functional forth
for the BBC computer

Understand your computer through Forth

boris allan

functional forth
for the BBC computer

functional forth
for the BBC computer

Understand your computer through Forth

boris allan

First published 1983 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12 – 13 Little Newport Street,
London WC2R 3LD

Copyright © Boris Allan
ISBN 0 946408 04 1

Reprinted 1984

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

Cover design by Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

		Page
0	Becoming Forthright	1
1	Backwards and Forthwards	7
2	Stack and Colon Steering	19
3	Circular Sums	33
4	Editorial Freedom	49
5	Complete Control	65
6	A Logical Language	77
7	Son et Lumière	91
8	Creative Writing	101
9	Turning Turtle Graphics	107
A	Further with Forth?	115

Contents in detail

SCREEN 0
Becoming Forthright
An outline of the way Forth is organised, and an idea of its speed.

SCREEN 1
Backwards and Forthwards
What is a Forth 'word' and how it is composed, the use of pointers. An analysis of one word in detail, and how it is stored in memory.

SCREEN 2
Stack and colon steering
How to study the stack, by use of special Forth functions, given in colon definitions. The nature of colon definitions.

SCREEN 3
Circular sums
How arithmetic is performed, and the use, the need for complement arithmetic. A major application to emulate computer arithmetic, using execution vectors.

SCREEN 4
Editorial freedom
The difference between instant execution, and delayed execution, the difference between source and compiled applications. The use of the operating system commands from within Forth, the use of the Editor.

SCREEN 5
Complete control
An intensive study of control structures (definite and indefinite mechanisms), and control as exemplified in the design of applications. The construction of a random number generator, and a guessing game.

SCREEN 6
A logical language
The development of a logical vocabulary based on one logical function. The use of Polish notation and reverse Polish notation.

SCREEN 7
Son et lumière
Sound effects in Forth, and an in-depth study of the use of assembler facilities for two applications. Graphics, and the flexible use of VDU drivers.

SCREEN 8
Creative writing
Creating new structures in Forth; and the particular example of the simple manipulation of user-defined characters.

SCREEN 9
Turning turtle graphics
The final application, a major implementation of Turtle Graphics.

SCREEN A
Further with FORTH?
Further sources of information.

SCREEN 0
Becoming Forthright

For the things we have to learn before
we can do them, we learn by doing
them..
Aristotle, *Nicomachean Ethics*

Sit in front of your computer, and type in this line, when in BASIC (eg as soon as you have switched on):

```
10 MODE 4
20 FOR I=&5800 TO &7FFF
30 ?I = I
40 NEXT I
```

This is a simple program to put the value I into location I, where the locations vary from &5800 (the hexadecimal way of writing 22528) to location &7FFF (ie 32767). These are the locations taken up by high resolution graphics in mode 4.

Running the program fills the screen with a pretty patterned effect. I time the operation as taking 18½ seconds (from hitting return after the RUN, to the complete filling of the screen).

This is the only BASIC program in this book, and is here to make you feel unhappy. 18½ seconds is a very long time, and pretty striations are little compensation.

Load the Forth system, and wait. Type in the following, taking great care:

```
HEX
: SCREENFILL 8000 5800 DO  I  I  C!  LOOP ;
DECIMAL
```

If all is right (and if it is, you get an OK output) then merely type in

```
SCREENFILL
```

to fill the screen in about two seconds. If you did not get it right, try, try again. The Forth word SCREENFILL achieves the same object as the BASIC program, but was about ten times as swift. With graphics that speedy, the need to use machine code is lessened. Speed can be obtained in graphics applications by use of Forth, a speed which is sufficient for most purposes (and certainly most video games).

Forth should be used, then, for its speed; but, as or more important, use Forth for its ability to be extended, and for its sophisticated approach to programming. All three features are related by the Forth philosophy 'keep it simple'. A simple solution has elegance: but a simple solution means that we have had to understand the real problem, and have not ducked the problem by hiding behind complexity.

Charles H Moore, the inventor of Forth, says that simplicity provides confidence, reliability, compactness and speed. If you understand Forth, you will, at the end of the day, find that you understand far more than you ever did before about computers and computer languages. Because Forth is so powerful, and can help you find out so much more about any computer, most of the examples given in this book will help you to understand other machines, and other Forths (and most Forths are very similar).

The key characteristic of Forth is that it is a 'threaded interpretive language', a name which tries to indicate how Forth works. Each word in Forth (eg SCREENFILL above) points to other words (eg DO, I, C!, or LOOP), which each point to other words (which each point to other words, . . .). The threading through the definitions comes to an end when a word is defined totally by a machine code routine.

Already we can see the truth of Aristotle's dictum: to learn Forth we need to know how it works, but to learn how it works we need to use Forth. If the instructions in a repair manual are

> Unscrew the nut holding the wurble plate to the ding box, but only after disconnecting the mains supply to the ding box, or you will be electrocuted.

there may be a fair number of fatalities. In a TIL (a threaded interpretive language) the manual has to be written in a sensible order:

1) Disconnect the mains supply to the ding box.
2) Unscrew the nut holding the wurble to the ding box.

That is, no nasty surprises. If we read ahead in the manual before acting then we may be safe, but the whole operation is that much slower (and how far ahead do we have to read to be perfectly sure?)

Any computer program is no more than a set of instructions, and sometimes the same set of instructions are repeated. A truly ignorant person might have to be told, on each occasion, how to unscrew a nut (and you cannot be much more ignorant than a computer). For absolute certainty the manual would have to read:

1) Disconnect the mains supply (see line 21) to the ding box.
2) Unscrew the nut (see line 75) holding the wurble to the ding box.

where the 'see line . . . ' instructions are pointers to other places in the manual. That is, we have the name of the operation, and then the location of the instruction with that name.

The manual itself is itself an operation — repair a thing — and is composed of smaller operations, which can then be seen to be composed of smaller operations, until one reaches certain primitive operations (those which have to be left to the basic human operations, eg 'pick up a screw-driver'). Going through the manual, new operations can be explained only in terms of previous operations, and one threads one's way through the operations to arrive at the most basic level. To learn about Forth one follows on a thread, from one operation down to the basic.

The manual is not indexed by pages (in fact it has no index) and we can only refer to line numbers. Each particular operation, and its attendant instructions, extends over a number of lines.

For each operation, the first line contains information which says over how many lines the name of the operation extends, and whether the instructions are valid. The next few lines contain the name given to the operation. We call these lines (including the first line) the *name field* of that operation. The line number for the first line will be called the *address* of the name field.

After the name field, we come to a number which extends over a fixed number of lines and tells us the name field address of the immediately prior operation. To work out what operations there are in the manual we can start with the last operation in the manual, and find where the previous operation started, by using the pointer to the previous name field address. (We can call this the *link field*, it links operations together in the order they were defined). Once we have

worked out the name of the previous operation, we can use that operation's link field to work out where is the operation previous to that.

There will be operations of many differing natures: some might be mechanical, some may be purely taking readings from dials, or some might be instructions to wait for a certain period of time. After the link field, we have lines which contain the line number (address) corresponding to the *type* of operation, and where the definition of that *type* is to be found. Call this the *compilation field*, the lines which tell us how the instructions are to be put together (or compiled).

Last of all, there will be the lines containing the instructions, where in most cases the instructions will refer to the addresses of lines which contain more primitive operations. Call this the *parameter field*, because this sets the bounds upon what the operation can do.

The manual has no index, partly because the manual is open-ended — new operations are being added all the time. To find, therefore, whether there is an operation with a certain name in the manual, we start with the latest definition. Then we look at its link field, find the previous operation with its own link field, and so continue, until we exhaust all the definitions — or find the one we need.

If we listed out all the names of the operations (we could define an operation which did this, called (say) VLIST we have built up the names of the operations, as in a dictionary. The first spare line above the dictionary — where we would define a new operation — might be found by HERE (why not?) To find the address of the line number by use of the operation HERE, we would first have to find how to calculate by use of HERE — this means we would have to search through to find what the operation called HERE did. This may sound tedious, but it is the kind of activity that a moron could do well — computers are morons.

The manual will start at a certain line number (address), and there might be a manual-using operation (called +ORIGIN) which took the value of the bottom line, and added it on to the value of any number we supplied. Adding zero to +ORIGIN will tell us at what line the manual starts.

Forth has a dictionary of names and definitions (like our imaginary manual), where the names are produced by entering (ie typing in) VLIST, and each name corresponds to a portion of space in the computer's memory. The computer does not have the addresses of lines on pages, it has the addresses of locations in the computer. Different names perform different operations (having differing fun-

ctions) and most words, as they are known in Forth, can be seen to be the equivalent of functions or procedures in other languages.

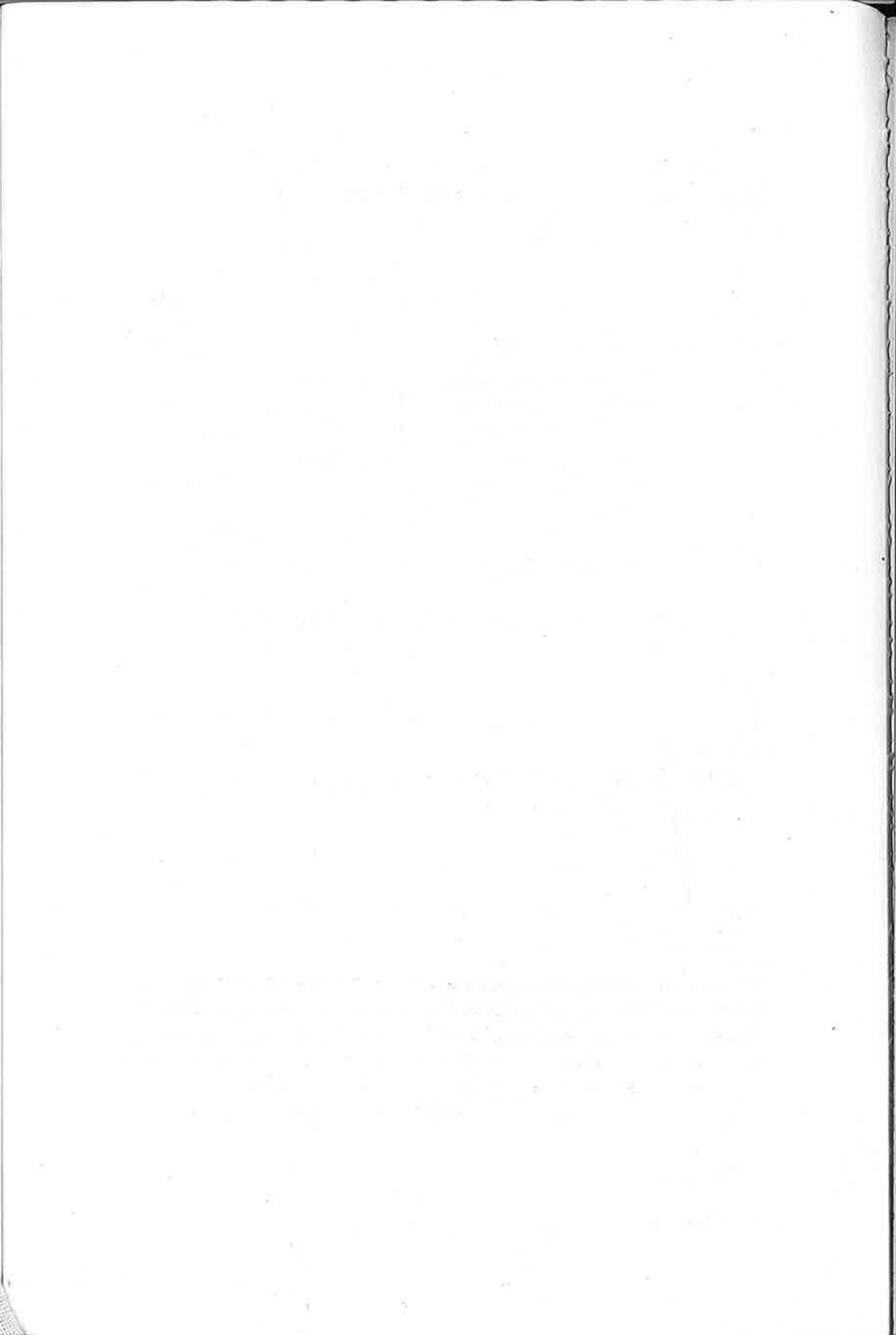
To type in VLIST is to activate a procedure which lists out the names defined in the dictionary, and it has no other effects. To type in HERE, however, is to produce a number — the number which gives the address of the first free location above the words already in the dictionary. The number produced by HERE does not appear on the screen. The number is left in a portion of the memory (a location) outside the limits of the dictionary ie it is left on the 'stack'.

The stack is best considered, at the moment, as a *series* of locations in memory, where you can leave numbers to be later used for whatever you need. To print out the number which is on top of the stack, the one most accessible, you type in the dot ".", a *word* in Forth which is easily overlooked, as it is only a dot.

Though Forth is highly regular and singleminded, any book about Forth cannot be so (as Aristotle said . . .). If every item had to be explained before it was used, then progress would be slow indeed: we have to assume that the reader is rather more flexible and intelligent than a computer and so we ask the reader to use some Forth words before there has been a chance to explain them.

I am not aiming to write a manual-there is an excellent one to go with Acornsoft Forth (by Richard DeGrandis-Harrison). Rather I am aiming to try to show a way through Forth, by looking at how it works, how it functions. Any explanations I give are not the strict definitions one expects from a manual, rather they are explanations which show how I learnt about Forth and how it functions. My choice of applications is also idiosyncratic, being less concerned with numerical applications as such; and far more concerned with its use in analysing logical implications, or in the use of graphics.

My aim is to take the mystery out of mastery.



SCREEN 1
Backwards and Forthwards

We shouldn't forget the *Goofus Bird* that builds its nest upside down and flies backward, not caring where it's going, only where it's been.
Jorge Luis Borges, *The book of imaginary beings.*

You may know where you have been, but you can only guess where you are going. Load Forth and type

VLIST

to watch the screen be filled with a large number of words, ending with

BRANCH @EXECUTE EXECUTE LIT OK

If you then enter

' LIT .

the number

3666 OK

appears. The OK appears automatically in Forth whenever an action is successfully completed, and in this case what has been completed is the parameter field address of the Forth word LIT. Already we have been faced with a strange term (the parameter field address is usually shortened to PFA), but one which we note includes 'address'.

Before we investigate what the PFA of LIT (or any other words) actually means, try

' EXECUTE .

to which the response is

3751 OK

At this point some of you may be confused because nothing seems to have happened apart from an **OK**. The missing number has not been lost, it is merely sleeping, to be woken by

.

where the dot means print out the number left on the stack from the previous operation. We will examine the stack in a while, so at the moment remember to use the dot if it is needed.

For one last use of the tick, (') enter

' @EXECUTE .

3777 OK

and then look at the three numbers we have had output.

The Forth words produced by VLIST going from the top down to **LIT OK** at the bottom, are in an order, and that order is given by their arrangement in memory. The PFA of the Forth word @EXECUTE is at 3777 which is higher than that of EXECUTE (3751), which is itself higher than that of LIT (3666). The words in the VLIST are thus ordered in some way which corresponds to their addresses. What address, where?

Introducing locations

Computers have to store their information somewhere, and there are *locations* in the computer's memory where information can be stored. Each location has its own address, just as every line in a manual has a number, and the basic location in which we store information is called a byte. The byte can hold values from 0 to 255, and this value can be interpreted in differing ways depending upon the context (more on bytes in the third screen). The parameter field address is the address of a byte in the computer's memory, and the PFA of @EXECUTE is the address of the location at which the routine associated with @EXECUTE starts.

Sometimes (most of the time) one byte is not enough to store the information we need. If we put two bytes together to form a composite, this is called a *cell* in Forth. In other languages two bytes are often called a word, but word has special meaning in Forth — all those names you see in the VLIST. Try to VLIST again, but this time press the escape-key almost instantaneously. You will now see the

top of the list. The list is called the dictionary, ordered not alphabetically but by the order in which the words were defined. The words can be seen to correspond to the instructions in a manual, with no surprises.

At the top of the dictionary, assuming we are sufficiently agile to press the ESC key before it disappears off the top of the screen, is

CREATE-SCREENS DISK TAPE TR/W TW

and onwards. Type the following

HERE . CR ' CREATE-SCREENS . CR ' DISK . CR

to which the response may be

12645
12497
12442
OK

and when I say 'may be', different versions of Acornsoft Forth may well give differing values. The differences in the exact locations are unimportant, it is the order that is important — as with everything in Forth. Now to investigate the line that has just been entered.

HERE is a word which tells us where the first available location lies above the dictionary. When we form a word of our own, the definition will start to fill up memory starting in the location to which HERE points. The dot outputs the value of the location left by use of HERE, and CR moves to a new line (it is a carriage return). The tick tells us where is the PFA of CREATE-SCREENS (and then dot and CR), and so for DISK. Note the decreasing values of the addresses, from the address at the first location above the dictionary, down to the address 3666 for the PFA of LIT. Is LIT at the very bottom? Can we go any further?

The Forth word +ORIGIN when added to a number tells us how far 'up' the system is that number of locations; that is, it gives the address of that number of locations from the start of Forth. If we go up zero locations, then this must be where Forth starts. So enter

Ø +ORIGIN .

to which the response is

3584 OK

so that the Forth system starts at address 3584. Now type in the closely associated line

Ø +ORIGIN H.

EØØ OK

which indicates that the Forth system starts at the *hexadecimal* address EØØ; and if the memory maps in the User Guide are consulted, this corresponds to the User's BASIC Program Area when not in Forth.

Press function key 9, and you will see

CALL&ØEØ2

CALL&ØEØ2 ?

which is translated to mean that function key 9 has been programmed to produce CALL&ØEØ2, and Forth does not understand (ie the query). The reason why Forth does not understand is because key 9 is not to be used within Forth. Press the BREAK key, and you will read

BBC Computer
BASIC
>

and to then press key 9 sends you back to Forth. CALL&ØEØ2 is a call within BASIC to a machine code routine which is executed at address ØEØ2 (in hexadecimal). Forth seems to be started at an address two locations above the start of the system as a whole. This is a cold start — a start from the beginning — and if you do not want to lose your newly-defined words, and for some reason have jumped out of Forth, then key 8 (CALL&ØEØ5) will perform a warm start — as little as possible is lost.

By use of the two words +ORIGIN and HERE we have thus found that the basic Forth system (in the version I am using) goes from hex (short for hexadecimal) ØEØØ to hex 3165, the latter by using

HERE H.

3165 OK

or, if so desired, (as HERE leaves the value 12645)

12645 H.

3165 OK

This shows the power of the language — there are few other languages which enable you to do so much so easily.

Adding to the dictionary

At this point, it makes sense to define a function key to perform a VLIST, a very common request. We do this (for key Ø) by

Ø KEY' VLIST M

where the space between KEY' and VLIST is very important, as are all spaces between Forth words. See if your definition has worked, by depressing key Ø.

Now we have the VLIST mechanised, we will add a word to the Forth dictionary.

Enter these three lines

HERE .

12645 OK

356 CONSTANT ONE

OK

HERE .

12655 OK

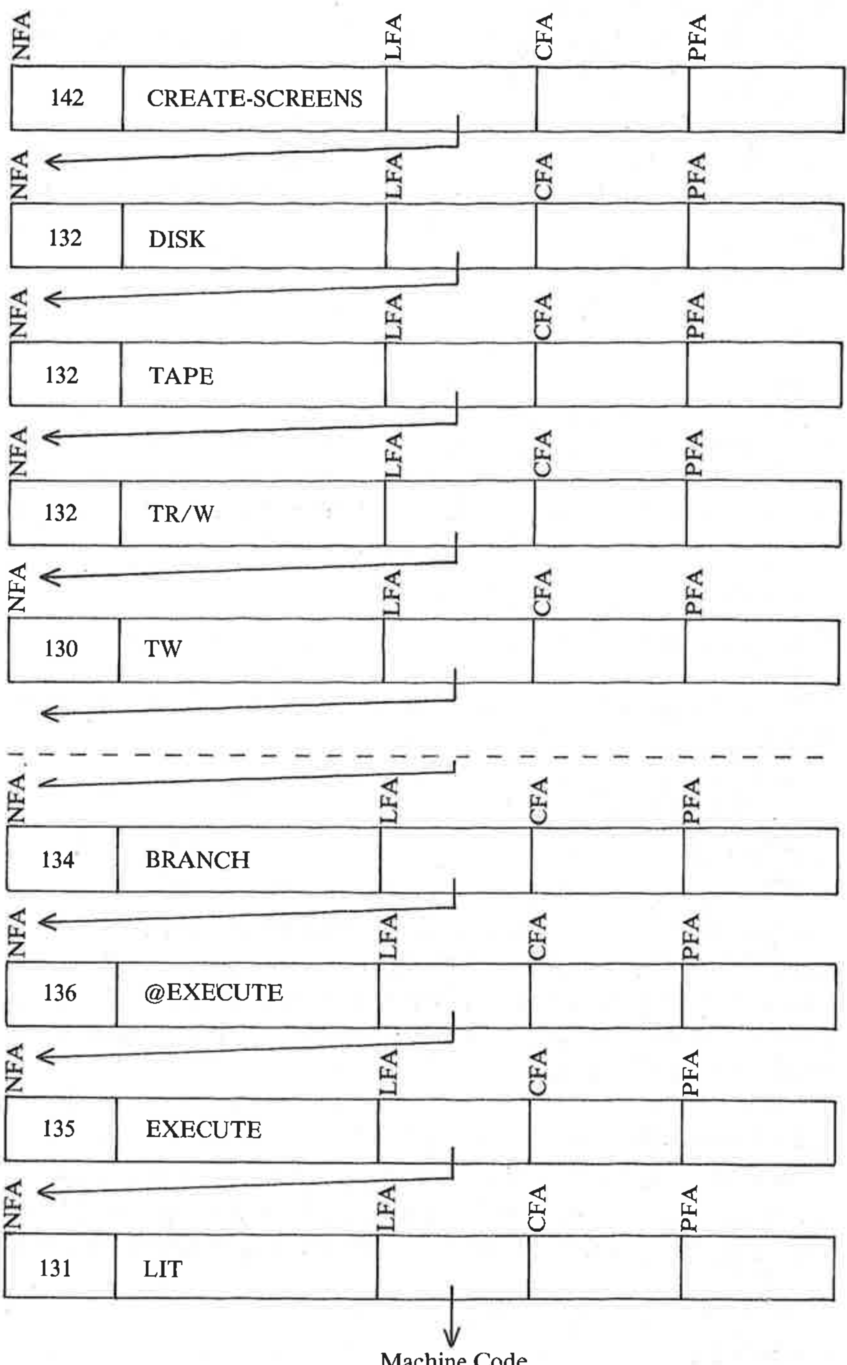
and we can see that whatever happened in our second line (356 CONSTANT ONE) moved the first location above the dictionary from 12645 to 12655. The word ONE (a constant of value 356) thus takes up ten locations in memory (ie ten bytes). What we shall now do is thread our way through these bytes, and in so doing we unravel a large part of Forth. (Different implementations, and different versions will use different numbers, but the exercise is general).

The first location we will examine is the byte at address 12645, the first byte in the portion of memory devoted to the word ONE, and so we enter

12645 C@ .
131 OK

and the first byte is a composite affair. Rather simplistically, at the moment, if 128 is subtracted from the value in the first byte then that

Threading of Words in VLIST



is the number of characters in the name of the word (other things can happen, but more of that later).

131 less 128 is 3, the number of letters in ONE: the calculation is

131 128 – .

3 OK

and so we can now move on to finding how the name is saved. But first try

38 EMIT CR

&

OK

which is how we output the character corresponding to the ASCII code 38. The word C@ obviously gives the value stored in the location specified, and so to give the values stored in the next three locations (12646, 12647, and 12648) we use

12646 C@ . 12647 C@ . 12648 C@ .

79 78 197 OK

which just produces numbers. Trying to use EMIT is rather more helpful:

12646 C@ EMIT 12647 C@ EMIT 12648 C@ EMIT CR

ONE
OK

because we can see that the next three bytes contain the name of the word ONE. The first byte plus the bytes which contain the name are called the name field, and the first byte is at the name field address (NFA). Once we have the NFA, we can (from the first byte) work out all else about the name.

The link field

The next location is at address 12649, and is not treated as a byte, but as a cell (ie two locations at once). To find the contents of a cell we use the Forth word @ (called 'fetch'), not C@ (c-fetch). We enter

12649 @ .

12478 OK

and a mystical number appears, which seems to be unrelated to anything, apart from that it is near to 12649. Now for some more mysticism:

```
12478 C@ .
```

142 OK

and if 128 is subtracted from 142 (ie 142 128 −) the result is 14. If you count the number of characters in CREATE-SCREENS, then you find that there are 14. Entering

```
12479 C@ EMIT 12480 C@ EMIT 12481 EMIT CR
```

CRE
OK

reveals all. The cell (two bytes) which follow the name field contains the link field, (which starts at the link field address, LFA) that is, a pointer to the start of the name field address of the previous word. When we use VLIST, the VLIST procedure jumps from the LF to the NFA, prints out the name, and then uses the following LF to point to the next (ie earlier) NFA, and so continues.

If we return to the word ONE, the LFA was at 12649, and two locations on from there is 12651. We enter

```
12651 @ .
```

5612 OK

and wonder what that means. If we try

```
' CONSTANT .
```

5606 OK

perhaps there is something there, concerning the definition of CONSTANT (a word we used in the definition of ONE?) and what happens is that 5612 contains machine code instructions which define what a constant does. This is the compilation field (also known in earlier versions of Forth as the code field) and it contains the compilation field address (the CFA). The CF points to earlier addresses in memory, where the machine instructions corresponding to the type of word are found.

The standard types of word are given by the following identifiers:

: (a colon definition)

CONSTANT
VARIABLE
USER
CREATE
VOCABULARY

and new types can be formed by using the CREATE word in conjunction with DOES> — but more of this later. If you want to find where each hangs out (approximately) use the tick, eg

' : .

7954 OK

and so forth. Then compare the order of the addresses with that on the VLIST.

Two locations on from 12651 is 12653. Enter

12653 @ .

to which the response is

356 OK

that is, the number we stored originally in the constant we called ONE. Try

ONE . CR ' ONE CR

356
12653
OK

and you find that to enter ONE as such gives the value of the constant, and the tick applied to ONE gives the address at which the value is stored, so that

' ONE @ .

356 OK

Parameter fields

The two locations following the compilation field contain the parameter field, and the parameter field address (PFA) is given by the use of tick (as I noted earlier). In the case of a constant, the parameter field is two bytes long, and the field contains the value of

the constant. In the case of ONE the PFA is 12653, and the cell stored from that address is fetched. Suppose we c-fetch each individual byte:

```
12653 C@ . CR 12654 C@ . CR
```

100
1
OK

and then try to relate the values in these two bytes to the value stored in the cell. It is clear that $356 = 1 \times 256 + 100$, but what is surprising is that the two bytes (at 12653 and 12654) seem to be in the wrong order. I faintly expect the more significant byte (the one multiplied by 256) to come first — but, then, the most significant byte is higher in memory.

How can we alter the value stored in the constant ONE? Obviously all we need to do is store a new value at the PFA. We know that the PFA is 12653 (in this case) but in general it would be tedious to have to search our way through a definition to find it. We use the tick, of course, and we will store the value 689 in ONE. Thus first enter

689

OK

Second, find the PFA of ONE

' ONE

OK

Third, store that value

!

OK

and fourth, to find whether it has worked

ONE .

689 OK

This sequence would normally be written

689 ' ONE !

and that is how the value stored in a constant is changed.

Enter

FORGET ONE VLIST

and you will see that ONE has disappeared from the dictionary, and printing HERE takes you back to 12645. As an exercise, enter

VARIABLE ONE 356 ONE !

and see if you can investigate how a constant and a variable differ, concentrating on the CFA and the PFA. For a start, trying

ONE . CR ' ONE CR

produces

12653
12653

which is unlike the constant we called ONE. To find the value stored by the variable we have to fetch it, that is

ONE @ .

356 OK

But to store a value we simply use

689 ONE !

OK

and if we then

12653 C@ . CR 12654 C@ . CR

177
2
OK

(note 689 = 2×256 + 177). Suppose we alter the value in the byte at address 12653 from 177 to Ø? The value stored in that cell should then appear to be 2×256, ie 512. We c-store in the byte by using

Ø 12653 C! ONE @ .

512 OK

(think about it). Now try

1ØØ ONE C! ONE @ .

612 OK

and work out what has happened.

For reference, **Figure 1.1** shows schematically how the constant ONE is stored in memory. You might find it useful to attempt a similar exercise for the following

VARIABLE BIGGER

isolating the meaning of every byte.

Figure 1.1 The Definition of "356 Constant One"

	NFA				LFA		CFA		PFA	
Address	12645	12646	12647	12648	12649	12650	12651	12652	12653	12654
Content	131	O	N	E	12478		5612		356	
Designation	Name Field				Link Field		Compilation Field		Parameter Field	

Name Field	The name of the word
Link Field	Points to the Name Field Address (NFA) of previous word, CREATE-SCREENS
Compilation Field	Points to the machine code routine corresponding to a CONSTANT
Parameter Field	The machine code routine uses the data in the Parameter Field.

SCREEN 2
Stack and Colon Steering

I'm walking backwards to Christmas,
across the Irish Sea . . .
Spike Milligan, Song (?)

Suppose you are a mouse trying to find your way through a maze. Not having a piece of string, and also being mechanical, you have to remember where you have been, so that you can retrace your steps where necessary.

You start at the entrance, and find that there are three different ways you can go. You choose one direction, remembering which of the three it was. You continue in this way until you are blocked, and there is no way forward, so turning round. You continue forward (back the way you came) until you reach the previous junction: either you have tried all the possible openings at that junction, (and carry on to the junction before that) or you try a new opening, remembering which openings you have tried. The process continues until you find your way out, or find there is no way out.

You have reached the ninth junction (J9) and remember details of all the previous junctions, that is

. . . J6 J7 J8 J9

then from the ninth junction you reach your tenth

. . . J6 J7 J8 J9 J10

From the tenth junction there are two openings. You try one of them, and find your way blocked

. . . J6 J7 J8 J9 J10 STOP

So you go back to junction 10

... J6 J7 J8 J9 J10

and try the new way, which also is blocked

... J6 J7 J8 J9 J10 STOP

As there were only two ways from junction ten, and you have used both of them, you retrace your steps back to the ninth junction

... J6 J7 J8 J9

and try the remaining opening, to reach a new junction ten (different from the last, which has been totally forgotten).

... J6 J7 J8 J9 J10

When junction ten is reached, we can reach a new junction, junction eleven

... J6 J7 J8 J9 J10 J11

The story continues until we are out.

Stacking

We have been using a *stack*. A stack is an ordered list of elements for which the last element (in this case a junction) is the first element to be examined — unlike a bus queue in which it is first come, first served. In our maze-running example the stack seemed to be the natural mode of operating, and Forth (with its concern for predictability and simplicity) finds the stack a very useful way of organising operations.

To examine the stack properly we need to use special Forth words, but to define these special Forth words we will first have to try to understand something of colon definitions (which need some understanding of the stack to be understood properly...). First, however, remember

356 CONSTANT ONE

The key word in this definition is CONSTANT, and, in effect, CONSTANT works both forwards and backwards. The word CONSTANT is an instruction to the Forth system to take the next word in line (in this case ONE) and make ONE be described by the mode of operation we call CONSTANT. Part of the mode of operation of CONSTANT is to take the number from the top of the stack (TOS), ie the last number (ie 356), and make that the value of the constant

ONE. Another part of the definition of CONSTANT says that when a constant is called by name, it leaves the value stored on the TOS. Sounds complicated?

Enter the following

SP@ . CR 356 SP@ . CR CONSTANT ONE SP@ .

which produces

88
86
88 OK

which has an interesting interpretation. SP@ (s-p-fetch) is a variable which points to the TOS, but with the slight modification that the value is given instantly, like a constant. When s-p-fetch is called, it itself leaves a number TOS, but it does not count itself. When the stack is empty the TOS is 88, which means that the empty stack starts at address location 88 (first line). Typing in the number 356 means that this number has to go somewhere, so it goes on the stack, and the TOS is now 86 (two locations further downwards. The stack descends in memory until it reaches location Ø). When we define the constant ONE, it takes its value from the number TOS, ie 356, and so the TOS then goes back to 88.

We

FORGET ONE SP@ CONSTANT SBASE VLIST

and discover that we have a new word SBASE in our dictionary. Typing

SBASE .

gives us the illuminating answer

88 OK

so that we now have a constant which is set to the base location of the stack. If SP@ gives a result other than 88 (the value of SBASE), there must be at least one number on the stack. Put the number 3 on the stack

3

OK

and then fetch the number held at the address 86, and print it out by dot

86 @ .

3 OK

So we have shown that the first item (the lowest item) on the stack is at address SBASE less 2. Try

56 SBASE 2– @ .

56 OK

This needs some examination.

First we put 56, and that goes into the cell (two bytes) at address 86:

Address 86 84 82 80 ....
Value 56

and then we put the value of the constant SBASE in the cell at address 84

Address 86 84 82 80 ...
Value 56 88

The word (not number) 2– means subtract 2 from the value TOS, and so

Address 86 84 82 80 ....
Value 56 86

The fetch word (@) takes the value TOS (ie 86) and replaces it with the value stored at that location

Address 86 84 82 80 ....
Value 56 56

The dot takes the value at the TOS, and prints it out (that value disappears).

Address 86 84 82 80 ....
Value 56

We are now where we were when we started, and a further dot will print out the remaining value on the stack (yet another 56).

A test definition

If we

HERE .

12657 OK

we find that the constant SBASE takes up two more bytes than the constant ONE. Why? This is an important address to remember. Type in, very carefully,

: TEST ;

a colon definition which does nothing, and then

12657 C@ .

132 OK

which indicates that the definition has four characters (132 128 –). VLIST to make sure it is there, and then FORGET TEST. Now, just as carefully, enter this incorrect colon definition

: TEST escape-key

(ie without the concluding semi-colon). The system will log the escape

Escape
OK

and a VLIST will show that TEST is in the dictionary. However, when we

TEST
TEST ?

Forth cannot find TEST, so

12657 C@ .
164 OK

and not 132 as before. The extra 32 has come from the fifth bit in the first byte being set to 1, to flag an illegitimate definition (more on bits later). Forth does not recognise TEST as a valid word — it isn't. Try to

FORGET TEST

TEST ? MSG #24

What on earth is TEST? Error message number 24 means declare vocabulary, ie is it somewhere else?

```
SMUDGE 12657 C@ .
132 OK
```

We have changed the 'smudge-bit' back to zero and we can safely FORGET TEST. All this is to prepare you for possible problems if you make a mistake with colon definitions.

The first word is an unsophisticated means of printing out the contents of the stack, together with the associated address for each value, starting at address SBASE 2− (ie 86). The word (called STNUM) contains an example of a loop in Forth, but first the definition:

```
:
STNUM SP@ SBASE 2- DO I . I @ . CR -2 +LOOP ;
```

STNUM consists of a loop from SBASE minus 2 to the current TOS (s-p- fetch). The loop counter (I) is loaded and the content of the location is fetched and printed. The loop counter is then decremented by −2, and if the limit has not been reached, then the loop repeats.

We try

```
1 2 3 STNUM

86  1
84  2
82  3
OK
```

and see how those three values are stored, with 3 being TOS. If there is nothing on the stack — eg press the escape-key — we find

```
STNUM

86 86
OK
```

and I leave you to find out why (remember the loop always goes through at least once). Type in

```
1 2 3 DUP STNUM

86  1
84  2
82  3
80  3
```

OK

which shows very clearly that DUP takes the TOS (3) and makes another copy on top of that value,

ROT STNUM

86 1
84 3
82 3
80 2

whereas we can see that ROT takes the third element from the top and places it TOS, moving the other two downwards.

There are many stack manipulation words, eg

DUP DROP SWAP OVER ROT PICK ROLL ?DUP

whose exact meaning is not always immediately clear, but whose meaning can be made visible by use of STNUM. For example, enter

86 84 82 80 78 76 74 72 STNUM

and you produce

86 86
84 84
82 82

and so on down to

72 72
OK

which provides a very helpful way of trying out the words.

Another interesting word is DEPTH — how many items there are on the stack, eg

DEPTH .

8 OK

(count 'em up).

Clear the stack when you have finished by using the escape-key, and then

DEPTH .

0 OK

then try

DEPTH DEPTH . .

1 0 OK

or

DEPTH DEPTH DEPTH STNUM
86 0
84 1
82 2
OK

and then

2 STNUM

which after 82, gives

80 2

that is, the value 2 in the top two locations.
If we now enter

= STNUM

86 0
84 1
82 1

What has happened is that the top two values have been compared for equality, both have been removed and a flag (1 for true) replaces the lower of the two. If we use

0= STNUM

we find that address 82 now contains 0, as 1 does not equal 0. It is well worth playing around with the other comparison words, that include

< => 0< 0= 0> NOT

and in my improved version of displaying the stack, the word STACK, I use a comparison.

Improved stacking

STACK lists out the contents of the stack without giving their addresses, horizontally not vertically (to save space on the screen), and it also says explicitly if the stack is empty. It is still short

: STACK DEPTH O= IF ." EMPTY " CR ELSE SP@ SBASE 2- DO I @ . -2 +LOOP CR THEN ;

but perhaps not completely transparent.

The new word STACK is given a colon definition, and the first thing to happen is that we find the stack depth, and compare that to zero. If it is true that the depth is equal to zero (ie there is a 1 left TOS) we dot-quote (print) EMPTY, ending at the quote, and CR; else we do a loop from SBASE less 2 to the TOS (using SP@). For each value of the counter we fetch the value stored at that location, and print it — the counter decreases in units of 2. Then we end. Trying

1 2 3 4 STACK

produces

1 2 3 4
OK

Whereas, when the stack has been cleared, to enter

STACK

EMPTY
OK

shows that the word gives the correct answer.

If we now enter

11 STACK

11
OK

and, then, to type in

HEX STACK

B
OK

shows things can change. What has happened? Well, try

DECIMAL STACK

11
OK

and perhaps it becomes clearer. The value of eleven in decimal is shown as 11, and in hexadecimal it is B. So we can progress further by entering

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 STACK HEX STACK DECIMAL

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11
OK

and can see that 10 in hex is written as A. The number TOS is 17 (in decimal) and so to print out that value we can use the dot, or we can use the h-dot

H.

11 OK

which (although we are in decimal mode) prints out the value TOS in hex.

Changing modes

It is useful to be able to move between both decimal and hex modes (eg we can relate to memory maps where values are given in hex). Are any other modes possible? The answer is yes. The way in which we change from one mode (ie numerical base) to another is by use of the special variable BASE. BASE points to a location in which is stored the present numerical base. BASE is a special type of variable called a USER variable, and it is special in that the location in which the value is stored is not alongside the rest of the word in memory. When we created the variable ONE, to enter

‘ ONE . ONE .

produced

12653 12653 OK

because the PFA of ONE (given by tick) was also where the value was stored (ie address 12653). Enter

‘ BASE . BASE . BASE C@ .

5999 1Ø62 1Ø OK

and 5999 does not equal 1Ø62.

The parameter field of BASE starts at 5999, but this is not where the value is stored. The value is stored at location 1062 (using hex Ø426). The section of memory from hex Ø4ØØ is the language ROM workspace in the User Manual, and when there is a cold start in Forth all the variables in that area are initialised to the correct starting values (eg BASE always points to the value 1Ø). User variables mean that initialisation and suchlike are easier, and more compact to mechanise.

Acorn usually prefixes a hexadecimal number by &, and so it seems reasonable to try to make h-dot do the same. We can do this either by inventing a new routine, or using h-dot in some way. Try the latter first, and start by

' H. .

9421 OK

and then we define a new version of H.

: H. ." &" H. ;

H. MSG #4 OK

which tells us that we have already used the name H. before, for a word, otherwise it is OK. We know this, of course, because we used H. in the body of the procedure which defined H.. But first

' H. .

12652 OK

or

' H. H.
&316C OK

and the position of H. has moved up the memory. A VLIST will show there are two examples of H., one at the top, and one where it always was, just before DEC..

When we were defining H. (above), which H. did the system understand when it came to it in the body of the declaration? Did it get confused? (Forth, confused? Never.)

When the Forth system comes to the H. in the definition, it searches through for that name (by use of the link fields and name fields) but when it comes to the latest H., it does not recognise it as a word. Whilst a word is being defined, the NFA is 'smudged' so that it

is not used by the system: it is only unsmudged when the definition has been successful. This is why, when the definition of TEST was incorrect, we could not FORGET it, and had to use the word SMUDGE.

If we now use H. then the one used will be the latest definition. But if we had used H. in a definition before the new version, the old version would be used, because the pointers would all be to that old version. The new version is not that terrific, however, because

```
-1 H.
```

&-1 OK

and so we

```
FORGET H.
```

and only lose the new version.

```
: H. DUP 0< IF ." -&" ELSE ." &" THEN ABS H. ;
```

and then

```
-1 H.
```

-&1 OK

which seems far better. Note that the new definition is rather more complex, and — as with many colon definitions — the stack is explicitly used a great deal. Let's try to work that out.

We need a number to be on the stack before H. can be used, otherwise we get an error message number 1 – nowt ont' stack. Suppose we have a –1, we duplicate this value, and so have two –1 in the top two locations. The –1 (TOS) is examined to see if it is minus, it is, so it is replaced by 1 (which means true). As the TOS is true then –& is printed (otherwise it would be &). Then, when that is over, the absolute value of –1 replaces it TOS. Finally, that positive value is output in hex.

For my last trick this screen, I will define a word to print numbers out in binary, and it will be called BIN.,

```
: BIN. BASE C@ 2 BASE C! SWAP . BASE C! ;
```

and it takes the current base (c-fetches), leaving it TOS (the number to be printed is now second on stack). The number 2 is then c-stored in the location to which BASE points; the top two elements are swapped (the number to be printed is now TOS); the number TOS is

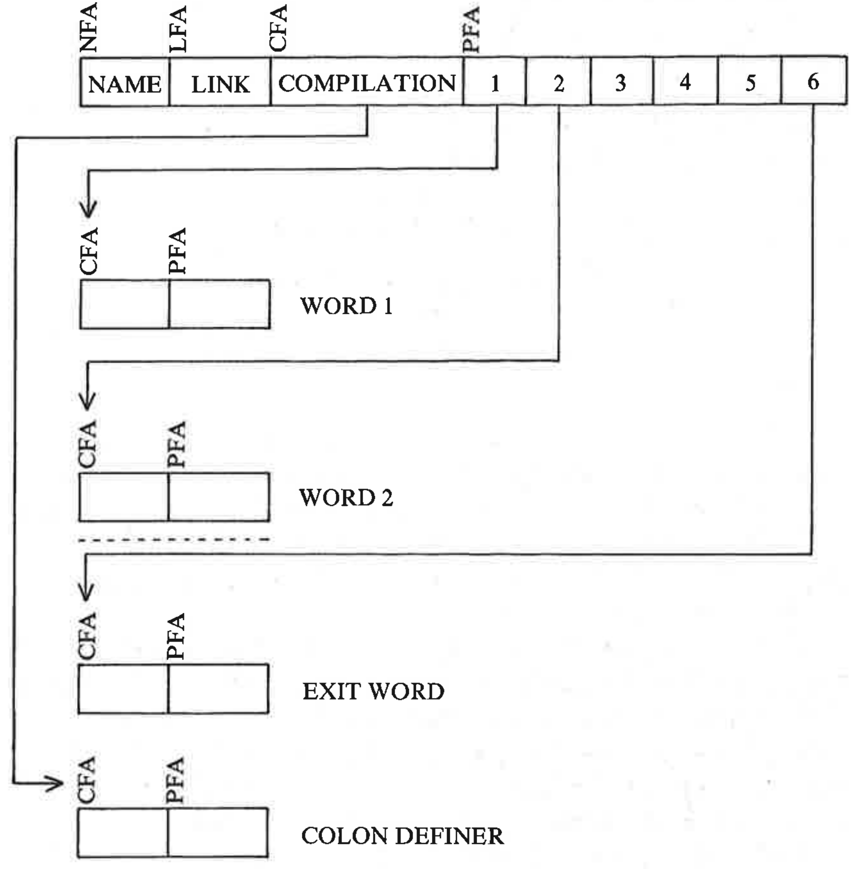
printed (in BASE 2); and then the number now TOS (ie the original base) is c-stored in the location to which BASE points.

Plain dot can provide some surprises, eg

```
32768 .
-32768 OK
```

and it is perfectly correct.

The Form of Colon Definitions



Notes

a) The operation of the Link Field is not shown

b) The final pointer in any colon definition is *always* to EXIT

c) EXIT may also be used to end a colon definition (eg in a conditional expression).

SCREEN 3
Circular Sums

Progress might be a circle, rather than a straight line.
Eberhard Zeidler, *Contemporary Architects*.

The time is one o'clock, and five hours ago it was eight o'clock. In normal arithmetic, $1 - 5 = -4$, whereas for the twelve hour clock, $1 - 5 = 8$. A computer measures numbers by a form of clock arithmetic, and clock arithmetic gives answers which are not expected — unless you expect them.

The world, and most of its machines, follow clock arithmetic. To turn through 450 degrees is to turn to 90 degrees from one's original direction; 366 days on from now is usually tomorrow's date, one year on; on a car milometer, one mile on from 999 999 is 000 000 (as 1 000 000 is too large to fit); and if I fly round the world in a straight line, I will end up where I started (OK, only approximately).

A computer is only a machine, and it can only repeat itself. The basic element of the computer — and one which can only repeat itself — is the byte (noted earlier). Larger machines might have basic elements which are different to bytes, but we will ignore them. The byte takes values from 0 to 255: $3 + 4 = 7$, but for $255 + 1$ the result is 0, and the result of $1 - 5$ is 252 (clock arithmetic to a base of 256, as there are 256 numbers from 0 to 255).

We can easily investigate some of these operations, and first we define a variable

```
VARIABLE CELL
```

which we know points to a value which is contained in two bytes (ie a cell). On page 17, to the end of that screen, I showed how we can access each of the two bytes separately, and, based on that exami-

nation, we can define several words which will allow us to examine the contents of both bytes, independently. The first two words allow us to point to each of the two locations separately,

```
: BYTELOW CELL ;
: BYTEHIGH CELL 1+ ;
```

that is, BYTELOW points to the byte at the lower of the two locations, and BYTEHIGH points to the higher location (1+ further from the lower address). There is a further word (and more of that later)

```
VARIABLE BYTE
```

We will now store a number in CELL, a number which occupies both bytes,

```
258 CELL ! BYTELOW C@ . BYTEHIGH C@ .
```

2 1 OK

and, as $1 \times 256 + 2 = 258$, then we can see that the low byte contains the less significant part of the number stored in the cell (we have already found this out in Screen 1, but it is worth remembering). Now, I want to use both BYTELOW and BYTEHIGH for similar purposes, and thus I want to design some way of swapping them for each other. The question then becomes, how do they differ and how do they agree?

Execution vectors

Both are variables, and thus have a similar structure in memory (the only real difference is in the length of the name and, thus, the name field). The way in which they differ is shown simply by

```
' BYTELOW . ' BYTEHIGH .
```

12668 12685 OK

(your numbers may differ), in that the PFA is different. The PFA is a value, and any value can be stored in a variable, but to store the PFA is not always useful for types other than constants or variables. What we need to store is the CFA (compilation field address) which tells us, then, where it is possible to EXECUTE the word. Try this

```
' VLIST CFA EXECUTE
```

and you find that you are VLISTing. To omit the CFA in the above line is a good way of getting the system to hang around doing nothing. Save the situation by pressing BREAK and then, when back to BASIC, press f8. The word CFA assumes that the PFA is on the stack (it has been given by ' VLIST in this case) and it replaces the PFA with the compilation field address. The word EXECUTE assumes that a CFA is on the stack, and executes that routine (omitting CFA means it tries vainly to execute from the PFA).

Suppose, therefore,

```
' BYTELOW CFA BYTE !
```

which means that the compilation field address is now stored in the variable BYTE. When we then try to

```
BYTE @ EXECUTE C@ .
```

2 OK

we find that the value contained in the lower byte is output. To then try

```
' BYTEHIGH CFA BYTE ! BYTE @ EXECUTE C@ .
```

1 OK

shows that by changing the CFA in BYTE allows us to swap between the two locations without altering other words. The use of *execution vectors*, which is what the use of BYTE entails, for this specific example, is shown schematically in **Figure 3.1**.

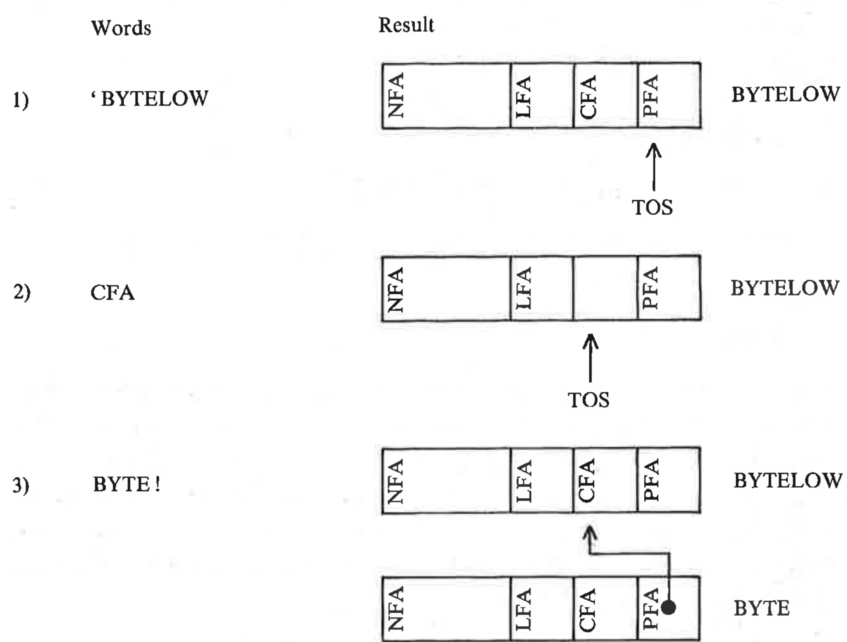
Form some words:

```
: EVEC CFA BYTE ! ;
: BLOC BYTE @ EXECUTE ;
: BSTORE BLOC C! ;
: BFETCH BLOC C@ ;
```

where EVEC needs a PFA on the stack upon which to operate; BLOC executes the routine to which the CFA stored by BYTE is pointing; BSTORE then stores the value on the stack in the location to which BYTE points ultimately; and BFETCH fetches the value of the byte stored at the appropriate location. Here is a sequence which can, if you wish, be placed all on one line:

```
' BYTELOW EVEC
67 BSTORE
BFETCH .
```

Figure 3.1 An Example of Execution Vectors



Notes

a) This places the parameter field address of BYTELOW on the top of stack.
b) This takes the PFA from the TOS and replaces it with the corresponding compilation field address.
c) This stores the value TOS, in the parameter field of the word BYTE. To execute the value fetched from the parameter field of BYTE (by @ EXECUTE) is to execute BYTELOW.

67 OK

BYTELOW C@ .

67 OK

which shows that we are now using the byte at BYTELOW.
Now try

CELL @ .

323 OK

(obviously $1 \times 256 + 67 = 323$), and then

CELL @ H. CR BYTEHIGH C@ H. BYTELOW C@ H.

143
1 43 OK

(note that the new improved version of H. is not being used, as I had lost it by switching off). The first value, hex 143, is that contained in the cell to which CELL points, and the two values on the next line, hex 1 and hex 43, are those contained in each of the two separate bytes. As is easy to see, when we use hexadecimal notation we know, without further calculation, what is contained in each byte when the value in a cell is examined.

Negative numbers

Enter

– 1 BSTORE BFETCH .

255 OK

and so the number we placed in the byte (–1) is not the number we retrieve (255). Why? Look at **Figure 3.2**: it is a special kind of location which can only take the values from Ø to 7 — it is displayed in clock form, modulo 8. To add numbers we count in a clockwise direction, and so it would seem reasonable to count in a counter-clockwise direction to subtract. For example, 2 less 5 is 5, ie $2 - 5 = 5$; and $0 - 5 = 3$ (or $-5 = 3$, and $3 + 5 = 0$). As -5 is also 3, then $2 - 5 = 2 + 3 = 5$. To subtract we can add, as long as we add the complement of the negative number. The complement of the number (for this example) is equal to 8 minus the size of the negative

number. For a byte (arithmetic to modulo 256) the number we fetch is always in the complement form, eg −1 is equivalent to 256 − 1.

We need to know, however, when an answer is meant to be negative, as all values we use in the machine are positive. In **Figure 3.2** there are eight separate values, and to allow positive and negative values we have to split them in the middle, between 3 and 4 (see **Figure 3.3**). And in this case we can see that 3 + 1 = −4 — something which does not happen in ordinary arithmetic.

Put the number 65535 on the stack, put the number 1 on the stack, add them together, and then print out the result.

```
65535 1 + .
Ø OK
```

Cells in Forth can be seen to take values from 0 to 65535 (65536 different values = 256 × 256) and thus adding 1 to 65535 sends the clock round to zero. For further elucidation try

```
65535 DUP . U.
-1 65535 OK
```

which takes the value 65535, places it on top of the stack, duplicates the value, dot prints the value TOS, and then u-dots the value now TOS. The value 65535 is printed out as −1, as we might expect, but the u-dot prints out the value as an unsigned number (the positive value stored in that cell).

Further elucidations:

```
32767 1 + DUP . U.

-32768 32768 OK
```

which explains the puzzle at the end of the last screen. Just as 3 + 1 = −4 for our modulo 8 example, so 32767 + 1 = −32768 for our modulo 65536 cell. We have stored −1 in the lower byte, and we will store −1 also in the higher byte:

```
' BYTEHIGH EVEC -1 BSTORE BFETCH .

255 OK
```

We now have −1 stored in two separate bytes, both of which claim to have stored 255. What happens, then, when we put the two bytes together (as CELL) and output that value?

```
CELL @ .

-1 OK
```

Figure 3.2 Modulo 8 Arithmetic

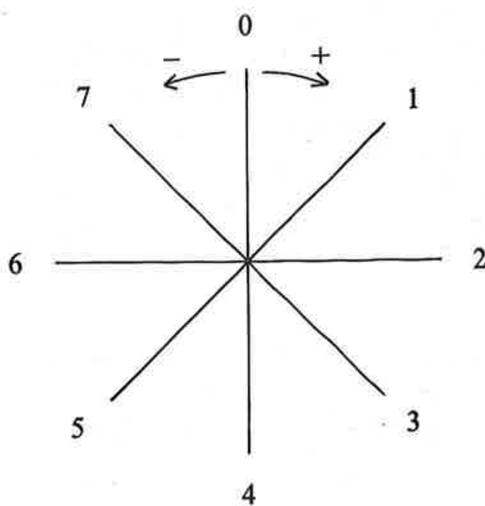
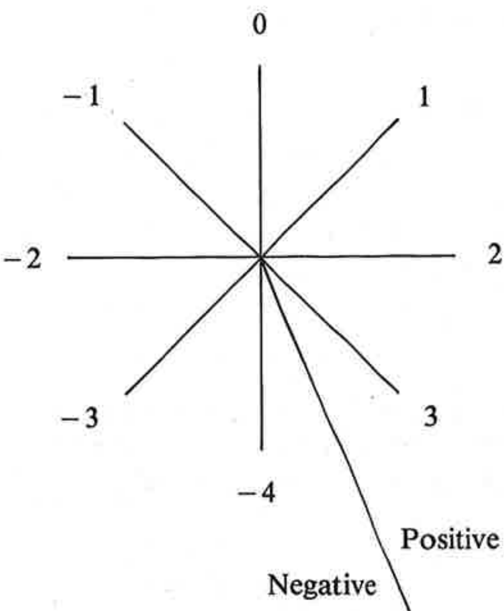


Figure 3.3 Complement Arithmetic — Modulo 8



Confusing isn't it? Remember what I said about using hexadecimal to examine the separate bytes in cell? So,

```
HEX CELL @ U. DECIMAL
FFFF OK
```

so that -1 (or 65535) printed as an unsigned number is FFFF in hexadecimal. Check that each byte has a value FF by

```
255 H.

FF OK
```

How does the computer tell (when using the dot, say) that a number is negative? **Figure 3.4** reveals all: when we are to treat a number as negative, there is 1 in the most significant bit of the corresponding binary number. To check this we will need a method of printing out unsigned binary numbers, B.,

```
: B. BASE C@ 2 BASE C! SWAP U. BASE C! ;
```

This word takes the existing base, and leaves TOS, 2 is placed TOS and c-stored in BASE, after swapping, the unsigned number is printed, and then the original base is retrieved from the stack and c-stored in BASE, again. Here we go:

```
32767 DUP B. 1 + B.

111111111111111 1000000000000000 OK
```

and if the 1s are counted up, we can see that there are 15 of them in the first number. 32767 is thus fifteen bits all set to 1: 32768 is sixteen bits, the first 1 and the other fifteen are zero. (Compare the binary values in **Figure 3.4**).

Now

```
255 B. CR 65535 B.

11111111
1111111111111111
```

and if we remember that 65535 is -1, and that 255 was stored as -1, and see that -1 in **Figure 3.4** is 111, we can realise that when all bits are true (or 1) then the result is -1. This is why for many systems true is represented as -1, because then all the individual bits are true. A cell, therefore, has 16 bits and a byte has eight bits — the first (most significant, leftmost) bit is frequently called the sign bit. The way of

storing numbers in this manner is called two's-complement arithmetic. A worthwhile exercise is to try sums such as

```
30000 6000 + . 6000 30000 - .
-29536 -24000 OK
```

and try to predict the results.

Manipulating numbers

Here is an interesting little pair of words,

```
: DBL DUP + DUP DUP DUP . U. B. ;
: DBLUP BEGIN DBL ?TAB UNTIL ;
```

the first of which duplicates the number TOS, adds the two numbers together, and by use of three DUPs makes three copies of that value. These three values are then output by dot, u-dot, and b-dot. The second word gives us our first encounter with indefinite loops; the loop begins by performing DBL, then tests to see if the TAB key has been pressed (and leaves 1 TOS if it has, 0 otherwise). If there is a zero TOS when the UNTIL is reached then control returns to BEGIN, otherwise the loop is left.

If we use DBLUP, remembering to stop by use of TAB

```
1 DBLUP

2 2 10
4 4 100
8 8 1000
16 16 10000
32 32 100000
```

and onwards to

```
8192 8192 10000000000000
16384 16384 100000000000000
-32768 32768 1000000000000000
0 0 0
```

Figure 3.4 Modulo 8 Arithmetic

Normal	Complement	Base 2
0	0	000
1	1	001
2	2	010
3	3	011
4	−4	100
5	−3	101
6	−2	110
7	−1	111

Figure 3.5 Addition Table, Complement Arithmetic Modulo 8

+	0	1	2	3	−4	−3	−2	−1
0	0	1	2	3	−4	−3	−2	−1
1	1	2	3	**−4**	−3	−2	−1	0
2	2	3	**−4**	**−3**	−2	−1	0	1
3	3	**−4**	**−3**	**−2**	−1	0	1	2
−4	−4	−3	−2	−1	**0**	**1**	**2**	**3**
−3	−3	−2	−1	0	**1**	**2**	**3**	−4
−2	−2	−1	0	1	**2**	**3**	−4	−3
−1	−1	0	1	2	**3**	−4	−3	−2

So we see that twice 32768 (ie 65536) is considered to be Ø by Forth. The first column of figures gives the two's-complement form of the value, the second column gives the raw (ie unsigned) value in decimal, and the third column gives the raw value in binary. It is very clear that to double is merely to add a zero to the binary number; and, when the binary number becomes 16 bits long, then to add a zero means that it becomes 17 bits long. It is not possible to store 17 bits in a cell, only the 16 least significant (ie rightmost, of smaller place value) are stored. To store 1 followed by 16 zero bits, is to store the 16 zero bits, and appear to store zero.

Incidentally the word DBLUP could be defined in a slightly different, but equally effective way:

```
: DBLUP BEGIN ?TAB NOT WHILE DBL REPEAT ;
```

which says begin to do this loop while it is not true that the TAB has been pressed. A final form could be

```
: DBLUP BEGIN DBL AGAIN ;
```

which simply loops round and round between BEGIN and AGAIN, until stopped by BREAK or ESCAPE.

How does the Forth system (or most other systems) work out how to add or how to subtract? Have a look at **Figure 3.5**. This is an addition table for the modulo 8 example, using complement arithmetic. All inadmissable (ie wrong) additions are underlined, and we can see that if the sign of the result is different to the sign of the two values being added (when they are both of the same sign), then it is wrong. Notice that the values being added in **Figure 3.5** are all in themselves admissable, we do not try to add −5 to 4 (both of which numbers are inadmissable).

We will now try to implement this for the addition of bytes, so first enter

```
COLD
```

to start Forth from the beginning. Our first task is to isolate three bytes (two values to be added and the result). We do this by

```
VARIABLE BYTES
```

which reserves two bytes (the cell which contains the value of that variable) and we need to extend the space available by one byte (and protect that byte from over-writing). Now enter

```
HERE . 1 ALLOT HERE .
```

12657 12658 OK

and find that the top of the dictionary is at 12657. By using the word ALLOT (which takes the value 1 off the TOS) we find that the top of the dictionary has moved up one byte to 12658. The word ALLOT sets aside an extra 1 byte for our use (1ØØ ALLOT would set aside 1ØØ bytes).

```
: 1B BYTES ; : 2B BYTES 1+ ; : 3B BYTES 2+ ;
VARIABLE PRESENT
```

Our three bytes are at the addresses 1B, 2B, and 3B, and we will use an execution vector called PRESENT to simplify our routines.

```
: CHOSEN CFA PRESENT ! ;
: LOCATION PRESENT @ EXECUTE ;
: STORE LOCATION C! ;
: FETCH LOCATION C@ ;
```

These are house-keeping words, they are the basic words we will use to store and fetch values. To store a word in a byte, we need to take a number off the TOS — to check whether the value is within bounds we can then compare the value we fetch back, with the value we stored.

Byte addition

How do we turn a value stored in a byte, into its complement form? If the value stored in the byte is less than 128 then the value is as it stands. When the value is 128 or greater then the number is negative and has to be complemented. If we divide the number by 128 then, if the result is 1 (there are no fractions in Forth, only whole numbers), this means that the number is greater than 127, and is supposed to be negative. So to two's complement

```
: 2COMP FETCH DUP 128 / IF 256 – THEN ;
```

that is, fetch whichever byte is currently being examined and put it TOS, duplicate that TOS, put 128 TOS and divide it into the lower value. If there is a truth flag TOS (ie a 1) 256 is subtracted from the lower value (the top value disappears when examined by IF), and the complemented value is left TOS.

We now have to compare the value as stored in the byte, with the value we thought we stored

: VALID? DUP STORE 2COMP = NOT IF ." INVALID" CR THEN ;

The number TOS is duplicated, stored and fetched back, and the original value is then checked with the number fetched back for equality (it is turned into a two's-complement form by 2COMP). The result of the test is reversed by NOT, and if the number TOS is true then "INVALID" is printed. Whether the number(s) are valid or not, they are added together:

: ADDITION ' 1B CHOSEN FETCH ' 2B CHOSEN FETCH + ' 3B
CHOSEN STORE ;

and this word shows how execution vectors can come into their own. First 1B is chosen, and its content fetched; then 2B is chosen, and its content fetched; the two numbers, TOS and next, are then added together; and, finally, 3B is chosen, and the result is stored in that location.

We now check to see if the addition is valid

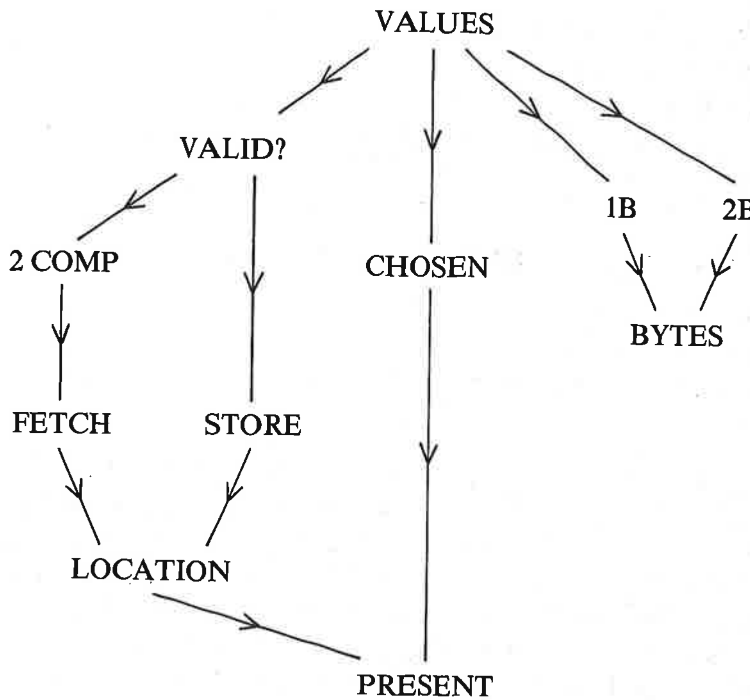
: NEGSIGN? 128 / ;
: SIGNCHECK ' 1B CHOSEN FETCH NEGSIGN? ' 2B CHOSEN
FETCH NEGSIGN? = ;
: 1&3SIGN ' 1B CHOSEN FETCH NEGSIGN? ' 3B CHOSEN FETCH
NEGSIGN? = ;
: RESULTCHECK ." RESULT " SIGNCHECK IF 1&3SIGN NOT IF
." INVALID" THEN THEN ;

and the words are interpreted as follows:

NEGSIGN? leaves a 1 if the number TOS is 128 or greater (thus it is negative in two's-complement); SIGNCHECK checks to see if the signs of 1B and 2B are the same; 1&3SIGN checks to see if the signs of 1B and 3B are the same; and these are all brought together by RESULT-CHECK. After printing RESULT a signcheck is made, and if it is true a check is then made to see if the signs of 1B and 3B are the same — if it is not then the result is invalid.

The final word does it all, using these earlier words,

Figure 3.6 Threading "Values"



Only words defined in this application are shown, standard Forth words are omitted.

```
: ARITHMETIC VALUES ADDITION RESULTCHECK ' 3B
CHOSEN 2COMP ." = " . CR ;
```

and so to find what is the result of adding 127 and 127 in two's complement, modulo 256, we enter

```
127 127 ARITHMETIC
FIRST
SECOND
RESULT INVALID = -2
OK
```

(fun?) and perhaps

```
128 127 ARITHMETIC
FIRST
SECOND INVALID
RESULT = -1
OK
```

where the result is also invalid, but it is not flagged as so (but we know an input was invalid). Try experimenting with different values, to see if you can crack the word, that is, have two valid inputs and an invalid output. You should not be able to so do, unless it is flagged.

If you look back through the words we used to create ARITHMETIC, it is easy to see the threading in practice. Take the word VALUES, to see how that is built from the bottom upwards, as is shown in **Figure 3.6**. Within its definition VALUES only refers to the words 1B, 2B, CHOSEN, and VALID? (apart from standard Forth words such as ' or CR). 1B and 2B both refer to BYTES in their definitions, CHOSEN refers to PRESENT, VALID? to 2COMP and to STORE, which in turn refer to . . . (see **Figure 3.6**). Finally, explain why the word FACTOR does what it does.

```
: FACTOR 1 10 1 DO I . I * DUP . CR LOOP ; FACTOR
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 -25216
9 -30336
```

Numerical Progressions

UNSIGNED	SIGNED	BINARY
65534	– 2	1111 1111 1111 1110
65535	– 1	1111 1111 1111 1111
0	0	0000 0000 0000 0000
1	1	0000 0000 0000 0001
2	2	0000 0000 0000 0010
⋮	⋮	⋮
32765	32765	0111 1111 1111 1101
32766	32766	0111 1111 1111 1110
32767	32767	0111 1111 1111 1111
32768	– 32768	1000 0000 0000 0000
32769	– 32767	1000 0000 0000 0001
32770	– 32765	1000 0000 0000 0010
⋮	⋮	⋮
65533	– 3	1111 1111 1111 1101
65534	– 2	1111 1111 1111 1110
65535	– 1	1111 1111 1111 1111
0	0	0000 0000 0000 0000

SCREEN 4
Editorial Freedom

Then, rising with Aurora's light,
The Muse invoked, sit down to write;
Blot out, correct, insert, refine,
Enlarge, diminish, interline.
Jonathan Swift, *On Poetry*.

When you were entering the definitions of the words for the ADDITION example, no doubt you kept making errors, and found that you had to SMUDGE and then FORGET. This can be extremely tedious — extremely is to put it mildly — and it is made worse by the simple fact that, when a word defined a while ago is giving incorrect results, sometimes we are unable to check to see if we had typed in what we thought we had typed (I am forever forgetting to put the dot in definitions).

This might be made easier, if there was a facility to, perhaps,

```
LISTDEF ADDITION ( THIS IS NOT LEGAL )
```

where the word LISTDEF, when applied to a word, listed out the contents of the definition of the word. This might be seen to be the equivalent of listing a program in other languages — there is at least one variety of Forth (that for the Jupiter Ace) for which this is possible. The question is: why is such a facility not usually available?

What happens when we type in the definition of the word

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

is that the Forth system works its way through the definition, building up pointers (as we have seen) and then, once the word has been stored away at the top of the dictionary, the actual line we have entered is forgotten. All that remains is the *compiled* version in the directory. Compiled means that the instructions are saved as

machine instructions which do not require any further translation when they are executed by use of the word

DEMO

though we have to remember that DEMO expects a number TOS, which is either 1 or Ø.

The actual series of words in the definition of DEMO corresponds to what, in some other languages, might be called the *source* definition of the word. One reason why Forth is so speedy is because it is a *compiled* language (as, for example, is Pascal) and does not save the language in source form. Many other interactive languages (eg BASIC or COMAL) only use the source form, and do not compile the program.

Immediate execution

Though to compile the language directly, as does Forth, produces a speedily running application, there needs to be a slightly greater understanding of what happens, than for some other languages. Consider the word DEMO. Now, normally, as words are used in a definition, so pointers to the compilation addresses of these words are stored in the parameter field of the word being defined (compare the threading for VALUES, **Figure 3.6**). This will work, seems sensible, for the words ." and CR in the definition of DEMO but what is to happen about IF, ELSE, and THEN? All these three words are intimately interrelated in that, when IF is executed, depending on the number TOS, whether the portion after IF or the portion after ELSE is used, depends upon the state of the stack.

When DEMO (or any other word) is compiled, IF has to be executed immediately, because it has to calculate how far forward it has to go to reach the ELSE or THEN markers in the parameter field. In other words, when IF is encountered in the definition, immediately, it compiles a special routine (the same one, normally, as is used by UNTIL and WHILE), which, if the flag TOS is false, branches to another location in the parameter field. For IF, the branch is forward, as is that for WHILE; whereas that for UNTIL is backward (to BEGIN). When ELSE and THEN are encountered, other routines are immediately compiled to work out other branches in the parameter field. You will find more information on IF ELSE THEN, and the conditional and unconditional loops in Screen 5.

IMMEDIATE words can be very useful, and one of the most useful applications comes with Forth vocabularies. One of the most useful Forth vocabularies (and the most commonly used after the main Forth vocabulary) is the EDITOR. The reason why I have given the strange name screen to my chapters is because screens are the way in which source definitions are saved in Forth. If we save the definitions as source then, obviously, we must save them by use of files; and that is why I will start by examining screens as if they are nothing more than ordinary files. The screens are not what might normally be termed ordinary files, because they correspond to what is termed in the User Guide a section of memory (eg page 392).

Loading files

To load a section of memory, normally, one uses *LOAD, but to enter *LOAD in Forth only ends with a confused system — what is it? Right, with your cassette in the player, or your disk in the drive, try

```
OS' CAT
```

which is a command to the Operating System to CATalogue. Note that the asterisk (which precedes operating system commands as a rule, page 416 of the User Guide) is not necessary. The result of cataloguing will depend upon at which point the cassette had been stopped, or whether disks were attached, but the result should follow the familiar form.

Position the tape in the cassette just before the file ØØØ3, and then

```
OS' LOAD"ØØØ3"
```

to watch the first of the files (ie screens) of the Editor being loaded. When the file has been loaded, then

```
Ø LIST
```

and screen #Ø is then listed (it should be blank). To follow this by

```
1 LIST
```

may either be accompanied by the information that the operating system is Searching, or screen #1 is listed, or nothing might happen. If the screen is not listed then press the escape-key (possibly

jumping into BASIC, to be returned by use of key 8), and try again to list screen 1.

If you then manage to list screen 1, it should read

```
SCR #1  1 H
Ø ( TAPE EDITOR )
1 FORTH DEFINITIONS HEX
2 VOCABULARY EDITOR IMMEDIATE
```

down to

```
15 - - >
```

or something similar. This is a screen. Though the number of the screen is given as 1 (and 1 hex), really it is screen #3; but we have fooled Forth into thinking it is screen #1 (by loading the screen ourselves, and not by the use of Forth — which has to be given the correct screen number). We will come back to the contents of this screen in a while.

Now try

```
PREV @ H.

4FF8 OK
```

(and this may differ depending upon implementation), and then

```
PREV @ 2+ C@ EMIT CR

(
OK
```

shows that the screen itself starts two locations on from the value of a variable PREV: the line PREV @ etc finds a value (a location) adds 2 onto that value, fetches the value of the byte from that location; and then prints out the character corresponding to that value (the ASCII code). The whole screen may be listed by

```
: SCREENLIST PREV @ 3 + B/BUF + PREV @ 2+
DO I C@ EMIT
LOOP ;
```

so then

```
SCREENLIST
```

will output (in less compressed form than the usual LIST) the contents of the screen ie that portion of memory which is 1024 (B/BUF) bytes long.

So this is what happens when there is a screen loaded: the file is loaded into memory, and Forth recognises that portion of memory as being occupied by a screen; and when a screen is saved, all that is saved is a portion of memory.

Let us do it properly. Rewind the cassette to the correct position (if necessary), and then enter

3 LOAD

which starts to load the Editor. The first file to be loaded is ØØØ3, and the final line (as we have seen) is

15 – – >

which is an instruction to the filing system to then load the next screen (in numerical order). Each screen, until the final screen 10 (A in hex), then causes the next screen to be loaded.

Redefined words

Part way through the Searching and Loadings you will notice

ØØØ6 Ø3 Ø4ØØ
I MSG #4 Searching

which means that in screen 6, the word I has been re-defined. We have already met I, as the loop counter, and we used I in the word SCREENLIST. If we use SCREENLIST now, we have a listing of the last screen to be loaded (ie screen 10 or A in hex), and if you would like a rather better view try

1Ø LIST

and then enter

: FLOOP 1Ø Ø DO I . LOOP ;

to

FLOOP

Ø 1 2 3 4 5 6 7 8 9 OK

is as you would expect. If you now VLIST then you will notice that there are new words in the dictionary, some of which you were unaware, such as WHERE, LINE, and EDITOR. Type

EDITOR VLIST

and more words will appear before your eyes — words like C, TILL, and all those to FLOOP.

: ELOOP 1Ø Ø DO EDITOR IMMEDIATE I . LOOP ;

and then

ELOOP

Ø 1536 ELOOP ? MSG # 23

where the message number 23 means 'off the current editing screen'. Forth must consider, therefore, the I of ELOOP as different to the I of FLOOP. The difference comes from the addition of EDITOR IMMEDIATE to ELOOP. When the definition is being built up in the parameter field of ELOOP, it comes to EDITOR and just sets up a pointer to the compilation address of EDITOR. However, when it encounters the word IMMEDIATE, it executes that last word to which there is a pointer — it executes EDITOR.

EDITOR instructs the system to change to the vocabulary (set of words) we have defined as being called EDITOR. When the system reaches I, it first searches through the EDITOR vocabulary, and there it finds a definition of I (a definition which differs from the original definition, as loop counter). The standard vocabulary is called Forth.

Studying a screen

Now is the time to return to the listing of the first screen for the editor (see page 52). Line Ø of the screen shows how we can insert comments in the source to help understand what is happening. A comment is opened by ((known as paren) followed by a space, as it is a word. The text of the comment then follows, to be terminated by a close-paren). Though there is no need to leave a space between the text and close-paren, one normally does for neatness. The next line (line 1) sets the vocabulary to the standard Forth vocabulary, and sets the arithmetic base to hex.

The next line (2) defines a new vocabulary, called EDITOR and acts on that definition immediately. All the other definitions in this screen are in the Forth vocabulary, which can be checked by FORTH VLIST, and then EDITOR VLIST. The words defined in the screen are still in the Forth vocabulary, because the definition of EDITOR has not yet been activated by EDITOR DEFINITIONS (something which comes in screen #5, that is, #LOCATE onwards).

Examine lines 10 to 12

```
10 : PROGRAM CR ." 1st screen number? "
11 QUERY INTERPRET SCR ! CLRSCR
12 [COMPILE] EDITOR ;
```

remembering that if you are in the default 7 MODE then the square brackets look like little arrows. These lines define a word PROGRAM which asks you to input the screen number, then uses QUERY to input the number, which is then INTERPRETed to leave a number TOS. The number TOS is then stored in the variable SCR (which holds the current screen number), and the screen in memory is then cleared (CLRSCR). Finally, the system is left with the current vocabulary being the EDITOR: the word [COMPILE] informs the system that the next word (in this case EDITOR) — though a word which normally executes immediately — is not to be executed, it is to be compiled as if it were an ordinary word. Rather than setting the vocabulary to EDITOR there and then, whilst the word PROGRAM is being compiled, EDITOR is only activated when the word PROGRAM is executed.

We did not need to tell Forth that EDITOR was IMMEDIATE when we defined ELOOP — it already knew. We needed only to have defined

```
: ELOOP 10 0 DO EDITOR I . LOOP ;
```

and, as you can check, the result is the same.

If you now

```
7 LIST
```

and look at line 9 within that screen (different screens, different lines, for other versions) you see

9 SAME @ IF FORTH I EDITOR

as part of a loop which starts at line 8 and extends to line 10. When that screen is loaded the vocabulary is the EDITOR, and so unless

otherwise instructed — when I is encountered it will be used in its redefined version, and in this case it is needed in its original form. FORTH I EDITOR changes the vocabulary to Forth, sets up the pointer to the compilation address of I, and then reverts to the Editor vocabulary.

Indexing screens

If using cassette, reposition, if using disk be thankful, and then enter

3 1Ø INDEX

and slowly (or quickly, depending on your storage medium) you will have printed out the first line for each screen in the range from 3 until 1Ø. (Remember that the first line is line Ø.) The first lines are conventionally only ever comments, and that is why the first line is picked out by the word INDEX. If you examine the first line for screen #6, it reads

Ø (MORE LINE EDITOR)

and if we

6 LIST

then at line 4 we find the definition of I for the EDITOR vocabulary

4 : I DUP S R ;

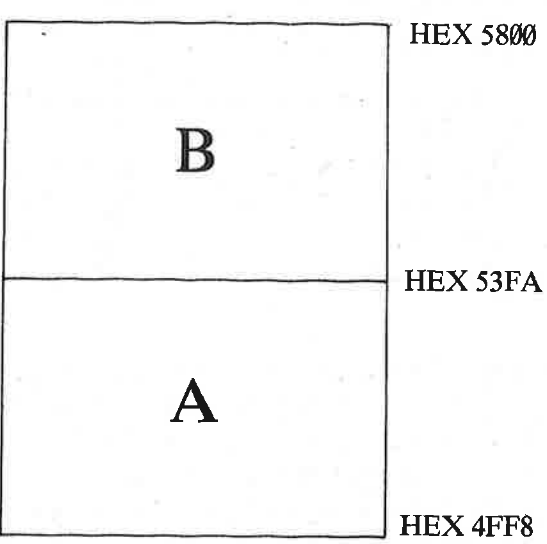
The meaning of I is that the text following I is stored in a portion of memory called the PAD, and I takes the number TOS and replaces that line on the present screen with the information from PAD. All the lines that extend from the line at which the insertion is made to the end, are moved down one line.

If you enter

PAD .

1472 OK

Mass Storage In Forth



Notes

a) PREV indicates where the previous screen was loaded, either hex 4FF8 or hex 53FA.

b) When a screen has been loaded (eg in the block A), the *next* screen is loaded into the other block (in this case block B).

c) The penultimate screen is not lost, so — for example — when screen A of the Editor has been loaded, it is still possible to list screen 9.

it can be seen that the PAD is a constant which starts at location 1472, and this is where the information is stored (the length is 64 bytes in all). To put information in the PAD, try

Ø TEXT (ANOTHER LINE)

which puts 0 TOS (so that the line is finished if wished by the ASCII character with code Ø), and the portion after TEXT is placed in the PAD locations. As well as being terminated by ASCII Ø, the text is terminated by a return or, if too long, the first 64 characters are counted as significant. (Note that if at any time any Editor commands are unrecognised, just type EDITOR to bring it back.)

1 I

inserts the text from PAD at line 1, moving all other lines one down (and losing the last line). The definition of I, for example, is now at line 5, and not line 4.

The definition of I uses two other Editor words, S (insert an extra line of spaces at the line, whose number is TOS) and R (replace the contents of the line, whose number is TOS, with the contents of the text held at PAD). If you study screen #5 (the previous screen) there are many new words which at first might seem rather confusing: it is remarkable how simple most of them are. It will be well worth your effort to go through the Editor screens, to work out how they do what they do — it might help to learn more advanced techniques.

Forming a screen

Figure 4.1 lists the standard fig-Forth words for Line Editing, and to show how we can use them to save an application we will store the definitions for STACKNUM and STACK (from my Screen 2). First of all we enter

PROGRAM

and then (turning back to page , where the definition of PROGRAM is given) we expect to be asked

1st screen number?

and thus we have to decide on the number we will give to the screen. As they will be definitions from my Screen 2, I suggest they be given the screen number hex 20 (or decimal 32): so enter 32. The screen

Figure 4.1 fig-FORTH Line Editing Commands

COMMAND	ACTION
H	Holds contents of line at PAD. Line still remains in screen.
E	Erases the line, that is fills the line with blanks.
S	Inserts a blank line (spread lines).
D	Deletes line, holds contents of line at PAD, and moves all lines up by one. (Line 15 becomes blank.)
M	Moves cursor by number of characters given by preceding value. (Really a string edit function, but defined as part of Line Editor.)
T	Types contents of line, and also holds in PAD. Text remains in screen.
L	Lists the screen.
R	Replaces contents of line with text held at PAD.
P	Puts text onto line.
I	Inserts a line, with text held at PAD.

number is stored in the special variable SCR (i.e. SCR !), and the portion of memory put aside for that screen is filled with blanks (i.e. CLRSCR). The Editor vocabulary is then activated. 32 has been entered and an empty screen 32 (20 hex) has been listed on the screen.

We can try

```
Ø P ( STACK ANALYSIS WORDS )
```

and then enter

```
L
```

to watch the screen being listed again, but with a new line Ø which says (STACK ANALYSIS WORDS). Press the escape-key, then

```
L
L ?
```

shows that we lose the Editor vocabulary when there is some error,

```
EDITOR L
```

shows how we get back to listing.

```
2 P SP@ CONSTANT SBASE
```

OK

```
L
```

will list and show that we now have a new line 2. The definition of the word P is (Editor screen #6)

```
: P 1 TEXT R ;
```

so what P does is take the text which follows P, puts that text in the PAD, and replaces the line whose number was TOS with the contents of the text stored in the PAD. Now try

```
3 R L
```

and you will find that line 3 is the same as line 2 (as the content of the PAD was the text which followed 2 P in the line above). To rid ourselves of line 3, we

```
3 P
```

and it goes.

Now to build up the definition of STNUM,

```
4 P : STNUM FORTH
```

OK

```
5 P   SP@ SBASE 2-
```

OK

```
6 P   DO I . I @ . CR  -2  +LOOP
```

OK

```
7 P   EDITOR ;
```

OK

where line 4 sets the vocabulary to Forth, line 5 sets the parameters for the loop, line 6 is the loop, and line 7 returns control to the Editor vocabulary. We have to change vocabularies to accommodate the differing definitions of I. If at any time you make an error, at the moment the simplest thing to do is to redo the line as a whole. The definition of STACK follows a similar pattern, as can be seen from the listing of screen #2Ø H.

The astute observer will notice that on lines 1, 3, 8, there is a null comment, ie ( ), instead of a blank line. The reason I have done this is

simple: sometimes (and sometimes not, without any rhyme or reason) compilation stops at a blank line — ignoring the rest of the screen — so to be on the safe side I have de-blanked all lines. It is quite easy to do:

```
Ø TEXT ( )
1 R 3 R 8 R
```

and to see if our definitions work — as they should — enter, after pressing the escape-key,

```
32 LOAD
```

and the response should be an **OK**. To VLIST will show that STACK, STNUM, and SBASE, are there. The escape-key had to be pressed before the screen was loaded because the stack had to be cleared, for SBASE to store the correct value.

Saving the screen

The next stage is to save the screen as a file on cassette or disk, but first enter an operating system command

```
OS' OPT 1,2
```

which "gives detailed information including load and execution addresses" (User Manual, page 398). The next stage is to

```
SAVE
```

which then produces a listing of screen 32 and a query **OK?:** pressing the Y key gives

```
OK? RECORD then RETURN
ØØ2Ø Ø3 Ø4ØØ ØØØØ53FE ØØØØ53FE
```

which means that file ØØ2Ø is Ø4ØØ bytes long, and the start and execution address is 53FE. (*OPT 1,2 has other uses, see later.) Type

```
ANOTHER
```

and a blank screen #33 (21 H) will be listed. If we had entered MORE we would have both saved and set up another screen, ie

```
: MORE SAVE ANOTHER ; ( SCR #3 of Editor )
```

which shows how easy it is to extend the set of words to produce personalised input and output for the tape system or the disk system.

To alter the items on a screen, there are several other words (shown in **Figure 4.2**), and the only way it is possible to learn to use these words, is by using them. Practise on dummy screens, or list screens from the Editor and try to modify them. It might be worth trying to set up the application ARITHMETIC by use of screens. As the application will extend over several (or at least two) screens, at the end of all but the last screen you will put – – > (as you will remember from the examination of the Editor screens, this means carry on to the next screen). You will also find PROGRAM and MORE useful words: start with screen #48, that is 3Ø in hex.

Finally, enter

```
OS' OPT 1,2
OS' CAT
```

and catalogue the Editor files. You will find

```
ØØØ3  Ø3  Ø4ØØ  ØØØØ53FE  ØØØØ53FE
ØØØ4  Ø3  Ø4ØØ  ØØØØ4FFA  ØØØØ4FFA
ØØØ5  Ø3  Ø4ØØ  ØØØØ53FE  ØØØØ53FE
```

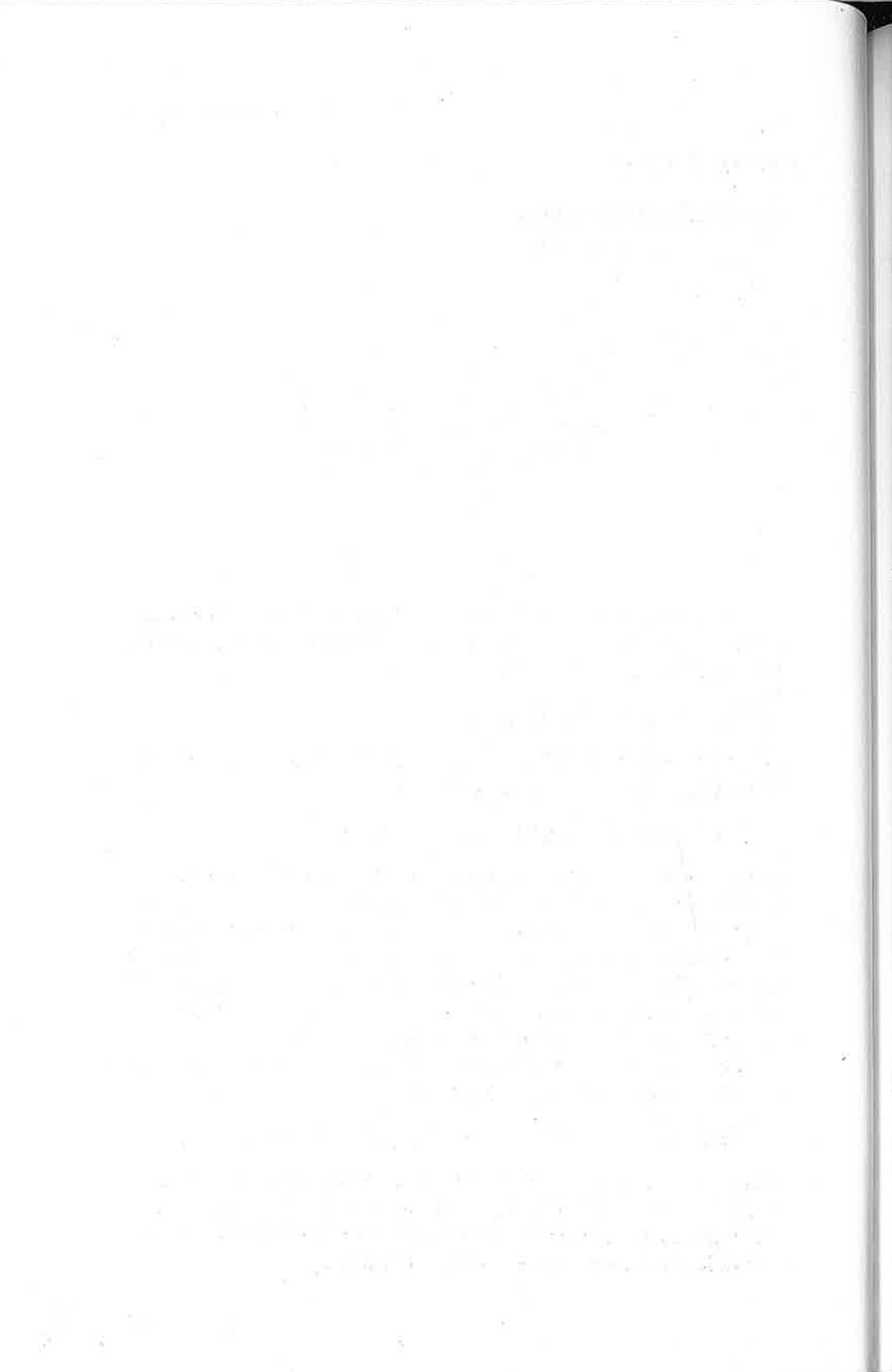
and so on. You can see the reason for our results near the beginning of this screen (page 52 and following). Editor screens are loaded into two patches of memory, from hex 53FE or hex 4FFA, and one is able to interchange the contents.

Learning about the Editor system is far more than mere editing: it is a study of how Forth is organised, on your own computer.

Figure 4.2 String Editing Commands

COMMAND	ACTION
DELETE	Deletes the given number of characters, backwards from current cursor position.
N	Finds the next occurrence of the text held at PAD.
F	Finds the first occurrence of following text.
B	Moves the cursor back by the number of spaces given by the preceding number.
X	Finds and deletes the first occurrence of the given text.
TILL	Deletes all text from the current cursor position to the end of the text given. (One line only)
C	Inserts text at current cursor position.

```
HEX 20 LIST

SCR £ 32        20 H
  0 ( STACK ANALYSIS WORDS )
  1 ( )
  2 SP@ CONSTANT SBASE
  3 ( )
  4 : STNUM FORTH
  5     SP@ SBASE 2-
  6     DO I . I @ . CR -2 +LOOP
  7     EDITOR ;
  8 ( )
  9 : STACK FORTH
 10     DEPTH 0= IF ." EMPTY" CR
 11     ELSE SP@ SBASE 2-
 12     DO I @ . -2 +LOOP CR
 13     THEN EDITOR ;
 14
 15
OK
H
OK
HEX 50 LIST
```



SCREEN 5
Complete Control

It is not enough to take steps which may
some day lead to a goal; each step must
be itself a goal and a step likewise.
Goethe, *Conversations* (ed J P Ecker-
mann)

Already we have learnt to control what happens by use of loops and conditionals, and in my last screen we spent some time looking at the loop index I. Enter this definition

```
: TRY 5 Ø DO I . CR LOOP ;
```

and then execute TRY. A list of numbers, from Ø to 4 is printed out. What happens when we modify TRY, by

```
: TRY 5 Ø DO I . RP@ 1+ @ . CR LOOP ;
```

(clever use of the cursor keys and COPY makes the modification easier). Two columns of numbers from Ø to 4 are output on the screen: the action of I, and the action of RP@ 1+ @, would seem to be identical. RP@ gives an address which is one byte less than the address of the cell atop the **Return Stack**, and the loop counter is stored atop the return stack. (After all, it had to go somewhere.)

What is further down (higher in address) the return stack? One cell (two bytes) on from the address of the TORS (top of return stack) is the address RP@ but 3 greater. We try

```
: TRY 5 Ø DO I . RP@ 1 + @ . RP@ 3 + @. CR LOOP ;
```

and find that it gives three columns, the two as before, plus a column of 5s. After the loop index comes the loop limit, and when the loop index equals the loop limit the loop ends. (Sensible?) In some implementations the loop limit is called I', defined by

```
: I' RP@ 3 + @ ;
```

and note that, though we entered 1 + as one word, 3 + is two distinct words (: 1 + 1 + ;). Incidentally if you have not been FORGETting a VLIST will show numerous TRYs.

We can now extend our adventures to two loops

```
: TRY 3 Ø DO 12 1Ø DO I . J . RP@ 5 + @ . CR LOOP LOOP ;
OK

TRY

1Ø Ø Ø
11 Ø Ø
1Ø 1 1
11 1 1
1Ø 2 2
11 2 2
OK
```

and perhaps this needs some explanation. The first column gives the loop index (I) for the inner loop (from 1Ø to 12); the second column gives the loop index for the loop which encompasses the inner loop (J); and the third column shows that the outer loop index (ie J) is stored four bytes on from the TORS (ie two cells, corresponding to I and I').

For slightly more clarification

```
: TRY 3 Ø DO 12 1Ø DO I . J . RP@ 5 + @ . CR LOOP I . CR
LOOP ;
```

and after each inner loop (two lines) we have an extra line with one value, the value of the outer loop index (ie Ø, then 1, then 2). The I between the two LOOPs is different to the I in the inner loop. In the second instance (between the LOOPs) it thinks that the loop from Ø to 3 is the inner loop. It might be useful to have a word which can be used to an arbitrary depth of nesting —

```
: IND 4 * 1+ RP@ + @ ;
```

where Ø IND gives the index of the innermost loop, and, for example, 2 IND gives the index for 2 loops out from the inner.

Filling the stack

What else can be done with the return stack?

: LITTER-BUG 14 Ø DO I . I 2– >R LOOP ;

and then to

LITTER-BUG

Ø –1 –2 –3 –4 –5 –6 –7 –8 –9 –10 –11 –12

and it continues until you stop it. The Forth word >R takes the number TOS and puts it TORS, and in this case it takes a number 2 less than the loop index (I) which is at that moment TORS. The number TORS is thus 2 less than the number underneath it. When LOOP is reached, the number TORS is incremented by 1 and checked to see if it is equal to the number next on stack (the old value of the loop index). It isn't and so the loop continues. The loop counter (TORS) gets 1 less each time.

This means that the return stack gets filled up with litter, and if the execution is not stopped in time, the system may crash (then BREAK, and key 9). In any application the use of >R has to be balanced by R> , because what the good programmer giveth (by >R) the good programmer taketh away (by R>). For example,

: KEEP-COMPUTER-TIDY 2Ø Ø DO I . R> LOOP ;

so that when we enter

KEEP-COMPUTER-TIDY

Ø 21 8194

and a flashing cursor, no OK, — a dead system. To use >R or R directly (that is, outside a colon definition) is another good way of crashing the system and the return stack is far more susceptible to being treated strangely because it coincides with the 6502 stack.

We can use the return stack for temporary storage, so that we can more easily access lower items on the computation stack, then replacing these higher items. For example,

: TOP 3 >R >R >R. R> R> R> CR;

to then

1 2 3 4 5 TOP3 TOP3

2
1
OK

so the top three elements on the computation stack (3, 4, 5, with 5 being TOS) are removed and the element underneath is then printed (first the 2, and then 1). As the return stack is longer than the computation stack, it is a good place for temporary storage, as long as the values are not confused with indexes for loops.

Many of the loops we have used earlier have ended with a +LOOP, with a value before, so now try a definition

```
: SOOPER-LOOP 16 1 DO I. 3 +LOOP;
```

which produces numbers from 1 to 13 in steps of 3. The word

```
: SOOPERER-LOOPERER 17 1 DO I. I +LOOP;
```

produces

1 2 4 8 16 OK

and this is one way of producing powers of two up to a fixed limit. As loops make use of the return stack, sometimes it is simpler to use other forms of control so that the return stack is freer of confusions.

Branching out

The other forms of control are users of the same basic routines as IF (ØBRANCH used by UNTIL and WHILE) and ELSE (BRANCH used by AGAIN and REPEAT). To try to come to terms with the structure of IF ... ELSE ... THEN we can alter the names of the words to make them rather more explanatory:

```
: IFTRUE [COMPILE] IF ; IMMEDIATE
: IFNOT [COMPILE] ELSE ; IMMEDIATE
: IFEND [COMPILE] THEN ; IMMEDIATE
```

so we can then define

```
: TRIAL IFTRUE ."T" CR IFNOT ."F" CR IFEND ;
1 TRIAL Ø TRIAL
```

T
F
OK

and the similarity to the word DEMO page 49 is clear (both are doing the same thing).

Examine the three definitions: all follow a common course, the word [COMPILE] preceding an immediate word (IF, ELSE, and THEN). If the word [COMPILE] is not there, when the word IF, say, is encountered, it executes the routine ØBRANCH and then on reaching the ending semicolon there is an error — because there was no ELSE or THEN. [COMPILE] stops the instant execution of IF: it instructs Forth to only compile the correct pointers for IF, and not to execute. IFTRUE thus becomes another way of referring to IF, IFNOT is ELSE, and IFEND is THEN.

If **Figure 5.1** is examined then it can be seen that all the indefinite loops are variants of IF ELSE THEN, where IF is, as it were, part way into the loop. IF THEN corresponds to BEGIN UNTIL, and IF ELSE THEN corresponds to BEGIN WHILE REPEAT. BEGIN AGAIN is a rule unto itself. These control structures can produce tidier applications than definite loops, and jumps out of loops.

The best way to illustrate the utility of indefinite loops is to design an application. One of my favourite examples is HILO, which is a very old number guessing game. You try to guess a number, and are only told whether you should go higher or lower. Effectively, you BEGIN and make a guess, and WHILE not true, you increase the tally of tries, and then REPEAT the process.

Make a dummy definition

```
: HILO INITIALIZE BEING INPUTSEG WHILE COMPARE
REPEAT RESULT ;
```

where INITIALIZE will be a procedure to set up the unknown number, and set the tries count to zero. INPUTSEG will be a procedure to input the user's guess, and test to see if the guess agrees with the number; WHILE they are not equal, COMPARE makes a comparison, to see if the user should guess higher or lower, then all (except INITIALIZE) is REPEATed; and finally the RESULT will be produced.

Randomization

One item we will definitely need will be a random number generator, and so we will make that our first little task. Consider the function R(I) where R(I) is the I'th random number formed by the sequence

$$R(I) = K * R(I-1) + \; - \; (MOD\ 99)$$

and we will use a starting value 1 (ie R(Ø) = 1), a constant of 13 (ie K = 13), to produce

1 14 84 4 53 96 61 2 27

a series of fairly random-looking numbers. This method of producing pseudo-random numbers is simple to implement, which seems a good reason to use it: so what do we need?

We need a value K, some value to be used for the modulo, and some starting value (ie R(Ø)). Start with the modulo: I chose 16383 for this purpose, partly because it is then easier to keep the numbers within bounds, and not to have to worry about complement arithmetic — I do not want to have to worry about negative random numbers. The constant I will choose to be 8195, because it is 3 more than half 16384, and sequences of the type generated by such constants have been found to be the most robust, in general. This leaves the starting value: either the user could choose it or it could be randomly chosen by the computer; so why not both?

The successive values of the random number will be stored in a variable called SEED, and so for the user to start the sequence all that is needed is to store a number in SEED. To randomly start supposes a random number, which brings us back to where we started. Where on the BBC Computer can we find something which is continually changing, and whose value is essentially unpredictable? Why not the internal clock? How do we get to use the internal clock? We ask SHEILA (User Manual, page 437). If we look at locations FE64 and FE65 (hex) we find that they contain timer counters, so if we fetch the cell at those two locations then we have current time value — essentially unpredictable.

Screen # 5ØH gives the construction of the random number generator, and we will start with RANDOMIZE. Before any of the words are defined we set the arithmetic to HEX, which is so much simpler to check. RANDOMIZE fetches the value from location FE64, and, if an added touch of randomness if wanted, we could use >< to follow @; >< swaps the bytes around. Try

1234 ><

−11772 OK

and you can see that swapping the bytes is a good way of increasing unpredictability, if nothing else. Try

1235 ><

– 11516 OK

and note that 11772 – 11516 is 256 which is 256 times the difference between 1235 and 1234. However, it is not really necessary.

Line 5 begins to manipulate the number we have fetched from hex FE64. That number is TOS and then hex 4ØØØ (ie 16384) is placed TOS; MOD takes the number TOS and calculates the value of the number one beneath, in modulo arithmetic to the base given by the number TOS (ie hex 4ØØØ). The number resulting from this operation is then checked to see if it is less than zero, and if negative it has hex 4ØØØ added. This number is then stored in the locations to which SEED points. At this point it might be worth noticing how the easy use of hex simplifies the numbers for this case.

RANDOMIZE, therefore, produces an initial random number, but we still have to create a series of random numbers. In the equation above, we have R(Ø) and so we have the rest to do. This is what RANDOM does (line 7), and until we get to U* all is familiar, hex 2ØØ3 and the value stored by SEED are atop the stack. Now try

18ØØØ 2 * . 18ØØØ 2 U* D.

– 29536 36ØØØ OK

to investigate what is the meaning of U*. The unsigned multiply (U*) is used when we are multiplying two ordinary numbers together, but we wish 36ØØØ to be treated as 36ØØØ and not as its complement form – 29536. The two numbers stored in the top two cells of the stack are multiplied together, and the result is left as a double length number (which occupies two cells, ie four bytes).

Double numbers

As double length numbers are different to single length numbers, and Forth has no sense of initiative, there have to be different words to print out and so on. The word D. (d-dot) is an instruction to print out the top two cells of the stack, as if it were a double length number. Enter

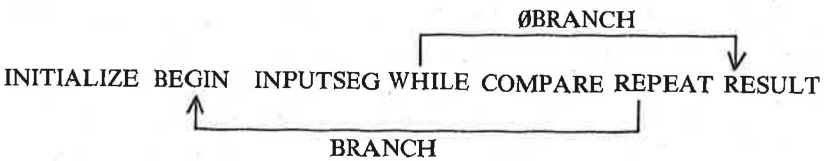
12. 2DUP D. . .

12 Ø 12 OK

which enters a double length number 12. , where the trailing dot indicates to Forth that this is a double length number (it is not a decimal point). The double length number is then duplicated (by 2DUP), and

Control Structures for HILO

a) HILO — Forth Version



b) HILO — Flow Diagram

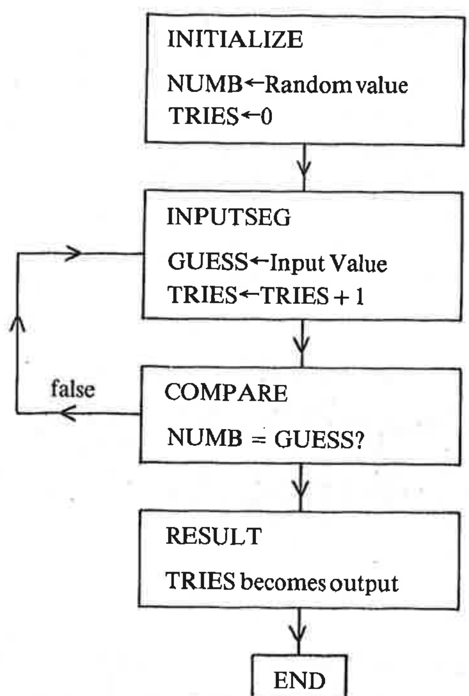
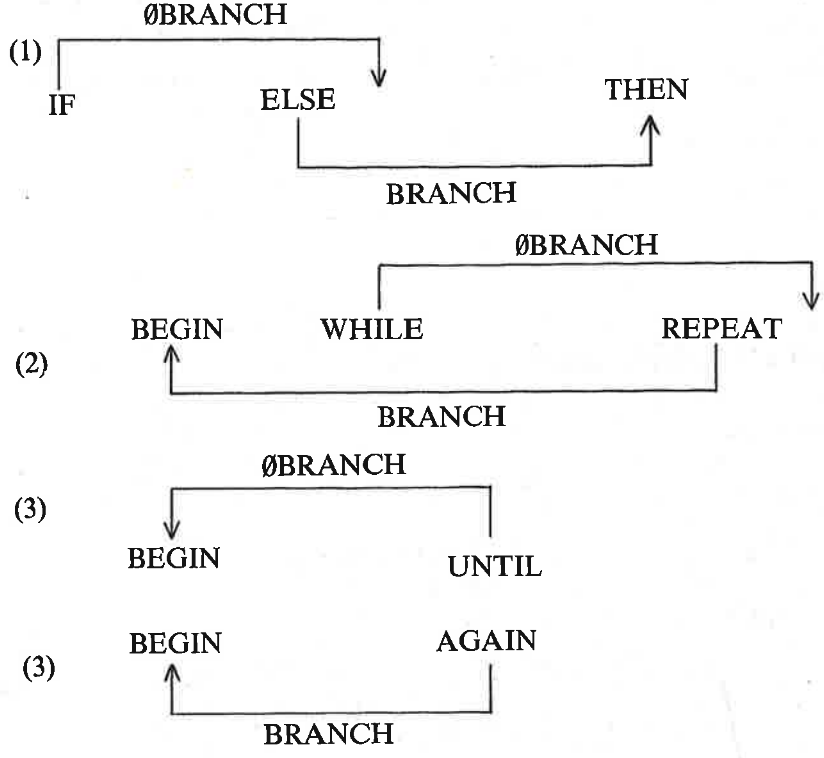


Figure 5.1 Branching Control Structures



Notes
BRANCH is an unconditional branch, it always occurs if that point is reached.
ØBRANCH is only activated if the condition is *false*

the double length number is then printed by d-dot. This still leaves a double length number TOS, and so we print the two top cells separately by dots. The first cell contains nothing, and the second cell contains 12. When I was discussing how the numbers were stored in bytes (page 16) I commented that they seemed to be in the wrong order, as do the cells, but I also noted that they were stored in numerical order of addresses. Is this the case here? Just try

```
70000. . .
```

1 4464 OK

```
70000. 84 C@ . 85C@. 86 C@ . 87 C@ .
```

1 0 112 17 OK

and guess that 0x16777216 + 1x65536 + 17x256 + 112x1 = 70000 (you did guess that, didn't you?) to see how the arrangement is not so simple as you (or I) might have hoped, but we can cope.

As a salutary reminder to remember whether it is double or single, and how it counts:

```
70000. 1 + 0. 70000. 1. D+ D.
```

135536 70001 OK

so we must use a double length 1. and a double length addition D+. Before we analyse line 7, try as your last effort

```
19. 4 U/MOD . .
```

4 3 OK

to see that the double length number 19 is divided by the single length number 4, to give the answer 4 remainder 3 in that order on the stack.

We have entered hex 2003, and the value to which SEED points, on the TOS; we then multiply them together to produce a double length number to which we add the double length number 1. The double length number TOS is then divided by the single length number 3FFF, and the dividend is left TOS, with the remainder being one under the TOS. We DROP the number TOS, as it is not wanted, the number now TOS (ie the remainder) is duplicated, and one of the values is then stored at SEED, the other value is left TOS.

The final line, line 9, defines RND. RND expects a number TOS, and this gives the range of random values, eg 9 RND will give random values in the range from Ø to 8. RANDOM is called and leaves another number TOS, the two numbers TOS are SWAPped, and the random number is then given MODulo the number TOS when RND was used.

The game

Screen #51 starts by defining three variables, all more or less self- explanatory. The word INITIALIZE stores a random number from Ø to 255 in the variable NUMB, and sets the TRIES to zero. NUMIN finds the user's guess, stored in GUESS; and CORRECT? leaves a flag TOS depending upon whether the GUESS is equal to the NUMB. In line 7, RESULT gives the result of the game, and says how many TRIES were taken.

COMPARE (lines 8 and 9) finds if NUMB is less than GUESS, and if it is, it tells you to go lower, else higher. INPUTSEG adds 1 to the number of TRIES, takes the NUMIN, and finds if it is CORRECT? — rather NOT so. The word HILO (as mentioned earlier) ties them all together by use of a BEGIN WHILE REPEAT control loop.

What does this do?

```
: XX XXA BEGIN XXB WHILE XXC REPEAT XXD ;
```

Actually, I do not know, but it could easily be another way of writing HILO (XXD is RESULT). It pays to have explicit names, quite often comments are not then needed. Though Forth is a logical language, it helps to make it look sensible . . .

```
HEX 50 LIST

SCR £ 80       50 H
  0 ( RANDOM NUMBER GENERATOR )
  1 (  )
  2 VARIABLE SEED
  3 (  )
  4 HEX : RANDOMIZE FE64 @
  5     4000 MOD DUP 0< IF 4000 + THEN SEED ! ;
  6 (  )
  7 : RANDOM 2003 SEED @ U* 1. D+ 3FFF U/MOD DROP DUP SEED ! ;
  8 (  )
  9 : RND RANDOM SWAP MOD ; DECIMAL
 10
 11
 12
 13
 14
 15
OK


OK
HEX 51 LIST

SCR £ 81       51 H
  0 ( HILO - NUMBER GUESSING GAME )
  1 (  )
  2 VARIABLE GUESS VARIABLE TRIES VARIABLE NUMB
  3 (  )
  4 : INITIALIZE 256 RND NUMB ! 0 TRIES !  ;
  5 : NUMIN ." WHAT IS YOUR GUESS? " QUERY INTERPRET GUESS ! ;
  6 : CORRECT? NUMB @ GUESS @ = ;
  7 : RESULT ." CORRECT IN " TRIES @ . ." ATTEMPTS" CR ;
  8 : COMPARE NUMB @ GUESS @ <
  9    IF ." LOWER" CR ELSE ." HIGHER" CR THEN ;
 10 (  )
 11 : INPUTSEG 1 TRIES  +! NUMIN CORRECT? NOT ;
 12 (  )
 13 : HILO INITIALIZE BEGIN INPUTSEG WHILE COMPARE REPEAT RESULT ;
 14
 15
OK
```

SCREEN 6
A logical language

> The very first lesson that we have a right to demand that logic shall teach us is, how to make our ideas clear; and a most important one it is, depreciated only by minds who stand in need of it.
> Charles S Peirce, *Collected Works*.

All of simple logic, the logic embodied in machines which can perform arithmetic, can be reduced to just one basic form.

The logical form we are most likely to know is NOT (and we have already used NOT in earlier screens), where

NOT T is F
NOT F is T

and T is true, F is false. Almost as well known are the two connectives AND and OR:

p AND q

is only true if *both* p and q are true, whereas

p OR q

is true when *either* p or q is true. If p OR q is true when either p or q is true, then p OR q is only false when *both* p and q are false — this latter statement has resemblances to the definition of AND. This resemblance is codified as one of **DeMorgan's rules**:

p OR q is NOT(NOT p AND NOT q)

where the other rule is

p AND q is NOT(NOT p OR NOT q)

Logic Gates: Compare User Guide pages 503-506

NAME	SYMBOL
NOT	
AND	
OR	
XOR	
NAND	
NOR	
EQUIV	

and so one of the two forms is (theoretically) superfluous. The NOT function is essential, however, if one is only to use, say, AND and not OR.

The four most commonly used connectives are AND, OR, the implication, and the equivalence.

p EQUALS q

is true when either both p and q are true, or both p and q are false. (T EQUALS T, and F EQUALS F, are both true). The only time that

p IMPLIES q

is false is when p is true, but q is false. The implication is true when p is a sufficient condition for q (eg a BBC computer IMPLIES television or monitor, whereas it is possible to have a television or monitor without having a BBC computer). The various ways in which these logical functions are related to the truth (or not) of p and q are shown in **Figure 6.1** (1 is true, and 0 is false).

It is quite simple to create rules to relate the implication and equivalence to AND or OR:

p EQUALS q is p AND q OR NOT (p OR q)
p IMPLIES q is NOT p OR q

and whenever there is an AND, one could put the equivalent expression for OR (and vice versa). Note, however, the ubiquitous NOT function.

One basic connective

If it were possible to derive the NOT function from AND or OR, we could reduce all of simple logic to just one function. Charles Peirce found such a function, (in 1880) a function which meant "neither...nor..." (usually known as NOR). The function can be defined by

p NOR q is NOT (p OR q)
            is NOT p AND NOT q

where the first is the usual definition, and the second follows from DeMorgan's rules. The truth table for the NOR function is simple, because the only time p NOR q is true is when both p and q are false.

H M Sheffer (in 1913) discovered another function which could be used, and it meant not p or not q (usually known as NAND)

Figure 6.1 Four Key Logical Connectives

P	Q	AND	OR	IMPLIES	EQUALS
1	1	1	1	1	1
1	0	0	1	0	0
0	1	0	1	1	0
0	0	0	0	1	1

Figure 6.2 Two Basic Logical Connectives

P	Q	NAND	NOR
1	1	0	0
1	0	1	0
0	1	1	0
0	0	1	1

p NAND q is NOT(p AND q)
is NOT p OR NOT q

(the second from DeMorgan's rules). Sheffer used the symbol | for his NAND function (Sheffer's stroke) and Peirce had used the symbol ↑ for NOR (Peirce's dagger).

The obvious question seems to be 'why are these two functions so powerful? What sets them apart?' **Figure 6.2** shows that when p and q are both true then the result is false, and when p and q are both false the result is true. Remember that to use AND or OR required the use of NOT, that NOT could not be derived from either AND or OR, so when we discover

NOT p is p NOR p
is p NAND p

we see that all we need is NAND or NOR. (Have you noticed just how often we use the words 'and' or 'or'? Useful aren't they?)

A list of results

p AND q is NOT(p NAND q)
p OR q is NOT p NAND NOT q
p NOR q is NOT(NOT p NAND NOT q)

and

p AND q is NOT p NOR NOT q
p OR q is NOT(p NOR q)
p NAND q is NOT(NOT p NOR NOT q)

which may seem interesting, but not very relevant. It is relevant, however, as a glance at the circuit diagram for the BBC computer disc interface shows (User Manual, page 505). In that diagram there are many shapes which look like a temple dome with a circle on top: these are NAND gates, and are very popular because they are easy to fabricate. They can also be used (in combination) to produce any desired effect. We have shown that, theoretically, all one needs to give all possible effects are NAND functions: practically, this has also been found to be the case.

A new notation

Nicod (in 1917) showed that the whole of logical calculus could be based on one single axiom (which used the Sheffer stroke function as the only function). In the second edition of *Principia Mathematica*

(1925) Bertrand Russell suggested that Nicod's formulation replace Russell's original form.

The NOT function (eg NOT p) can be written with brackets — though usually they are omitted — such as NOT(p). p NAND q can also be expressed as NAND(p q), and though (again) this might seem pointless, it does at least emphasise that the logical functions are functions, functions which produce a result (ie true or false). LISP is a functional language, which when it calculates p AND q uses the function (AND p q) with a different use of brackets.

To save space I will give the logical functions one letter names, those in **Figure 6.3**. Most are self-explanatory, except D for NOR (from the Dagger) and S for NAND (from the Stroke). XOR stands for exclusive-or and exists as a connective in some programming languages (it is called EOR in BBC BASIC), and most machine codes. XOR(p q) is true when p and q differ in value, and is the reverse of EQUAL(p q).

To come to terms with this new form of functional notation, here are DeMorgan's rules:

OR(p q) is NOT(AND(NOT(p) NOT(q)))
AND(p q) is NOT(OR(NOT(p) NOT(q)))

which show up the similarity structure quite well. Try the new one letter method, without brackets, for the rules

Opq is NANpNq
Apq is NONpNq

and then try to relate the following formula to the original form

Epq is OApqNOpq

This is an easy way of implementing a logical vocabulary for LISP or BBC BASIC using functions.

It will pay to study these definitions

Np is Spp
Apq is NSpq
Dpq is ANpNq
Opq is NDpq
Epq is OApqDpq
Xpq is NEpq
Ipq is ONpq

where, for example, the fifth line is read as

EQUALS(p q) is OR(AND(p q) NOR(p q))

If we had a computer wired for logic, all we would need is a NAND gate, and arrangements of NAND gates.

Consider:

If my computer then my portable TV
If my portable TV then a monochrome picture
THEREFORE
If my computer then monochrome picture

or

If p then q
If q then r
THEREFORE
If p then r

or

IMPLIES(p q)
IMPLIES(q r)
THEREFORE
IMPLIES(p r)

or we could say that the first two statements imply the third statement (the conclusion):

so

IMPLIES(AND(IMPLIES(p q) IMPLIES(q r))
IMPLIES(p r))

(it is worth the effort to tease this out). We can then turn this into

IAIpqIqrIpr

which we could then test in some automated way for the truth of the whole. The whole is no more than a complex function, the complex function called the 'hypothetical syllogism'.

A reversed notation

This extremely concise (and readily mechanised) method was invented by a Polish logician J Łukasiewicz, and is called (funnily enough) *Polish Notation*. The main difference between the original notation, and my version, are the letters which stand for the fun-

Figure 6.3 Notational Equivalences

FULL	SHORT
NAND	S
NOT	N
AND	A
NOR	D
OR	O
EQUALS	E
XOR	X
IMPLIES	I

ctions: Łukasiewicz used Polish clues for the letters, I use English — a computer could not care less. Suppose that we write the definition of EQUALS in reverse order? We will ignore brackets, and so produce

p q EQUALS is p q NOR p q AND OR

and the method is naturally called *reverse Polish notation* (RPN).

When we evaluate this formula, we take the parameters in strict order, and when we reach a function we use parameters we have stored for that function. In RPN the hypothetical syllogism becomes

p r IMPLIES q r IMPLIES p q IMPLIES AND IMPLIES

and let us work through the line. Variables p and r are stored (you guessed! — on a stack), and then when IMPLIES is reached it expects two parameters on the stack, operates on them and leaves a value TOS. The sequence is repeated for q r IMPLIES, and p q IMPLIES, so that, when it is reached, there are three values on the stack and AND operates on the top two values, leaving a value TOS. There are now two values on the stack, and it is those upon which the last IMPLIES operates. Sounds just like Forth, does it not?

LISP would add the two numbers together by

(PLUS 2 3)

which plusses 2 and 3. The process of addition is really the application of a function, the function PLUS operates on 2 and 3 to produce the sum of the two numbers. As we have already seen, LISP works in a Polish manner whereas with Forth we proceed by an RPN procedure. To add 2 and 3 together, therefore, we must use

2 3 +

to leave the number 5 TOS. It is all perfectly logical, but somewhat annoying if you are trying to turn $(2+3)\times(3-4)+9/2$ into RPN (answer at end of this screen).

Words in Forth sometimes, as we have noted, correspond to functions, and so the logical functions we have been discussing would seem to fit Forth well: examine screen #6Ø hex.

Screen #6Ø H starts, at line Ø, with the title: these are logical functions, and this is the first page. We declare (line 2) that we will be using a special vocabulary to be called LOGIC, and LOGIC is an immediate word (ie it will execute immediately even within colon definitions). Line 4 then says that the following definitions are to be

placed in the LOGIC vocabulary (so as to make it a special set of words, such as those of the Editor).

The other words on this screen are not actual logic functions, rather they are preparatory operations: T has the value 1 for true, F has the value Ø for false (both constants); TABLE is a variable whose value will be treated as two bytes, and not one cell (similar to CELL (page 33), or BYTES (page 43). To store the value 1 in the cell is to store 1 in the lower byte, and Ø in the higher byte. You can check that here and now by

```
VARIABLE TABLE 1 TABLE ! TABLE C@ . TABLE 1 + C@ .
FORGET TABLE
1 Ø OK
```

The word UNDER2 places the value TOS under the two values next to it on the stack, making those two values TOS and TOS-1 (it uses the ROT function twice — try it out on values by checking with the word STACK, (page 27) and screen #2Ø H).

All the words in this screen are preparatory to the actual definition of functions in the next screen #61 H. So look at that screen.

Line 15 of the previous screen had an arrow, – – > , two minuses and the inequality sign, which means that when we

```
HEX 6Ø LOAD DECIMAL
```

we also load #61 H. On line 2 of the next screen there is another assertion that the words to be defined are to be placed in the LOGIC vocabulary, in case there is a hiccup in the loading. The logic functions are defined in lines 4 to 11, starting with NAND. On loading this screen you will notice many messages #4, because quite a few of these functions already exist as standard in Forth. As long as we ask for LOGIC when we use these redefined words, there will be no confusion (setting up the screens and testing involve many changes between the EDITOR and LOGIC vocabularies, apart from involuntary moves to the Forth vocabulary).

The logic functions

Each function expects either 1 parameter (NOT) or two parameters (the rest) on the stack. NAND expects two parameters (1 or Ø), and finds the correct result for NAND by looking it up in the appropriate locations I have called TABLE. In more technical jargon, the variable TABLE constitutes a very simple look-up table:

we do not calculate the value we look it up in a table held somewhere in memory. (This is used extensively in the Turtle Graphics application). I have only calculated the result for NAND in this manner as a simple illustration, because obviously

```
: NAND * 1 SWAP - ;
```

would work equally as well. I could have added an extra byte onto TABLE, by using ALLOT, and then I could have merely added the parameters (with a few other modifications, work them out).

NOT expects one parameter, which has to be duplicated to allow NAND to function: AND is merely NAND NOTted; and so we can then use NAND, NOT, or AND, when we come to the next definition. We define OR: on page 82 we see that OR is defined in Polish notation by

Opq is NANpNq

which in RPN is reversed to produce

qpO is qNpNAN
is qNpNS

because AND NOTted is NAND (ie Sheffer's stroke). Line 7 gives the definition of OR, we then NOT one parameter, SWAP and then NOT the other parameter, and NAND the two parameters — check the congruence between this line and qNpNS.

The only other complex definition is that for XOR. Now, on page 82 we have (reversed here)

qpE is qpDqpAO
qpX is qpEN

so that

qpX is qpDqpAON

now OR(ie O) can be expressed in terms of AND and NOT, i.e. (page 82)

xyO is xNyNAN

so, by extension, we find

qpX is qpDNqpANANN

Stop to think.

After you have thought, think about

qpX is qpOqpSA

and how that is what seems to happen in line 9. The parameters p and q occur twice, once for the OR and once for the NAND, and the word 2DUP duplicates the top two words (it also is used to duplicate double length numbers). UNDER2 puts the top value under the two numbers underneath, so that NAND can be used: this leaves a parameter, plus the first result, and we AND.

IS is a word, like others we have seen before, which merely looks at the number TOS, and outputs whether it is true or false. Forth DEFINITIONS makes sure that once the LOGIC definitions are over, we do not add any by mistake: to add new LOGIC definitions we will have explicitly to say so at the time we produce the new definition.

To use the new functions we might try

LOGIC (TO ACTIVATE LOGIC) T T XOR IS

FALSE
OK

(try to define a new function to produce the hypothetical syllogism, but remember LOGIC DEFINITIONS).

Finally, a few words on the ways in which Forth differs from some other languages, and we will use the syllogism as an example.

IMPLIES(AND(IMPLIES(p q) IMPLIES(q r)) IMPLIES(p r))

A language decoder, when it comes to this expression, encounters IMPLIES, a function it knows to have two parameters. This information will have to be stored, probably on a procedure/function stack. The decoder then encounters AND, not a parameter, and so it will store this information on the procedure stack: and then AND is followed by yet another function, IMPLIES, which information is stored on the procedure stack.

At last we encounter parameters, p and q, and they are stored on a computation stack: IMPLIES then operates on the two values on the stack, leaving the result on the stack. The result of this IMPLIES becomes the first parameter value for the AND, the second parameter value for AND is left on the stack by the operation of IMPLIES on q r. The result left by AND becomes the first parameter value for the first IMPLIES we encountered.

This is tedious, though easily enough done on a computer, but it does require two stacks — for the computation, and one for the procedures. In **Figure 6.4** the syllogism is shown as a directed graph

(something akin to the decoding of VALUES, in **Figure 3.6**), and an ordinary functional language goes through the graph from the top downwards.

In an elemental functional language (such as Forth) we work from the bottom line upwards, eg in **Figure 6.4**.

p r IMPLIES q r IMPLIES p q IMPLIES AND IMPLIES

and only one stack is needed, because the user has already decoded the order in which the functions are to be applied.

The RPN version of

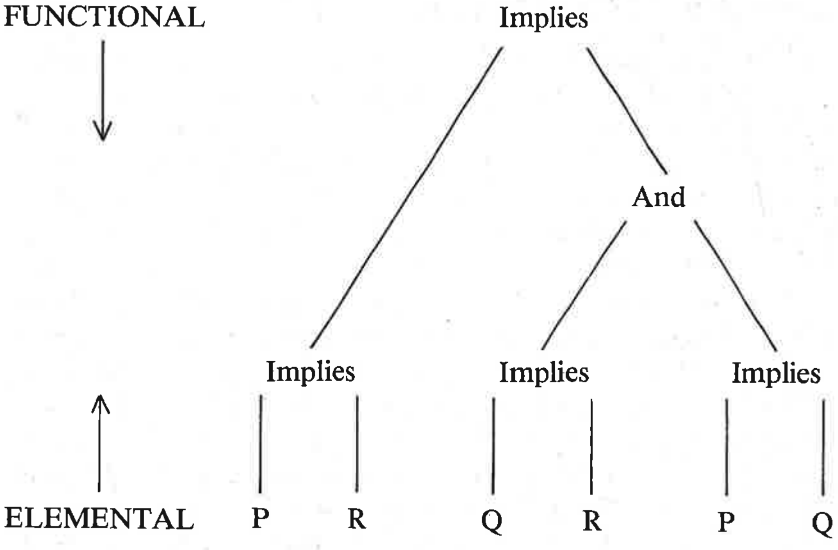
(2 + 3) × (3 − 4) + 9/2

might be (there are other arrangements)

9 2 / 3 4 − 2 3 + x +

Simple?

Figure 6.4 The Hypothetical Syllogism



```
OK
HEX 60 LIST

SCR £ 96        60 H
  0 ( LOGICAL FUNCTIONS -- P 1 )
  1 (  )
  2 VOCABULARY LOGIC IMMEDIATE
  3 (  )
  4 LOGIC DEFINITIONS
  5 (  )
  6 1 CONSTANT T ( TRUE  )
  7 0 CONSTANT F ( FALSE  )
  8 VARIABLE TABLE 1 TABLE !
  9 (  )
 10 : UNDER2 ROT ROT ;
 11
 12
 13
 14
 15 -->
OK


HEX 61 LIST

SCR £ 97        61 H
  0 ( LOGICAL FUNCTIONS -- P 2 )
  1 (  )
  2 LOGIC DEFINITIONS
  3 (  )
  4 : NAND * TABLE + C@ ;
  5 : NOT DUP NAND ;
  6 : AND NAND NOT ;
  7 : OR NOT SWAP NOT NAND ;
  8 : NOR OR NOT ;
  9 : XOR 2DUP OR UNDER2 NAND AND ;
 10 : EQUALS XOR NOT ;
 11 : IMPLIES SWAP NOT OR ;
 12 (  )
 13 : IS IF ." TRUE" CR ELSE ." FALSE" CR THEN ;
 14 (  )
 15 FORTH DEFINITIONS
OK

OK
HEX
```

SCREEN 7
Son et Lumière

Lying in bed would be an altogether perfect and supreme experience if only one had a coloured pencil long enough to draw on the ceiling.
G K Chesterton, *Tremendous Trifles.*

Personally, I find the use of sound in programs fairly boring, and I do not revel in yet another chorus of 'we are the champions' or some funereal march. This may be why, when I come to write about sound using Forth, my enthusiasm stutters.

Sometimes, in a long application, I might want to beep to wake me up — ready for the next section, etc. Try

```
7 EMIT
```

and you will hear the beep (ASCII CTRL G), so why not

```
: BEEP 7 EMIT ;
: BEEPS Ø DO BEEP LOOP ;
```

and then

```
1ØØØØ BEEPS
```

will drive you mad. This will not satisfy those of you who want to do more complex effects, so the first thing to do is to read all of section 30, then pages 244, 347 and 461 in the User Manual.

All attempts to make sounds (other than the BEEP) use an assembly language routine (for general details see section 43 of the User Manual, and, in particular, section 10.3 of the Acornsoft Forth Manual). The routine used is OSWORD (an operating system routine) with various parameters. In fact I could have used OSWORD to read the internal clock, to start my random number.

The OSWORD routine is invoked by a call to location hex FFF1, which points to location 20C (hex), in normal programming, as can be seen by

HEX FFFI C@ . FFF2 @ .

6C 20C OK

because hex 6C is machine code for a jump to a new location, anywhere in memory, and the address to which the jump is to be made is *contained* in location hex 20C. Therefore,

20C @ .

– 1815 OK

This is not terribly helpful, but if we attempt to

20C @ U.

E7EB OK

this is more pleasing to the eye. We have now jumped into the operating system ROM (User Manual, page 502).

A sound routine

Examine then what has to be done to activate this sequence: on page 461 of the User Manual, we find that a call to OSWORD with A = 7 makes a sound. A stands for the 6502's accumulator, and A = 7 means that we have to have placed, in some way, the value 7 in the accumulator. The eight bytes pointed to by locations XY to XY + 7 are treated as four 2-byte values, the User Manual tells us, or — as Humpty Dumpty said — inpenetrability. X and Y are special locations in the 6502 processor, called *registers*, and when the OSWORD procedure finds 7 in the accumulator it then expects the two bytes (X and Y are both one byte) to together point to the start of eight bytes of information.

I do not wish to invest substantially in the explanation of assembler commands, but will explain in outline how the routine ((SOUND) — Acornsoft Forth Manual, section 12.2.1) executes. The first thing any routine using the X register must do, is save the present value stored in the register (the X register contains a pointer to the current state of the stack, Forth Manual section 10.4) and this is done by storing the content of X at location XSAVE (usually 68 hex).

We next need to store the value 7 in the accumulator (ie $ LDA), and then jump to the subroutine (JSR) at location FFF1. After the subroutine has run its course we take the value in XSAVE and load it back into the accumulator (LDA). We clear the carry bit, and then add 8 onto the value in the accumulator ($ ADC). The value in the accumulator is now 8 more than the original value of the X register (it is as if eight bytes have been dropped from the stack). When the value in the accumulator is transferred to the X register (TAX), it points to 4 cells higher. A jump to NEXT means that execution is transferred to the next word in the sequence.

Here is the sequence again, when we have loaded the ASSEMBLER vocabulary by 12 LOAD:

```
HEX ( IT KEEPS IT SIMPLE )
CODE (SOUND) ( DEFINE AN ASSEMBLER WORD )
XSAVE STX, ( SAVE THE VALUE IN X REGISTER )
7 $ LDA, ( LOAD 7 INTO ACCUMULATOR )
FFF1 JSR, ( CALL OSWORD AT FFF1 )
XSAVE LDA, ( PUT ORIGINAL X VALUE IN ACC )
CLC, ( CLEAR THE CARRY FLAG BEFORE )
8 $ ADC, ( ADDING 8 TO ACCUMULATOR )
TAX, (TRANSFER VALUE FROM ACC TO X )
NEXT JMP, ( AND FOR THE NEXT WORD . . . )
END-CODE ( BACK TO YER FORTH )
DECIMAL ( FINGERS CAN BE USEFUL )
```

and if the code for (ENVELOPE) is compared, the differences are minor. To wit,

```
8 $ LDA, replaces 7 $ LDA,
ØE $ ADC, replaces 8 $ ADC,
```

because OSWORD expects 8 in the accumulator, and there are 14 (hex ØE) items on the stack and not 8. As an exercise (to which I give no solution) try to invent a routine which will operate as either (SOUND) or (ENVELOPE) — depending on parameters.

(SOUND) expects four cell values on the stack (eight bytes), and (ENVELOPE) expects fourteen bytes on the stack, treating them as fourteen different values. As the routine for ENVELOPE is going to be more complex, I will discuss that routine and ignore SOUND (see section 12.2 of the Forth Manual). We have to remember that if a number is to be stored as a byte it can only take 256 distinct values, and an examination of the envelope command in the User Manual

(page 244) shows that values vary from − 128 to 127, in some cases, and from Ø to 255 (and less in some cases).

Type in the number (where it is the Forth AND we use)

```
HEX -1 DUP U. FF and U.
```

FFFF FF OK

and we know (from Screen 3) that −1 is hex FFFF (ie decimal 65535), and obviously (?) to AND FFFF with FF will produce FF. Enter (still in hex)

```
2345 FF AND U.
```

45 OK

and try to work out why (I suggest you try to examine numbers in their binary form). We have, it seems, a way of losing one of the two bytes in a number. Slightly modify the preceding input,

```
2345 FF AND >< .
```

45ØØ OK

and it is clear that the bytes have swapped round in their order (we used this, or nearly used this, in our application RANDOMIZE, page 70).

The word EXPACK (section 12.2.2 of the Forth Manual) sets the higher order byte to zero by ANDing, and uses the return stack for temporary storage. I will make only the slightest of modifications, in that I will stay in hex:

```
: EXPACK
     >R ( TOS TO RETURN STACK )
     FF AND ( SET VALUE NOW TOS TO Ø HIGH-BYTE )
     R> (VALUE FROM TORS BECOMES TOS )
     FF AND ( HIGH-BYTE SET TO Ø )
     >< ( SWAP BYTES )
     + ; ( ADD TWO TOP CELLS TOGETHER )
```

and I have added a few comments. EXPACK is great, we can now reduce two cells to one (ie two bytes), if they are TOS and TOS − 1, but not all of them can be.

If we are going to allow the parameters for ENVELOPE to be given in the order of the User Manual, then this is back to front for Forth. This is something else we have to sort out. We can cope with

the top two cells, so we can EXPACK them without further ado. We are now left with one cell where there were two, and we want to access the two cells under the TOS — we can use UNDER2 defined in the last Screen (page 86 and screen #6). We now have two cells atop, in the correct order for Forth as well.

Now it is possible to define ROT as 3 ROLL, and 1 ROLL does nothing. To retrieve the next two cells, that is, those under the two cells we modified, we can 4 ROLL 4 ROLL. We have seven cells to fill, from 14 original cells, and as 2 ROLL 2 ROLL leaves the top two cells as they were (try it), we can modify the ENVELOPE word defined in the Forth Manual — purely in the interests of showing the structure —

```
: ENVELOPE ( MODIFIED VERSION )
9 2 DO ( FOR ALL VALUES FROM 2 TO 8 )
I ROLL I ROLL ( BRING TO TOS )
EXPACK (BYTE MANIPULATION)
LOOP ; ( AND AGAIN )
```

This may well be slower than the version suggested in the manual, and is only given in this form to accentuate the structure. How you use these words is up to you: all you really need is BEEP.

VDU commands

Graphics are far more interesting than sound, and graphics in Forth on the BBC computer take some beating. The graphics are so good that my final application — Screen 9 — is an in-depth use of graphics. At the moment I want to examine the outline of graphics on the BBC, using Forth.

For a start, because of the way Acornsoft Forth uses memory, we can get no higher resolution than Mode 4 (effectively Model A graphics). When you switch on, and load Forth, you are in Mode 7, and thus to use any of the graphics capabilities you need to change mode. Type

```
4 MODE
```

and the screen clears, and you are in Mode 4. As a frequent user of Forth, and a user of BASIC, you may find that you enter 4 MODE in BASIC, and nothing (of course) happens. Nothing, that is, except a line 4 MODE in your program — beware.

To drive graphics, we need to use the VDU drivers, because these allow us to perform all manner of useful operations. The list (User Manual, page 378) is extensive, and can be emulated in various ways. VDU 3Ø homes the text cursor to top left, so

DECIMAL 3Ø EMIT

will home the text cursor to the top left. Alternatively

7 > VDU

will beep, because > VDU is almost the same as EMIT; but EMIT, you will realise, only takes values from Ø to 127. That a VDU driver is little more than a collection of EMITs helps explain the differences between the use of commas and semi-colons in VDU commands. Incidentally try

22 EMIT 7 EMIT

and you will change to mode 7. To get to mode 4

22 EMIT 4 EMIT

and we are away.

Certain VDU commands are followed by numbers separated by commas, and certain are followed by semi-colons. The distinction is not that important for BASIC, as long as you use the correct separator, but in Forth the distinction is crucial. The distinction is explained on page 386 of the User Manual, where it is explained that

VDU 25,4,1ØØ;5ØØ;

is equivalent to

VDU 25,4,1ØØ,Ø,244,1

because 5ØØ is equal to 1 × 256 + 244. Try to

5ØØ >< 255 AND . 5ØØ 255 AND .

1 244 OK

which is in the opposite order to that we need.

A few definitions seem in order. First we give > VDU (and also EMIT) a new name

: ,, > VDU ;

to be followed by

```
: ;; DUP >< 255 AND SWAP 255 AND ,, ,, ;
```

so that VDU 25,5,100;500; (ie plot to the coordinates 100, 500) can be simply written

```
25 ,, 5 ,, 100 ;; 500 ;;
```

and a little line is drawn. If you enter

```
7 ,,
```

we beep again.

The random walk

Make sure RND (and associated words) is loaded, and enter the following (compare with the list of VDU commands, page 378 of the User Manual):

```
: RANDWALK ( A RANDOM WALK APPLICATION )
RANDOMIZE ( GET IT OVER EARLY )
12 ,, 16 ,, ( CLEAR TEXT, GRAPHICS, AND HOME CURSOR
)
25 ,, 4 ,, 600 ;; 600 ;; ( MOVE TO 600,600 )
BEGIN 25 ,, 1 ,, ( THIS WILL BE A REL PLOT )
55 RND 27 - ;; ( THE RANDOM X CHANGE )
55 RND 27 - ;; ( THE RANDOM Y CHANGE )
?TAB UNTIL ; ( STOPPED BY TAB )
```

and when activated this will produce a haphazard squiggle — a random walk. Unfortunately the squiggle repeats: my random number generator is not all that random, it would seem. Apart from the fact that my random number generator isn't (it repeats itself quite quickly), the RANDWALK word is a perfect example of how not to write an application. We will try again.

Here are the words:

```
: CLS 12 ,, ;
: CLG 16 ,, ;
: WALK DUP RND SWAP 1+ 2/ - ;
: RANDOMWALK
CLS CLG 4 600 600 PLOT
BEGIN 1 55 WALK 55 WALK PLOT ?TAB UNTIL ;
```

and the sense is there to be seen, with no need for comments. PLOT is an Acornsoft Forth command which exactly corresponds to the ordinary commands given in the User Manual.

User-defined graphics

VDU 23 can be used to produce user-defined graphics shapes (pages 384 and 427). The VDU command takes ten values, the first is 23 which gives the nature of the VDU command, and the second is the ASCII code for the character to be re-defined (the defaults are those between 224 and 255 inclusive). The next eight bytes are the values corresponding to the lines in the redefined character, stored as the equivalent to the binary number so formed: page 170 of the User Manual gives full details.

To store a simple man we have to use VDU 23,240,28,28,8,127, 8,20,34,65 (page 171), where — for example — 28 is formed by blobs in the fourth column from the left (16), the fifth (8), and the sixth (4). Right, first define a new word

```
: BIN 2 BASE C! ;
```

and then enter

```
BIN 00011100 DECIMAL .
```

28 OK2

so we have an easy way to enter the values for shapes, without any need to convert to decimal. If we enter in the values from the top row downwards, as we do normally with VDU 23, then there will be a fair deal of stack manipulation. The question is how to do the manipulation, and I will use the method we used together with EXPACK in the definition of ENVELOPE.

```
: DEFSHAPE 23 ,,
1 9 DO I ROLL ,, -1 +LOOP ;
```

So to enter the little man from page 170 of the User Manual, all we need to do is (to store in character 240)

```
240 BIN 00011100 00011100 00001000
01111111 00001000 00010100 00100010
01000001
```

OK

DECIMAL DEFSHAPE 24Ø ,,

and a little man appears.

To define the suits for playing cards (page 389 of the User Manual) all we need do is

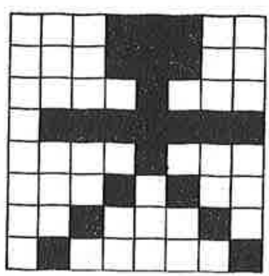
```
224 8 28 28 1Ø7 127 1Ø7 8 28 DEFSHAPE
225 8 28 62 127 62 28 8 Ø DEFSHAPE
226 54 127 127 127 127 62 28 8 Ø DEFSHAPE
227 8 28 62 127 127 127 28 62 DEFSHAPE
```

To fill the screen with little men takes about 0.7 seconds in Forth, in BASIC it takes about 1.4 seconds using VDU 24Ø, or 1.7 seconds using PRINT CHR$(24Ø); . The speed differential, in this case, does not seem to be as great as that for filling the high resolution memory (as explained in Screen Ø).

Graphics will be discussed extensively in my final application (Screen 9), but first some creative writing.

Defining A Little Man

Shape



Binary Value	Decimal Equivalent
0 0 0 1 1 1 0 0	28
0 0 0 1 1 1 0 0	28
0 0 0 0 1 0 0 0	8
0 1 1 1 1 1 1 1	127
0 0 0 0 1 0 0 0	8
0 0 0 1 0 1 0 0	20
0 0 1 0 0 0 1 0	34
0 1 0 0 0 0 0 1	65

Note

The shape of the man is that given in the User Manual, and the binary and decimal values correspond to the shape, in that where the square is black there is a 1 in the binary number — also zero.

SCREEN 8
Creative Writing

> Had I been present at the creation, I would have given some useful hints for the better ordering of the universe.
> Alfonso the Wise

The compilation field points to a machine code routine somewhere in memory, and the routine indicates what is to be done by the word which points to the routine. The parameter field provides the information upon which the routine can operate, but as was noted on page 15 there are different types of operation.

Consider what happens when we define

```
356 CONSTANT ONE
```

in terms of what happens at the time of creation, and then what it does when we refer to ONE later. When we create the constant ONE, the value 356 is placed into the parameter field which immediately follows the compilation field (if in doubt refer to Screen 1). When ONE is executed (or any other constant), it does something: it fetches the value in the location we call the parameter field. To define a type-forming word, therefore, we have to define two aspects: first, how the structure is created; and, second, what the type does when it is executed.

Define a new form of constant, exactly like the old one, but with a different name:

```
: CONST CREATE , DOES> @ ;
356 CONST NEW-ONE (AN EXAMPLE)
```

and then see how it works. We have a colon definition, so that the name field, link field, and compilation field, will be as usual. The first word in the parameter field is CREATE: CREATE is a word

which means that when CONST is executed, Forth will expect the name of a new word to follow. The new word will be defined as type CONST (eg NEW-ONE), and the next portion of code — until DOES> or ";" — is the way the word is structured. In the case of CONST the word "," (the comma) means take the number TOS, and compile (store) it into the next two vacant locations, the first two locations of the parameter field. The existence of the comma is why, in the previous screen, I use double-comma ,, for the VDU command.

When we just enter NEW-ONE we leave the parameter field address, unless there is something to follow DOES> , and for the type CONST, we fetch (@) the value stored in the parameter field. That is, when we enter a constant, we automatically produce the value stored.

The way VARIABLE is defined is

: VARIABLE CREATE 2 ALLOT ;

which means allocate two bytes for the parameter field, but when a variable is executed leave only the PFA (Screen 1). The two bytes are uninitialised, so if we define

: VAR CREATE , ;

OK

89 VAR TT TT @ .

89 OK

we have a way to use variables, where we are able to initialise the variable — a useful function.

Creating shapes

Return to the last screen, and consider the word DEFSHAPE. Would you like to be able to say (compare page 93)

240 MAN DEFSHAPE

where MAN is some word which is defined as those eight relevant numbers, eg

: MAN 28 28 8 127 8 20 34 65 ;

(User Manual, page 171), but this is slightly messy in that this is like defining a word

```
: ONE 356 ;
OK
ONE .
350 OK
```

which is a strange way of defining a constant. Just as we can define the constant by use of the defining word CONSTANT, so perhaps we can define shapes by a defining word SHAPES?

The defining word SHAPES will have to create a structure with eight values (bytes); and then, when it is used, what it does is to present the eight values in the correct order to DEFSHAPES. Here goes

```
: SHAPES CREATE 8 Ø DO C, LOOP
DOES> DUP 7 + DO I C@ -1 +LOOP ;
OK
28 28 8 127 8 2Ø 34 65 SHAPES MAN
OK
MAN . . . . . . . .
65 34 2Ø 8 127 8 28 28 OK
```

which seems correct. The portion after CREATE compiles eight values in the eight bytes of the parameter field of the word which will be defined; and when one DOES> the PFA is duplicated, with 7 being added to one of the values so that we operate a loop which goes from the last value on the stack (PFA + 7), to the first value off the stack. Study the operation to make sure you see how it works.

We have a man, suppose we want to invert him? To invert a defined character, we have to start at the bottom instead of the top, and go from right to left, instead of left to right (e.g. 1111ØØØØ is turned into ØØØØ1111). This means we have to find how to reverse, but we do not have to create this time, we only need to define some ordinary words.

Start simply : 1Ø1Ø is equal to 1Ø, and its reverse Ø1Ø1 is equal to 5; 111Ø is equal to 14, and its reverse Ø111 is equal to 7; but this does not appear to be the way to proceed. What I was trying to do was not to have to analyse the number as a binary number, but find some simple trick way out: I cannot. There might be a simple way,

but it escapes me, and so I would be only too pleased to hear of such a method — until then I have to operate in rather a long-winded manner.

The first thing, then, is to turn the value into a series of binary digits (ie bits): the number 2 is 1Ø in binary, and the number 3 is 11 in binary; 2 divided by 2 is 1 remainder Ø, and 3 divided by 2 is 1 remainder 1, eg

```
2 2 /MOD . . 3 2 /MOD . .
```

1 Ø 1 1 OK

Here lies our salvation. If we continually divide by 2, leaving the remainder at each division, this is how we can produce a binary number: we need seven divisions.

```
: BITS 7 Ø DO 2 /MOD LOOP ;
67 BITS . . . . . . . .
```

Ø 1 Ø Ø Ø Ø 1 1 OK

We have produced a representation of 67 as the eight bit binary number Ø1ØØØØ11, with the most significant (ie leftmost) bit being TOS. The reversed value is therefore capable of being produced if we treat the value TOS as being the least significant (ie rightmost) bit.

When we discussed loops I showed how it was possible to produce a loop whose counter (I) increased as a binary progression (page 68): we use this type of loop here,

```
: REVAL Ø 255 1 DO I * + SWAP I +LOOP SWAP ;
67 BITS REVAL .
```

194 OK

and (as you will no doubt check) 11ØØØØ1Ø is binary version of the decimal number 194. To operate on the eight numbers stored in a shape we use

```
: REVERSE 8 Ø DO BITS REVAL 8 ROLL LOOP ;
```

so that we can

```
MAN REVERSE SHAPES NAM
```

so that the reverse of MAN is now saved as a new shape NAM.

It might appear to be simple to flip the numbers upside down, all one does is take the numbers on the stack and reverse their order. This is a common need, and can be satisfied in one way by

```
VARIABLE 8BYTES 6 ALLOT
: FLIP 8BYTES DUP 7 + DO I C! -1 +LOOP
8BYTES DUP 7 + DO I C@ -1 +LOOP ;
1 2 3 4 5 6 7 8 FLIP . . . . . . . .
```

1 2 3 4 5 6 7 8 OK

and there are other ways. To turn a MAN upside down

```
MAN REVERSE FLIP SHAPES RMAN
```

and RMAN is the shape of an upside down MAN. How about

```
: ROTATE REVERSE FLIP ;
```

so that to rotate a shape (turn upside down) we enter

```
ANYSHAPE ROTATE SHAPES ROTATEDSHAPE
```

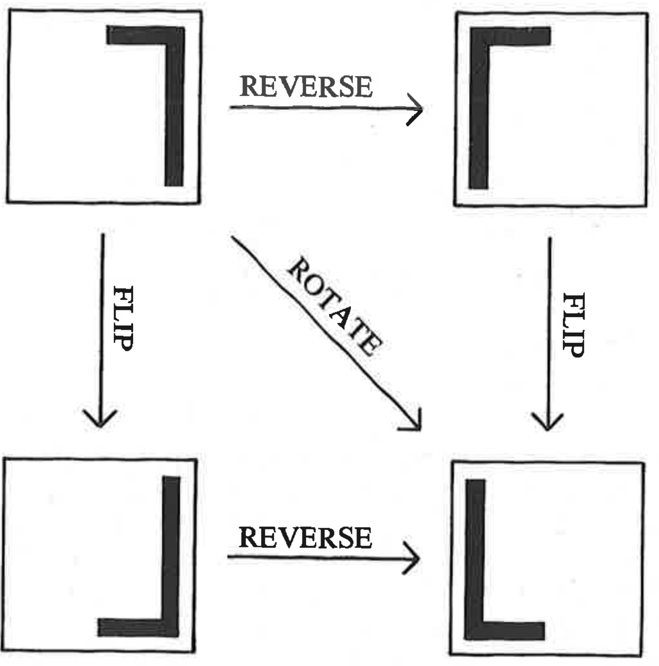
so by entering one shape we can already form up to three other derivative shapes.

The potential of the use of CREATE is thus vast, already we have simplified the use of user-defined graphics beyond recognition. Suppose we want alternatively to print a shape and then its rotated version (eg MAN)

```
5 MODE (IT IS CLEARER TO SEE)
: ONWARDS BEGIN 240 MAN DEFSHAPE 240 ,, 240
RMAN
DEFSHAPE 240 ,, ?TAB UNTIL ;
```

and it can be seen that we can build up a repertoire of shapes (stored on screens), and we need not be restricted by the number of characters we are able to define. It is, of course, quicker not to redefine the same character, and to use the 32 characters from ASCII 224 to 255 without having to worry about 'exploding' the memory (User Manual, page 427). It is possible that you might need more than 32 different shapes, but if you store the 32+ shapes as SHAPES, then there are no other complications. In addition, if more than one shape is a variant on a basic shape then, by designing the correct words, new shapes can be defined purely with respect to that first shape. The power is clear to see (no pun intended).

Creating Shapes



Notes

The action of FLIP and REVERSE is commutative, in that REVERSE FLIP gives the same result as FLIP REVERSE.

SCREEN 9
Turning Turtle Graphics

The turtle lives 'twixt plated decks
Which practically conceal its sex.
I think it clever of the turtle
In such a fix to be so fertile.
Ogden Nash, *The Turtle.*

Forth — as we have seen — tries to keep things simple. Another language which tries to keep things simple is LOGO, and one of the most arresting qualities of LOGO is the ability to use Turtle Graphics.

Here is a simple notion: on the display is a turtle, sometimes it is visible, and sometimes (our time) it is invisible. You instruct the turtle by three commands: MOVE forward a certain amount; TURN an amount: and draw a line when moving, or do not. The application in screens #9ØH to #94H contain all that is needed for a sophisticated system, BUT (not that big a but) we turn in units of five degrees at a time. The reason for this will become apparent if we examine the first screen — P 1, VARIABLES.

The first word is TURGRA and this delineates the turtle graphics application as a whole: if we FORGET TURGRA then we do. There are certain global variables (those in lines 11 to 14) and to make absolutely sure the values are defined from the outset, I CREATE a word VAR which sets the variable automatically to zero. The variables are fairly self- explanatory: ANGLE gives the present direction in which the turtle faces (directly upwards is zero and we measure counter-clockwise); PEN is 1 or Ø depending on whether the pen colour is white or black; and X and Y are the coordinates of the turtle (on this, more later).

To perform any plotting, where we use diagonal lines, we need to be able to calculate various trigonometric functions of angles, most of which take on fractional values. As you will only be too aware,

Forth only deals in terms of integer values. Lines 7, 8, and 9, contain my solution to this little problem: we define a variable SIN, and then store (ie compile) 18 further values immediately after the parameter field of SIN (which already contains zero). The further values are sine values (for increments of 5 degrees) taken from standard tables, with .ØØØ1 being expressed as 1, and 1.ØØØØ being given as 1ØØØØ — we have a look-up table of sine values.

Screen #91 (P 2, DISPLAY) makes extensive use of VDU commands, and the words ,, and ;; given earlier, in Screen 7 (it is as well to study section 34 of the User Manual). The words CLG, CLS, COLOUR, and GCOL, are of the same meaning as the equivalents given in the User Manual: the other words are concerned with the way in which we describe our display. The word GRASCR (a VDU 24 command) clears a graphics screen which extends from line 128 upwards, ie leaving a space at the bottom of the display; ORIGIN sets the graphics origin (VDU 29) to the middle of the new graphics screen. CLRGRA sets the foreground colour to the colour given by the value of PEN, and the background is set to its reverse, then the sequence GRASCR CLG ORIGIN is executed (note that text is unaffected).

The text window is set up by TEXSCR (VDU 28), to give four lines at the bottom of the display, in mode 4 (exactly contiguous with the graphics window). CLRTEX sets the foreground and background colours to the reverse of those for the graphics window, and then COLOUR TEXSCR CLS. And to the next screen.

The next screen (#92H — P 3, UTILITIES) contains the house-keeping words. PCOL takes the value TOS, and assigns that to PEN (simpler than PEN ! for non-Forthers who may be using the routines). It is not imagined that it will be greatly used. The word INVERT takes the current foreground colour and reverses it (eg black to white, or white to black). This is one way it is possible to paint over lines.

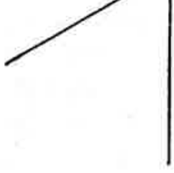
The next four words can be conveniently assigned to function keys (assume VLIST is on key Ø)

```
1 KEY' ALLCLR  M
2 KEY' RESTART  M
3 KEY' STARTALL  M
```

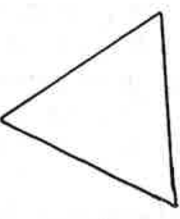
where key 1 is the same as 'clear everything', key 2 is the same as 'clear the graphics, and initialise everything', and key 3 is the same as 'clear both windows, and initialise'. To program the keys in this way

Constructing A Triangle

1 200 MOVE 24 TURN



1 200 MOVE 24 TURN



1 200 MOVE 24 TURN

: TRISIDE DUP 1 SWOP MOVE 24 TURN;

: TRIANGLE TRISIDE TRISIDE TRISIDE DROP;

can greatly aid in the development of effects. The only word left to explain is CENTRE — this sets the angle to zero, and both coordinates to zero.

Enter

-99 72 MOD .

-27 OK

so that when we take the modulus of a negative number, the result is negative. Line 2 of screen #93, therefore makes sure that the angle lies between Ø and 71, and that the result is always positive. TURNTO sets the angle to the value TOS, and TURN adds the value TOS to the present angle (all within the correct bounds). If you are not sure, LOC tells you where you are, and which way you are pointing. When we use trigonometrical functions we only need to consider one quadrant, and its plus and minus versions: STANG chooses the correct angle from the first quadrant.

SVAL uses STANG and also the size of the angle (ie whether it is greater than 36) to work out the value of the sine, using the SIN look-up table and negativing its value if the angle is greater than 36. To work out the correct cosine value, all we need do is modify the angle slightly (ie subtract 18, ie 90 degrees) and then use SVAL: this is how CVAL is constructed.

The final screen (#94H) contains the actual commands to draw lines (all the turtle does is draw straight lines). Before we can draw a line we need to know the coordinates of the point to which we are drawing. XCO and YCO calculate the new coordinates, given the angle and the number TOS (which is the distance to be moved). To do this we need to know the value of the sine and cosine, which we have stored in the look-up table as integers. We multiply the number TOS by the value from the SIN table, save it as a double length number and then divide that number by 1ØØØØ (that is the meaning of the word */ — the result is saved in X or Y.

The MOVE command is the only drawing command. It expects two numbers on the stack, TOS the distance to be moved and next 1 or Ø (1 to draw, and Ø to move). MOVE is a re-definition.

Screen #C8H is a simple demonstration of the potential of TURGRA, but like Forth the key is expansion. TURGRA is a dummy. If the system is not available (eg we are trying to run the demonstration without having loaded the graphics system) we are instantly told. SIDESQ draws a line, and then turns through a right

angle — the side of a square. To put four SIDESQ together produces a SQUARE, and SQTURN turns and rotates (and makes larger) a square — to be ended by use of the TAB key, or boredom (ie 900).

This is the end, Forth is what you make of it — now you have mastery without mystery.

```
HEX 90 LIST

SCR £ 144       90 H
  0 ( TURTLE GRAPHICS - P 1, VARIABLES )
  1 ( )
  2 : TURGRA ;  ( DELINEATES THE APPLICATION )
  3 ( THESE ARE ALL INITIALIZATIONS )
  4 ( )
  5 : VAR CREATE 0 , ; ( INITIALIZED TO ZERO )
  6 ( )
  7  VAR SIN  (  TABLE ) 872 , 1736 , 2588 , 3420 , 4226 , 5000 ,
  8 5736 , 6428 , 7071 , 7660 , 8192 , 8660 ,
  9 9063 , 9397 , 9659 , 9848 , 9962 , 10000 ,
 10 ( )
 11  VAR ANGLE
 12  VAR PEN
 13  VAR X
 14  VAR Y
 15    -->
OK
```

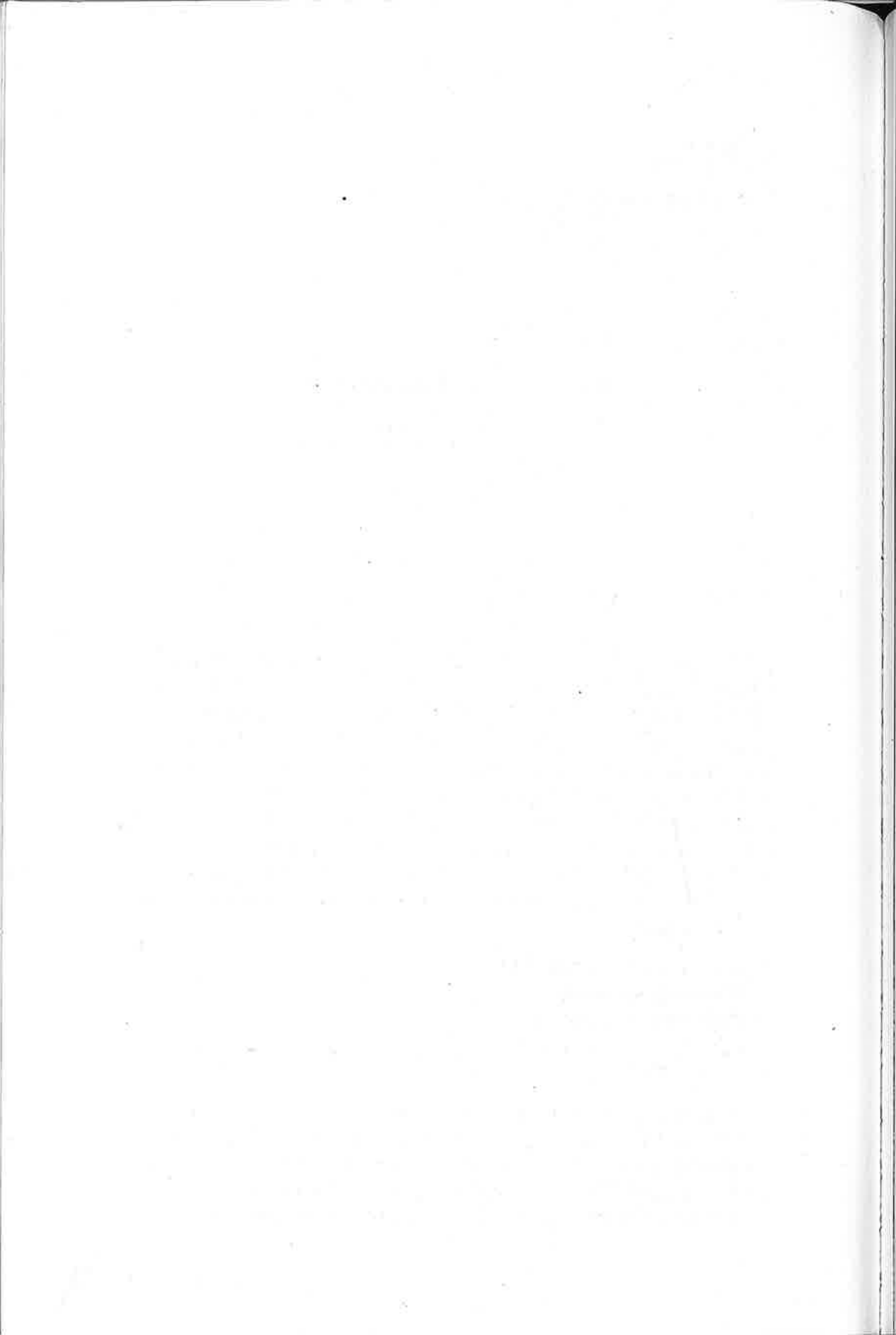
```
HEX 91 LIST

SCR £ 145       91 H
  0 ( TURTLE GRAPHICS - P 2, DISPLAY )
  1 ( )
  2 : ,, >VDU ;    : ;; DUP >< 255 AND SWAP 255 AND ,, ,, ;
  3 : CLG 16 ,, ;    : CLS 12 ,, ;
  4 : COLOUR 17 ,, ,, ;    : GCOL 18 ,, SWAP ,, ,, ;
  5 ( )
  6 ( SET-UP GRAPHICS WINDOW )
  7 : GRASCR 24 ,, 0 ;; 128 ;; 1279 ;; 1023 ;; ;
  8 : ORIGIN 29 ,, 639 ;; 575 ;; ;
  9 : CLRGRA 0 PEN @ GCOL 0 129 PEN @ - GCOL GRASCR CLG ORIGIN ;
 10 ( )
 11 ( SET-UP TEXT WINDOW )
 12 : TEXSCR 28 ,, 0 ,, 31 ,, 39 ,, 28 ,, ;
 13 : CLRTEX 1 PEN @ - COLOUR 128 PEN @ + COLOUR TEXSCR CLS ;
 14 ( )
 15 -->
OK
```

```
HEX 92 LIST

SCR £ 146        92 H
  0 ( TURTLE GRAPHICS - P 3, UTILITIES )
  1 ( )
  2 : PCOL PEN ! ;
  3 : INVERT 1 PEN @ - PCOL 0 PEN @ GCOL ;
  4 ( )
  5 : ALLCLR 4 MODE ;
  6 : CENTRE 4 0 0 PLOT 0 ANGLE ! 0 X ! 0 Y ! ;
  7 ( )
  8 : STARTALL ALLCLR 0 PCOL CLRTEX CLRGRA CENTRE ;
  9 : RESTART 0 PCOL CLRGRA CENTRE ;
 10 ( )
 11    -->
 12
 13
 14
 15
OK
```

```
HEX 93 LIST

SCR £ 147        93 H
  0 ( TURTLE GRAPHICS - P 4, ANGLES )
  1 ( )
  2 : FNANG DUP 0> IF 72 MOD ELSE 72 MOD 72 + THEN ; ( 0-72 ANGLE )
  3 ( )
  4 : TURNTO FNANG ANGLE ! ;
  5 : TURN ANGLE @ + FNANG ANGLE ! ;
  6 : LOC ." X,Y ARE " X @ . Y @ . CR ." ANGLE IS " ANGLE @ . CR ;
  7 ( )
  8 : STANG 36 MOD DUP 18 > IF 36 SWAP - THEN ;
  9 : SVAL DUP STANG 2 * SIN + @ SWAP 36 > IF NEGATE THEN ;
 10 : CVAL 18 SWAP - FNANG SVAL ;
 11    -->
 12
 13
 14
 15
OK
```

```
HEX 94 LIST

SCR £ 148       94 H
  0 ( TURTLE GRAPHICS - P 5, DRAWING )
  1 ( )
  2 ( CALCULATION OF COORDS )
  3 ( )
  4 : XCO ANGLE @ SVAL 10000 */ NEGATE X @ + X ! ;
  5 : YCO ANGLE @ CVAL 10000 */ Y @ + Y ! ;
  6 ( )
  7 : MOVE DUP XCO YCO 0= IF 4 X @ Y @ PLOT
  8 ELSE 5 X @ Y @ PLOT THEN ;
  9
 10
 11
 12
 13
 14
 15
```

```
DECIMAL 200 LIST

SCR £ 200       C8 H
  0 ( TURTLE GRAPHICS DEMO - P 1 )
  1 ( )
  2 TURGRA ( MAKE SURE IT'S THERE )
  3 ( )
  4 : SIDESQ 1 SWAP MOVE 18 TURN ;
  5 : SQUARE 4 0 DO DUP SIDESQ LOOP DROP ;
  6 : SQTURN 900 0 DO Q I MOVE  I SQUARE 6 TURN ?TAB IF LEAVE THEN
  7 LOOP ;
  8
  9
 10
 11
 12
 13
 14
 15
OK
```



SCREEN A
Further with Forth?

> It is a great nuisance that knowledge can only be acquired by hard work. It would be fine if we could swallow the powder of profitable information made palatable by the jam of fiction.
> W Somerset Maugham, *Novels and their authors.*

The type of Forth used on the BBC computer is called Forth-79, and this standard differs from an earlier standard Forth called fig-Forth. Fig stands for Forth Interest Group, and if you become serious about really developing your Forth expertise, join FIG UK. It is possible to purchase (more cheaply than elsewhere) much useful documentation, including implementation notes. You may have noticed the many versions of Forth which are now advertised, most are fig-Forth based on details from that documentation.

Join FIG UK and buy the documentation for both fig-Forth and Forth-79, as Acornsoft Forth does actually draw from both standards. The address

Forth Interest Group UK
Honorary Secretary
15 St Albans Mansion
Kensington Court Place
London W8 5QH

and the name is Keith Goldie-Morrison.

There are many books on Forth, and apart from Richard DeGrandis-Harrison's *Forth on the BBC Microcomputer* (the manual to go with Acornsoft Forth) I cannot recommend any. I do recommend a more general book, which talks of the theory of Forth and other

threaded interpretive languages: *Threaded Interpretive Languages* by R G Loeliger (Byte Books, 1981).

I also recommend *Lewis Carroll's Bedside Book* (Dent, 1979), 'cos it's fun.

Other titles from Sunshine

THE WORKING SPECTRUM

David Lawrence

0 946408 00 9 £5.95

THE WORKING DRAGON 32

David Lawrence

0 946408 01 7 £5.95

THE WORKING COMMODORE 64

David Lawrence

0 946408 02 5 £5.95

DRAGON 32 GAMES MASTER

Keith Brain/Steven Brain

0 946408 03 03 £5.95

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best-selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listings for the Spectrum, Dragon, BBC, Vic 20 and 64, ZX 81 and other popular micros. Only 35p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £8.00 in the UK and £14.00 overseas.

Notes

Printed in England by Commercial Colour Press, London E.7.

There is no better way to learn about how computers really work than to study and practise the logic of Forth. In this novel approach to the language Boris Allan finds a way through Forth, by looking at how it works and how it functions. This is not a manual — Boris Allan is more concerned with the use of Forth in analysing its logical implications and in the use of graphics than with the more mundane numerical applications.

The topics covered include:-

- **Forth words and pointers**
- **the stack**
- **arithmetic and vectors**
- **source and compiled applications**
- **operating system**
- **control structures**
- **logical functions**
- **sound effects**
- **creative graphics**
- **Turtle graphics**

Master Forth and for most applications you can dispense with the need for machine code, such is its speed and dexterity.
The author, Boris Allan, is a regular contributor to a wide range of computer magazines. He also writes a weekly column for Popular Computing Weekly.

ISBN 0 946408 04 1

£5.95 net