes are necessary.

10 What ‹BUILDS and DOES› Does

10.1 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, e.g. 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 which is 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, e.g. 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 <BUILDS and DOES> are used for this purpose. The < and > signs (which are not pronounced!!) are present to indicate that the two words should be used together and to show their order of use.

10.2 The Actions of <BUILDS and DOES>

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

```
: FAMILY <BUILDS ..... 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 <BUILDS and DOES> , however, make it a level 2 definition.

The words following <BUILDS 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 though 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 <BUILDS 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 <BUILDS executes it generates the dictionary entry header for MEMBER , including its name, the link pointer to the previous dictionary entry and a code field pointer as normal. It also reserves an extra two-byte space immediately following the code field pointer. The words, if any, after <BUILDS then execute and will usually act to place values, or reserve space, in the parameter area of MEMBER .

The execution of DOES> takes the address of the first word after DOES> in the dictionary entry of FAMILY and places it in the two-byte space reserved by <BUILDS , creating a pointer to the list of words that will be executed when MEMBER is used. It also re-writes the contents of MEMBER's code-field address to point to a routine that will handle this execution.

When MEMBER is executed (level 0) the address of its parameter area is placed on the stack and then the words following DOES> , in FAMILY , are executed. Execution of all words defined by FAMILY begins with this code, so FAMILY produces a group of words with related actions. <BUILDS and the words following it 'customise' each new word by compiling items unique to it (e.g. values from the stack, 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.

10.3 The use of <BUILDS and DOES>

Some simple examples may make the use of these words more clear.

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. The following definition of VARIABLE is identical except that the values of the variables it creates are not initialised.

```
: VARIABLE <BUILDS 2 ALLOT DOES> ;
```

When this is executed by typing:

```
VARIABLE SIZE
```

<BUILDS 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 ! and @ as normal.

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

```
: VARIABLE <BUILDS , DOES> ;
```

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

The definition of CONSTANT is:

```
: CONSTANT <BUILDS , DOES> @ ;
```

The compilation stage is identical to that of VARIABLE , 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 <BUILDS C, DOES> ;
```

and

```
: CCONSTANT <BUILDS C, DOES> C@ ;
```

These both expect a value in the range 0 to 255 on the stack to initialise the value of the dictionary entry they create.

10.4 Arrays and Tables

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

10.4.1 One-dimensional Arrays

A simple definition for a one-dimensional array is:

```
: ARRAY
      <BUILDS  2* ALLOT
      DOES>    SWAP 2* +
;
```

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

```
10 ARRAY VALUES
```

The words after <BUILDS reserve two bytes for each element. When VALUES is executed, the index is taken from the stack and 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
```

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
    <BUILDS DUP 1 - ,   ( STORE MAXIMUM INDEX )
            2* ALLOT    ( RESERVE SPACE )
    DOES>   2DUP        ( DUPLICATE INDEX AND PARAMETER ADDRESS )
            @ OVER <    ( CHECK IF INDEX TOO LARGE )
            SWAP 0< OR ( OR IF NEGATIVE )
            5 ?ERROR   ( ISSUE ERROR MESSAGE IF NEEDED )
            2+         ( OTHERWISE STEP OVER MAXIMUM 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.

10.4.2 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 could be added in a similar manner to that given for one-dimensional arrays.

```
: 2ARRAY
     <BUILDS 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 )
;
```

The use is

```
10 5 2ARRAY RECTANGLE
```

to create a 10 by 5 array called RECTANGLE. In this example the array indices may range from 0,0 to 9,4 inclusive. The address of, say, the 2,3 element is left on the stack by:

```
2 3 RECTANGLE
```

10.4.3 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
      <BUILDS 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 LIST 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
     <BUILDS DUP ,                 ( store no. of items )
             2* ALLOT              ( reserve space )
     DOES>   DUP 2+                ( get start address )
             SWAP @                ( max number of items )
;
```

LIST is then created by

```
n TABLE LIST
```

where n is the required maximum number of items.

10.5 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 to handle strings up to 255 characters in length.

```
: STRING              ( max length ... )
        <BUILDS
              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.

The definition of $IN 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 8.2.3) and for numeric conversion (Section 7.2.4).

```
: $IN                               ( ... addr\length )
    HERE C/L 1+ BLANKS              ( CLEAR MEMORY AT HERE )
    1 WORD                          ( INPUT STRING TO HERE )
                                    ( TERMINATED BY CARRIAGE RETURN )
    HERE 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
```

10.6 A CASE Statement

10.6.1 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.

10.6.2 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 package to select the resolution mode, and has an added check to restrict the choice to the integers from 0 to 3 inclusive.

Here is the definition of the defining word CASE: :

```
: CASE:
      <BUILDS 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 a variety of possible forms for case statements in FORTH see 'FORTH Dimensions' Vol.2 No.3 (1980) (see Appendix E).

11 Further Examples

11.1 Fast Divide-by-Two

Division reqires a large number of operations and is usually very slow in a microcomputer. The following routine is a machine code primitive which will divide by two. Its action is identical with

```
2 /
```

but is about sixty times faster.

```
HEX

CREATE 2/
    18  C,  1B5  ,  910  ,  F6   ,
    2D0  ,  1F6  ,  1F0  ,  38  C,
    176  ,   76  ,  4C  C,  2842 ,
SMUDGE
```

Assembly Listing

```
18                    CLC
B5 01                 LDA 1,X
10 09                 BPL 2D1V
F6 00                 INC 0,X
D0 02                 BNE NOINC
F6 01                 INC 1,X
F0 01        NOINC    BEQ 2DIV
38                    SEC
76 01        2DIV     ROR 1,X
76 00                 ROR 0,X
4C 42 28              JMP NEXT
```

11.2 Recursion

A recursive routine is one which uses itself. In FORTH a dictionary entry cannot find a reference to itself while it is being defined. This problem is solved by defining the IMMEDIATE word MYSELF.

```
: MYSELF
     LATEST           ( get name field address of latest word )
     PFA              ( convert to parameter field address )
     CFA              ( convert this to code field address )
      ,               ( compile the address )
;    IMMEDIATE
```

11.2.1 Factorials

The following example shows its use to form a recursive definition to calculate factorials.

```
: (FACT) ( n1\n2 ... n3 )
      -DUP IF DUP ROT * SWAP
              1 - MYSELF
           THEN
;

: FACT ( n ... )
      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. Attempting to calculate factorials of numbers greater than 7 will cause an arithmetic overflow. Factorials of negative numbers are not defined. Error checking is confined to FACT. This then uses (FACT) to calculate the result, which is displayed.

The following, alternative, definition of FACT makes use of MD*, defined in Section 4.4. The result is left as a double precision number, allowing the calculation of factorials of numbers up to and including 12.

```
: (FACT)
      -DUP IF DUP 1 -
              >R MD* R>
              MYSELF
           THEN
;

: FACT ( n ... )
       DUP 0< OVER 12 > OR 5 ?ERROR
       1 0 ROT (FACT) D.
;
```

11.2.2 Sorting an Array

The word QUICKSORT (n1\n2 ...) uses the quicksort algorithm to sort elements n1 to n2 inclusive of the array NUMBERS into increasing numerical order.

Before typing in the following definitions you must first define

```
2/       ( Section 11.1 )
ARRAY    ( Section 10.4.1 )
MYSELF   ( Section 11.2 )

0 VARIABLE TEMP                  ( TEMPORARY STORAGE )
256 ARRAY NUMBERS                ( OR WHATEVER SIZE YOU WISH )

: EXCHANGE ( n1\n2...)           ( EXCHANGE ELEMENTS n1 & n2)
                                 ( OF NUMBERS )
  NUMBERS DUP >R @ SWAP
  NUMBERS DUP @ >R
  ! R> R> !
;

: PARTITION ( n1\n2 ... n3\n4 )  ( SPLIT ARRAY INTO SMALLER SECTIONS,
                                 EXCHANGING ELEMENTS WHERE NECESSARY )
  BEGIN 2DUP > 0= WHILE
  SWAP BEGIN DUP NUMBERS @ TEMP @ < WHILE 1+ REPEAT
  2DUP EXCHANGE SWAP 1+ SWAP 1 -
  REPEAT
```

```
;

: QUICKSORT ( n1\n2 ... )
    2DUP 2DUP + 2/ NUMBERS @ TEMP ! PARTITION
    >R ROT DUP R < R> SWAP
    IF      MYSELF ELSE 2DROP THEN 2DUP >
    IF SWAP MYSELF ELSE 2DROP THEN
;
```

The routine can be checked by filling NUMBERS with integers in reverse order by, for example

```
: NFILL ( n ... )
   DUP 0 DO
           DUP I NUMBERS ! 1 -
         LOOP DROP
;
```

Then execute, for example,

```
100 NFILL
```

which should fill the first 100 elements of NUMBERS with reverse-order integers. Executing

```
0 99 QUICKSORT
```

should then leave the elements of NUMBERS in ascending order. The contents can be checked by, for example:

```
: NCHECK ( n ... ) 0 DO I NUMBERS ? LOOP ;
```

Then execute, for example:

```
100 NCHECK
```

11.3 A Screen Copying Utility

Copying screens, particularly with a tape-based version, can be very tedious. The following routines will allow the copying of up to five screens at a time from one tape to another. The screen contents are temporarily stored in the graphics memory from #8200 to #8BFF. The screens may be renumbered during the copying process but the input and output screens must be numbered consecutively. They require the prior loading of the tape interface.

```
BASE @                   ( SAVE CURRENT BASE )
HEX

: OFFKEY                 ( DISABLE KEYBOARD )
    -1 20A +! ;

: ONKEY                  ( RE-ENABLE KEYBOARD )
     1 20A +! ;

: PAUSE                  ( WAIT FOR A KEYPRESS )
     KEY DROP CR ;

: INSCR ( first screen no.\no. of screens to input ... )
     CR ." PLAY " PAUSE
     OFFKEY
     0 DO
        DUP I + LIST         ( LOAD AND LIST SCREEN )
        FIRST 200 DUP
        I * 8200 +
        SWAP CMOVE           ( MOVE TO GRAPHICS MEMORY )
     LOOP DROP
     ONKEY
;

: OUTSCR ( first screen no.\no. of screens to output ... )
     OVER SCR !              ( SET NEW SCREEN NO. )
     ." RECORD" PAUSE
     OFFKEY
     0 DO
        I 200 * 8200 +       ( TRANSFER FROM    )
        FIRST 200 CMOVE      ( GRAPHICS MEMORY )
        SAVE
        1 SCR +!             ( INCREMENT SCREEN NO. )
        LOOP DROP
     ONKEY
     -1 SCR +!               ( RESET FOR NEXT BATCH )
;

: COPY ( first input screen no.\first output screen no.\
                             no. of screens ... )
     5 MIN                   ( ENSURE NOT MORE THAN 5 SCREENS )
     >R SWAP R               ( KEEP NO. OF SCREENS ON RETURN STACK )
     INSCR
     R>                      ( RECOVER NO. OF SCREENS )
     ." NEW TAPE " PAUSE
     OUTSCR
;

BASE !                       ( RESTORE ORIGINAL BASE )
```

To copy screens 6, 7 and 8 to a new tape, numbered as 15, 16 and 17, execute:

```
6  15  3  COPY
```

Separate use of INSCR and OUTSCR will allow intermediate editing of the screens provided each screen is moved from the graphics memory to the tape buffer, edited and then returned to the same area of the graphics memory. In HEX,

```
        n FIRST 200 CMOVE ( graphics -> tape buffer )
      FIRST  n  200 CMOVE ( tape buffer -> graphics )
```

where n may be

8200, 8400, 8600, 8800 or 8A00,

depending on which of the 5 screens is to be edited.

11.4 Use of the Operating System Monitor Routines

It may be necessary, from within FORTH, to use the cassette operating system commands. The following definitions allow this to be done either by direct execution from the keyboard or from within a definition.

```
BASE @ HEX

CREATE OSCLI ( call the OSCLI routine of the monitor )
      8E86 , 20 C, FFF7 ,
      8EA6 , 4C C, 2842 ,
SMUDGE

: (MONITOR)
    1 WORD HERE PAD C/L 1+ CMOVE   ( TRANSFER KEYED INPUT TO PAD )
    PAD COUNT                      ( GET ADDRESS & LENGTH OF STRING )
    2DUP + 0D SWAP C!              ( ADD 'RETURN' TO END )
    1+                             ( INCREASE STRING COUNT )
    100 SWAP CMOVE CR              ( TRANSFER TO COS BUFFER )
    IN @ 60 IN !                   ( PUT RTS HERE FOR MONITOR )
    OSCLI                          ( INTERPRET & EXECUTE )
    IN !                           ( RESTORE VALUE OF IN )
;

: MONITOR
     STATE @ IF                    ( COMPILING A DEFINITION )
               COMPILE QUERY
               COMPILE
             THEN
  (MONITOR)
; IMMEDIATE

BASE !
```

Note that THEN does not create a compiled address but only marks the end of a conditional, for calculation of an offset. The second COMPILE will, therefore, compile (MONITOR) and not THEN.

Any of the Cassette Operating System commands of Chapter 19 of 'Atomic Theory and Practice' pages 139 to 142 may then be used. Note that the initial * of these commands should not be used. For example, to use the COS command *CAT to obtain a catalogue of a tape, the sequence:

```
MONITOR CAT
```

should be used (and not MONITOR *CAT).

If MONITOR is used in a definition, it will, on execution of the definition, wait for the command to be entered from the keyboard. This allows the command to be selected at execution time rather than being fixed at the time of definition.

11.5 WAIT

The following example is an implementation of the WAIT instruction of ATOM BASIC, which waits until the next 'tick' of the 60Hz sync signal. Each tick is signalled by a zero in the most significant bit of Port C of the 8255 PIA, at address #B002 in the ATOM.

A simple implementation is as follows:

```
HEX

: WAIT
     BEGIN B002 C@ 80 < UNTIL
     BEGIN B002 C@ 7F > UNTIL ;

DECIMAL
```

This is equivalent to the machine code subroutine at #FE66 in ATOM BASIC.

11.6 Using the Internal Loudspeaker

11.6.1 Generating Tones

The simplest way of generating a tone from the internal loudspeaker is to TOGGLE bit 2 of the output port at address #B002. This is done by the word BLIP1 which is used by TONE1 to generate a short note.

```
HEX

: BLIP1
    B002 4 TOGGLE ;

: TONE1
    100 0 DO BLIP1 LOOP ;
```

A second tone may be generated by toggling the speaker twice within a definition.

```
: BLIP2
   B002 4 2DUP TOGGLE TOGGLE ;

: TONE2
   100 0 DO BLIP2 LOOP ;
```

The words BOP and BIP are useful for sound effects for paddle-type games:

```
: BOP
   30 0 DO BLIP1 LOOP ;

: BIP
   30 0 DO BLIP2 LOOP ;
```

Finally, we can produce a warble tone by:

```
: WARBLES
   0 DO BIP BOP LOOP;
```

This is used by typing, for example:

```
10  WARBLES
```

11.6.2 Music

The following definitions will allow the keyboard to be used to play music via the internal speaker.

```
CREATE NOTE ( 0\FRE\LEN ... )
   EAEA ,   EAEA ,   6A0 ,    88 C,
   FDD0 ,    18  C,  2B5 ,   475 ,
    495 ,    3B5 ,   575 ,   595 ,
    890 ,    4A9 ,   4D  C, B002 ,
    8D  C,  B002 ,   5B0 ,  EAEA ,
   EAEA ,    EA  C,   B5 ,   5D0 ,
    1D6 ,    18  C,  590 ,  EAEA ,
   EAEA ,    EA  C,   D6 ,  C7D0 ,
    1B5 ,   C6D0 ,  E8E8 ,    4C C,
   291E ,
SMUDGE

DECIMAL

: PITCH   ( FREQUENCY ... FRE )
     256 5 */ ;

: TONE    ( FREQUENCY ... )
     PITCH 0 SWAP 2500 NOTE ;

: TABLE   ( BYTESIZE ... )
    <BUILDS ALLOT DOES> ;

20 TABLE KEYS 40 TABLE FREQUENCIES

: CFILL ( BYTEVALS\TABLENAME\BYTELENGTH ... )
   0 DO
      DUP I +
      >R SWAP R> C!
   LOOP DROP
;

   32 93 91 64 59 80 76 75
73 74 85 72 89 71 70 82 68
69 83 65   KEYS 20 CFILL

: TFILL   ( VALS\TABLENAME\LENGTH ...)
   0 DO
     DUP  I 2* +
     >R SWAP R> !
   LOOP DROP
;


  0   683   640   608   576   542   512   483
456   457   406   387   352   341   320   304
288   271   256   243   FREQUENCIES  20  TFILL

: KBD ( C\ADDR ... OFFSET )
   0 BEGIN
      >R 2DUP
      R 19 > IF CR ." ?" 2DROP 1
             ELSE R + C@ =
             THEN R> 1+ SWAP
      UNTIL
   1 -
;

:  KEYBOARD
     BEGIN
       KEY KEYS KBD
       >R 2DROP R>
       2* FREQUENCIES + @ TONE
     ?ESC UNTIL
;
```

The tones produced by NOTE vary in timbre as well as pitch and are not 'pure' unless the value of FRE is an integral power of two. This is because the speaker is only toggled when the value third on the stack overflows. This occurs at slightly irregular intervals unless the above condition is met. One effect is an apparent shift in the pitch of some notes. The values in FREQUENCIES are therefore not exactly those expected for a true scale, but are chosen for the best perceived scale.

Executing KEYBOARD allows the notes to be played. Pressing the ESC key will terminate execution. The arrangement of the keys used is as follows:

```
   |E|R| |Y|U|I| |P|@|
 |A|S|D|F|G|H|J|K|L|;|[|]|
```

11.7 Using the VIA Timer

Execution times of FORTH words can be found by use of the VIA timer as shown in the following examples. To avoid timing the compilation of the words as well as their execution the timing should be carried out within a colon-definition.

```
HEX

: TIMER-ON      ( START TIMER )
     0 B80A ! FFFF  B808 ! ;

: .TIME ( adjustment ... )  ( DISPLAY ELAPSED TIME )
      20  B80A !
     FFFF B808 @ -
     SWAP -
     . ." MICROSECONDS "
;

DECIMAL
```

The word .TIME, which displays the time interval in microseconds from the instant the timer was started, requires a time adjustment on the stack to correct for the time taken to read the counter and for any other words whose timing is not required. The time to read the counter is 620 microseconds.

Examples:

```
: DROPTEST
    620 0 TIMER-ON DROP .TIME ;

    DROPTEST  48 MICROSECONDS  OK
```

(i.e. the word DROP executes in 48 microseconds)

```
: DUPTEST
    620 TIMER-ON DUP .TIME DROP ;

    DUPTEST  71 MICROSECONDS  OK


:  CONTEST
    668 TIMER-ON 0 DROP .TIME ;

   CONTEST  77 MICROSECONDS  OK
```

This last example gives the execution time of a constant, in this case the constant 0 . Its value must be dropped from the stack before executing .TIME so the execution time of DROP is added to the time adjustment, giving a total adjustment of 668 microseconds.

The following table gives execution times for some of the more common words.

WORD:	TYPE:	EXECUTION TIME (MICROSECONDS) :
DROP	CODE	48
DUP	CODE	71
OVER	CODE	71
ROT	FORTH	424
PICK	FORTH	1168
ROLL	FORTH	9481 + 1636(n-4) (n = 4, 5, 6 ...) (NOTE: 1 ROLL = NOOP, 2 ROLL = SWAP, 3 ROLL = ROT)
@	CODE	80
!	CODE	84
DO ... LOOP	CODE	80 + 116n (n = no. of loops)
I	CODE	79
*	CODE	759 to 1379 depending on the number of non-zero bits in the multiplier
/	CODE	4230 to 4394 largely depending on the number of digits in the result.

The words * and / are colon definitions but most of the execution time is spent in the machine-code primitives U* and U/ .The relatively long execution time for ROLL is due to it being (like PICK) a FORTH definition, rather than a machine code definition, and including an error check. It is particularly slow since it involves a fairly high degree of movement of the stack contents.

11.8 Further Graphics

The following application will plot a figure similar to that shown on the front cover of this manual. It uses recursive calls to 3SIDES to plot three sides of a rectangle, their orientation being controlled by a case statement (see Section 10.6.2).

The application requires the GRAPHICS package to have been loaded previously. This contains the definition of CASE: . The variables X and Y are defined in the FORTH vocabulary and are therefore distinct from the X and Y in the GRAPHICS vocabulary. Note that this duplication is not detected by the system since, when FORTH is the CONTEXT vocabulary, no other vocabularies are searched.

```
FORTH DEFINITIONS DECIMAL

4 CONSTANT N    128 CONSTANT H0
0 VARIABLE X      0 VARIABLE Y
0 VARIABLE X0     0 VARIABLE Y0
0 VARIABLE H

: YSCALE  ( SCALE Y VALUE TO FILL SCREEN )
    3 2 */  ;

: XYLINE  ( DRAW LINE TO X,Y*3/2 )
    X @ Y @ YSCALE GRAPHICS LINE  ;

: X+      ( LINE IN +VE X-DIRECTION )
    H @ X +! XYLINE  ;

: Y+      ( LINE IN +VE Y-DIRECTION )
    H @ Y +! XYLINE  ;

: X-      ( LINE IN -VE X-DIRECTION )
    H @ MINUS X +! XYLINE  ;

: Y-      ( LINE IN -VE Y-DIRECTION )
    H @ MINUS Y +! XYLINE  ;

: 3SIDES  ( INDEX\CASE NO ... INDEX )
    OVER DUP          ( 2 COPIES OF INDEX )
    IF                ( NON-ZERO INDEX )
       1 - SWAP       ( DECREMENT INDEX AND )
                      ( BRING CASE NO. TO TOP )

       [ HERE H ! ] NOOP
                      ( RESERVE SPACE FOR ORIENTATION )
                      ( - DEFINED LATER - AND SAVE )
                      ( ADDRESS IN H )
    ELSE 2DROP        ( CASE NO. AND INDEX )
    THEN
;

: ORA ( THESE DETERMINE THE 4 ORIENTATIONS )
    3 3SIDES X-
    0 3SIDES Y-
    0 3SIDES X+
    1 3SIDES DROP
;
```

```
: ORB
    2 3SIDES Y+
    1 3SIDES X+
    1 3SIDES Y-
    0 3SIDES DROP
;

: ORC
    1 3SIDES X+
    2 3SIDES Y+
    2 3SIDES X-
    3 3SIDES DROP
;

: ORD
    0 3SIDES Y-
    3 3SIDES X-
    3 3SIDES Y+
    2 3SIDES DROP
;

CASE: ORIENTATION
    ORA ORB ORC ORD
;

' ORIENTATION CFA H @ !
    ( PLACE ADDRESS OF ORIENTATION IN 3SIDES )
    ( TO COMPLETE RECURSION )

: INITIALISE
    H0 DUP H !
    2 / DUP X0 ! Y0 !
    GRAPHICS 3 CLEAR WHITE
;

: XYSET ( START POSITION AND SIZE FOR EACH PLOT )
    H @ 2 / DUP H !
    2 / DUP X0 +! Y0 +!
    X0 @ Y0 @
    2DUP Y ! X !
    YSCALE GRAPHICS MOVE
;

: PLOT-IT
    INITIALISE
    0 BEGIN 1+ ( INCREMENT INDEX )
         XYSET
         0 3SIDES
         DUP N =
      UNTIL
    DROP
    KEY DROP 12 EMIT
;
```

When the application has been entered the figure is displayed by executing PLOT-IT. The number of iterations is governed by the constant N . Its value may be changed by, for example:

```
3 ' N !
```

In graphics mode 3, the largest value of N for a clear display is 5. However an interesting textured effect can be produced by changing the WHITE in INITIALISE by, for example:

```
GRAPHICS ' INVERT CFA ( EXECUTION ADDRESS OF INVERT )
' XYSET NFA 4 -       ( LOCATION OF WHITE IN INITIALISE )
!
```

and executing PLOT-IT with N = 6.

11.9 Non-destructive Stack Print

The following short application will allow the stack contents to be displayed without destroying them. It is useful, for example, in the development and testing of an application.

It requires the previous loading of 2/ (Section 11.1). Alternatively the somewhat slower <2 /> may be used. In addition the silent user variable S0 must be given a dictionary header by

```
6 USER S0
```

The definitions are as follows:

```
: DEPTH ( ... n ) ( returns the number of stack items )
   SP@ S0 @ SWAP - 2/ ;

: .S ( ... )       ( non-destructive stack display )
   CR DEPTH
   IF S0 @ 2 - SP@ SWAP
      DO I ? -2 +LOOP
   ELSE ." EMPTY "
   THEN
;
```

The stack items are printed with the top stack item on the right.

12 Error Messages

Most detected errors in ATOM FORTH result in an error message of the form:

```
? cccc MSG # n
```

where cccc is the word where FORTH thinks the error has occurred. The general rule in error handling is that both the return and computation stacks are cleared. The one major exception is error message 4, indicating the redefinition of an existing word, when the message is simply a warning. The message number is printed in the current base so may not be immediately recognisable. In the following error message list the message number is given in decimal and hex.

DECIMAL	HEX	Message
0	0	Unrecognised word or invalid character
1	1	Empty stack
2	2	Dictionary full
3	3	Has incorrect address mode (Assembler)
4	4	Not unique (warning only)
5	5	Index or parameter outside valid range
6	6	Tape/disc screen number out of range
7	7	Full stack
8	8	(Reserved for disc use)
9	9	)
10	A	)
11	B	) User definable
12	C	)
13	D	)
14	E	)
15	F	(Reserved for disc use)
16	10	(Reserved for disc use)
17	11	Compilation only
18	12	Execution only
19	13	Conditionals not paired
20	14	Definition not finished
21	15	In protected dictionary
22	16	Use only when loading
23	17	Off current editing screen
24	18	Declare vocabulary

Glossary of FORTH Words

This glossary contains all words present in ATOM FORTH. Each entry is of the following form:

Word   Stack Action   Uses   Leaves   Status   Pronunciation

followed by a description and in many cases a numerical example.
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 that the word uses and leaves are also shown. Some words have an additional letter indicating their status.

A	only for the Acorn ATOM - not a standard FORTH word
C	may only be used in a colon definition
E	intended for execution only
P	has precedence bit set; will execute even when in compile mode

Where not obvious, the standard pronunciation is given in square brackets after the status.

The glossary contains all words that are immediately available to the user when the basic system is loaded. This includes headerless entries (see Appendix C), which are listed with their code field (execution) addresses. Since they have no names in the dictionary their names are completely arbitrary, but the names given are those preferred in a standard FORTH system.

!   (n\addr ...)   2 0   [store]

Stores the value n at the address addr.

before: 7 35 -1234 4128
after: 7 35

(-1234 is stored in the two bytes from address 4128)

!CSP   0 0   [store C-S-P]

Stores the stack pointer value in user variable CSP. Used as part of the compiler security.

#   (ndl ... nd2)   2 2   [sharp]

Converts the least-significant digit (in the current base) of the double-precision number ndl 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 #> .

If BASE is DECIMAL

before: 9 32 1234567
after: 9 32 123456

(ASCII code #37 is stored in PAD)

#>   (nd ... addr\count)   2 2   [sharp-greater]

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 .
.Hbefore: 23 19 0   0
after: 23 19 4128 3

(TYPE would display the 3-character string starting at address 4128)

#S   (ndl ... nd2)   2 2   [sharp-S]

Converts the double-precision number ndl 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 a value of zero. #S is used beween <# and #> .

before: 27 5 1234567
after : 27 5 0000000

(ASCII codes #37, #36, #35, ... #31 are in consecutive memory locations at PAD)

'   (... addr)   0 1 P   [tick]
(during execution)
0 0
(during compilation)

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 in the definition.

(**P** **[paren]**

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".

(+LOOP) **(n ...)** **1 0** **[bracket-plus-loop]**

Headerless code; execution address #28E0. The run-time procedure compiled by +LOOP that increments the loop index by the signed quantity n and tests for loop completion. See +LOOP .

(.") **[bracket-dot-quote]**

Headerless code; execution address #31E4. The run-time procedure compiled by ." that transmits the following in-line text to the output device. See ." .

(;CODE) **C**

Headerless code; execution address #3146. The run-time procedure 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) **[bracket-abort]**

Headerless code; execution address #347A. Executes after an error when WARNING is -1. It normally causes the execution of ABORT but the contents of the parameter area can be changed (with care) to point to to a user-defined error-handling procedure.

(DO) **C**

Headerless code; execution address #2910. The run-time procedure compiled by DO that moves the loop control parameters to the return stack. See DO .

(ENTER) **0 0** **[bracket-enter]**

Headerless code; execution address #3AF5. Interprets the current contents of the tape input buffer.

(FIND) **(addr1\addr2 ... pfa\b\tf)** **2 3** **[bracket-find]**
(found)
(addr1\addr2 ... ff) **2 1**
(not found)

Headerless code; execution address #2955. Searches the dictionary starting at the name field address addr2 for a match with the text starting at addr1. For a successful match the parameter field 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.

(LOAD) **(addr\f ...)** **2 0 A** **[bracket-load]**

Headerless code; execution address #3B69. The implementation-dependent routine used by LOAD and --> to load screens from tape or disc. The addr is that of the zero-page data required by the operating system. This data is completed by the creation of a 3-digit file name from the screen number and the insertion of its address as the first item. If the rest of the data is created by OSDATA then addr must be #62. The flag determines the appearance of the prompt on the display. In all cases an indication is given of the screen number for which a search is being made. If the flag is false a further prompt PLAY TAPE with a wait for a keypress is given. If the flag is true (non-zero) these further actions do not occur.

(LOOP) **[bracket-loop]**

Headerless code; execution address #28BA. The run-time procedure compiled by LOOP that increments the loop index by one and tests for loop completion. See LOOP .

(NUMBER) **(ndl\addrl ... nd2\addr2)** **3 3** **[bracket-number]**

Headerless code; execution address #33B7. Converts the ASCII text beginning at addrl + l according to the current numeric conversion base. The new number is accumulated into ndl, being left as nd2. Addr2 is the address of the first non-convertable digit. (NUMBER) is used by NUMBER .

***** **(nl\n2 ... n3)** **2 1** **[times]**

Leaves as n3 the product of the two signed numbers nl and n2.

before: 7 -3 9
after: 7 -27

***/** **(nl\n2\n3 ... n4)** **3 1** **[times-divide]**

Leaves as n4 the value nl*n2/n3. The product nl*n2 is kept as a double precision intermediate value, resulting in a more accurate result than can be obtained by the sequence nl n2 * n3 / .

before: 7 3 17 5
after: 7 10

***/MOD** **(nl\n2\n3 ... n4\n5)** **3 2** **[times-divide-mod]**

Leaves, as n4 and n5 respectively, the remainder and the integer value of the result of nl*n2/n3. The product nl*n2 is kept as a double precision intermediate value, resulting in a more accurate result than can be obtained by the sequence nl n2 * n3 /MOD .

before: 7 3 17 5
after: 7 1 10

+ **(nl\n2 ... n3)** **2 1** **[plus]**

Leaves as n3 the sum of nl and n2.

before: 19 7 24
after: 19 31

+! **(n\addr ...)** **2 0** **[plus-store]**

Adds n to the value at addr.

```
before: 25  -2  8427 (addr 8427 contains 29, for example)
after:  25           (addr 8427 now contains 27)
```

+- **(nl\n2 ... n3)** **2 1** **[plus-minus]**

Leaves as n3 the result of applying the sign of n2 to nl.

```
before: 17  4  -7
after:  17 -4
```

+LOOP **(n ...)** **1 0 P,C** **[plus-loop]**

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 or equal to the loop limit.

Execution then continues with the word following +LOOP .

+ORIGIN **(n ... addr)** **1 1** **[plus-origin]**

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.

, **(n ...)** **1 0** **[comma]**

Stores (compiles) n in the first two available bytes at the top of the dictionary and increments the dictionary pointer by two.

- **(nl\n2 ... n3)** **2 1** **[subtract]**

Leaves as n3 the difference nl - n2.

--> **P** **[next screen]**

Continues interpretation with the next screen of source code from tape.

-DUP **(ff ... ff)** **1 1** **[dash-dup]**
or **(tf ... tf\tf)** **1 2**

Duplicates the top stack value if it is true (non-zero).

-FIND **(... pfa\b\tf)** **0 3** **[dash-find]**
(if found)
(... ff) **0 1**
(if not found)

Used as -FIND nnnn . The CONTEXT and then the CURRENT vocabularies are searched for the word nnnn . If found, the entry's parameter field address, name length byte, and a true flag are left; otherwise just a false flag is left.

-TRAILING (addr\n1 ... addr\n2) 2 2 [dash-trailing]

Changes the character count n1 of the text string at addr so as not to include any trailing blanks, and leaves the result as n2.

. (n ...) 1 0 [dot]

Prints the number n on the terminal device in the current numeric base . The number is followed by one blank space.

." P [dot-quote]

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.

.R (n1\n2 ...) 2 0 [dot-R]

Print the number n1 at the right-hand end of a field of n2 spaces. Unlike <.> no following space is printed.

/ (n1\n2 ... n3) 2 0 [divide]

Leaves the value n3 = n1 / n2.

```
before: 13  27  6
after:  13  4
```

/MOD (n1\n2 ... n3\n4) 2 2 [divide-mod]

Leaves the remainder n3 and quotient n4 of n1/n2. The remainder has the sign of the dividend.

```
before:  13  27  6
after:   13   3  4
```

0,1,2 (... n) 0 1

These often-used numerical values are defined as constants in the dictionary to save both time and dictionary space.

0< (n ... f) 1 1 [zero-less]

Leaves a true flag if n is less than zero, otherwise leaves a false flag.

0= (n ... f) 1 1 [zero-equals]

Leaves a true flag if n is equal to zero, otherwise leaves a false flag.

0BRANCH (f ...) 1 0 [zero-branch]

Headerless code; execution address #28A2. 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 .

1+ (n1 ... n2) 1 1 [one-plus]

Increments n1 by one to give n2.

2* **(nl ... n2)** **1 1** **[two-times]**

Multiplies nl by two to give n2. Faster in execution than 2 * .

2+ **(nl ... n2)** **1 1** **[two-plus]**

Increments nl by two to give n2.

2DROP **(nd ...)** **2 0** **[two-drop]**

Drops the double-precision number nd (or two single precision numbers) from the stack.

2DUP **(nd ... nd\nd)** **2 4** **[two-dup]**

Duplicates the top double-precision number (or the top two single-precision numbers) on the stack.

: **P,E** **[colon]**

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 its precedence bit is set (P), in which case it is executed immediately.

; **P,C** **[semi-colon]**

Terminates a colon definition and stops further compilation.

;S **P** **[semicolon-S]**

Stops interpretation of a screen from tape. It is also the word compiled by <;> at the end of a colon definition to return execution to the calling procedure.

< **(nl\n2 ... f)** **2 1** **[less-than]**

Leaves a true flag if nl is less than n2, otherwise leaves a false flag.

before: 15 2 17
after: 15 1 (true)

<# **[less-sharp]**

Sets up for numeric output formatting. The conversion is performed on a double number to produce text at PAD . See also # , #> , #S , SIGN .

<BUILDS **C** **[builds]**

Used within a colon definition in the form

: cccc <BUILDS .... DOES> .... ;

to create a new defining word cccc

When cccc is executed in the form:

cccc nnnn

a new dictionary entry is created for nnnn with a name and a parameter area produced by <BUILDS and a high level execution procedure defined by DOES> .

When nnnn is executed it has the address of its parameter area (defined by <BUILDS) on the stack and executes the words after DOES> in cccc .

<CMOVE (from\to\count ...) 3 0 [reverse-C-move]

As CMOVE , but the source byte with highest address moves first and bytes are transferred in sequence of decreasing addresses. It is useful for short forward movements of memory contents, in cases where CMOVE would over-write the source data.

<TABLE! (addr1\n ... addr2) 2 1 [reverse-table-store]

Headerless code; execution address #3B3F. Places a value n at address addr1 and decrements the address by two to give addr2, ready for a new value. This can be used to store data in a table which is filled from the highest address towards lower addresses. It is used by LOAD to construct the address table for OSLOAD .

= (n1\n2 ... f) 2 1 [equals]

Leaves a true flag if n1 is equal to n2, otherwise leaves a false flag.

```
before:  53  27  27
after:   53  1
```

> (n1\n2 ... f) 2 1 [greater than]

Leaves a true flag if n1 is greater than n2, otherwise leaves a false flag.

```
before: 12 0 -1
after:  12  1
```

>R (n ...) 1 0 C [to-R]

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 temporarily to remove a number from the stack to access a lower number. See R> .

? (addr ...) 1 0 [question-mark]

Prints the value contained in the two bytes starting at addr. Equivalent to <@ .> .

?COMP [query-comp]

Issues an error message if not compiling.

?CSP [query-C-S-P]

Issues an error message if stack position differs from that saved in CSP . Used as part of the compiler security.

?ERROR (f\n ...) 2 0 [query-error]

Issues error message number n if the boolean flag f is true. Uses ERROR . The stack is always empty after an error message.

?ESC (... f) 0 1 [query-esc]

Tests the keyboard to see if the ESC key is depressed. A true (non-zero) flag is returned if the key is down at the time of the test, otherwise a false (zero) flag is returned. In many FORTH systems this function is carried out by the word ?TERMINAL which may test for any key being pressed.

?EXEC [query-exec]

Issues an error message if not executing.

?LOADING [query-loading]

Issues an error message if not loading from tape.

?PAIRS (nl\n2 ...) 2 0 [query-pairs]

Issues an error message if nl 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 [query-stack]

Issues an error message if the stack is out of bounds.

?TERMINAL

See ?ESC .

@ (addr ... n) 1 1 [fetch]

Leaves on the stack the 16-bit value s found at addr.

```
before:  11  4123
after:   11   375
```

(assuming the value 375 was stored in the two bytes from address 4123).

ABORT

Clears both stacks, enters the execution state, prints the start-up message on the terminal device and returns control to the keyboard.

ABS (n ... u) 1 1

Leaves u as the absolute value of n.

```
before: 12   -17
after:  12    17
```

AGAIN P,C

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 a endless loop.

ALLOT **(n ...)** **1 0**

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.

AND **(n1\n2 ... n3)** **2 1**

Leaves as n3 the bit-by-bit logical AND of n1 and n2.

(assuming binary)

```
before: 1101 1010  1100
after:  1101 1000
```

BACK **(addr ...)** **1 0**

Calculates the backward branch offset from HERE to addr and compile into the next available dictionary memory address. Used in the compilation of conditionals (AGAIN UNTIL etc.)

BASE **(... addr)** **0 1**

A user variable containing the current number base used for input and output conversion.

BEGIN **0 1 P,C**

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 **(... c)** **0 1** **[B-L]**

A constant that leaves the ASCII value for 'blank' or 'space' (Hex 20).

BLANKS **(addr\n ...)** **2 0**

Fill n bytes of memory starting at addr with blanks.

BLK **(... addr)** **0 1** **[B-L-K]**

A user variable indicating the input source. If BLK is zero, input is taken from the keyboard. If it is non-zero input is taken from the tape input buffer area.

BRANCH

Headerless code; execution address #288D. 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 .

C! **(b\addr ...)** **2 0** **[C-store]**

Stores byte b (8 bits) at addr.

before: 53 29 3127
after: 53

(29 is stored in the single byte at address 3127)

C, **(b ...)** **1 0** **[C-comma]**

Stores (compiles) b in the next available dictionary byte, advancing the dictionary pointer by one.

C/L **(... n)** **0 1** **[C-slash-L]**

A constant containing the number of characters per line. This is normally 64, so a full FORTH 'line' will occupy two lines of the VDU display.

C@ **(addr ... b)** **1 1** **[C-fetch]**

Leaves b as the 8-bit contents of addr.

CFA **(pfa ... cfa)** **1 1** **[C-F-A]**

Converts the parameter field address of a word to its code field (execution) address.

CLIT **(... n)** **0 1** **[C-lit]**

Headerless code; execution address #285B. Within a colon-definition CLIT can be compiled before an 8-bit literal value. When the word containing CLIT is later executed the 8-bit value (in the range 0-255) is pushed to the stack as a single-precision (16-bit) number with its most-significant part set to zero.

This word is used by a number of system words but is not available to the user via the keyboard interpreter, which uses only LIT .

CMOVE **(from\to\count ...)** **3 0**

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.

COLD

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 . It may be called from the keyboard to remove all application programs and restart with the nucleus dictionary alone.

COMPILE

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 (n ...) 1 0

A defining word used in the form

n CONSTANT cccc

to create a word cccc , with the value n contained in its parameter field. When cccc is executed the value n will be left on the stack.

CONTEXT (... addr) 0 1

A user variable leaving the address of a pointer to the VOCABULARY in which a dictionary search will start.

COUNT (addr1 ... addr2\n) 1 2

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.

(assuming that the text string 3 65 66 67 starts at 6124),

before: 47 6124
after: 47 6125 3

(TYPE would then display the 3 characters ABC)

CR [C-R]

Transmit a carriage return and line feed to the terminal output device.

CREATE

A defining word used in the form

CREATE cccc

to create a dictionary header for the word cccc with its code field pointing to the first byte of the parameter field.

One common use is, with the aid of <,>, to compile machine code into the parameter area to produce a machine code primitive. This does not need an assembler vocabulary.

CSP (... addr) 0 1 [C-S-P]

A user variable used for temporary storage of the stack pointer in checking of compilation errors.

CURRENT (... addr) 0 1

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+ (nd1\nd2 ... nd3) 4 2 [D-plus]

Leaves as nd3 the double number sum of double number nd1 and nd2.

D+-   (nd1\n ... nd2)   3 2   [D-plus-minus]

Applies the sign of single number n to the double number nd1, leaving the result nd2. See +- .

D.   (nd ...)   2 0   [D-dot]

Prints the signed double number nd according to the current BASE. One blank is printed after the number. See <.>.

D.R   (nd\n ...)   3 0   [D-dot-R]

Prints a signed double number nd on the right of a field n characters wide. See .R . No trailing blank is printed.

DABS   (nd ... ud)   2 2

Leaves the absolute value ud of a signed double number nd. See ABS .

DECIMAL

Sets BASE to decimal numeric conversion for input and output.

DEFINITIONS

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 .

DIGIT   (c\n1 ... n2\tf)   2 2
(valid)
(c\n1 ... ff)   2 1
(invalid)

Converts ASCII character c, wih base n1, to its binary equivalent n2 and a true flag. If c is not a valid character in base n1, then only a false flag is left.

In hexadecimal base, but displaying stack in binary.

a) Valid
Hex character 'D' has ASCII code #44, or 01000100 in binary, and represents the value 1101 in binary

```
before: 00010110   01000100   00010000
after:  00010110   00001101   00000001
```

b) Invalid
Character 'G' has ASCII code #47, or 01000111 in binary, and does not represent a hexadecimal value

```
before: 00010110   01000111   00010000
after:  00010110   00000000
```

DLITERAL (nd ...) **2 0 P**
(compiling)

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.

DMINUS (nd1 ... nd2) **2 2**

Change the sign of nd1, leaving it as nd2.

DO (n1\n2 ...) **2 0 P,C**

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> **[does]**

Used with <BUILDS to create a new defining word cccc . When a word nnnn (created with cccc) executes it uses the sequence of operations following DOES> in cccc . At the start of this sequence the address of the parameter field of nnnn will be put on the stack so that the execution can refer to the particular values associated with nnnn . See <BUILDS .

DP (... addr) **0 1** **[D-P]**

A user variable, the dictionary pointer, leaves addr, whose contents point to the first free byte at the top of the dictionary.

DPL (... addr) **0 1** **[D-P-L]**

A user variable containing the number of digits to the right of the decimal point on double number input. It may also be used to contain the column location of a decimal point in user-generated output formatting. On single number entry the value in DPL defaults to -1.

DROP (n ...) **1 0**

Drops the top number on the stack.

```
before: 53  21
after:  53
```

DUP (n ... n\n) **1 2**

Duplicates the top number on the stack.

```
before: 53  21
after:  53  21  21
```

ELSE P,C

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 (c ...) 1 0

Transmits ASCII character c to the output device. The contents of OUT are incremented for each character output.

```
before:  23     65
after:   23
(A is displayed on the output).
```

ENCLOSE (addr\c ... addr\n1\n2\n3) 2 4

Headerless code; execution address #29B1. The text-scanning primitive used by WORD .

The text starting at 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 (00) 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 (addr\n ...) 2 0

Sets n bytes of memory starting at addr to contain zeros.

ERROR (n ...) 1 0

Gives an error notification. The value in WARNING is examined and if it is -1 a system ABORT is executed, via (ABORT) which is a headerless dictionary entry. Otherwise an error message number n is given, the stacks are emptied and control is returned to the keyboard. In the system as supplied, WARNING is set to zero so the error message is given. Changing WARNING to -1 and also the vector in (ABORT) allows the user to define his own error response.

EXECUTE (addr ...) 1 0

Executes the definition whose code field (execution) address is on the stack.

EXPECT (addr\count ...) 2 0

Transfers characters from the keyboard to the memory starting at 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 addr. One or more nulls are added at the end of the text.

FENCE (addr ...) 1 0

A user variable containing an address below which the user is not allows to FORGET . In order to use FORGET on an entry below this point it is necessary to alter the contents of FENCE . In the ATOM, changing the contents of FENCE to a value less than #8000 (the start of the upper block of RAM) may produce unpredictable results for FORGET .

FILL (addr\n\b ...) 3 0

Fills n bytes of memory starting at addr with the value b.

FIRST (... n) 0 1

A constant that leaves the address n of the first byte of the tape input/output buffer area.

FORGET E

Used in the form

FORGET cccc

to delete the definition with name cccc and all dictionary entries following it. An error message is given if the CURRENT vocabulary is not the same as the CONTEXT vocabulary, i.e. if the entry cccc is not in the vocabulary that is searched first. An error message is also given if cccc is in the protected area of the dictionary, below FENCE . Regardless of the value stored in FENCE the nucleus dictionary, all of which is necessary for the correct operation of the system, is protected against FORGET .

FORTH P

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.

HERE (... addr) 0 1

Leaves the contents of DP i.e. the address of the first unused byte in the dictionary.

HEX

Sets the numeric conversion BASE to sixteen (hexadecimal).

HLD (... addr) 0 1 [H-L-D]

A user variable containing the address of the latest character of text produced during numeric output conversion (by #).

HOLD **(c ...)** **1 0**

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 **(... n)** **0 1 C**

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 beween DO and LOOP .

ID. **(addr ...)** **1 0** **[I-D-dot]**

Prints the name of a word from its name field address on the stack.

IF **(f ...)** **1 0 P,C**

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

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] .

IN **(... addr)** **0 1**

A user variable containing the byte offset to the present position in the input buffer (terminal or tape) from where the next text will be accepted.

INTERPRET

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 CURRENT 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 part of a number the position of the decimal pointer will be stored in DPL and a double number will be left on the stack. The number itself will not contain any reference to the decimal point.

KEY **(... c)** **0 1**

Leaves the ASCII value of the next terminal key pressed.

LATEST **(... addr)** **0 1**

Leave the name field address of the most recently defined word in the CURRENT vocabulary.

LEAVE **C**

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 **(pfa ... lfa)** **1 1** **[L-F-A]**

Convert the parameter field address, pfa, to its link field address, lfa.

LIMIT **(... addr)** **0 1**

A constant leaving the address of the first byte after the highest memory available for the tape I/O buffer.

LIT **0 1 C**

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 **(n ...)** **P,C**

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 **(n ...)** **1 0**

Searches the tape file for screen n and, when found, loads it into the tape buffer. The PLAY TAPE response is as described in *LOAD, page 140 of "Atomic Theory and Practice", with the addition that the message '>n' is given, indicating that a search is being made for screen n. The contents of the buffer is then interpreted. Interpretation will end at the end of the screen or at ;S unless --> is found, in which case loading will continue with screen n+1.

LOOP **P,C**

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* **(n1\n2 ... nd)** **2 2** **[M-times]**

Leaves as the double-precision number nd the signed product of the two signed single-precision numbers n1 and n2.

M/ **(nd\n1 ... n2\n3)** **3 2** **[M-divide]**

Leaves, as n2 and n3 respectively, the signed remainder and signed quotient from the division of the double number dividend nd by the single divisor n1. The sign of the remainder is that of the dividend.

M/MOD **(ud1\u2 ... u3\ud4)** **3 3** **[M-divide-mod]**

Leaves, as ud4 and u3 respectively, the double quotient and remainder from the division of the double dividend ud1 by the divisor u2. All are unsigned integers.

MAX **(n1\n2 ... max)** **2 1**

Leaves as max the larger of n1 and n2.

MESSAGE **(n ...)** **1 0**

Prints on the output device error message number n.

MIN **(n1\n2 ... min)** **2 1**

Leaves as min the smaller of n1 and n2.

MINUS **(n1 ... n2)** **1 1**

Changes the sign of n1 and leaves the result as n2. The sign is changed by forming the two's complement.

MOD **(n1\n2 ... mod)** **2 1**

Leaves as mod the remainder of n1/n2 with the sign of n1.

NFA **(pfa ... nfa)** **1 1** **[N-F-A]**

Converts the parameter field address of a definition to its name field address.

NUCTOP **(... addr)** **0 1 A**

Headerless code, execution address #2822. A constant containing the address of the top of the nucleus dictionary. Used by FORGET to prevent accidental forgetting of the nucleus dictionary, all of which is required for the correct operation of the system.

NUMBER **(addr ... nd)** **1 2**

Converts the character string starting at addr with a character count byte, to the signed double number nd using the current numeric base. If the string contains a decimal point its position will be given in DPL. If a valid numeric conversion is not possible an error message will be given. Used by INTERPRET .

NOOP **[no-op]**

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.

OR **(n1\n2 ... or)** **2 1**

Leaves as or the bit-by-bit logical OR of n1 and n2.

OSDATA (addr1\n ... addr2) 0 1 A [O-S-data]

Headerless code; execution address #3B4B. The implementation-dependent routine that creates the zero-page data required by the operating system load or save, with the exception of the pointer to the filename (see "Atomic Theory and Practice" p.191). The beginning of this data, left as addr, is at #62.

OSLOAD (addr\f ...) 2 0 A [O-S-load]

Headerless code; execution address #3B24. The implementation-dependent machine code used by (LOAD) to make the appropriate call to the operating system load routines. The meanings of addr and f are as for (LOAD) .

OUT (... addr) 0 1

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 (n1\n2 ... n1\n2\n1) 2 3

Copies the second stack item over the top item.

```
Before:  15  23  17
after:   15  23  17  23
```

PAD (... addr) 0 1

Leaves the address of the text output buffer. This is always 68 bytes above HERE . Numeric output characters are stored downwards from PAD , character text is stored upwards.

PFA (nfa ... pfa) 1 1 [P-F-A]

Converts the name field address, nfa, of a dictionary entry to its parameter field address, pfa.

QUERY

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

Clears the return stack, stops compilation and returns control to the keyboard. No message is given.

R (... n) 0 1

Copy the top of the return stack to the computation stack. The action is identical to that of I .

R0 (... addr) 0 1 [R-zero]

A silent user variable (no dictionary entry) containing the initial address of the top of the return stack. It may be given a header in the dictionary by:

```
8 USER R0
```

R#   (... addr)   0 1   [R-sharp]

A user variable which contains the location of the editing cursor for the Editor.

R>   (... n)   0 1   [R-from]

Removes the top value from the return stack and leaves it on the computation stack . See >R .

REPEAT   P,C

Used in a colon definition in the form

BEGIN ... WHILE ... REPEAT

In execution REPEAT forces an unconditional branch back to BEGIN .

ROLL   (n ...)   1 1

Rotates the top n1 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
```

An error message is given if n is less than 1.

ROT   (n1\n2\n3 ... n2\n3\n1)   3 3

Rotates the top three items on the stack, bringing the third item to the top

```
before:  92  17  28  12
after:   92  28  12  17
```

RP!   [R-P-store]

Initialises the return stack pointer.

RP@   (... addr)   0 1   [R-P-fetch]

Leaves the address of the return stack pointer. Note that this points one byte below the last return stack value.

S->D   (n ... nd)   1 2   [S-to-D]

Leaves as nd the signed single-precision number n converted to the form of a signed double-precision number (with unchanged value).

S0   (... addr)   0 1   [S-zero]

A silent user variable (no dictionary entry) containing the address which marks the initial top of the computation stack. It may be given a header in the dictionary by:

6 USER S0

SCR   (... addr)   0 1   [S-C-R]

A user variable containing the number of the most recently referenced source text screen.

SIGN **(n\nd ... nd)** **3 2**

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

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! **0 0** **[S-P-store]**

Initialises the computation stack pointer.

SP@ **(... addr)** **0 1** **[S-P-fetch]**

Leaves the value of the stack pointer on the stack. The value corresponds to the state of the stack before the operation.

```
before:  1   2
(address 58  56   54)
after:   1   2    56
```

SPACE

Transmits an ASCII blank to the output device.

SPACES **(n ...)** **1 0**

Transmits n ASCII blanks to the output device.

STATE **(... addr)** **0 1**

A user variable indicating the state of compilation. A zero value indicates execution and a non-zero value indicates compilation.

SWAP **(n1\n2 ... n2\n1)** **2 2**

Exchanges the top two items on the stack.

```
before: 17  23  59
after:  17  59  23
```

THEN **P,C**

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 **(... addr)** **0 1**

A user variable containing the address of the terminal input buffer.

TRAVERSE (addr1\n ... addr2) 2 1

Headerless code; execution address #2FB8. 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.

TOGGLE (addr\b ...) 2 0

Complements the contents of addr by the bit pattern b.

```
before: b = 00110000, contents of addr = 01101010
after:                contents of addr = 01011010
```

TYPE (addr\count ...) 2 0

Transmits count characters of a string starting at addr to the output device.

U* (u1\u2 ... ud) 2 2 [U-times]

Leaves the unsigned double-precision product of two unsigned numbers.

U. (n ...) 1 0 [U-dot]

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/ (ud\u1 ... u2\u3) 3 2 [U-divide]

Leaves the unsigned remainder u2 and unsigned quotient u3 from the division of the unsigned double dividend ud by the unsigned divisor u1.

U< (un1\un2 ... f) 2 1

Unsigned comparision. Leaves a true flag if un1 is less than un2, otherwise leave a false flag. For correct operation the difference between un1 and un2 should not exceed 32767.

UNTIL (f ...) 1 0 P,C

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 .

USER (n ...) 1 0

A defining word used in the form

n USER cccc

to create a user variable cccc . Execution of cccc 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.

VARIABLE (n ...) 1 0

A defining word used in the form

n VARIABLE cccc

to create a variable cccc with initial value n. Execution of cccc leaves the address, in the parameter area of cccc , containing the value of cccc .

VLIST [V-list]

Display, on the output device, a list of the names of all words in the CONTEXT vocabulary and any other vocabulary from which the CONTEXT vocabulary is chained. All VLIST's will therefore include a listing of words in the FORTH vocabulary. The listing be be interrupted by pressing the ESC key and resumed by pressing the space bar. If, after interruption, The ESC key (or any other key except the space bar) is pressed, the listing will be aborted.

VOC-LINK (... addr) 0 1

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 E

A defining word used in the form

VOCABULARY cccc

to create 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 .

WARNING (... addr) 0 1

A user variable whose value determines the action on detection of an error. If WARNING contains -1 an error causes a system ABORT which may, by changing a pointer in (ABORT) , be altered to a user-defined response. If WARNING contains zero then an error message number is given. In the system provided, WARNING is set to zero on initialisation . See ERROR .

WHILE (f ...) 1 0 P,C

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 (... addr) 0 1

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.

WORD (c ...) 1 0

Accepts text characters from the input buffer (terminal or tape) until a delimiter character c is found. The string starting with a length count byte, is then placed in the WORD buffer starting at HERE and two or more banks are added to the end. The choice of input buffer is determined by BLK . See BLK , IN .

X

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 both input buffers have at least one null character at the end.

XOR (n1\n2 ... xor) 2 1

Leaves the bit-by-bit logical exclusive-OR of n1 and n2.

[P [left-bracket]

Used in the creation of a colon definition in the form

: nnnn ... [...] ... ;

to suspend compilation of the definition and allow words to execute . See] .

[COMPILE] P,C [bracket-compile]

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 e.g.

[COMPILE] FORTH

to delay the change of the CONTEXT vocabulary to FORTH until the word containing the above sequence executes.

] [right bracket]

Used during the creation of a colon definition, to resume compilation after the suspension of compilation by [.

Appendix A
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):

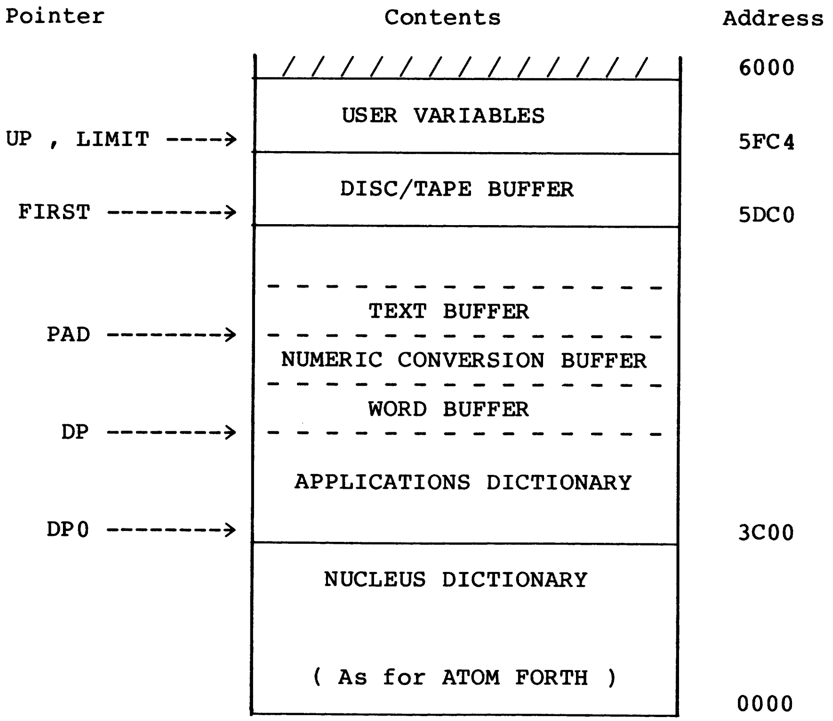
```
            1 1 1 1 1 0 1 1
```

add 1 (form the two's-complement):

```
            1 1 1 1 1 1 0 0    (-4)
```

Appendix B
System FORTH

FORTH for the Acorn Systems 3 and 4 is almost identical to ATOM FORTH, except for the changes in the memory map. FORGET is changed to work in the range #3C00 and #8000 and no memory is needed above #8000.



The system can be modified to use memory up to #8000 by changing UP, LIMIT and FIRST, for example:

```
HEX
LIMIT 2000 + ' LIMIT !
FIRST 2000 + ' FIRST !
10 +ORIGIN @ 2000 + 10 +ORIGIN !        ( change UP )
COLD                                    ( initialise )
```

The system should then be resaved:

```
SAVE FORTHA 240 400 2EB
SAVE FORTHB 2800 3C00 2EB
```

Appendix C
Dictionary Entry Structure

All dictionary entries in FORTH have the same general form:

NFA	Name length (1 byte), Characters of the name (up to 31 bytes)	Name field
LFA	Link pointer to previous NFA (2 bytes)	Link field
CFA	Pointer to machine code to execute (2 bytes)	Code field
PFA		Parameter field

The name length byte contains, in its least significant five bits, the number of characters in the name of the word (maximum 31 characters). The sixth bit is the 'smudge' bit which, when set to 1, will prevent the dictionary entry from being found on a dictionary search (except by VLIST). This is mainly used to prevent the finding and use of a partly-completed dictionary entry. The seventh bit is known as the precedence bit and marks a word as being IMMEDIATE when set to 1. The eighth or most significant (sign) bit is always set to 1, as is the sign bit of the last character of the name. This is to allow the operation of TRAVERSE , which will move, in either direction, across the name field of the word.

The link field contains the address of the start of the name field of the preceding dictionary entry to allow a dictionary search to be made. A link field containing zero marks the end of the dictionary.

The various types of dictionary entry 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 machine code is placed in the parameter field of the entry and the code field points to its start.

b) Constants

The value of the constant is contained in a two-byte parameter field and the code field points to a machine-code routine to place the contents of the parameter field on the stack.

c) Variables

The value of the variable is contained in a two-byte parameter field and the code field contains a pointer to a machine-code routine which places the address of the parameter area on the stack.

d) User Variables

The offset from the start of the user variable area to the address where the variable is stored is contained in a one-byte parameter area. The code field points to a machine-code routine which adds the offset to the address of the start of the user variable area and places the result on the stack.

e) Colon definitions

The code field contains a pointer to a machine-code routine which interprets the contents of the parameter field as a list of addresses of other FORTH words to be executed.

f) Words constructed using <BUILDS and DOES>

The first two bytes of the parameter area contain the address of the words following DOES> in the creating word. The remainder of the parameter area contains a series of values placed there by the words (if any) following <BUILDS in the creating word.

The code field contains a pointer to machine code which will

(i) place the address of the third byte of the parameter area (the start of the values placed there by <BUILDS) on the stack;

(ii) execute the list of words starting at the address contained in the first two bytes of the parameter area (the words following DOES>).

Saving Dictionary Space

The maximum length of the name of a dictionary entry is contained in the user variable WIDTH . This may at any time be reduced from its default value of 31 characters, with a consequent saving of space in the name field of a dictionary entry. If, for example, the value of WIDTH is reduced to 3 by

```
3 WIDTH !
```

then any new dictionary entry will be with the actual length of its name, but only the first three characters saved. All words must then be uniquely determined by their length and their first three characters (i.e. LOOK and LOOP will not be distinguished). The use of a value of WIDTH less than 3 is not recommended but, as this demonstrates,

IT IS VER- EAS- TO REA- FOR-- IF YOU ONL- HAVE- THE FIR-- THR-- LET---- AND THE LEN--- OF THE WOR-

The value of WIDTH may be increased or decreased at any time (subject to its remaining in the range 3 to 31 inclusive) and will not have any effect on previously defined words.

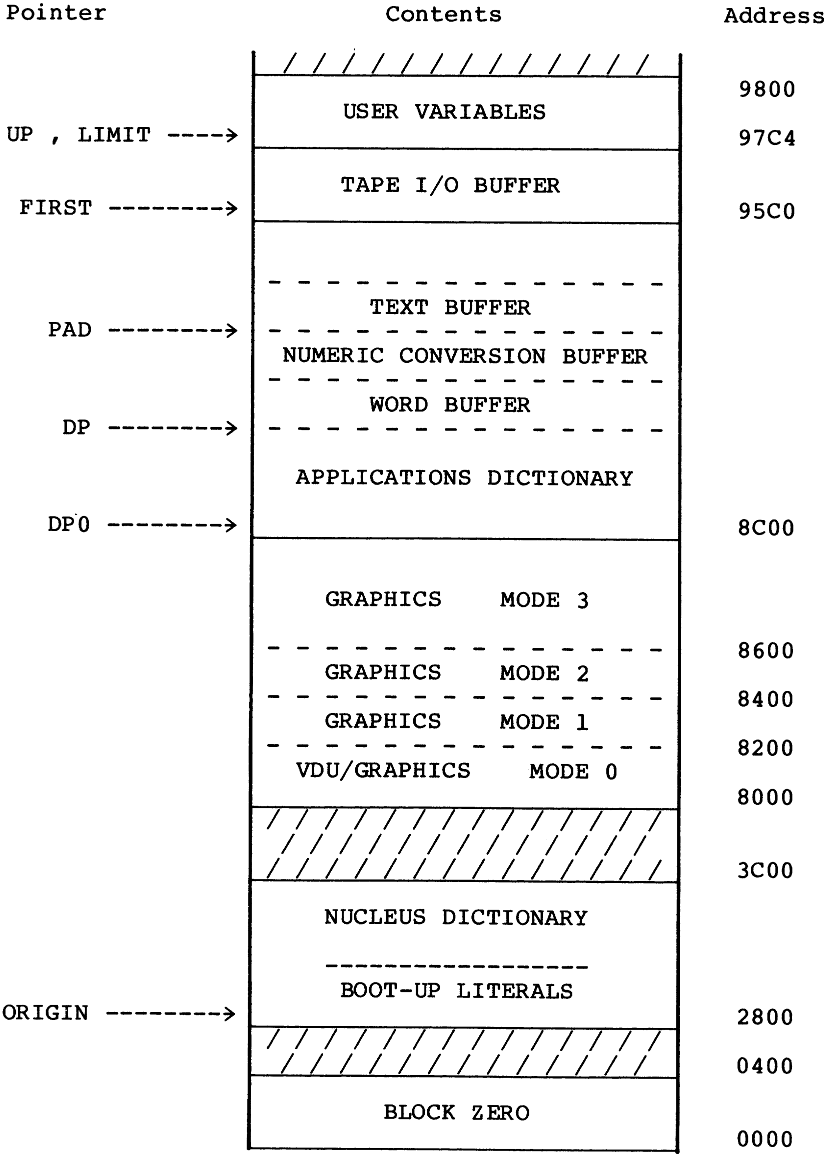
Headerless Code

Some dictionary entries in ATOM FORTH are 'headerless'. This means that their heads do not include name and link fields. They cannot, therefore, be found by a dictionary search, or included in a new definition unless their code field (execution) addresses are known to the programmer. The glossary gives this address for each headerless entry.

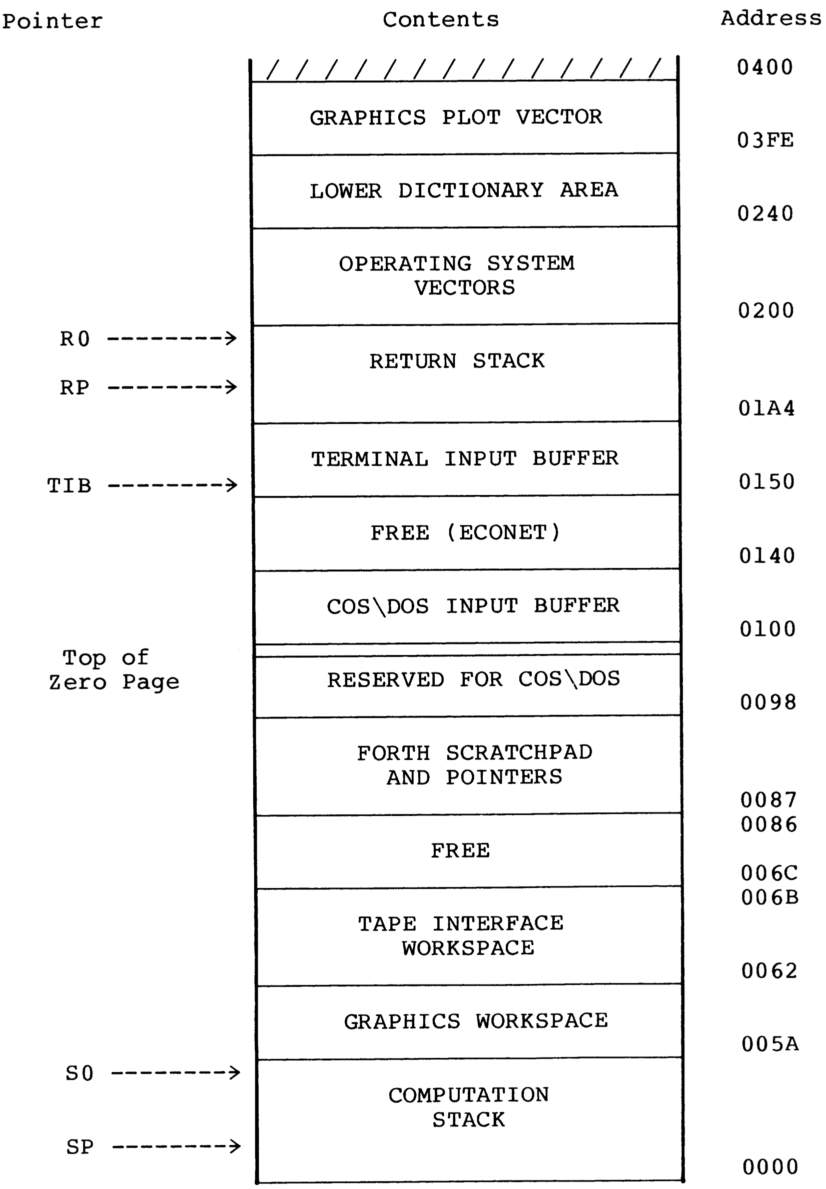
Creating headerless code is a very efficient way of saving memory, but it does reduce the flexibility of the system since headerless entries are relatively difficult to use.

The main use of headerless code is in producing a stand-alone system whose action is fixed, such as a dedicated control system.

Appendix D
Memory Allocation



Block zero Memory Map



Relocation of the Applications Dictionary

The applications dictionary, tape buffer and user variable area normally reside in memory between #8200 and #97FF. All addresses in this range are, if treated as signed single-precision integers, 'negative'. This has two important consequences for relocation of the applications dictionary in the address range below #8000:

1. Before compiling a dictionary entry, FORTH checks whether there is sufficient room between the top of the dictionary (contents of DP) and the start of the tape buffer (value of FIRST). A 'positive' value in DP and a 'negative' value of FIRST will cause the test to fail and give error message 2 (dictionary full). In order to avoid extensive changes to the system, relocation of the applications dictionary below #8000 should be accompanied by a corresponding relocation of the tape buffer.

2. The word FORGET performs an address validation check before allowing part of the dictionary to be discarded. This check assumes that valid addresses are 'negative', and they will therefore fail in the relocated applications dictionary. The definition of FORGET must be modified for the relocated system.

The modifications given here also relocate the user variables. This is not strictly necessary but will leave the upper RAM completely free so that mode 4 graphics can be used. The changes are made permanent so that the new system can be saved on tape. The areas to be saved are #240 to #400 and #2800 to #3C00, both with an execution address of #2800. A minimum of 2K of RAM is required in the region between #3C00 and #7FFF inclusive. In the code, XXXX represents the RAM start address and YYYY the end address + 1.

```
COLD                     ( start on an empty applications dictionary )
HEX

XXXX DUP DP !            ( relocate applications dictionary )
DUP 1E +ORIGIN !         ( change DP boot-up parameter )
DUP 1C +ORIGIN ! FENCE ! ( FENCE and its boot-up parameter )

YYYY 3C -                ( start of new user variable area )
DUP 10 +ORIGIN !         ( change user variable boot-up parameter )

DUP ' LIMIT !            ( relocate end of tape buffer )
204 - ' FIRST !          ( and the beginning )

: TEMP                   ( a temporary definition of the new FORGET )
    CURRENT @ CONTEXT @
    - 18 ?ERROR          ( CONTEXT = CURRENT? )
    [COMPILE] '
    DUP FENCE @
    [ 2822 , ]           ( top of nucleus )
    MAX < 15 ?ERROR      ( below FENCE or in nucleus? )
    DUP NFA DP !
    LFA @ CURRENT @ ! ;
```

If this definition is compiled successfully, all is well so far.

```
' TEMP HERE OVER -       ( PFA and parameter field length )
' FORGET SWAP CMOVE      ( overwrite FORGET )
                         ( the new version is shorter, so OK )
```

```
FORGET TEMP

               ( If TEMP is forgotten successfully, everything is OK )

COLD           ( Finally, check out modified boot-up parameters )
```

Appendix E
Further Reading

1. The FORTH Interest Group in the U.S.A. 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 an SAE please) to:

FORTH Interest Group,
P.O. Box 1105,
San Carlos,
Ca. 94070.

2. The FORTH Interest Group U.K. is the British branch of the U.S.A. group. At present it meets on the first Thursday of every month at 7 p.m. at the Polytechnic of the South Bank in London. Membership includes a bi-monthly newsletter entitled "FORTHWRITE". Like its parent group, F.I.G. U.K. 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 (S.A.E. please):

The Honorary Secretary,
F.I.G. U.K.,
15, St. Albans Mansions,
Kensington Court Palace,
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

The August 1980 issue is now unobtainable, but reprints of all the BYTE articles are available from the U.S.A. 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 (Nov. 1981).

Available from: 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 month's time!

Index

ACORNSOFT

SECOND EDITION

ISBN 0 907876 05 6