Alan Winfield

THE COMPLETE FORTH

A new way to Program Microcomputers

SIGMA PRESS

THE COMPLETE FORTH

A.F.T. Winfield

Sigma Technical Press

ISBN 0-905104-22-6

Published by:

SIGMA TECHNICAL PRESS,
5 Alton Road,
Wilmslow,
Cheshire,
U.K.

Distributors:

Europe, Africa:
JOHN WILEY & SONS LIMITED,
Baffins Lane, Chichester,
West Sussex, England.

Australia, New Zealand, South-East Asia:
Jacaranda-Wiley Ltd., Jacaranda Press,
JOHN WILEY & SONS INC.,
GPO. Box 859, Brisbane,
Queensland 40001, Australia

Printed in Great Britain by
J. W. Arrowsmith Ltd, Bristol

Preface

FORTH is an exciting computer language that was first developed in the early 1970's for scientific applications, but not until recently has FORTH become widely available for microcomputers. Indeed, FORTH has emerged at a time in which microcomputers have 'come of age' and many users have gone beyond the tentative exploratory stages – and are programming for serious and demanding applications.

Most of the existing languages suffer serious limitations; BASIC is too slow for many applications; yet assembler is not user-friendly, is difficult to learn, and worse still, is limited to one processor. FORTH overcomes all of these difficulties to provide a compact and friendly language, with fast execution.

This book is a complete guide to FORTH programming. The first half of the book introduces the language through examples and frequent comparison with BASIC. The later chapters delve into some of the more unusual capabilities of FORTH, many of which have no equivalent in other languages. The FORTH-79 standard dialect of FORTH is adopted throughout the book, although common departures from this standard are detailed as footnotes.

The book is intended for anyone who wishes to learn and use FORTH. Some familiarity with microcomputers and the language BASIC is assumed, but no prior knowledge of FORTH is required. The book should be equally well serve as a useful reference of ideas and techniques for practising FORTH programmers.

I would like to acknowledge Charles Moore and Elizabeth Rather – the inventors of the language, and the FORTH Standards Team – the originators of the FORTH-79 standard dialect used in this book. Grateful thanks are due to Graham Beech of Sigma Technical Press for suggesting the book, Ian Mitchell and Peter Cain for the technical proofing of the manuscript, and my better half Mary for putting up with me during its writing!

Alan Winfield,
Hull.
January, 1983.

Contents

Introducing FORTH viii

How is FORTH Different? viii
How best to read this book ix
For Handy Reference xi
For Computer Professionals xi

Chapter 1 FORTH Fundamentals

1.1 Speak FORTH 1
1.2 The Stack 1
1.3 FORTH Arithmetic 2
1.4 Further FORTH Arithmetic 3
1.5 About the Numbers 4
1.6 Some DUPlication 6
1.7 More Stack Manipulation 7
1.8 Summary and Exercises 8

Chapter 2 The FORTH WORD

2.1 FORTH in Action 11
2.2 The FORTH Error 12
2.3 The FORTH Variable 13
2.4 A Closer look at VARIABLE 15
2.5 The FORTH CONSTANT 16
2.6 Summary and Exercises 17

Chapter 3 The COLON Definition

3.1 Colon Calculations 21
3.2 More Percentages 22
3.3 Colon Definition or Program? 23
3.4 Interpret? .. Compile? 24
3.5 Creating tables and arrays 25
3.6 The Stack Notation Extended 27
3.7 Summary and Exercises 28

Chapter 4 FORTH Structures 1, IF

4.1 True or False? 31
4.2 The IF structure defined 32
4.3 Nested IF structures 33
4.4 Logical Operators for Complex Conditionals 34
4.5 The Missing Comparison operations
4.6 Summary and Exercises 37

Chapter 5 FORTH Structures 2, Loops

5.1 The DO loop 41
5.2 The DO loop in action 42
5.3 Loop Calculations 43
5.4 +LOOP for interesting increments 44

5.5	Nested DO loops and other specialities	44
5.6	The UNTIL loop	46
5.7	The WHILE loop	47
5.8	FORTH structures in action	48
5.9	Summary and Exercises	49
Chapter 6	**Editing, Saving and Loading FORTH programs**	
6.1	The FORTH LOADing concept	53
6.2	The Editor	55
6.3	More BLOCK handling	58
6.4	Vocabulary Management	60
6.5	Summary	62
Chapter 7	**Number and String Input and Output**	
7.1	Character input-output, the basics	65
7.2	String input-output 1	66
7.3	String input-output 2	68
7.4	Number Bases	70
7.5	Alternative number input	71
7.6	Summary	72
Chapter 8	**Double Precision and beyond**	
8.1	Double Precision Numbers	75
8.2	Mixed Precision	77
8.3	The Return stack for High Speed Definitions	78
8.4	Formatted Number Output	80
8.5	Fixed Point Arithmetic	82
8.6	Summary	84
Chapter 9	**Extending FORTH**	
9.1	Defining new Defining words	87
9.2	The last word on ARRAYs	89
9.3	A STRING variable	90
9.4	Self Modifying Data structures	93
9.5	A Closer Look at the Dictionary	94
9.6	Defining new Compiling words	97
9.7	Summary	100
Chapter 10	**FORTH Finale**	
10.1	The Calendar Vocabulary	103
10.1.1	Zeller's Congruence	103
10.1.2	Daynumber and day	105
10.1.3	Month, year and daysleft	106
10.1.4	The Calendar Vocabulary blocks listed	107
10.2	A Video Game Vocabulary	109
10.2.1	The Ball handling routines	110
10.2.2	Bat handling	112
10.2.3	The Squash game	112
10.2.4	The Videogame Vocabulary blocks listed	114
Bibliography		116

Answers to Problems 117
Glossary of FORTH Terminology 123
Index
FORTH Handy Reference Card

We could even request a calendar for the whole year by typing simply:

```
1982 year
```

Having specified our ideal end result we must now write a FORTH program for each of the functions 'day', 'daysleft', 'month' and 'year'. The 'year' program would probably look something like this:

```
: year
        12 0 DO                 ( loop for twelve months )
                I OVER month    ( print each month )
             LOOP
        DROP
;
```

The program has been placed inside a special FORTH structure called a 'Colon definition' and given the name 'year'. Providing that FORTH already recognises the word 'month', all of the above may be typed in, and will be compiled and added into FORTH, with the name 'year'. The 'year' program has now become a part of FORTH that may be run at any time by simply typing, for example:

```
1982 year
```

What could be easier!

Of course, before typing in 'year', we must already have entered a program for 'month'. Again, this will be in the form of a colon-definition:

```
: month       ... a FORTH program ...   ;
```

This time the program inside the colon definition is likely to be written entirely in standard FORTH and may be typed directly into a standard FORTH system.

Don't worry if you do not understand the actual syntax of the examples given above or terms like 'compile'. All of this will be explained during the course of the book, including more detailed programs for the 'calendar' words above. The important message of this introduction is that the FORTH programmer builds a special 'vocabulary' of functions, (a calendar vocabulary in the example above). Simply typing one of the words in the vocabulary will cause the corresponding program to be run. The finished vocabulary may then be stored on disk or cassette.

For the hobbyist and professional alike this is an interesting and refreshing alternative approach to programming. For the computer professional I will say more at the end of this introduction, but now a few words about this book.

How best to read this book

Like any new programming language FORTH must be learned from the ground floor up. There may be quite a few floors in the FORTH building, but the effort is exceedingly worthwhile, as I hope I showed earlier in this introduction! Nevertheless, FORTH is an interactive language, meaning that programs may be developed and tested 'at the keyboard', or, what is more useful at this stage, FORTH may be learned 'at the keyboard' as well. This means that as each new facility is explored we may actually try out the facility on a microcomputer running FORTH, and that is true right from the very beginning!

This book is an ideal accompaniment to a brand new FORTH system running on

your microcomputer, but don't worry if you do not have a microcomputer to hand, the examples will make sense anyway. Virtually all of the FORTH which appears throughout the text may be typed in, and accepted by most of the standard FORTH systems currently available. If your system conforms to the FORTH-79 standard (produced by the FORTH Standards Team), then all of the examples will run without modification. If not, you may have to consult the documentation for your system, to identify any differences.

In all of the examples which are suitable for trying out on a FORTH system I have indicated the response from FORTH in italics. The part which you actually type in is not italicised. Apart from this, there are only two additional points to be remembered when talking to a FORTH system. These are:

i) FORTH doesn't start to execute any of your typed input until after you hit 'return'. This is a single key on the far right hand side of the keyboard, normally labelled 'return', or sometimes 'newline'.
This is called 'buffered' input, and has the great advantage that if you make a typing error you are able to correct it by using the 'backspace' key.

ii) FORTH likes each number, or symbol in the input to be separated by at least one space.
The reason for this will be explained later.

Rather than state these conventions each time an example occurs, we will just assume them, so that for example:

```
." I AM FORTH "  I AM FORTH ok
```

means that you typed `." I AM FORTH"` and then hit 'return', and FORTH responded by printing *I AM FORTH ok*. The final 'ok' is FORTH's way of saying "I've finished processing that line of input and I am ready for the next".

Chapters one to five inclusive form a self contained introductory course in FORTH programming which requires only a basic familiarity with computer concepts and terminology. For readers familiar with BASIC, examples of FORTH and BASIC are given side by side where appropriate (and possible!). Also a short set of exercises is included at the end of each chapter, with full solutions at the end of the book.

Chapter six covers the FORTH editing and disk (or cassette) handling operations. FORTH editors differ considerably from system to system and so this chapter presents only a typical set of editing operations.

Chapters seven, eight and nine present a selection of some of the more exotic and sometimes obscure techniques of FORTH programming. The chapters are not essential reading for the absolute beginner to FORTH but are intended more as reference material for the practising FORTH programmer, who will, it is hoped, delve into these chapters for hints or ideas to incorporate into his own programs. The newcomer to FORTH is, nevertheless, recommended to skim through these chapters, to whet the appetite and to become aware of some of the more unusual capabilities of FORTH, many of which have no parallel in most other computer languages.

Finally, chapter ten presents two major FORTH programs which are both interesting programs in themselves, and provide examples of how a FORTH programmer approaches the design of large programs.

For Handy Reference

Included in this book is a tear-out FORTH handy reference card, which gives very brief details of all of the words and symbols which FORTH recognises. In FORTH terminology this is called the 'Dictionary'. If you have a FORTH system on your micro-computer, it is unlikely that the dictionary is identical to the one in the handy reference, but they will probably be very similar. The notation used in the handy reference for summarising the action of each FORTH word, or symbol, may be confusing at first, but will be explained in some detail in chapter one. As soon as you start writing your own FORTH programs, which should be very shortly, you will find this handy reference invaluable. It is recommended that you should refer to this, when trying out examples as soon as possible after chapter one. Likewise, if any of the terminology needs clarification a glossary of FORTH terminology is included at the end of the book – which may be easier to use than digging up the appropriate section in the text.

The remainder of this chapter is a summary of FORTH, for computer professionals. If you don't speak 'computerese' then you can easily skip this section and go straight into chapter one!

For Computer Professionals

By any standards FORTH is a most unusual computer language. Certainly FORTH has little in common with mainstream languages such as BASIC or Pascal. Nevertheless, FORTH is a high-level language; it embodies structured programming concepts, and FORTH programs are both modular and portable. At the same time, the FORTH programmer has access to primitive operations, or symbolic assembler if needed – so in some respects FORTH may be likened to a macro-assembler.

A FORTH system is both an interpreter and compiler. Normally, all input to FORTH (which may come from either the keyboard, or backing storage), is interpreted and executed directly. However, if the same input is enclosed inside a colon-definition (as illustrated earlier in this introduction), then it is compiled into a compact threaded code. Thus, FORTH has the unusual feature of providing an interactive interpeter like environment for testing and debugging programs, which is friendly and easy to use, but at the same time the final programs are truly compiled and therefore fast and efficient. Runtime speeds comparable to compiled Pascal, or better than ten times faster than interpreted BASIC, can easily be achieved in FORTH.

Another feature of FORTH which results in short development and debugging times is the unusual nature of programming by extending the language. All FORTH operations (which may be likened to the 'reserved' words of BASIC – 'LET', 'PRINT', '+', '-' etc.), are contained in a dictionary. Programming proceeds by defining new words using the existing set and adding these new words to the dictionary using the special 'colon-definition' construct. These new words are in turn used to develop still more complex operations and in this way the FORTH programmer builds a special 'vocabulary', which is tailored to his problem. Each new operation is fully tested at the keyboard before proceeding to the next – and in this way bugs are caught and cured early!

Most complete FORTH systems will already have a number of special purpose 'vocabularies' built in, editor, assembler, and disk handling vocabularies are examples. This means that a FORTH system is normally completely self-sufficient – no other development software is needed whatever. Furthermore, the whole system is fairly compact, typically under 16k bytes. This is of particular advantage to the software engineer, since he can often use the target computer system as the development system as well.

In case the above description has made FORTH sound like the answer to every programmer's dream (!), let me describe briefly some of the features which some programmers may find less attractive...

The first is the use of a stack, and as a result, Reverse Polish notation. To a large extent these features are a part of the fundamental structure of FORTH and certainly contribute to the speed and compactness of FORTH. The stack also results in some very neat ways of passing parameters into programs. As an example, in the 'calendar' operations outlined early in this chapter – to print out a whole year calendar the user simply types, say:

```
1982 year
```

When FORTH interprets this line of input it 'pushes' the number 1982 onto the stack (as it does for any numbers in an interpreted input stream). The program 'year' then 'pops' the number off the top of the stack, and uses that as its parameter.

The second, possibly controversial, feature is the use of integer arithmetic. The FORTH philosophy here is that integer arithmetic is much faster than floating-point, and the majority of applications only need integers anyway. For those applications where decimal arithmetic must be used, FORTH provides a set of operations with double-precision (32 bit) integers (±2,147,483,648), which may be used to implement fixed-point arithmetic.

To conclude this introduction, it is worth remarking that FORTH is not an easy language to master. Surprisingly, FORTH is easiest for absolute beginners to computing! For readers like myself, who were raised on algebraic languages such as Pascal and BASIC – learning FORTH means learning a completely new and fascinating approach to programming.

1
FORTH Fundamentals

At its very simplest a FORTH system may be used like a calculator, to evaluate arithmetic expressions and directly print out the results. Although this seems a humble beginning, it does demonstrate the unusual way in which FORTH uses a 'stack', and as a direct consequence, 'Reverse Polish' notation. This chapter introduces and explains these two concepts, and develops a notation for describing the stack which will be used throughout the rest of the book.

1.1 Speak FORTH

Imagine being seated in front of a computer which speaks FORTH, and suppose, for example, that you would like some help in multiplying the two numbers 23 and 34. In FORTH you would type:

```
23 34 * .
```

to which FORTH will agreeably respond,

```
782 ok
```

It is clear, therefore, that we have achieved the same result as typing `PRINT 23*34` in BASIC. You will have noticed, however, the peculiar position of the `(*)` multiplication symbol in the FORTH version of this operation, namely, after the two numbers which are to be multiplied, rather than between them as is the normal convention. Additionally, what is the significance of the full stop `(.)` symbol in the FORTH input?

The answer to these questions lies in the realisation that FORTH uses a 'Stack' for arithmetic.

1.2 The Stack

Let's try a simpler example than the one above, just type a single number:

```
27
```

FORTH will respond with the reply 'ok', on the same line:

```
27 ok
```

and nothing appears to have happened (except that FORTH seems to think it's 'ok'!). But actually something rather crucial has happened, namely, the number '27' has been 'PUSHed' onto the stack. The symbol full stop `(.)`, which we encountered before has exactly the opposite effect – it 'POPs' the number off the top of the stack, and prints it out:

```
.
```

and FORTH will print:

```
27 ok
```

So, the stack had the effect of remembering a number for us as if we had jotted it down on a notepad.

Let me explain the STACK, and the operations of PUSH and POP in a little more detail. If you are already familiar with the operation of a stack you can easily skip through to the next section.

A stack is simply a special type of storage buffer for numbers, in which numbers are 'pushed' onto the stack, for storage, and later retrieved by 'popping' back off the stack. The last number to be pushed onto the stack will be the very first to be popped back off it, and so the stack is often called a Last-in First-out or LIFO store.

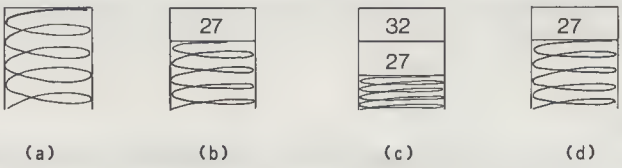


Figure 1.1 The Spring Loaded Stack

A good way of picturing a stack is like a spring loaded plate store, of the type often used by cafeterias. An empty stack will look like (a) in figure 1.1. Push 27 onto the stack and it will appear like (b). You can see that there's plenty of room still left in the stack, so we could push another number onto the stack, as in (c). Pop the number off the stack in (c), and the number 32 is retrieved, and the stack reverts to (d), exactly as it was in (b).

1.3 FORTH Arithmetic

Let me now examine, in more detail, the multiplication example used in the previous section by illustrating the contents of the stack before and after each number and symbol in the FORTH expression:

```
23 34 * .
```

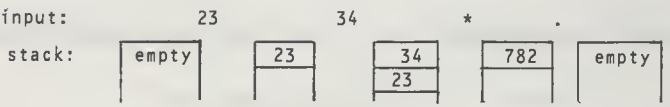


Figure 1.2

Reading figure 1.2 from left to right, we can see that FORTH simply pushes numbers onto the stack whenever they occur in the input. Thus, by the time we reach the symbol `{*}`, the stack already has the two numbers 23 and 34 on it. FORTH responds to `{*}` by popping the top two numbers off the stack, multiplying them, and then pushing the result back onto the stack. The stack after the `{*}` just has the result 782 on it. The final symbol `{.}`, as explained already, simply pops the result off the stack and prints it.

So, we are now in a position to write down the first rule of programming in FORTH; it is:

All FORTH arithmetic is executed on a stack.

Let me go a stage further, and say that all arithmetic operations work on the numbers on the stack, and place their results back onto the stack. This now explains the peculiar order of:

```
23 34 * .
```

instead of the usual `PRINT 23*34`.

1.4 Further FORTH Arithmetic

A question you may well be asking yourself, at this stage, is "Doesn't this mean that if I want to do complicated arithmetic in FORTH, I will have to change around the expression first in order to make it work on the stack?". The simple answer to this question is "yes", you do have to alter the expression before entering it into FORTH, but that process is very easily learned. Let's take as a simple example, the expression:

```
(1 * 2) + (3 * 4)
```

and consider how to evaluate this expression mentally. We say "Oh, that's simple, it's the sum of 1 multiplied by 2 (=2) and 3 multiplied by 4 (=12). So the answer is 2 + 12 which equals 14".

What we really did then was calculate 1 * 2, and save the result for later, then calculate 3 * 4, and finally add the two results together. Let's write that down, in FORTH:

```
1 2 *  3 4 *  +
```

Figure 1.3 shows exactly how FORTH executes this, to produce the correct result of 14.

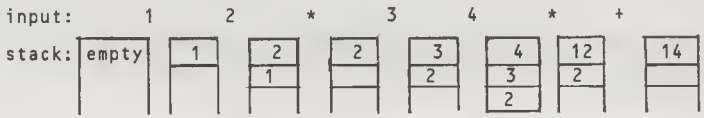


Figure 1.3

Notice the clever way in which FORTH saves the result of the first multiplication, on the bottom of the stack, until the second multiplication is complete and the addition can take place.

Ordinary arithmetic notation is often called 'infix' because the operators (`+`, `-`, `*`, `/` etc.) are fixed in-between the numbers. FORTH arithmetic is called 'postfix', or, more commonly, 'Reverse Polish', after the Polish logician who invented the notation. In reverse Polish each arithmetic operator comes after the numbers upon which it operates (termed 'operands'). Converting from ordinary arithmetic notation to reverse Polish is simply a matter of looking at the expression and deciding how you would evaluate it on paper. Once you've decided the order in which to evaluate the individual operations, it is highly likely that FORTH will also best evaluate them in the same order. Notice that the operands in a reverse Polish expression remain in the same order in which they occur in the equivalent 'infix' expression; only the operators change position. Finally, you should try the

expression with a picture of the stack, to make sure that FORTH will really get it right.

All of this talk of infix, and Reverse Polish may, by now, have you wondering "Why have I bothered with FORTH at all, since most other computer languages like BASIC and Pascal understand ordinary arithmetic anyway". This is certainly a fair criticism and is best answered by considering that reverse Polish arithmetic is very easy to execute and FORTH arithmetic is, as a result, very fast, certainly much faster than BASIC arithmetic. The stack in FORTH is, however, used for far more than just the execution of arithmetic. In fact, almost all FORTH operations use the stack to 'pass parameters' (that is, get input values, and deposit output results). The use of Reverse Polish in arithmetic follows naturally from the fact that FORTH is a stack-orientated language.

1.5 About the Numbers

In all of the examples so far the numbers have been whole numbers or, to use the correct term, *integers*. The reason for this is that FORTH arithmetic works on integers only. In FORTH we cannot have numbers like 3.14 E -2 (which is the same as 0.0314), properly termed 'floating-point' numbers.

This is not such a dreadful limitation as it might at first appear, because FORTH will allow us to handle 'fixed-point' decimal numbers – like 100.23 or 1.234 – using a set of double-precision arithmetic operations which will be explained in detail in chapter 8.

But for the moment let us confine ourselves to integers. FORTH will handle negative, as well as positive, integers in the range:

```
-32,768 to +32,767
```

so that `-9999`, `-1`, `0`, or `10000` are all examples of valid FORTH numbers. In FORTH terminology, values in this range are 'signed single precision numbers', and are represented on the stack as 16 bit binary (beginners should look up the entry on two's complement arithmetic in the glossary of FORTH terminology for a more detailed description).

Alternatively, FORTH will allow us to enter 'unsigned single precision numbers' in the range:

```
0 to 65,535
```

This is useful if we should require an extended positive range, but do not require a negative range. Notice that `.` will not print the correct value for unsigned numbers greater than 32767:

```
50000 . -15536 ok                ( wrong !! )
```

Instead we must use the 'unsigned' number print operation `U.`:

```
50000 U. 50000 ok
```

Most FORTH arithmetic operations will work for unsigned numbers providing that the result of the operation is within the unsigned number range, for example:

```
50000 67 + U. 50067 ok           ( 50000 + 67 )
40000 1 - U. 39999 ok            ( 40000 - 1 )
20001 3 * U. 60003 ok            ( 20001 * 3 )
```

But care should be exercised here!

Throughout this book we shall employ the FORTH convention that the term 'number' implies 'signed single precision number'. Whenever we are referring to unsigned numbers this will be explicitly stated.

The use of integers means that division in FORTH will sometimes give an answer which is not quite correct. For example, dividing 11 by 3:

```
11 3 / . 3 ok
```

gives the answer 3, whereas the true answer is 3 with a remainder of 2. To get round this there is a FORTH operation {/MOD}; a special form of division which leaves the remainder on the stack as well as the actual result (quotient). So if we POP and print both of the results from the stack after a {/MOD}, as in:

```
11 3 /MOD . . 3 2 ok
```

then we get the complete answer e.g. 3 remainder 2. Figure 1.4 shows the stack as this example is executed.

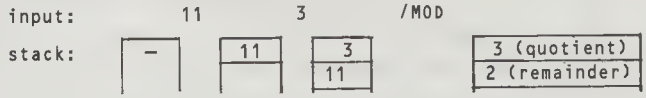


Figure 1.4 The {/MOD} operation

If we require only the remainder from a division, the FORTH operation {MOD} should be used:

```
11 3 MOD . 2 ok                    ( calculate remainder only )
```

Two more unusual arithmetic operations are {MAX} and {MIN}. Both need two values on the stack, and leave a single value; {MAX} leaves the largest of the two values, and {MIN} the smallest. For example:

```
10 20 MAX . 20 ok
-5 5 MIN . -5 ok
```

{MAX} and {MIN} both take note of the 'sign' of the two values, using the normal convention that negative numbers are 'less than' positive numbers.

Finally, FORTH has two 'sign' changing operations, {ABS} and {NEGATE}. {ABS} has the effect of making the number on top of the stack positive whatever its sign, for example:

```
100 ABS . 100 ok
-2 ABS . 2 ok
```

Numbers that are already positive are not affected by {ABS}. {NEGATE} has the effect of always reversing the sign:

```
100 NEGATE . -100 ok
-2 NEGATE . 2 ok
```

1.6 Some DUPlication

Suppose that we wish to add up a set of 4 numbers, but instead of producing a single result, print out a cumulative result at each stage in the calculation. So that for example, adding 1 plus 2 plus 3 plus 4 will print out:

```
3
6
10
```

where 3 is the result of 1+2, 6 is the result of 1+2+3, and 10 is the result of 1+2+3+4.

The FORTH version of the calculation could be as in figure 1.5.

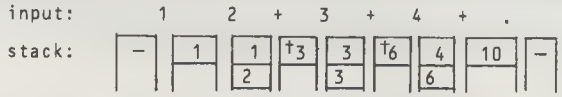


Figure 1.5

If we try typing the input in figure 1.5 it will indeed produce the final result of 10, but will not print out any of the required intermediate results. Looking at the picture of the stack contents in figure 1.5, we see that the intermediate results do, in fact, appear on the stack during the calculation, at the positions indicated by †. What we require is a method of printing out these intermediate values without actually removing them from the stack, and the FORTH word `DUP` will help.

`DUP` is not an arithmetic operation, it is one of a special set of FORTH operations called 'Stack Manipulations'. `DUP` has the effect, when executed, of DUPlicating the number on the top of the stack, so that, for example:

```
5 DUP
```

will result in the number 5 as the first and second items on the stack as in figure 1.6.

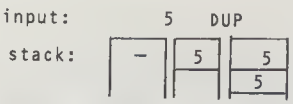


Figure 1.6

Let me, at this stage, introduce a new shorthand notation for picturing the stack before and after the execution of a FORTH operation:

```
input:          DUP
stack action: (n  →  n  n)
```

The 'stack action' is in the format:

```
(stack before   →   stack after)
```

Either side of the arrow (→) is a list of the stack contents, the rightmost element in the 'stack before' list is the item on top of the stack before the operation is executed, and the rightmost element in the 'stack after' list is the item on top of the stack after the operation. If this does not seem too clear, it will after a few more examples!

However, to get back to our problem, it should now be evident that to print the number on top of the stack without actually losing it we use the FORTH:

```
DUP  .
```

If there is only one number on the stack, then in our new stack notation {.} can be pictured as:

```
( n → )  and print n
```

If there happen to be two numbers on the stack, {.} will print the top number and leave the second number on top of the stack:

```
( n2 n1 → n2 ) and print n1
```

The combined effect of {DUP} and {.} is, therefore, to leave the stack unchanged, however many numbers it contains, but also print a duplicate of the number on top of the stack.

```
          DUP              .

( n    →      n   n    →    n )  and print n
```

We can now rewrite the solution to our cumulative sum as:

```
1  2  +  DUP  .  CR  3  +  DUP  .  CR  4  +  .
```

And this will print exactly the result required -

```
3
6
10 ok
```

Notice that I've also included the FORTH operation {CR} which prints a carriage-return line-feed on the terminal and therefore puts each result on a separate line. {CR} does not affect the stack at all:

```
  CR
( → )
```

1.7 More Stack Manipulation

Most of the stack manipulation operations are designed to overcome one of the main limitations of a stack, which is that the order of the numbers on a stack is always the reverse order in which the numbers were pushed onto the stack. Normally this means that we can only ever pop numbers in a last-in first-out fashion, for example:

```
100 200 300 ok
. . . 300 200 100 ok
```

prints the numbers in reverse order. The FORTH stack manipulation operations overcome this limitation. For example, the duplication words {DUP}, {OVER} and {PICK} allow us to pick out any number from within the stack and push a new copy of it onto the top of the stack. So in:

```
100 200 300 ok
3 PICK .  100 ok
```

the operations {3 PICK .} have picked out and printed the third item on the stack, without changing the stack in any way. Figure 1.7 shows the effect of {3 PICK} on the stack. {OVER} picks out the second item on the stack, and pushes a duplicate of it onto the stack, {OVER} is therefore the same as {2 PICK}.

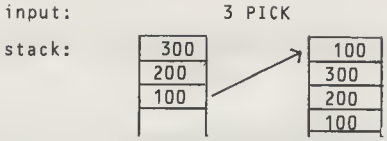


Figure 1.7 The operation {PICK}

The stack re-ordering words {SWAP}, {ROT} and {ROLL} allow us to change around the order of the numbers on the stack. For example in:

```
100 200 300 ok
SWAP ROT ok
. . . 100 200 300 ok
```

the operations {SWAP ROT} have completely reversed the order of the three numbers on the stack. Figure 1.8 shows the stack during the phrase {SWAP ROT}.

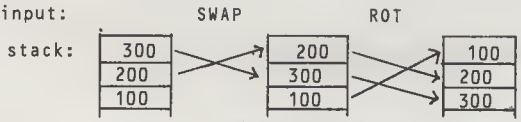


Figure 1.8 {SWAP ROT}

{ROLL} is the more general form of stack re-ordering operation. {n ROLL} will have the effect of rotating the nth item on the stack up to the top of the stack. {2 ROLL} is thus the same as {SWAP}, and {3 ROLL} the same as {ROT}.

Finally, the operation {DROP} is worth mentioning. {DROP} has the effect of simply losing the number off the top of the stack – like {.} but without actually printing it.

1.8 Summary and Exercises

Here is a summary of the FORTH operations covered in this chapter, followed by a set of practice problems in the use of these operations. Suggested solutions are given at the end of the book.

In the stack descriptions of the following operations 'n' indicates a single precision number, and 'un' an unsigned single precision number. Also note that n1, n2 etc. does not indicate the order of numbers on the stack. Top of stack is always on the right of each of the stack-before and stack-after lists.

The recommended pronunciation is given, on the right hand side of the page, in "quotes", for those words whose pronunciation may not be obvious. For example {DUP} is pronounced "dupe".

Stack Manipulation:

DUP (n → n n) "dupe"

Duplicate the number on top of the stack.

DROP (n →)

Discard the top number from the stack.

`SWAP` `(n1 n2 → n2 n1)`

Exchange the top two stack numbers.

`OVER` `(n1 n2 → n1 n2 n1)`

Duplicate the second number on the stack.

`ROT` `(n1 n2 n3 → n2 n3 n1)` `"rote"`

Rotate the top three numbers bringing the third to the top.

`PICK` `(n1 → n2)`

Duplicate the n1'th number down on the stack. n1 must be greater than zero.

`ROLL` `(n → )`

Rotate the n'th item (not counting n itself) up to the top of the stack. n must be greater than 1.

Arithmetic:

`+` `(n1 n2 → sum)` `"plus"`

Add n1 and n2 leaving the sum.

`-` `(n1 n2 → diff)` `"minus"`

Subtract n2 from n1 leaving the difference.

`*` `(n1 n2 → prod)` `"times"`

Multiply n1 and n2 leaving the product.

`/` `(n1 n2 → quot)` `"divide"`

Divide n1 by n2 leaving the quotient which is rounded toward zero.

`MOD` `(n1 n2 → rem)` `"mod"`

Divide n1 by n2 leaving the remainder with the same sign as n1.

`/MOD` `(n1 n2 → rem quot)` `"divide-mod"`

Divide n1 by n2 and leave the remainder and quotient. The remainder has the same sign as n1.

`MAX` `(n1 n2 → max)` `"max"`

Leave the greater of the two numbers n1 and n2.

`MIN` `(n1 n2 → min)` `"min"`

Leave the lesser of the two numbers n1 and n2.

`ABS` `(n → |n|)` `"abs"`

Leave the absolute value of n (reverse the sign of n if it is negative, otherwise leave it unaltered).

`NEGATE` `(n → -n)`

Reverse the sign of n. (Two's complement.)

Output printing:

`.` `(n → )` `"dot"`

Print n (in current base – see chapter 7.4) as a signed single precision number with one trailing space.

`U.` `(un → )` `"u-dot"`

Print n (in current base) as an unsigned single precision number with one trailing space.

`CR` `( → )` `"c-r"`

Print a carriage-return and line-feed.

Exercises

1) Write the following arithmetic expressions in FORTH:

```
(1 + 2) * (3 - 4)
10 + 100/9 + 5
2 * (3 * (4 * (5 + 6)))
```

2) Convert the following FORTH expressions back into 'infix':

```
20 10 +  20 10 -  /
1 2 3 4 + + +
20 1 2 * -
```

3) Show how the stack will be affected by the following operations, assuming in each case that the stack is initially empty:

```
100 -200 ABS MAX
-10000 0 MIN NEGATE
1 2 SWAP OVER
10 DUP DUP * *
10 20 30 40  3 PICK  +
```

4) How would you calculate the sum, difference, product and quotient of two numbers, without having to re-enter the two numbers for each separate calculation? (Hint – you must use the stack duplication words).

2
The FORTH WORD

It may seem unusual to ask the question "What does FORTH actually do when it executes a line of input?" so early in this book, but FORTH is an unusual language and the answer to this question is not complex, but it will increase our understanding considerably and make programming in FORTH that much easier. This chapter describes a simple model of a FORTH system as it executes a line of input. At the same time, some useful terminology is introduced; terminology that will be used often throughout the rest of the book. Knowing how FORTH works enables us to predict some of the things that can go wrong and so the chapter continues by introducing FORTH error handling. Finally, we extend our vocabulary to include variables, constants and arrays and thereby introduce the important concept of the 'defining word'.

2.1 FORTH in Action

All of the FORTH operations which we have discovered so far (the ones I've enclosed in {curly brackets}) are called, in FORTH terminology, WORDs. Even the single character symbols like {*} or {.} are FORTH words. Each word is contained in the FORTH dictionary so that when FORTH *interprets* a line of input each word in the input stream is looked-up in the dictionary. Figure 2.1 describes two of the entries in the dictionary, {*} and {.}.

```
Word     Definition..

  *      (n1 n2 → prod)    Multiply n1 by n2
  .      (n → )            Print n
```

Figure 2.1 Two DICTIONARY entries

Just like an ordinary dictionary, the FORTH dictionary contains words, and for each word a definition. The definition specifies the ACTION of the word when executed. Figure 2.1 shows the definition both as a verbal description, and more precisely using the (stack before → stack after) notation proposed in chapter 1.6. The FORTH Handy Reference (at the end of the book) shows the FORTH-79 standard dictionary, which contains about 130 words, in the same format.

Let us imagine that the dictionary contained only the two words {*} and {.}, and examine in more detail how FORTH interprets our simple multiplication example:

```
23 34 * . 782 ok
```

FORTH will go through (very quickly) the following steps after we press return:

i) FORTH finds the first word in the input stream, which is {23}, and searches the dictionary for a match. {23} is *not* in the dictionary so FORTH assumes instead that it is a number. Indeed {23} is a valid decimal number, so it is converted into binary and pushed onto the stack.

ii) Likewise, the second word in the input stream, {34} is not found in the dictionary, so again it is assumed to be a number and pushed onto the stack.

iii) The third word in the input stream {*} is found in the dictionary so the word is executed causing the two numbers on top of the stack to be multiplied, and the result pushed back onto the stack, as defined by the dictionary definition of {*}.

iv) The fourth word is {.}, again this word is successfully found in the dictionary and executed, causing the number on top of the stack to be printed.

v) There is no more input left, so FORTH prints the message 'ok' and waits for another line of input.

We can now write the second rule of programming in FORTH:

> All input to FORTH consists of a sequence of words. Each word must be either in the dictionary, in which case it is executed, or a valid number, in which case it is pushed onto the stack.

2.2 The FORTH Error

The above description of FORTH in action begs the question – "What happens if you type a word which is not in the dictionary, and not a number either?". Well, if we type something like:

```
PQRXYZ  PQRXYZ ?
```

FORTH simply replies with the message 'PQRXYZ ?' which means, predictably, that {PQRXYZ} is not in the dictionary, and it's not a number either! The FORTH error message '?' is similar to the BASIC 'Syntax error' message, except that FORTH helpfully prints out the word in the input which it doesn't understand. This is useful if you have a long input line with an error in the middle, for example:

```
1 !2 * 3 4 * +  !2 ?
```

An additional rule of FORTH is that each word in the input must be separated by at least one space. It is doubly important that this is observed, since confusing errors can occur if it is not; for example, missing the space between a number and a valid FORTH word, or between two FORTH words:

```
23 34* .  34* ?
```

FORTH treats {34*} as one word, and cannot find it in the dictionary or interpret it as a number, so we get the syntax error message '?'.

Missing the space between two numbers is even worse:

```
2334 * . 0 STACK EMPTY
```

FORTH reads the first word as {2334}, and pushes the number two thousand three hundred and thirty four onto the stack. But when FORTH comes to execute the multiplication {*}, which needs two numbers on the stack, there is only one and so FORTH prints the error message 'STACK EMPTY'. Of course, if the stack had any numbers left over from previous operations still on it, then {*} would probably use one of these for the missing operand and produce a confusing result. For this reason it is a good idea to clear the stack down from time to time by simply typing {.} a few times until you get the message STACK EMPTY.

Notice also from the above example that a spurious zero has been printed before

the error message. This is because {.} has actually printed a number from off the end of the stack, which is usually a zero.

The error condition STACK EMPTY is one of the most common error conditions in FORTH, and because it is peculiar to the use of the stack, there is no equivalent to STACK EMPTY in BASIC. STACK EMPTY occurs whenever there are less numbers on the stack than a FORTH operation needs in order to execute correctly. Almost all operations need input arguments or parameters and could potentially cause the error condition STACK EMPTY. The stack notation (stack before → stack after), defined in chapter 1.6, tells us exactly how many arguments an operation needs. For example:

{.} (n →)

needs one argument,

{*} (n1 n2 → prod)

needs two arguments, and

{ROT} (n1 n2 n3 → n2 n3 n1)

needs three arguments. Later in the book I will describe a technique for keeping a note of the contents of the stack whilst writing a FORTH program – and thereby minimising the likelihood of STACK EMPTY.

2.3 The FORTH Variable

Chapter 1 introduced the stack, and described how all FORTH arithmetic is performed on the stack. In addition, we saw how temporary results may be saved on the stack for later use, or how, using the stack manipulation words, a result may be used by more than one FORTH operation. The stack may be thought of as a useful short-term memory, or as a scratchpad for doing rough work in, but is clearly not suitable for long-term storage of numbers. Instead we must use the *variable*.

The BASIC programmer is familiar with the use of variables, since in BASIC almost all arithmetic is performed between variables. The BASIC statement:

```
LET A1=100
```

simply sets the variable 'A1' to the value 100. The equivalent FORTH statement is:

```
100 A1 !
```

but if you were to type this, FORTH would reply with the error message 'A1 ?', the reason being that in FORTH we must define the variable first, using the word {VARIABLE}:

```
VARIABLE A1 ok
```

This has the special effect of reserving a memory location labelled 'A1'. BASIC has no equivalent to this since, in BASIC, variables are implicitly pre-defined. In FORTH this is not so and a variable must be declared, or 'defined', before it may be used. In case this is not too clear, imagine that you are writing a BASIC program and need a new variable which you decide to call 'I'. You would simply write 'LET I= ...'. In FORTH you must define the variable before you can assign a value to it or use it in further operations, by typing:

```
VARIABLE <name>
```

to create a variable named <name>.[1] It is worth noting that there are no restrictions on the number of characters in <name>, or the characters themselves. Some FORTH systems place the whole variable name into the new dictionary entry, but others save only the first three or four characters, in which case you must ensure that these are unique (the documentation for your particular system will tell you more about this). Either way we can use meaningful variable names like:

```
VARIABLE Year ok
```

The FORTH input:

```
1982 Year ! ok
```

sets 'Year' to the value '1982'. Here {1982} is pushed onto the stack, {Year} gives the name of the variable, and {!} stores the number on top of the stack into the variable.

In the FORTH input:

```
Year @ . 1982 ok
```

the word {@} has the converse effect of fetching the value of the variable {Year}, and pushing it onto the stack. {.} then prints it out, so we have the FORTH equivalent of the BASIC 'PRINT Year' (which some BASIC's would not allow!). The phrase {@ .} is used so often that FORTH has a special operation {?} with exactly the same effect, so that we could simply type:

```
Year ? 1982 ok
```

to print the value of the variable.

The words {!} and {@} are called "store" and "fetch" respectively, and may be used together to perform arithmetic on variables, for example:

```
A1 @  1 +  A1 ! ok
```

is directly equivalent to the BASIC, 'LET A1=A1+1'. This FORTH example does not seem quite so peculiar if you consider that the two words {A1 @} fetch the value of A1 onto the stack, and the two words {A1 !} store the value on top of the stack in A1. The whole line simply breaks down into the three operations:

i) Fetch the value of A1 onto the stack.

ii) Add one, using FORTH stack arithmetic.

iii) Store the value on the stack, back into A1.

We may write any calculation involving variables in FORTH. Figure 2.2 illustrates how the BASIC 'LET A=2*X + Y' would be written, in FORTH, assuming that A,X and Y have been pre-defined.

[1]Some FORTH systems require an initial value to be supplied when the variable is defined, so that:

```
n VARIABLE <name>
```

defines a variable named <name> with initial value *n*.

```
BASIC:    LET A = 2 * X  +  Y

FORTH:    2  X @  *  Y @  +  A !
```

Figure 2.2 FORTH Variable Arithmetic

It is certainly fair comment to say that FORTH variable arithmetic is somewhat peculiar, but it is also true that variables are used far less often in FORTH than in other languages. The reason for this will become clear later.

Now, however, a few more words about {VARIABLE}.

2.4 A Closer look at {VARIABLE}

The reason that I have spent some time looking at variables in a chapter which started by talking about the FORTH dictionary is that {VARIABLE} is one of a special set of words called *defining* words.

The FORTH dictionary is rather like a dictionary in which some pages have been deliberately left blank. New words, together with their definitions, may be written in the blank pages, so that when subsequently a word is looked up in the dictionary the newly added words will be searched along with the rest. The defining words are a special class of FORTH operations which have the effect of writing new words into the 'blank pages' of the dictionary.

So the actual effect of typing:

```
VARIABLE Year ok
```

is to add a new word to the dictionary; {Year}. But back in section 2.1 we saw that all of the words in the FORTH dictionary have an *action* which takes place when the word is typed in – and so it is with new words defined by {VARIABLE}. Their action is to push the *address* of the variable onto the stack. Typing, for example:

```
Year . 23967 ok
```

will cause FORTH to print out a very peculiar number – which you've certainly never seen before! (It's unlikely to be '23967' either!). This is the actual address of the memory location which FORTH has reserved for the variable {Year}.

Figure 2.3 takes a look inside the dictionary to show a simplified view of the new dictionary entry for Year.

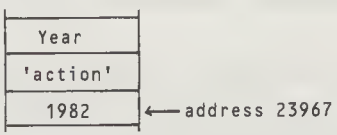


Figure 2.3 The Dictionary entry for {Year}

The dictionary entry has three distinct parts. The first is a 'name' part which contains the word "Year". The second specifies the 'action' of the variable, which is to push onto the stack the address of the third part of the dictionary entry containing its actual value.[2]

If we now look at the store {!} and fetch {@} operations in more detail, we see that they are equivalent to the BASIC operations POKE and PEEK. FORTH defines {!} and {@} as follows:

```
!       ( n addr → )     Store n at addr
@       ( addr → n )     Fetch contents of addr
```

These are, like the majority of FORTH operations, 16 bit number operations. FORTH does have 8 bit store and fetch operations which are useful for single character manipulation, and are called {C!} and {C@} respectively, more of these in chapter 7.

The FORTH input:

```
1 Year +! ok
```

actually causes the following sequence of events:

i) The first word {1} is not in the dictionary, and is taken to be a number, which is pushed onto the stack.

ii) The second word {Year} is looked up in the dictionary. If it is found (and it will only be found if it has been pre-defined using {VARIABLE}), then the address of Year will be pushed onto the stack.

iii) The third word {+!} has the special effect of adding the second number on the stack into the contents of the address on top of the stack. So year becomes equal to 1982+1 = 1983.

Of course, it is not necessary to remember this sequence of events every time you use FORTH variables. Like any self-respecting high-level language, FORTH will let you use variables without ever knowing where, in memory, the variables are stored.

2.5 The FORTH {CONSTANT}

Constants are simply convenient ways of representing often-used numbers by a meaningful name, rather than having to quote the number each time it is needed. In FORTH, constants must be defined before they can be used, just as variables must be defined. For example:

```
1234 CONSTANT Phonenumber ok
```

defines a constant called 'Phonenumber', with the value '1234'. Typing:

```
Phonenumber . 1234 ok
```

will cause the number 1234 to be pushed onto the stack, as if it had been explicitly typed (this is the actual *value* of the constant, not its address), and then printed.

The operation {CONSTANT} is, in fact, another example of a 'defining word'. In the example above, the constant {Phonenumber} becomes a new word in the dictionary, which returns its value when executed. The value is, however, fixed at the value supplied when the constant was defined, and may not be changed in the same way

[2]The 'action' part contains the address of a program which has the action specified.

that a variable may be altered. This is not strictly true; after all, if your phone number should change, you could simply type:

```
5678 CONSTANT Phonenumber ok
```

which defines a new constant 'Phonenumber', with the new value, even though we already have a constant called 'Phonenumber'.[3]
In fact this example illustrates another important feature of the FORTH dictionary:

> When a word is looked up in the dictionary, the dictionary is searched in the reverse order to the order in which new words were added.

Thus, after redefining 'Phonenumber', we actually have two definitions of 'Phonenumber' in the dictionary – but the most recently defined version will always be used.

If we should need to revert back to the old definition of Phonenumber, then we may simply 'forget' the new one, using the word {FORGET}:

```
FORGET phonenumber ok
Phonenumber . 1234 ok
```

{FORGET} is a useful 'housekeeping' operation; we can use it to clear out the dictionary from time to time. But use it with care since {FORGET <name>} will forget the most recently defined version of <name> and, in addition, anything defined since <name>.

2.6 Summary and Exercises

Here is a summary of the FORTH words covered in this chapter, followed by some practice problems on their use.

In the stack descriptions 'byte' refers to a 16 bit value, but with only the lower 8 bits set or used by the operation. The upper 8 bits are usually set to zero. 'addr' refers to a 16 bit value which represents the address of a byte in memory. The addressed byte may be the first byte of a larger item (i.e. a 16 bit variable).

<name> refers to the next word, delimited by spaces, in the input stream. <name> can consist of any non-space characters in the standard ASCII character set (see glossary for a description of ASCII), although the maximum number of characters and the number of characters which will be stored in the new dictionary entry are implementation dependent.

Memory:

@ (addr → n) "fetch"

Fetch from memory the number contained at addr.

! (n addr →) "store"

Store n at address.

[3]Some FORTH systems print a 'warning' message whenever a word is defined which already occurs in the dictionary. This message should be ignored since re-defining existing words is a perfectly valid facility.

`C@` `(addr → byte)` `"c-fetch"`

Fetch the byte contained at addr.

`C!` `(byte addr → )` `"c-store"`

Store byte at address.

`?` `(addr → )` `"question-mark"`

Display the number stored at address, using the same format as `{.}`.

`+!` `(n addr → )` `"plus-store"`

Add n to the 16 bit value stored at address, using the `{+}` operation.

Defining Words:

`VARIABLE` `( → )`

When used in the form: VARIABLE <name> creates a dictionary entry for <name> with two bytes of storage (in the parameter field – see chapter 9). When <name> is later executed it will place the storage address on the stack:

```
<name>  ( → addr)
```

`CONSTANT` `(n → )`

When used in the form: n CONSTANT <name> creates a dictionary entry for <name> with n stored (in the parameter field). When <name> is later executed it will leave n on the stack:

```
<name>  ( → n)
```

Dictionary Management:

`FORGET` `( → )`

When used in the form: FORGET <name> the most recently defined dictionary entry for <name> is deleted, together with all words defined since <name>. An error occurs if <name> cannot be found.

Exercises

1) Define the following constants:

Name	*Value*
ten	10
fred	4*ten+1

2) Define the following variables, and set them equal to the values:

Name	*Initial value*
XYZ	-100
A	XYZ-fred

3) Write a FORTH expression equivalent to the BASIC statement:

```
LET  X = 1 + X + X*X
```

Assume that the variable X has been pre-defined. Can your solution be

improved by the use of DUPlication, or even {+!}?

4) Write a FORTH expression to evaluate the quadratic equation:

```
ax↑2 + bx + c
```

where x has been defined as a FORTH variable and a, b and c have been defined as FORTH constants.

5) Show how {CONSTANT} may be used to name any specified memory location, and then use it like a variable. Don't actually try this on your computer – it could be disastrous!

3
The COLON Definition

So far we have examined in detail two defining words {VARIABLE} and {CONSTANT}. Both of these have the effect of adding new words to the dictionary, new words which will have a special action when executed to make them into variables and constants. The FORTH word colon {:}, is also a defining word, but a much more general defining word. Using it we may not only add new words to the dictionary but actually define the action that the new words will have when executed.

3.1 Colon Calculations

Suppose that we wish to compute some percentages using FORTH. We could simply type, for example:

```
150 12 * 100 / . 18 ok
```

to calculate 12% of 150. But with more than just one or two percentages to compute we could really use a special 'percentage' operator, rather like the percent key on a calculator. With a simple colon definition we can easily define such an operator, as follows:

```
: %  * 100 /  ; ok
```

and we may now type {%} instead of the sequence {* 100 /}, for example:

```
150 12 % . 18 ok
```

This is clearly much neater! It is less typing (and therefore less prone to error), and more readable as well.

The definition of {%} is an example of a 'colon definition', which has added a new word to the dictionary that will have the same effect, when executed, as if the sequence {* 100 /} had been explicitly typed. Figure 3.1 shows how this colon definition is structured.

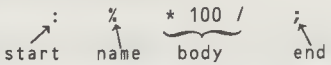


Figure 3.1 The Colon Definition of {%}

The colon {:} and the semi-colon {;} start and end the definition respectively. A colon definition always starts with a colon and is always terminated by a corresponding semi-colon. The <name> part of the definition is the first word following the colon and is the name given to the new dictionary entry. Just like any other FORTH input, the colon and the name must be separated by at least one space. Likewise, the name and the first word in the 'body' must also have at least one space separating them. The body of the definition may be any valid sequence of FORTH words (including numbers), and is compiled into the new dictionary entry. It is the body of the colon definition that defines the *action* of the new word when it is later executed.

Like the name of variables or constants, the <name> part of a colon definition can be any length and consist of any characters, but in many FORTH systems the first three or four characters must be unique.

In the {%} example, the action consists simply of multiplying the two numbers on top of the stack and dividing the result by 100. We may thus describe the new word {%} just like any other FORTH arithmetic operator by using the conventional (stack before → stack after) notation, as follows:

```
%       (n1 n2 → percent)   Take n2 percent of n1,
                            percent=n1*n2/100.
```

Indeed, it is a good idea to document all new words in this manner, so that at any time you will know exactly what your extended FORTH dictionary contains, and how to use it. This is particularly useful if you go on to use your newly defined word in further colon definitions.

3.2 More Percentages

Once {%} has been defined, it is treated just like all other FORTH operations, so that it may be included in complex expressions, like:

```
500 15 % 2 % . 1 ok
```

(which computes 2% of 15% of 500), or {%} may be included in the body of another colon definition, for example:

```
VARIABLE account ok
: invest
     account @  ( get account )
     12 %       ( compute interest )
     account +! ( add into account )
; ok
```

If we then place £200 in our account by typing:

```
200 account ! ok
```

We can discover how much the account will contain after 3 years of accumulating compound interest at 12%, by typing:

```
invest invest invest ok
account @ . 280 ok
```

Apart from illustrating the effect of a healthy rate of interest, this example demonstrates a number of new facilities:

i) A colon definition may occupy more than one line of input and, even though we type a 'return' at the end of each line, FORTH does not complete the definition and print 'ok' until after the terminating semi-colon. The additional spaces are ignored and are included merely to improve readability.

ii) Comments may be included by enclosing explanatory text in (round brackets). Remember that open-bracket {(} is a FORTH word and, like any other, must have at least one space on either side. The close-bracket is not a word, but simply a 'delimiter' to indicate the end of the comment.

iii) Newly defined variables (and constants) may also be included in the body of a colon definition. They are, after all, just words in the dictionary.

If we examine the definition of {invest} in a little more detail, we see that {invest} is not a new arithmetic operator, like {%}. It is, in effect, a complete, albeit very simple, program. The program may be run by typing:

```
invest ok
```

{invest} uses the stack during execution, but has no overall stack effect. Nevertheless, we may still document {invest} (and {account}) in the recommended manner:

```
account ( → addr)  User variable for invest
invest  ( → )      Add interest at 12% into account
```

Notice that I have documented these new words in the order in which they were defined.

3.3 Colon Definition or Program?

Figure 3.2 shows a very simple BASIC program with an equivalent FORTH colon definition.

```
BASIC                              FORTH .

                                  : Squared
10 INPUT X                                  ." = "
20 PRINT "Squared = ";X*X                   DUP * .
                                  ; ok

RUN
?4
Squared = 16                      4 Squared = 16 ok
```

Figure 3.2 BASIC in FORTH

Line 10 of the BASIC program asks the user to type a number, which is placed in the variable 'X'. Line 20 then prints the message "Squared = ", and finally the result of X*X.

The FORTH equivalent is simpler, because rather than asking the user for the number which is to be squared, the FORTH program takes the number off the top of the stack and uses that instead. The colon definition has been deliberately given the name {Squared} so that to run the program we simply type a number, followed by 'Squared', and hit 'return'. The action of {Squared} is to first print " = " then multiply the number on top of the stack by itself {DUP *} and finally print the result. This gives us a surprisingly neat and readable way of running a program, for example:

```
4 Squared = 16 ok
5 Squared = 25 ok
6 Squared = 36 ok
```

Notice that if you should forget to type the preceding number, FORTH will respond with the error message STACK EMPTY:

```
Squared = 0 STACK EMPTY
```

This is because {Squared} needs one argument, as we see from its stack description:

```
Squared   (n → )   Print n squared.
```

Two additional features of {Squared} are worth discussing,

i) {Squared} does not need to use a variable, as opposed to its BASIC counterpart which does. It is a characteristic of FORTH programs that variables are not often used, the stack being preferred for holding temporary values, as in {Squared}.

   Experienced programmers may be sceptical of this use of the stack, and ask the question "Does FORTH have an equivalent operation to the INPUT statement of BASIC?". The answer is that it is possible to request input from the user, in FORTH (as I will show in chapter 7), but for most applications the use of the stack to pass input values to a program is preferred. It is certainly easier and, after all, what could be neater than typing '4 Squared'?

ii) The use of dot-quote {."}. This is the FORTH equivalent of PRINT " ... " in BASIC, and will print all of the text following {."} until the next occurrence of the double-quote character ("). Most FORTHs will allow dot-quote to be used outside a colon definition, for example:

```
." Hi there "  Hi there ok
```

   When used inside a colon definition the enclosed message is compiled, and then printed out when the word is executed:

```
: GREET  ." Hi there " ; ok
GREET  Hi there ok
```

   Again, note that dot-quote must have at least one space on either side. The terminating quote (") is simply a 'delimiter' to mark the end of the text to be printed and need not be preceded by a space. If it is, then the space will form part of the printed text.

We can now answer the question posed by the title of this section "Colon Definition or Program?", by observing that programming in FORTH is achieved by writing one, or more, colon definitions – in other words, a colon definition is a program.

The comparison between a BASIC program and a FORTH colon definition serves to illustrate this principle but the analogy must not be taken too far; the BASIC programmer develops, extends and edits one program to achieve his goal, whereas the FORTH programmer achieves the same goal by writing a series of colon definitions, each of which is compiled and added to the dictionary, and tested separately. Furthermore to run a number of different BASIC programs requires that each one is separately loaded and RUN, but compiled FORTH is so compact that many different programs may coexist together in the dictionary and any one of them may be run just by typing its name.

3.4 Interpret .. Compile?

Since I have used both 'interpret' and 'compile' to describe FORTH on different occasions in this book so far, now is perhaps the time to clarify these terms. Let us consider the simple subtraction:

```
200 50 - . 150 ok
```

as soon as we finish typing the line of input and hit the 'return' key (after the {.} symbol), FORTH interprets the input in the manner already described; each word is

in turn looked up in the dictionary. If it is found, it is executed, otherwise it is treated as a number and pushed onto the stack.

Any FORTH which may be typed in and executed in this way may also be included in a colon definition, by simply preceding the input by `{: name}` and terminating it by `{;}`. The initial colon has the effect of switching FORTH from 'interpret' mode into 'compile' mode and the terminating semi-colon has the opposite effect. Treating our simple subtraction example in this manner gives us the following colon definition:

```
: example  200 50 - .  ; ok
```

The initial colon generates a new dictionary entry with the name 'example', and switches FORTH into 'compile' mode so that the following program, up to the semi-colon, is compiled into a compact set of instructions which are placed into the new dictionary entry. Figure 3.3 illustrates this example:

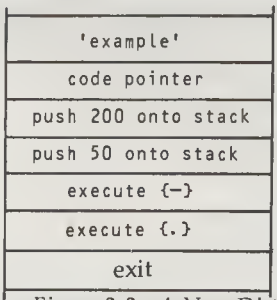


Figure 3.3 A New Dictionary Entry

The new dictionary entry starts with the name of our new word, in this case 'example' and contains four instructions, each one corresponding to a word in the body of the colon definition – the first is 'push 200 onto the stack' – the second is 'push 50 onto the stack' – and so on.

To execute these instructions we type:

```
example 150 ok
```

to produce exactly the same result as the original interpreted input.

In reality, the actual compiled instructions in the dictionary are not as long winded as they might appear from figure 3.3. Each consists simply of the 'address' of the corresponding dictionary entry (i.e. the dictionary entry for the word `{-}`, or `{.}`). The 'code pointer' points to a fast 'run-time' program which will execute the words in the definition by 'calling' each address in turn. This, and other details, are covered in depth in chapter 9, but for the present we shall note that:

> Ordinary FORTH is interpreted (and executed) straight away, but if the same input is enclosed in a colon definition, then it is compiled, and may be later executed by typing the name of the colon definition.

3.5 Creating tables and arrays

Array storage is a common requirement in many programs, and accordingly most languages provide facilities for setting up arrays. In BASIC arrays are 'dimen-

sioned' using the 'DIM' statement.

FORTH does not have an equivalent to 'DIM' in its standard word set, but does provide all of the necessary operations to 'build' arrays whenever they are needed. (Remember that because FORTH is an extensible language it doesn't need to have all of the facilities you are ever likely to require pre-defined.)

The easiest way of reserving array space in the dictionary is with the word {ALLOT}:

```
ALLOT   (n → )          Allot an extra n bytes of space to the
                        most recently defined dictionary entry.
```

{ALLOT} may be used together with {VARIABLE}. For example:

```
VARIABLE double 2 ALLOT ok
```

will have the effect of defining a new variable named {double}, with space for a single value, but {2 ALLOT} then reserves an extra 2 bytes so that the variable {double} has room for 2 values altogether. (A single value takes up 2 bytes of memory). Figure 3.4 illustrates the whole dictionary entry for {double}.

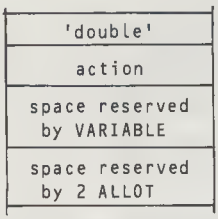


Figure 3.4 The Dictionary entry for {double}

We now have, in effect, a two element 'array'. The word {double} will return the address of the first number in the array, add 2 and we have the address of the second number. For example:

```
100 double ! ok
200 double 2+ ! ok
```

will initialise the array to contain the values 100 and 200,

```
1  double 2+  +! ok
double 2+  ? 201 ok
```

increments the second number, and prints its new value.

An alternative and, in many ways, neater way of setting up arrays is to use the defining word {CREATE}. {CREATE} is exactly like {VARIABLE}, except that it reserves no space in the dictionary entry. To set up an array using {CREATE} we must {ALLOT} the whole of the space required. For example:

```
CREATE array 40 ALLOT ok
```

defines an array with space for 20 numbers.[1]

Instead of using {ALLOT} we could use {,} (pronounced "comma") to both reserve space and set each element to an initial value. {,} has the effect of popping the number off the top of the stack, and storing it in the next free location (2 bytes) in the dictionary; thus, it is particularly useful for setting up tables of constants. For example:

```
CREATE TABLE -10 , -5 , 0 , 5 , 10 , ok
```

defines a five element array with values -10, -5, 0, 5 and 10. Again, we may pick out any value by adding an offset to the address returned by {TABLE}, as before, but a special colon definition is the best way to do this. For example:

```
: TABLE@  1- 2 * TABLE + @ ; ok
```

To pick out an entry simply precede the word {TABLE@} by the number of the required entry, for example:

```
5 TABLE@  . 10 ok        ( print the fifth entry )
1 TABLE@ ok              ( fetch the first entry )
2 TABLE@ ok              ( and the second )
+ . -15 ok               ( and add them )
```

Notice the use of the word {1-} in the definition for {TABLE@}, and the word {2+} in the examples with {double} earlier. FORTH defines four often used additions and subtractions for convenience, {1+}, {1-}, {2+} and {2-}, which are identical in action to their equivalent phrases {1 +}, {1 -}, {2 +} and {2 -} but are usually defined in machine-code for faster execution.

3.6 The Stack Notation Extended

In the last chapter I promised to describe a technique for illustrating the stack during program execution and it is just such a technique which we could use to clarify the operation of {TABLE@} above. The technique is to list vertically each word in the body of the colon definition. Then look up each word, in turn, in the FORTH Handy Reference, and note down the stack effect of that word, remembering that the 'stack after' list becomes the 'stack before' list for the next word down.

Figure 3.5 illustrates this notation in describing the operation of {TABLE@}:

WORD	Stack Effect	Comments
1-	(n† → n-1)	subtract 1 from index.
2	(n-1 → n-1 2)	push 2 onto stack.
*	(n-1 2 → offset)	multiply to give offset.
TABLE	(offset → offset addr)	fetch start address of TABLE.
+	(offset addr → offset+addr)	add offset.
@	(offset+addr → †n)	fetch value required.

Figure 3.5 The operation of {TABLE@}

One feature is worth noting in particular;

> The 'stack before' list of the first word, and the 'stack after' list of the final word, give the overall stack effect. These are indicated by † in figure 3.5 and

[1]Important note - on some FORTH systems the word {CREATE} cannot be used like this, and {VARIABLE} must be used instead, i.e.:

```
VARIABLE array 38 ALLOT
```

See your system documentation to find out which you must use!

enable us to formally describe {TABLE@} as follows:

```
TABLE@  (n1 → n2)       Fetch the entry from TABLE indexed by
                        n1.  n1 must be in the range 1 to 5.
```

The author has found this stack notation invaluable in developing FORTH programs with complex stack manipulations, and far from being cumbersome the technique soon becomes rapid as familiarity is gained. In particular, the experienced FORTH programmer will not have to refer often to the Handy reference, and will place words in the left hand column in groups of more than one, where the stack effect is very clear (or none at all) so that the whole diagram is much simplified.

Another useful technique for 'debugging' FORTH programs is to use the word {DEPTH} with {.} to print the number of values contained on the stack at certain key points in the definition under development. If we define:

```
: .S   CR DEPTH .  ; ok
```

{.S} will print the number of values on the stack, without affecting the stack at all. Including this in a new definition:

```
: TABLE@  1- 2 *  .S  TABLE +  .S  @ ; ok

. . . 0 STACK EMPTY      ( clear the stack first )

4 TABLE@
1
1 ok
```

shows us the number of stack values at the points marked by {.S} in the definition, and that all is well during execution! We can now FORGET {TABLE@} and redefine it without {.S}.

3.7 Summary and Exercises

The following new words have been introduced in this chapter:

Stack Manipulation:

DEPTH (→ n)

Leave the number of values contained on the stack, not counting n.

Arithmetic:

1+ (n → n+1) "one-plus"

Increment n by 1.

1- (n → n-1) "one-minus"

Decrement n by 1.

2+ (n → n+2) "two-plus"

Increment n by 2.

2- (n → n-2) "two-minus"

Decrement n by 2.

Defining Words:

`:` `( → )` `"colon"`

Used in the form:

`: <name> .... ;`

Creates a dictionary entry for <name>, and sets 'compile' mode so that subsequent words from the input stream are compiled into the new dictionary entry. These will be executed when <name> is later executed.

`;` `( → )` `"semi-colon"`

Terminate a colon definition and stop compilation.

`CREATE` `( → )`

Used in the form: CREATE <name> to define an empty dictionary entry for <name>. When <name> is later executed the address of (the parameter field for) <name> is left on the stack:

`<name> ( → addr)`

Dictionary Words:

`ALLOT` `(n → )`

Reserve n bytes in (the parameter field of) the most recently defined dictionary entry.

`,` `(n → )` `"comma"`

Allot two bytes in the dictionary and store n there.

Output:

`."` `( → )` `"dot-quote"`

When used in the form: `." text"` the text up to but not including the delimiter character " is printed. If dot-quote occurs within a colon definition, then the text is compiled so that it will be printed at execution time. Up to 127 characters may be enclosed.

Miscellaneous:

`(` `( → )` `"paren"`

When used in the form: `( text)` the enclosed text will be ignored. The delimiting "close-paren" character) is not a FORTH word and need not be preceded by a space, but must be separated from the following word by at least one space.

Exercises

1) Write a fast colon definition to triple the number on top of the stack.

2) Write a colon definition, `{newpage}`, which will print a form-feed, followed by `"Page - "`, and finally print the page number supplied on the stack.
(Hint: you will need to use the word `{EMIT}` to print the form-feed).

3) Create an array with 4 entries, set initially to the values, -10, 1, 10, and 1000. Define a word to calculate the address of the i'th entry, where i can be 0, 1, 2 or 3.

4) Define a word that will double the value of each of the current entries in the array of question 3.

5) Use the technique described in section 3.6 to explain the operation of the following colon definition, and therefore deduce its overall stack effect:

```
: example  DUP *  SWAP  DUP *  +  ;
```

(Hint: the stack initially needs two values on it).

4
FORTH Structures 1, IF

The one BASIC statement for which FORTH has no equivalent is 'GOTO' but the fact that FORTH is a fully structured language means that it doesn't need a 'GOTO'. This may seem an outlandish claim but it is not; indeed many experienced programmers feel that GOTOless programming is best. It makes for programs, they say, which are readable, self documenting and, above all, 'structured'. What does 'structured' programming mean? Well, three features make a structured language:

i) The ability to execute a sequence of operations, one after the other.
ii) Conditional testing and the execution of either one sequence or another sequence depending on the result of a conditional test.
iii) Repetitive execution of a sequence of operations, until some condition is met, or while a condition is true.

FORTH has each of these requirements, the first has already been illustrated by all of the examples so far, it is, of course, a pre-requisite of virtually any programming language. FORTH provides the second requirement in the conditional structure IF .. ELSE .. THEN, and the third in a set of looping structures, the DO loop, UNTIL loop, and WHILE loop. This chapter describes the conditional IF structure, and the comparison and logical operations which accompany it. The looping structures are covered in detail in chapter 5.

4.1 True or False?

A word which will test the sign of a number, and confirm whether that number is negative or not, could be defined as follows:

```
: Negative?   0 < IF  ." Yes "  THEN ; ok
```

and is used by simply typing a number, followed by 'Negative?':

```
-20 Negative? Yes ok
20 Negative? ok
```

A refinement of `{Negative?}` would be the inclusion of an `{ELSE}` clause:

```
: Negative?   0 < IF  ." Yes "  ELSE  ." No "  THEN ; ok
```

the new definition of `{Negative?}` supersedes the old one, so FORTH will now print a reply whether the number is negative, or positive:

```
-1 Negative? Yes ok
1 Negative? No ok
```

Let us examine this new definition of `{Negative?}`. Upon execution the value 0 is first pushed onto the stack. The word `{<}` then compares the top two numbers on the stack and tests for the second less than the first (on top of the stack). `{<}` replaces these two numbers by a single number called a *flag*, which may only have one of two values, *true* or *false*. In this particular example the number on top of the stack is 0, so `{<}` is actually testing for the second number less than 0, i.e. negative.

The next word is {IF} which pops the flag off the stack and causes the words immediately following to be executed if the flag were true, or those following the {ELSE} if the flag were false. In either case, execution continues after {THEN}, which must be included to properly terminate the {IF} structure even though the colon definition ends immediately after {THEN} as in this example. The {ELSE} clause is optional, as illustrated by the earlier version of {Negative} above.

The 'less-than' word {<} is one of a special class of FORTH words called 'comparison words', which usually occur immediately before the {IF} word. The formal definition of {<} is:

```
<       (n1 n2 → flag) flag is set 'true' if n1<n2,
                       false otherwise.
```

n1 and n2 may be any single-precision numbers, and flag is also a number but one which represents the logical values true or false, according to the following convention:

Logical value	Numerical value
true	1
false	0

A few examples will demonstrate the flag value:

```
-2 4 < . 1 ok
20 10 < . 0 ok
```

-2 is indeed less than 4, so {<} returns the value 1, which represents 'true'. 20 is not less than 10, so {<} instead returns 0, representing 'false'. Notice that the numbers which are to be compared are entered in the same order as they would be in ordinary notation, if we wish to test for n1 less than n2, we write in FORTH:

```
n1 n2 <
```

4.2 The IF structure defined

Figure 4.1 illustrates the general form of the IF structure:

```
conditional words
IF
        'true' words
ELSE
        'false' words
THEN
```

Figure 4.1 The full IF structure

The conditional words must place a logical 'flag' value onto the stack and will usually involve 'comparison' words like {<}, but not necessarily, since {IF} will treat any non-zero number as 'true' but only the value zero as 'false'.[1]

[1]Programmers who go for 'minimal' solutions can take advantage of this feature when testing for non-zero since such a test involves no comparison operation at all! For example:

```
: nonzero?  IF  ." yes"  THEN ; ok
34 nonzero? yes ok
```

The 'true' words are executed whenever the value taken off the stack by {IF} is non-zero, or true, the 'false' words are executed if the value is zero, or false. The true words, or false words, may be any sequence of valid FORTH, including further {IF .. ELSE .. THEN} structures and in this way IF structures may be nested to virtually any depth.

As illustrated by the first example in this section, the ELSE clause is optional and may be omitted so that the whole structure reduces to its simpler form, shown in figure 4.2:

```
conditional words
IF

      'true' words
THEN
```

Figure 4.2 The simple IF structure

Conditional structures, and indeed all of the structures which I describe in this chapter and the next, may only be used inside colon definitions. The reason for this is that such structures contain forward jumps which are not known until the whole colon definition has been compiled.

Figure 4.3 shows the correspondence between an IF statement in BASIC and the FORTH IF structure, and emphasises in particular the re-ordering which is characteristic of FORTH!

```
BASIC:    IF A=2 THEN PRINT "A=2"
FORTH:    A @ 2 =  IF  ." A=2"  THEN
```

Figure 4.3 BASIC IF → FORTH IF

The FORTH in figure 4.3 would be part of a colon definition and assumes that the variable A has already been defined using {VARIABLE}. A new comparison word {=} is illustrated, {=} is similar to {<} in that it replaces the top two values on the stack by a single flag value, but in this case the flag is set to 'true' only if the two values are equal.

4.3 Nested IF structures

Figure 4.4 illustrates a colon definition which incorporates two IF structures, one nested inside the other. The word defined {Grade} will have the effect when executed of printing one of three grades, 'Fail', 'Pass' or 'Distinction' depending on a score on top of the stack:

```
: Grade  DUP 40 < IF
                    ." Fail"              ( Less than 40 )
                    DROP
                 ELSE
                 70 < IF
                         ." Pass"         ( 40-69 )
                      ELSE
                         ." Distinction"  ( greater than 70 )
                      THEN
                 THEN ;
```

Figure 4.4 The definition of {Grade}

The whole of the second IF structure is encompassed within the ELSE clause of the outer IF structure, in the manner illustrated in figure 4.5:

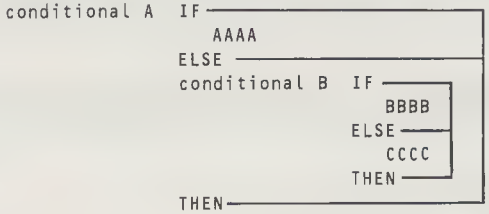


Figure 4.5 Nested IF structures

If corresponding {IF .. THEN} words are joined by lines, as in figure 4.5, then the lines should never cross, but remain nested one inside the other. If this is not so, then the program is not only meaningless but also invalid FORTH! It is common practice in FORTH to indent nested structures simply to aid readability by humans (computers have no trouble either way!). This is true also of other structured languages such as Pascal, but the BASIC programmer may find it unusual.

In figure 4.5, if conditional A turns out to be 'true', then the AAAA words are executed, but if conditional A were 'false' then conditional B will be tested, resulting in either BBBB or CCCC to be executed. Looking again at our original {Grade} example in figure 4.4, if the number on top of the stack is less than 40 the first IF will be true, 'Fail' is printed, {DROP} is executed, and the program finishes. If, however, the number on top of the stack was 40 or greater, then conditional B will be tested to decide between 'Pass' or 'Distinction':

```
25 Grade Fail ok
54 Grade Pass ok
70 Grade Distinction ok
```

Notice the use of {DUP} preceding the conditional A test in {Grade}. This is to ensure that the score being tested remains on top of the stack for conditional B, if necessary. The inclusion of {DROP} is to clear the extra value off the stack if it is not needed (that is if conditionalB will not be executed), so that the overall stack effect of {Grade} will be the same whichever route through the IF's is taken upon execution:

```
Grade   (n → )          Print 'Fail' if n<40, 'Pass' if
                        40<=n<70, or print 'Distinction'
                        otherwise.
```

This illustrates another point to be wary of in definitions involving conditionals:

> Make sure that, whichever route is taken through a conditional structure, the overall stack effect is the same.

Adherence to this principle will avoid many confusing bugs!

4.4 Logical Operators for Complex Conditionals

Quite often an IF statement involves more than one comparison, combined by using logical operations – AND, OR etc. For example, to test if X lies between a lower and an upper limit we could write, in BASIC:

```
IF (X>10) AND (X<100) THEN ....
```

which tests for X in the range 11 – 99. This could be written, in FORTH:

```
X @ 10 > IF
  X @ 100 < IF
              ....
            THEN
          THEN
```

but a much better, and simpler, solution is to use the word {AND}, as follows:

```
X @ 10 >  X @ 100 <  AND. IF
                              ....
                          THEN
```

Here we have written the two comparisons separately, {X @ 10 >} leaves a flag on the stack, and the second comparison {X @ 100 <} places a second flag onto the stack. These two flags are then combined by {AND}, to leave a single flag which will be true only if both comparisons were true. The operation of {AND} is described by the following:

```
AND     (n1 n2 → n3)          n3 = n1 AND n2

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

(0 = 'false', 1 = 'true')
```

Although {AND} is used in the above example to combine flag values, (0 or 1), it will perform a logical 'bitwise' AND between complete 16 bit numbers, as will the other FORTH logical operators, {OR} and {XOR}. We can easily demonstrate this as follows:

```
5 ok              ( = 101 binary )
6 ok              ( = 110 binary )
AND . 4 ok        ( = 100 binary )
```

Each bit in the binary equivalent of 4 is the result of separately ANDing the corresponding bits in the numbers 5 and 6.

Returning to our IF example, we see that the comparisons are performed upon a variable X, and this has simplified the expression to a certain extent because the value of X may be fetched each time it is needed for a comparison using the phrase {X @}.

More often, however, we shall need to perform a number of comparison tests upon the single value on top of the stack and this will require some stack manipulation. Suppose, for example, that we need to test if the number on top of the stack is less than 0 OR greater than 100. The condition test in front of an IF might appear:

```
DUP  0<  SWAP  100 >  OR  IF ....
```

Notice that the joining of 0 and < is not a printing error! {0<} is simply a more convenient (and usually faster) version of the two word sequence {0 <}. FORTH defines a number of often used 'comparisons with zero', whose overall stack effect is, in each case, identical to the effect of {0} followed by the comparison operation.

We can analyse the operation of the whole sequence above using the stack notation

described in the last chapter:

```
Word    Stack Effect                    Comment
DUP     (n → n n)                       Duplicate top of stack
0<      (n n → n flag1)                 flag1 true if n negative
SWAP    (n flag1 → flag1 n)             swap n and flag1
100     (flag1 n → flag1 n 100)         push 100
>       (flag1 n 100 → flag1 flag2)     flag2 true if n>100
OR      (flag1 flag2 → flag3)           OR the two flags
```

The key operations in this sequence are the initial duplication, which gives us two copies of the number n – one for each comparison operation, and the {SWAP} which rearranges the stack so that flag1 is preserved while the second comparison takes place – ready for the final {OR}. {OR} has the effect of combining the two flag values so that the resulting flag3 will be true if either flag1 OR flag2 (or both) were true.

4.5 The Missing Comparison operations

FORTH only defines three basic comparison operations; {<}, {=} and {>}, what about "less than or equal to", "not equal to" and so on? Well these operations are not defined in the standard system because we may very easily define them ourselves, if necessary, with the help of {NOT}, as follows:

```
: <=  > NOT  ;
: <>  = NOT  ;
: >=  < NOT  ;
```

In each case {NOT} reverses the truth value of the flag produced by the previous comparison operation, so that 'true' becomes 'false', and 'false' becomes 'true'.

It is a characteristic of FORTH that rather than clutter up the dictionary with an exhaustive set of operations, those whose definitions are trivial, like the ones above, may be defined by the user as needed.[2]

Another useful comparison operation is "unsigned less than", {U<}, which performs a 16 bit magnitude comparison. The operation can be used to compare unsigned numbers, as follows:

```
1 60000 U< . 1 ok        ( = true )
50000 40000 U< . 0 ok    ( = false )
```

or we can make use of the fact that negative numbers appear to be large positive numbers if treated as unsigned, to simplify certain comparisons. For example, the phrase:

```
100 U<
```

has exactly the same effect as the phrase:

```
DUP 0< NOT SWAP 100 < AND
```

and tests if the number on the stack lies within the range 0-99.

[2]Sceptical readers may ask "What about {0<}, surely this operation could have been left out, and defined by the user when needed?" In fact in most FORTH systems {0<} is a more 'primitive' operation than {<}, which is defined as:

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

Finally, a useful combined stack manipulation and comparison operation is {?DUP}, which duplicates the number on top of the stack only if it is non-zero (true). {?DUP} normally precedes {IF} so that if the number on top of the stack is non-zero, then it is duplicated for use within the IF structure, alternatively if the number is zero, then the stack is left cleared. For example:

```
: EXAMPLE ?DUP IF ." Non-zero number is" . THEN ; ok
34 EXAMPLE Non-zero number is 34 ok
0 EXAMPLE ok
```

4.6 Summary and Exercises

The following new words have been introduced in this chapter:

Stack Manipulation:

?DUP (n → n) *or* (n → n n) "query-dupe"

Duplicate n only if it is non-zero.

Comparison:

< (n1 n2 → flag) "less-than"

Flag is true if n1 is less than n2.

= (n1 n2 → flag) "equals"

Flag is true if n1 equals n2.

> (n1 n2 → flag) "greater-than"

Flag is true if n1 is greater than n2.

0< (n → flag) "zero-less"

Flag is true if n is less than zero (negative).

0= (n → flag) "zero-equals"

Flag is true if n is zero.

0> (n → flag) "zero-greater"

Flag is true if n is greater than zero (positive).

U< (un1 un2 → flag) "u-less-than"

Compare the magnitude of the unsigned 16 bit numbers un1 and un2, leaving the flag 'true' if un1 is less than un2.

NOT (flag1 → flag2)

Reverse the truth value, so that false becomes true or true becomes false.

Logical:

`AND` `(n1 n2 → and)`

Leave the bitwise logical AND of n1 and n2,

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

`OR` `(n1 n2 → or)`

Leave the bitwise logical OR of n1 and n2,

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

`XOR` `(n1 n2 → xor)` `"x-or"`

Leave the bitwise logical exclusive-or of n1 and n2,

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Control Structures:

`IF` `(flag → )`

Used in a colon definition in the form:

flag `IF ... ELSE ... THEN` or,
flag `IF ... THEN`

If the flag is true (non-zero) the words following IF are executed, and the words following ELSE are skipped. If the flag is false, then the words between IF and ELSE are skipped, and the words after ELSE are executed. The enclosed words may include control structures.

`ELSE` `( → )`

See IF above.

`THEN` `( → )`

See IF above.

Exercises

1) Show how the stack will be affected by the following comparison operations:

```
1 2 >
-4 0<
5 0> NOT
```

2) Define a new word, `{SIGN}`, which will have the effect of printing one of three

messages 'positive', 'zero' or 'negative', corresponding to the sign of the number on top of the stack.

3) What will be the results, in binary, of the following logical operations:

```
1101101 1010001 XOR
1010 101 OR
4 5 =  2 3 <  OR
```

4) How would you write the following BASIC IF statement in FORTH. Assume that the variables A and B have been pre-defined.

```
IF NOT((A=2) AND (B=2)) THEN LET A=4
```

5) Do the FORTH words {NOT} and {0=} have anything in common?

6) Can you deduce the effect of the following colon definitions, when executed:

```
: ex1  OVER OVER  >  IF  SWAP  THEN  DROP ;
: ex2  DUP  IF  DUP  THEN ;
```

5
FORTH Structures 2, Loops

In the introduction to the previous chapter, I mentioned that the third requisite of structured programming is the ability to execute a sequence of operations repetitively, in loops. FORTH provides three looping structures, the DO loop, the UNTIL loop and the WHILE loop. This chapter covers these three structures and then goes on to describe how 'nested' structures are constructed and debugged.

5.1 The DO Loop

The simplest (and most commonly used) of the three FORTH looping structures is the DO loop, used whenever we know beforehand, or can calculate, how many times a loop is to be repeated. Like the IF structure, DO loops may only occur inside colon definitions – so I will illustrate the DO loop with a simple definition:

```
: Underline  CR 16 0 DO  ." -"  LOOP ; ok
Underline
----------------ok
```

The two numbers 16 and 0 before the word {DO} tell the DO loop how many times it should be repeated. The first number is the 'limit' value and the second the 'index' value. The loop repeats (limit-index) times, which is (16-0 = 16) in this case, causing {."-"} to be executed 16 times with the effect shown.

The DO loop structure may be summarised as follows:

```
limit  index  DO   ...FORTH words...   LOOP
```

In this particular form of the DO loop, the limit value should always be greater than the index value, and the 'FORTH words' are always executed (limit-index) times. If the limit value is not greater than the index value, then the 'FORTH words' will just execute once, and the loop terminates. In either event, after the DO loop has finished, execution will continue of any words after {LOOP}.

Since the word {DO} simply takes its limit and index values off the stack we need not actually supply these values in the colon definition, but could make one or both of them into parameters for the newly defined word. In practice, it is most useful to have the index value, but not the limit value, supplied inside the colon definition. For example:

```
: Curses!  0 DO  CR ." Oh Dear!" LOOP ; ok
4 Curses!
Oh Dear!
Oh Dear!
Oh Dear!
Oh Dear! ok
```

You really could type {1000 Curses!} if you wanted to!

On reflection, it should be apparent that for most applications we must be able to use the index value as it counts up through the loop for calculations inside the loop.

FORTH does allow us to fetch the index value using the special word {I}, but before describing {I} we must look in more detail at the way a DO loop is actually executed.

5.2 The DO Loop in action

During execution the DO loop makes use of a second special purpose stack called the RETURN stack to hold the index and limit values.[1]

The sequence of events of a DO loop go something like this:

i) The word {DO} is executed once only, and simply transfers the top two values on the normal stack, onto the return stack as shown in figure 5.1.[2]

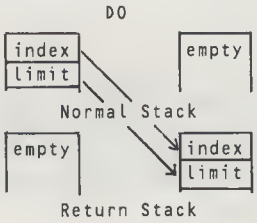


Figure 5.1 The Stack effect of {DO}

ii) The words enclosed inside inside the DO loop are executed as per normal.

iii) The word {LOOP} adds 1 to the index value on top of the return stack, and compares it with the second value on the return stack, the limit value. {LOOP} does not affect the normal stack in any way.

If the new (incremented) index value is less than the limit value, then a jump occurs to just after the {DO}, for another loop, shown in figure 5.2.

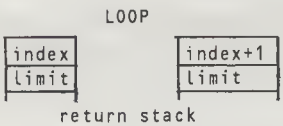


Figure 5.2 Stack Effect of {LOOP}, *index+1 < limit*

If, on the other hand, the incremented index value equals the limit, then the two values are cleared off the return stack, and execution continues after {LOOP}, as shown in figure 5.3.

[1]The RETURN stack is primarily used, by FORTH, to hold 'return' addresses during interpretation of the typed input. The return stack is, however, free for use inside a colon definition where it is used by the DO loop, as shown here. The FORTH programmer may also use the return stack inside colon definitions, as an extra stack, but with care! See chapter 8.3 for a note on this.

[2]By 'normal' stack I mean the stack we have been using throughout. Some FORTH programmers refer to it as 'normal stack', 'parameter stack', 'data stack' or just plain 'stack'!

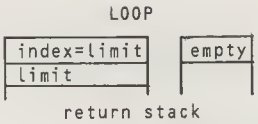


Figure 5.3 Stack Effect of {LOOP}, end of loop.

The overall effect of the DO loop on the return stack is thus to leave the return stack as it was before the DO loop – empty in the case illustrated above. Note that although the above description of the DO loop in action may seem complicated, in practice the FORTH programmer need not consider this since the DO loop takes care of itself.

5.3 Loop Calculations

The FORTH word {I} mentioned earlier is normally only used inside a DO loop and has the special effect of making a duplicate copy of the current index value, (on top of the return stack), and pushing this onto the normal stack – thus making it available for calculation, for example:

```
: Squares  0 DO
               I I * .
           LOOP ; ok
10 Squares 0 1 4 9 16 25 36 49 64 81 ok
```

Again, it is instructive to analyse the operation of {Squares} using the stack notation, remembering that we are picturing the normal stack only here, not the return stack:

```
Word     Stack Effect          Comment
0        (n* → n 0)            set index at 0
DO       (n 0 → )              set up loop from 0 to n
   I        (t → i)            fetch counter
   I        (i → i i)          fetch it again (same value)
   *        (i i → i*i)        square it
   .        (i*i → t)          and print the result
LOOP     ( → *)                terminate loop
```

The starred positions ☆ in the diagram indicate the overall stack effect of {Squares}:

```
Squares (n → )          Print n squares from 0 to n-1
                         squared.
```

Notice also that the stack is empty in the two positions indicated by †. It is important that any repetition of a loop should not cause an overall addition or removal of a number on the stack otherwise stack empty, or stack overflow (full) errors may result after multiple repetitions of the loop. It is generally good practice to ensure that the stack at the start and end of the loop, as in †, has the same number of values on it (zero in the above example) although there are exceptional cases where this rule is broken – by experienced FORTH programmers!

There is clearly a great similarity between the DO loop and the BASIC FOR statement and BASIC programmers may find the comparison shown in figure 5.4 helpful.

```
                              : Squares
10 FOR A=0 TO 9                 10 0 DO
20 PRINT A*A;                         I I * .
30 NEXT A                           LOOP
                              ;
  BASIC                         FORTH
```

Figure 5.4 BASIC FOR and FORTH DO

Notice in particular that the limit values are different in both cases. The BASIC FOR loop executes for A from 0 to 9 inclusive, but to achieve the same range in the FORTH DO loop requires setting the limit to 9+1 = 10.

5.4 {+LOOP} for interesting increments

A very necessary refinement of the DO loop is the use of the word {+LOOP} to specify loop step values other than +1. {+LOOP} is used in the place of {LOOP} and is similar, except that upon execution, {+LOOP} pops the number off the top of the stack and adds this to the index counter before deciding if the loop has finished or not. So, for example, if we required a loop to step through the values 3, 6, 9, 12, and 15, the appropriate DO loop construction would appear:

```
16 3 DO
        ......
  3 +LOOP
```

Likewise if we should want to step *down* through a set of values, say for example, 10, 5, 0, -5, -10, we would write:

```
-11 10 DO
        ......
  -5 +LOOP
```

Notice that the loop still finishes when the index value equals (or passes) the limit value.

Of course, the 'step' value could be a value calculated within the loop, as in the following example:

```
: Example  100 1 DO
                I .
           I +LOOP ; ok
Example 1 2 4 8 16 32 64 ok
```

Thus providing a very neat way of looping through an interesting set of values, which would otherwise have to be calculated!

5.5 Nested DO loops, and other Specialities

Just like all of the structures, DO loops may be nested inside each other within the same colon definition. In fact, in case you had not guessed already, any of the FORTH structures may be nested inside any other – so that we may have IF's within DO loops, or vice versa. First, however, let us look at an example of nested DO loops:

```
: Timestable  CR 11 1 DO
                        11 1 DO
                                J I * .
                            LOOP
                            CR
                        LOOP ;

Timestable
1  2  3  4  5  6  7  8  9  10
2  4  6  8  10  12  14  16  18  20
3  6  9  12  15 ...etc.
```

The word defined here {Timestable} has the effect, when executed, of printing a ten times table.[3]

The operation of {Timestable} relies on the word {J} which appears inside the inner loop. {J} is similar to {I} except that {J} pushes the index of the next outer loop onto the stack. (Which is actually the third value down on the 'return' stack, but you don't really need to know this in order to use {J}). Thus for each repetition of the outer loop, which you can think of as J stepping from 1 to 10, the inner loop repeats 10 times, that is, I steps from 1 to 10. Multiplying J and I thus gives us 1*1, 1*2, 1*3 .. 1*10, and then 2*1, 2*2, 2*3 .. 2*10, and so on up to 10*10 – in other words, a ten times table. In case the operation of {Timestable} still is not clear, here it is again together with an equivalent program in BASIC, in which I have deliberately chosen the FOR loop variables to be J and I respectively:

```
BASIC                       FORTH

                            : Timestable
                                    11 1 DO
10 FOR J=1 TO 10                        11 1 DO
20   FOR I=1 TO 10                          J I * .
30     PRINT J*I;                           LOOP
40   NEXT I                                 CR
50   PRINT                              LOOP
60 NEXT J                           ;
```

Figure 5.5 Nested DO loops, BASIC and FORTH

Remember that {J} and {I} in FORTH are not variables despite the fact that they are used rather like variables in this example.

One further word which like {I} and {J} is used exclusively inside a DO loop is {LEAVE}, which allows a loop to be terminated prematurely. The effect of {LEAVE} is simply to set the limit value which is second on the return stack equal to the current index value on top of the return stack. Thus, the next time that {LOOP} (or {+LOOP}) is executed the loop will not be repeated. {LEAVE} is normally placed inside an IF structure within the DO loop, as illustrated here:

[3]Most FORTH systems define a special printing word {.R}, which is the same as {.} except that it prints the number second on the stack, right justified, in a field width given by the value on top of the stack. Replacing {.} by the phrase {4 .R} in {Timestable} will thus result in a neater columnar output format. For a definition of {.R} see chapter 8.4.

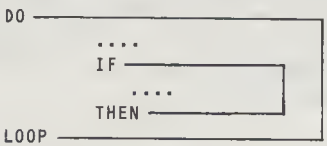
```
: example   10000 1 DO
                        I .
                        ?TERMINAL IF
                                    LEAVE
                                THEN
                    LOOP ;
```

This example assumes that a word {?TERMINAL} has already been defined, to have the effect of checking the keyboard to see if a key has been pressed. {?TERMINAL} should return the flag 'true' if a key has been pressed, 'false' if not. (Most FORTH systems do define such a word, although it is not in the FORTH-79 standard.)

```
?TERMINAL       ( → flag)       Set flag to true if a key
                                 has been pressed, false if not.
```

Upon execution {example} will simply count from 1 to 9999, but may be halted at any time by hitting any key on the keyboard. A useful facility if you do not want to wait until the program completes normally!

Before leaving this example, it is worth noticing the form of the nested IF structure inside the DO loop:



Joining the IF and THEN, and the DO and LOOP with lines shows that the IF structure is fully enclosed within the DO loop, and the whole structure is therefore correctly formed. If the lines should cross over, then the structure is illformed, and will not work correctly, as shown by figure 5.6.

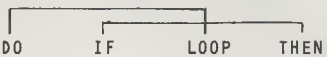


Figure 5.6 NOT a valid FORTH nested structure

If you are ever unsure of the correctness of nested structures, then simply apply this 'line' test, by joining corresponding DO .. LOOP, IF .. THEN (or BEGIN .. UNTIL, and BEGIN .. REPEAT) words, and if any lines should cross, then the structure is probably illformed.

5.6 The UNTIL loop

The DO loop is often referred to as a 'definite' loop because of the way that the number of repetitions is definitely determined before the loop starts. (Unless you use {LEAVE} within the DO loop!) The UNTIL loop and the WHILE loop are known as 'indefinite' loops because in both cases the number of repetitions cannot be known until during execution of the loop. In each case the continued repetition of the loop depends upon the result of a conditional test within the loop.

The general form of the UNTIL loop is as follows:

```
            BEGIN
                  FORTH words
condition UNTIL
```

in which the 'FORTH words' are executed repetitively *until* the 'condition' is 'true'. The 'condition' will normally involve comparison words, like the conditional part of the IF structure – and, moreover, the 'condition' normally tests a result of the FORTH words inside the loop. The condition test must leave a 'flag' on top of the stack which will be tested by {UNTIL}. An example of a colon definition involving an UNTIL loop is as follows:

```
: wait  BEGIN
            KEY
   32 = UNTIL ;
```

Another new word is introduced in this example {KEY}, which has the effect of waiting until a key has been pressed on the keyboard, and then leaves the ASCII value of the key on top of the stack:

```
KEY     ( → char)       Wait until a key has been pressed
                         and leave its value on the stack.
```

The 'condition' part of the UNTIL loop in {wait} simply tests the value of the key pressed for the value 32 (='space'), and loops indefinitely if it was not. The overall effect of {wait} will thus be to cause the system to wait until you press the 'space' key on the keyboard before responding. Here is {wait} described in more detail:

```
Word    Stack Effect              Comment
BEGIN   ( → )                     Start loop (no stack effect)
    KEY     ( → char)             Get character from keyboard
    32      (char → char 32)      ASCII value of 'space'
    =       (char 32 → flag)      flag is 'true' if char=32
UNTIL   (flag → )                 Loop back to BEGIN if flag 'false'
```

5.7 The WHILE loop

The structure of the WHILE loop is slightly more complex than the UNTIL loop and takes the general form as follows:

```
BEGIN   condition   WHILE
    FORTH words ...
REPEAT
```

If the 'condition' is 'true' then the 'FORTH words' will be executed and the loop repeats (back to BEGIN), otherwise if the 'condition' is 'false' then the 'FORTH words' are not executed, and the loops ends. Or, to put it another way, everything between BEGIN and REPEAT executes repetitively *while* the condition remains 'true'. Again the condition test must leave a flag on top of the stack which will be tested by {WHILE}.

The WHILE loop is often interchangeable with the UNTIL loop. We could, for example, rewrite our {wait} definition using a WHILE loop as follows:

```
: wait  BEGIN   KEY 32 = NOT  WHILE
            ( do nothing )
        REPEAT ;
```

noticing that the logic of the condition is reversed – this loop repeats *while* the key pressed is *not* 'space'.

Although the WHILE and UNTIL loops may seem to be almost identical, or even interchangeable, there is a crucial difference between them, namely, that the UNTIL loop *always* executes at least once, whereas the WHILE loop may not

execute at all – if the condition test turns out to be false first time. It is this difference which enables the FORTH programmer to decide which looping structure is appropriate to a given problem.

5.8 FORTH Structures in action

The order of execution of the words in a colon definition is strictly determined by the control structures within the definition. Furthermore, because we cannot have incomplete control structures (IF without THEN, DO without LOOP etc.), or GOTO, it is usually easy to determine the order of events within a colon definition at run-time. Let us take, as an example, the following definition:

```
: BARPRINT  DUP 0> IF
              0 DO ." *" LOOP   ( true words )
            ELSE
              DROP ." -"
            THEN
            CR ;
```

There are only two major paths through {BARPRINT}; either the true words or the false words of the IF structure will be executed, but never both. Figure 5.7 illustrates these two paths.

```
                                        repeat at least once
path1:  DUP 0> (IF) 0 (DO)  ." *"  CR ;

path2:  DUP 0> (IF) DROP ." -" CR ;
```

Figure 5.7 The order of events within a colon definition

The control structure words are not shown in figure 5.7 except where they affect the stack at run-time (and the word is shown bracketed). Two important things to notice are that execution always starts with the first word in the definition, {DUP} in the case above, and always ends with the terminating semi-colon. If {BARPRINT} is included in another definition, for example:

```
: BARTEST 11 -10 DO  I BARPRINT  LOOP ;
```

when the word {BARPRINT} is 'called' during the execution of {BARTEST}, either path1 or path2 of figure 5.7 executes, but it is the final semi-colon that 'returns' execution back to {BARTEST}.[4]

In this way, a complex program consisting of many 'levels' of colon definition still executes in an orderly and predictable manner. Providing that individual colon definitions are kept simple, making sure that a FORTH program executes in the right order presents no great problem. Simplicity is the key; FORTH programmers generally agree that a colon definition should never contain more than 3 or 4 control structures.

There are occasions when we do have to 'break out' of a word during execution. We might, for example, encounter an irrecoverable error that makes continued execution impossible. FORTH provides two words, {ABORT} and {QUIT}, both of which will halt execution and return control to the keyboard. {ABORT} clears all stacks and usually prints a message; {QUIT} leaves the normal stack intact and

[4]Chapter 9.5 will explain this mechanism in more detail.

prints no message. As an example, we could redefine the division operator {/} to check for division by zero, as follows:

```
: /     DUP                     ( duplicate the divisor )
          IF                    ( if non zero )
            /                   ( perform the division )
          ELSE
            ." Division by zero " ABORT
          THEN ;
```

This new division operator may be incorporated into subsequent definitions and will behave just like the old {/}, except that whenever division by zero is attempted the message "Division by zero" is printed, and execution is halted. We could have used the word {QUIT} instead of {ABORT}, in which case the normal stack remains intact and may be examined, for debugging purposes, when the division by zero error occurs.

5.9 Summary and Exercises

The following new words have been introduced in this chapter:

Control Structures:

`DO` `(n1 n2 → )`

Used in a colon definition in the form:

```
DO .. LOOP or,
DO .. +LOOP
```

DO sets up a definite loop with initial index value n2 and limit value n1.

`LOOP` `( → )`

Increment the DO .. LOOP index by +1 and terminate the loop when the index equals (or is greater than) the limit value.

`+LOOP` `(n → )` `"plus-loop"`

Add n to the DO .. +LOOP index using signed addition {+}, and compare the new index with the limit value. Terminate the loop if the index is equal to or greater than the limit, for n positive; or if the index is less than the limit, for n negative.

`I` `( → n)`

When used in the form DO .. I .. LOOP copies the index value onto the stack.

`J` `( → n)`

When used in the form DO .. DO .. J .. LOOP .. LOOP copies the index value of the outer loop onto the stack.

`LEAVE` `( → )`

Set the limit value of a DO loop equal to the current index value so that the loop is terminated at the next LOOP or +LOOP. The index remains unchanged and any words between LEAVE and LOOP or +LOOP are executed normally.

`BEGIN` `( → )`

Marks the start of an UNTIL loop or a WHILE loop, and is used in a colon definition in the form,

```
BEGIN ... flag UNTIL or,
BEGIN ... flag WHILE ... REPEAT
```

`UNTIL` `( flag → )`

In a BEGIN .. UNTIL loop, if the flag is false then execution loops back to BEGIN. If the flag is true the loop is terminated.

`WHILE` `( flag → )`

In a BEGIN .. WHILE .. REPEAT loop, if the flag is true then execution continues through to REPEAT and then loops back to BEGIN. If the flag is false the loop terminates and execution continues after REPEAT.

`REPEAT` `( → )`

Mark the end of a BEGIN .. WHILE .. REPEAT loop as above.

Input:

`KEY` `( → char )`

Wait for a key press and leave the ASCII value of the character on the stack.

Miscellaneous:

`ABORT` `(n1 n2 .... → )`

Clear the normal and return stacks and return control to the keyboard.

`QUIT` `( → )`

Clear the return stack and return control to the keyboard.

Exercises –

1) Define a word which will have the effect of printing a block of stars, so that we could type:

```
4 stars
****
****
****
****
ok
```

2) Write a program to add up all of the integers between a start value and an end value inclusive, where the start and end values are supplied as follows:

```
1 10 sumall
```

3) Write a program to print a 'countdown' from a specified start value, to zero, with a delay between each count of approximately one second. Print an appropriate message at zero, for example, 'We have liftoff'!

4) What will be printed out by the following loops:

```
: ex1  16 0 DO  I .  3 +LOOP ;
: ex2  0 10 DO  I . -1 +LOOP ;
: ex3  5 BEGIN  DUP .  5 +  DUP 100 > UNTIL  . ;
```

5) Write a program to print selectively only those numbers, between specified start and end values, which are exactly divisible by a third value, also specified at runtime.

6) Devise a {DUMP} program which will print out the contents of memory, starting at a specified address, in blocks of 8 lines by 8 bytes. At the end of each block halt and wait for a key to be pressed, if the key is 'space' then continue, otherwise exit the program. (Hint: you will need two nested DO loops inside an UNTIL loop.)

6
Editing, Saving and Loading FORTH programs

We have now reached the stage in the book of having covered sufficient FORTH words to be able to start constructing programs of a reasonable and useful complexity. But clearly the method of direct entry (into the keyboard) used for trying out examples so far is inappropriate for complex programs under development. We really need to be able to edit, save and load programs onto disk or cassette in 'source' form (that is, as they would be typed in directly). FORTH does provide these facilities but in a novel way in which the disk or cassette acts like an extension of memory. Computer scientists call this 'virtual memory' and in practical terms it means that FORTH program 'source' can be very large, without taking up much of the system memory.

6.1 The FORTH LOADing Concept

FORTH divides disk or cassette into 'blocks' of 1024 characters each. Blocks are fetched one at a time into 'block buffers' in RAM, for editing or execution. Programs may, however, extend over any number of blocks, and a block can contain commands to load successive blocks so that the FORTH programmer need not load each block separately.

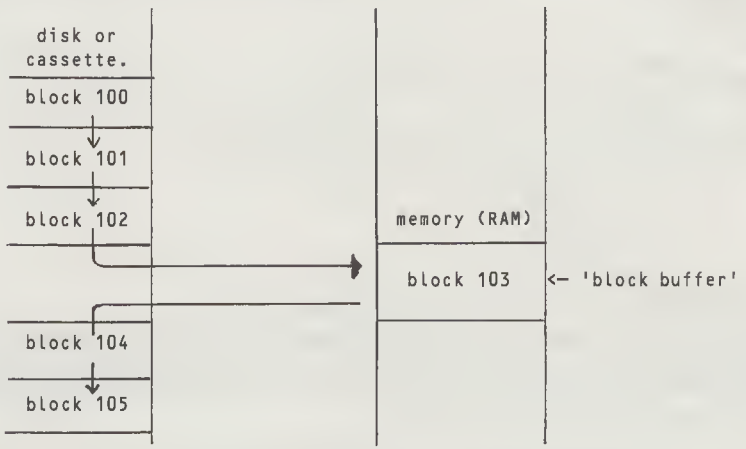


Figure 6.1 The FORTH LOADing Concept

Figure 6.1 illustrates this concept by supposing that a large FORTH program has been edited into blocks 100 to 105 inclusive. To load the whole program (which will almost certainly consist of a 'vocabulary' of colon definitions), the FORTH programmer may simply type:

```
100 LOAD
```

The word {LOAD} means 'fetch the block specified by the number on top of the stack from disk or cassette into a block buffer, then pass it into the FORTH interpreter as if it had been typed in directly'. Any colon definitions in the block will thus be compiled and appended to the dictionary as per normal. If the phrase {101 LOAD} has been included at the end of block 100, then, instead of returning control to the keyboard, block 101 will be loaded immediately after block 100, using the same block buffer. Any number of blocks may be 'chained' together in this way, so that a large program can be loaded all in one go, without taking up more than 1 kbyte of system memory (for the block buffer). Figure 6.1 shows block 103 in the process of being loaded.

The FORTH editor treats each 'block' as 16 lines by 64 characters – a convenient 'screen' full – and so our 6 block example of Figure 6.1 could accommodate a 96 line FORTH program. (Some FORTH systems refer to a block as a 'screen' but the two terms are generally interchangeable.) There are no special rules about what a block can contain. Anything you can type in directly (which is everything!), may be edited into a disk or cassette block. Our complete program in blocks 100 to 105 will probably consist of a collection of colon definitions, variable and constant definitions, lots of comment, and some FORTH intended to be executed directly during LOADing.

Here is what a single block might look like when LISTed. (The word {LIST} 'fetches' a block and then lists it as 16 numbered lines on the terminal.)

```
 0  ( The Complete FORTH, Chapter five examples )
 1
 2  : Squares              ( print 0 to 9 squared )
 3    10 0 DO
 4           I I * .
 5         LOOP ;
 6  Squares                ( try out Squares )
 7
 8  : Timestable           ( print a ten times table )
 9    CR 11 1 DO
10           11 1 DO
11                 J I * .
12               LOOP
13           CR
14           LOOP ;
15 Timestable              ( try out Timestable )
```

Line 0, by convention, consists of comments describing the content of the block. The two colon definitions of the block, on lines 2-5 and 8-14, are suitably indented for readability. Finally lines 6 and 15 will cause the newly defined words to be executed, for testing, again at LOAD time.

Each line in the block will be loaded into FORTH strictly in order from line 0 through to 15. This of course means that the content of a block – or series of blocks – cannot be in any order, but must follow the same rules that apply when typing in directly. In particular a newly defined word cannot be referred to until after its definition.

Notice that everything in the block could be typed in directly, but obviously there are great advantages in editing lengthy definitions onto disk or cassette blocks, since if there are errors in the definitions, as there are in any program under

development, a simple 'edit' and reLOAD will allow us to rapidly try the definition again without tedious retyping. Another good reason for using blocks for program development is that a well commented and laid out block is self-documenting, and readable by other FORTH programmers.

Most systems actually use more than one block buffer (normally two or three), and FORTH automatically decides which to use for a particular LOAD or LIST on a 'least recently accessed' basis. To illustrate what this means let us suppose that we are developing a set of colon definitions in block 100 and FORTH has assigned block 100 to block buffer number i (in a system with two block buffers, i and ii). Should we then wish to LIST block 95 for reference, FORTH will fetch the block into buffer number ii, since buffer number i has been more recently accessed. Having examined block 95, we can resume editing block 100 by typing `{100 LIST}`, and block 100 will not have to be re-read from disk or cassette since it is still contained in buffer i. In this way, disk or cassette transfers are kept to a minimum while developing one particular block.

If, in the example above, we had typed `{96 LIST}`, to examine block 96 instead of typing `{100 LIST}` to resume editing block 100, then buffer i would have been used for block 96, since buffer ii had been more recently used. But, before fetching block 96 FORTH will automatically save the contents of buffer i into block 100 on disk or cassette, so that the newly edited block 100 will not be lost. Thus, the FORTH programmer does not have to explicitly 'save' the block.

In pratice there are occasions when we do need to 'save' any newly edited blocks, such as before changing disks, or switching the power off, or just as a safety precaution before trying out some new (and hazardous) definitions! Accordingly FORTH does provide a 'SAVE-BUFFERS' command, and I shall examine this and other details later in the chapter.

6.2 The Editor

The FORTH-79 standard does not specify an editor vocabulary and, as a result, editors from different FORTH implementations often differ considerably. I will outline here a set of 'typical' editor words, (from a FORTH Implementation Group model). Readers with FORTH systems are recommended to consult their system documentation while reading this section, for more detailed information on their own editor words.

On many FORTH systems the editor vocabulary, (that is, the collection of editing words), is not normally accessible without either first LOADing a set of disk or cassette blocks containing the editor, or on some systems simply typing EDITOR. Again, your system documentation should tell you how to invoke the editor vocabulary if it is not present already.

Suppose that we would like to enter some newly devised FORTH into a block. First, we must locate an empty block, or one whose contents are no longer needed. (On some cassette based systems this is not necessary.) The best way to check that the block is suitable, and prepare it for editing, is to LIST the block, as follows:

```
100 LIST
 0
 1
```

```
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
ok
```

Usually the first thing we want to do is to enter new text into the block. This is best achieved line by line using the editing word {P}, for Put. Typing a line number, from 0 to 15, followed by {P}, and up to 64 characters of text terminated by the 'return' key will have the effect of Putting the text into the specified line. (If the line did previously contain anything this will be overwritten by the new text.) For example:

```
0 P ( Test block ) ok
2 P : Squares              ( print 0 to 9 squared ) ok
3 P   10 0 DO ok
4 P           I I * . ok
5 P         LOOP ; ok
```

Having typed in a few lines, we will probably want to list the block again to make sure they have gone in correctly. We could type {100 LIST}, but it is easier to use the editor word {L}, which has the effect of listing the block buffer currently in use:

```
L
 0 ( Test block )
 1
 2 : Squares              ( print 0 to 9 squared )
 3   10 0 DO
 4           I I * .
 5         LOOP ;
 6
 7
 8
 9
10
11
12
13
14
15
ok
```

If we want to look at just a single line in the buffer, the word {T}, for Type, is more convenient. For example:

```
2 T
2 : Squares              ( print 0 to 9 squared )
ok
```

To insert a new line into the buffer between two old lines requires two commands. First of all {S} which Spreads the text in the buffer, inserting a blank line, and then {P} to Put the new line into the buffer. For example:

```
2 S ok
2 P Squares                ( try out Squares ) ok
```

will move the definition of Squares to start on line 3. The reverse operation, of Deleting a line and moving up all lower lines to close the gap, is accomplished by the word {D}.

Finally, we could move a complete line from one place in the buffer to another, by using a single line scatchpad called the PAD. The Delete command {D}, in fact, places the deleted line into the PAD, which may then be copied out of the PAD onto another line of the buffer using the Insert command {I}, which Spreads the buffer, and then Inserts the text from the PAD into the new blank line. For example, to move the new line 2 into line 8, we would type:

```
2 D ok
8 I ok
```

and a full Listing of the block now appears:

```
L
 0 ( Test block )
 1
 2 : Squares                ( print 0 to 9 squared )
 3   10 0 DO
 4          I I * .
 5        LOOP ;
 6
 7
 8 Squares                  ( try out Squares )
 9
10
11
12
13
14
15
ok
```

Here is a summary of the editing words covered so far:

```
P text   (n → ) Put text (terminated by 'return') into line n.
L        ( → )  List the block buffer currently being edited.
T        (n → ) Type line n.  Also copy it into the PAD.
S        (n → ) Spread the buffer so that line n becomes blank.
D        (n → ) Copy line n into the PAD then move up the lower
                lines to close the gap.
I        (n → ) Spread the buffer then insert the text from the
                PAD into the new line n.
PAD      ( → addr)  Leave the start address of the PAD.
                    (FORTH-79 word).
```

These editing words are sufficient to allow FORTH to be entered and edited on a line by line basis. The FIG FORTH model editor does specify a number of additional words which allow more sophisticated editing including, for example, the

alteration of 'strings' within lines without having to retype the whole line.

Let us assume that we are ready to test the new FORTH in block 100. To load the block type:

```
100 LOAD
0 1 4 9 16 25 36 49 64 81 ok
```

and {Squares} has compiled correctly (lines 2-5), and executed correctly as shown by its output.[1]

Suppose, on the other hand, that we had mistyped one of the words in the definition for {Squares}, 'LLOP' instead of 'LOOP' in line 5:

```
100 LOAD
LLOP ?
```

FORTH will print the offending word, together with the error message '?' meaning 'word not found in dictionary'. The LOAD is then aborted and control returned to the keyboard, ready for us to edit and reLOAD the block.

As soon as the block is completed and tested we will want to save it back onto disk or cassette. As I indicated earlier, this will happen automatically should we go on to LIST and edit further blocks; as soon as the buffer occupied by our completed block 100 is needed for another block, then block 100 will be written back onto disk or cassette. Alternatively, if we have finished editing and testing we can save the completed block 100 by typing simply:

```
SAVE-BUFFERS
```

Newcomers to FORTH are recommended to use this command regularly until they become familiar with 'block handling'!

6.3 More BLOCK handling

The three operations LIST, LOAD and SAVE-BUFFERS, together with an editor vocabulary, are normally all that is required for the creation of applications programs on disk or cassette. FORTH does provide an additional set of block handling operations which are useful for setting up block input-output under program control so that, for example, a program may use disk or cassette 'data' blocks. This section will cover these techniques and is not essential reading for the newcomer to FORTH.

The word {BLOCK} is the basic block fetch operation (used to define LIST and LOAD); its effect is to fetch the specified block into the least recently accessed block buffer, if it is not already in memory, and to save the old contents of the block buffer first, if necessary. {BLOCK} does not process the block in any way after fetching it, but leaves the start address of the block buffer on the stack:

```
BLOCK          (n → addr)
```

This address may then be used to locate an item of data in the block, to extract the

[1]On some cassette based systems the LOAD command has the effect of always reading the block off cassette regardless of whether the block is already in a buffer or not. If this is the case, an alternative command is usually provided to load the block already in memory (ENTER or EXEC are two examples).

data, or modify it. As an example of the use of {BLOCK}, here is a definition for an {INDEX} word, to print line zero of each of a set of specified blocks:

```
: INDEX
        1+ SWAP DO                      ( loop through blocks )
                  I BLOCK               ( fetch a block )
                  64 0 DO               ( print line 0 )
                         DUP C@ EMIT 1+
                       LOOP
                  DROP CR               ( tidy stack, newline )
                LOOP ;
```

To index blocks 100 to 105 inclusive, type:

```
100 105 INDEX
```

The definition for {INDEX} uses the word {EMIT} which will be covered in detail in chapter 7. The important thing to notice here is the very simple way in which a block may be fetched, and data extracted from it.

The {BLOCK} operation can equally easily be used to define operations to 'save' or 'load' numerical data as a disk or cassette block, for example:

```
CREATE data 80 ALLOT            ( create a 40 element array )

: savedata                      ( save 'data' in block 150 )
            data                ( address of data array )
            150 BLOCK           ( fetch block )
            40 MOVE             ( move data into block )
            UPDATE ;            ( mark as updated )

: loaddata                      ( load 'data' from block 150 )
            150 BLOCK           ( fetch block )
            data                ( address of data array )
            40 MOVE ;           ( move data from block )
```

The numerical data are stored in the block in 'binary' form, which is an efficient and compact use of storage. (We could fit 512 single length numbers into one block.) It does mean, however, that LISTing the block will not produce an intelligible output. The word {MOVE} is used to transfer the data to and from the block buffer; the stack description of {MOVE} is as follows:

```
MOVE            (addr1 addr2 n → )
```

where n (16 bit) numbers stored in memory starting at addr1 are copied into memory at addr2 onward. The move starts by copying the number at addr1 into addr2, then addr1+2 to addr2+2, and so on.

Notice the use of the word {UPDATE} which has the effect of 'marking' the block buffer containing block 150 as 'updated'. This ensures that on the next LIST, LOAD, BLOCK or SAVE-BUFFERS operation which needs to use the same block buffer, the old contents will first be saved back into block 150 on disk or cassette. Whenever a block is edited it is automatically marked as 'updated', but when we alter a block under program control then we must explicitly update the block, hence the word {UPDATE}.

If we should need to initialise a block, then the word {BUFFER} is more useful than {BLOCK}. {BUFFER} has the effect of simply assigning the least recently accessed buffer to the specified block, saving its old contents if UPDATEd, but not fetching

the new block into the buffer. Its stack effect is similar to {BLOCK}:

```
BUFFER          (n → addr)
```

As an example here is a definition of a word to {CLEAR} a block to contain all spaces:

```
: CLEAR
        BUFFER                  ( assign a buffer )
        1024 32 FILL            ( fill it with spaces )
        UPDATE ;                ( and mark as updated )
```

To clear, for example, block 105, type:

```
105 CLEAR
```

and when the buffer contents are written out to disk or cassette (by SAVE-BUFFERS, for example), the old block 105 will be overwritten by the new empty block. (See chapter 7 for an explanation of {FILL}).

To conclude this section, three more words should be mentioned, {SCR}, {BLK} and {EMPTY-BUFFERS}.

{SCR} is a system variable containing the block number of the most recently LISTed block, and is useful when devising new editing words. {BLK} is another system variable, and contains the block number of the block currently being interpreted by LOAD. The FORTH interpreter (and the word {QUERY} covered in chapter 7), uses {BLK} to determine where the input stream should come from; if BLK is zero, then the input is from the keyboard, if BLK is non-zero, then input is from a block-buffer.

{EMPTY-BUFFERS} has the effect of initialising all of the block buffers by marking them as 'empty', so that none of the buffer contents will be written out to mass storage, even if UPDATEd. This is useful if you should accidentally corrupt the contents of a buffer (while editing, for example), and you do not want to overwrite the old disk or cassette block. Simply type {EMPTY-BUFFERS}, and then LIST the block, to restore it as it was.

6.4 Vocabulary Management

As we have seen already a complete program (or 'application' to use FORTH terminology) generally consists of a collection of colon definitions, or to put it more precisely, a 'vocabulary' of new 'words'. When a vocabulary is LOADed, the words are compiled and added into the dictionary in such a way that the new words are 'linked' into the existing dictionary. Figure 6.2 illustrates this linkage by showing two new dictionary entries; {ONE} and {TWO}.

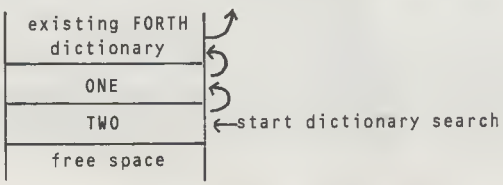


Figure 6.2 Dictionary Linkage

All dictionary searches will start with the most recently defined word, {TWO}, and work backwards. If the word in the input stream is not one of the newly defined

words, then the search will continue in the standard FORTH dictionary; the linkage ensures that new words, and old, may be freely intermixed in the input stream.

Any number of vocabularies may be LOADed and linked in this way, and a dictionary search will work through each vocabulary, in reverse order of LOADing, eventually working back into the standard dictionary. While this structure may be satisfactory, FORTH does provide a more elegant way of grouping vocabularies so that each is essentially separate but still easily accessible. The words, {VOCABULARY} and {DEFINITIONS} will achieve this, as shown by the following examples:

```
 0 ( Test vocabulary management )
 1 VOCABULARY FRENCH  IMMEDIATE
 2 FRENCH DEFINITIONS
 3 : ONE  ." un " ;
 4 : TWO  ." deux " ;
 5
 6 FORTH DEFINITIONS
 7 VOCABULARY GERMAN  IMMEDIATE
 8 GERMAN DEFINITIONS
 9 : ONE  ." ein " ;
10 : TWO  ." zwei " ;
```

Here we have defined two separate vocabularies, named {FRENCH} and {GERMAN}, both linked back into the FORTH vocabulary. The total dictionary linkage for this is shown in figure 6.3.

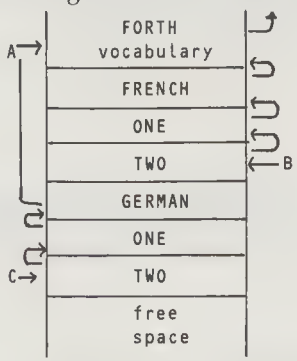


Figure 6.3 A Two Vocabulary link structure

The pointers A, B and C in figure 6.3 illustrate that there are three possible 'starts' for a dictionary search. We can start at point A, and miss out the two new vocabularies completely by writing:

```
FORTH ok
ONE ONE ?
```

To start at point B, and search FRENCH and then FORTH, write:

```
FRENCH ok
ONE un ok
```

or to start at point C, and search GERMAN and then FORTH, write:

```
GERMAN ok
TWO zwei ok
```

As illustrated here, different vocabularies can use the same name for different definitions. Each version of the same word is accessible simply by preceding the word by the name of its vocabulary.

In FORTH terminology the vocabulary in which a dictionary search will start is known as the 'context' vocabulary, and is determined by the value of the system variable {CONTEXT}. Words defined by the defining word {VOCABULARY} (such as {FRENCH} or {GERMAN}), will have the effect when executed, of setting the value of {CONTEXT} to the corresponding pointer (B or C). The word {FORTH} sets the FORTH vocabulary as the context vocabulary.

Notice also the use of the word {IMMEDIATE}, following the definitions of {FRENCH} and {GERMAN} on lines 1 and 7 of the example block above. This ensures that the words {FRENCH} and {GERMAN} will be always be executed, even when they occur within a colon definition; a necessary facility as we shall see shortly. (For a full explanation of {IMMEDIATE} see chapter 9.5.)

A second system variable, {CURRENT}, determines the vocabulary into which new definitions will be placed; the word {DEFINITIONS} sets {CURRENT} equal to {CONTEXT}. Thus:

```
FORTH DEFINITIONS
```

means that new definitions will be linked into point A in figure 6.3, and become part of the FORTH vocabulary. The new definition may still refer to other vocabularies, by altering the context during the definition, as follows:

```
FORTH DEFINITIONS ok
: TWO_TRANSLATE
        FRENCH TWO ." = "
        GERMAN TWO ; ok
FORTH TWO_TRANSLATE deux = zwei ok
```

The words {FRENCH} and {GERMAN} are executed at compilation time, (and produce no compiled code), they simply alter the context so that the first {TWO} will be found in the FRENCH vocabulary, and the second {TWO} in the GERMAN vocabulary.

6.5 Summary

The following new FORTH-79 words have been introduced in this chapter:

Mass storage input-output:

LIST (n →)

List the contents of block n. Set the variable SCR to n.

LOAD (n →)

Interpret block n by making it the input stream. (Preserve the pointers >IN and BLK into the present input stream so that it will be resumed when interpretation of the block ends.)

SCR (→ addr) "s-c-r"

System variable containing the number of the block most recently listed.

BLOCK (n → addr)

If block n is not already in memory, then fetch it from mass storage into the block buffer least recently accessed. (Saving the old contents of the block buffer first, if they had been modified i.e. UPDATEd.) Then leave the start address of the block buffer containing block n.

`UPDATE` `( → )`

Mark the most recently referenced block buffer as having been modified so that its contents will automatically be saved onto mass storage should the buffer be needed (by LIST, LOAD or BLOCK) for a different block, or upon execution of SAVE-BUFFERS.

`BUFFER` `(n → addr)`

Assign the least recently accessed block buffer to block n, first saving its old contents if they had been marked as UPDATEd. Do not fetch block n into the buffer, but leave the start address of the buffer.

`SAVE-BUFFERS` `( → )`

Save all block buffers that have been modified (i.e. UPDATEd).

`EMPTY-BUFFERS` `( → )`

Mark all block buffers as empty. Do not save any even if they are marked as UPDATEd.

Miscellaneous:

`PAD` `( → addr)`

Leave the start address of a general purpose 'scratch-pad' used to hold character strings for intermediate processing. The PAD has space for at least 64 characters (at addr to addr+63).

`BLK` `( → addr)` `"b-l-k"`

System variable containing the number of the block currently being interpreted by LOAD as the input stream. If BLK is zero then input is from the keyboard.

Memory:

`MOVE` `(addr1 addr2 n → )`

Copy n 16 bit values starting at addr1 into memory starting at addr2, proceeding toward high memory. If n is zero or negative do nothing.

Vocabulary Management:

`VOCABULARY` `( → )`

A defining word used in the form:

```
VOCABULARY <name>
```

to create (in the CURRENT vocabulary) a new vocabulary called `<name>`.

When <name> is later executed it will become the CONTEXT vocabulary, for dictionary searches, and using the word DEFINITIONS it may also become the CURRENT vocabulary for new definitions. All new vocabularies eventually link back to the FORTH vocabulary.

`CONTEXT` `( → addr)`

A system variable specifying the vocabulary in which dictionary searches will start during interpretation of the input stream.

`CURRENT` `( → addr)`

A system variable specifying the vocabulary to which newly defined words will be appended.

`FORTH` `( → )`

The name of the primary vocabulary. The FORTH vocabulary is normally both the CONTEXT and CURRENT vocabulary unless changed using VOCABULARY and DEFINITIONS. Executing FORTH restores FORTH as the CONTEXT vocabulary.

`DEFINITIONS` `( → )`

Sets CURRENT equal to CONTEXT so that subsequent definitions will be appended to the vocabulary previously selected as CONTEXT.

7
Number and String Input and Output

We have already used the "dot" and "dot-quote" operations for number and text output respectively, and for number input we have relied on the fact that numbers in the input stream are automatically pushed onto the stack. FORTH does, however, provide an extensive additional set of input-output operations which are used in combination, rather than individually, to develop new input-output words for virtually any application.

In an attempt to make the text of this (and the following) chapters as uncluttered as possible, many of the examples are presented without detailed (stack) analysis. All examples are, however, amenable to analysis using the stack notation of chapter 3.6, if the reader should seek further clarification.

7.1 Character input-output, the basics

The simplest of the character output operations is the word {EMIT} which will print the character whose ASCII value is on top of the stack.[1]

For example:

```
65 EMIT Aok
```

prints the single character "A", since 65 is the ASCII value for the character "A" (in decimal). To print a string of characters using {EMIT} just emit each character in turn, for example:

```
89 EMIT 69 EMIT 83 EMIT YESok
```

To avoid having to look up the ASCII value of each character, use the reverse operation {KEY}, which waits until a key has been pressed and then leaves its value on top of the stack:

```
KEY ok          ( After hitting 'return' press 'Y' )
. 89 ok
```

The string printing word "dot-quote" {."} is much easier to use than {EMIT} for character strings:

```
." YES" YESok
```

{EMIT} is more often used for printing control-characters, which cannot be included inside a dot-quote string. Three examples of these are already part of the standard vocabulary; {CR}, {SPACE} and {SPACES}:

```
: CR    13 EMIT  10 EMIT ;      ( print carriage return, line feed )
: SPACE 32 EMIT ;               ( print one space )
: SPACES  0 DO SPACE LOOP ;     ( print n spaces )
```

These definitions illustrate also that much of the standard vocabulary is itself defined in FORTH! Further special printing operations can easily be defined like this if needed, for example:

[1] A full description of ASCII is given in the glossary of FORTH terminology.

```
: TAB   9 EMIT ; ok              ( horizontal tab )
: CLRS 12 EMIT ; ok              ( clear screen )
CLRS TAB ." New page" CR

          New page
ok
```

As a final example of the use of both {EMIT} and {KEY}, suppose that we need to input a fixed length string into memory, from the keyboard, and later print the string. The best way to reserve space in memory to hold the string, is to {CREATE}[2] a new dictionary entry and {ALLOT} space in it, as described in chapter 3.5:

```
CREATE STRING 6 ALLOT ok         ( 6 bytes of space )
```

The newly defined word {STRING} will leave the address of the start of the allotted space on top of the stack and can be used to define {GETSTR} and {PRINTSTR} as follows:

```
: GETSTR  CR ." ?"               ( print a prompt )
          STRING                 ( address onto stack )
          6 0 DO                 ( loop thru characters )
                KEY                ( input a key )
                DUP EMIT           ( echo it )
                OVER C!            ( and store it )
                1+                 ( increment address )
              LOOP
          DROP ; ok              ( tidy up stack )

: PRINTSTR
          STRING                 ( address onto stack )
          6 0 DO                 ( loop thru characters )
                DUP C@ EMIT        ( fetch and print )
                1+                 ( increment address )
              LOOP
          DROP ; ok              ( tidy up stack )

GETSTR                           ( test GETSTR )
?ABCDEFok

PRINTSTR ABCDEFok                ( print STRING )
```

{GETSTR} has two limitations; one is that all six characters must be typed in (and not terminated by the 'return' key); the other is that the 'backspace' key cannot be used to correct typing errors. These limitations could be overcome with a more sophisticated definition for {GETSTR}, but since FORTH already has a number of powerful 'buffered' string input operations it is more sensible to use one of these.

7.2 String input-output 1

In general the FORTH programmer has two options available for string input to programs; one is to take the string input from the original input stream (for which BASIC has no equivalent); the other is to halt the program and wait for a line of input to be typed into the keyboard (as in {GETSTR} above, or the BASIC 'INPUT' statement). The first of these two options uses the important operation {WORD} and is described in this section. Section 7.3 covers the second.

[2]On non FORTH-79 systems you may have to type {0 VARIABLE STRING 4 ALLOT} to achieve the same effect.

{WORD} may only be used inside a colon definition (I shall explain why shortly) and the following example illustrates its use:

```
: PRINTNEXT  32 WORD COUNT TYPE ; ok
PRINTNEXT sillyexample sillyexampleok
```

{PRINTNEXT} has the rather pointless effect of printing the word typed in after it, but it does illustrate a number of interesting new operations.

{WORD} has the effect, when executed, of copying characters from the input stream into a 'word buffer', up to a delimiter character. Additionally, the number of characters copied is left at the head of the word buffer. Figure 7.1 illustrates this diagrammatically.

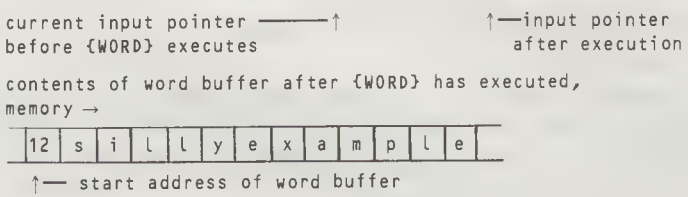


Figure 7.1 {WORD} in Action

{WORD} needs the delimiter character specified on the stack before execution, and leaves the start address of the word buffer on top of the stack after execution:

```
WORD    (char → addr)[3]
```

The effect of {32 WORD} in the example above is therefore to copy up to the next 'space' character. The next word in the example {COUNT} simply fetches the count byte pointed to by the address on top of the stack, and increments the address so that it points to the actual start of the string. Both values are left on the stack:

```
COUNT   (addr → addr+1 n)
```

The stack is now ready for the word {TYPE} which will print the string whose actual start address and character count are specified on the stack:

```
TYPE    (addr n → )
```

The reason that {WORD} cannot be used outside a colon definition is that FORTH itself uses the operation {WORD} and, as a result, the word buffer, while processing a line of input. Typing:

```
32 WORD sillyword COUNT TYPE
```

will not have the expected effect because by the time {TYPE} is being executed, the word in the word buffer is not "sillyword" (it is, in fact "TYPE"!).

Having established how {WORD} works we can devise a rather more useful operation {PUTSTR}, to copy the next word in the input stream into a special string buffer which we define ourselves:

[3]Some non FORTH-79 systems have a slightly different stack effect for {WORD}, in which the word buffer address is not left on the stack. If your system is one of these then you can probably replace {WORD} in the examples in this section by the phrase {WORD HERE}, but check your documentation first!

```
CREATE STRING 40 ALLOT ok                ( plenty of room )
: PUTSTR
        STRING 40 32 FILL                ( erase STRING )
        32 WORD COUNT                    ( get word buffer )
        STRING SWAP CMOVE ; ok           ( copy into STRING )
```

Two new words are introduced in this definition, {FILL} and {CMOVE}. {FILL} provides the useful function of filling a block of memory, byte by byte, with the same value. Its stack description is:

```
FILL    (addr n byte → )
```

and its action is to fill the 'n' bytes starting at 'addr' with the value 'byte'. The phrase {STRING 40 32 FILL} has the effect, therefore, of filling the 40 bytes starting at the address returned by {STRING} with the value 32. In other words, the STRING is filled with spaces.

The word {CMOVE} is the 'character block move' operation:

```
CMOVE   (addr1 addr2 n → )
```

The 'n' bytes starting at 'addr1' are moved to memory starting at 'addr2'. In the example above {CMOVE} moves the string placed into the word buffer by {WORD}, into our own STRING buffer. Of course, once the string has been copied there is no danger of it being corrupted by the FORTH interpreter, and it does not have to be processed in the same definition as {WORD}.

With a definition for {PRINTSTR} we may test out {PUTSTR} fully:

```
: PRINTSTR
        STRING 40 -TRAILING TYPE ; ok

PUTSTR teststring ok
PRINTSTR teststringok
```

{PRINTSTR} illustrates another useful string utility operation {-TRAILING} which has the effect of reducing the character count on top of the stack by the number of trailing spaces in the stored string (pointed to by the address second on the stack). {TYPE} will then only print the characters copied into the string (which must always be less than 40). A definition of {PRINTSTR} without the word {-TRAILING} would always print all 40 characters of the string.

Before leaving this section, two additional words should be mentioned which might be useful when using {WORD}: {HERE} which returns the address of the next available dictionary location (in most systems this is the same as the address returned by {WORD}, the word buffer moves up as the dictionary grows); {>IN} is a system variable containing the character offset into the input buffer (the input pointer shown in figure 7.1). {WORD} advances the value of >IN while scanning the input stream.

7.3 String input-output 2

The second method of achieving string input in FORTH is to use either the word {EXPECT}, or the word {QUERY}. Both have the effect of halting the program and waiting for a line of input to be typed into the keyboard, the only difference between the two is where the input is stored.

The more general of the two is {EXPECT}, which has the following stack description:

```
EXPECT  (addr n → )
```

{EXPECT} accepts characters from the keyboard into memory starting at 'addr', until either 'n' characters have been typed, or the 'return' key is pressed (whichever comes first). A very useful word {INSTR} can be developed using {EXPECT}, to input characters into our STRING buffer, from the keyboard:

```
: INSTR
        STRING 40 32 FILL                ( clear string )
        CR ." ?"                         ( print prompt )
        STRING 40 EXPECT ; ok            ( expect up to 40 chars )

INSTR
?This is a typed in line ok

PRINTSTR This is a typed in lineok
```

The operation {QUERY} is identical in action to {EXPECT}, except that it expects up to 80 characters, and places them in the FORTH terminal input buffer. {QUERY} is, in fact, another word used by the interpreter; it is the word that is executing whenever you are inputting normally to a FORTH system. The word {QUERY} is in many ways more useful to the FORTH programmer than {EXPECT}, because {QUERY} may be used together with {WORD} to 'parse' the typed input (that is, split it up into separate words).

Suppose, for example, that we are writing a program involving a 'question and answer' dialogue between computer and user (as employed in games programs), and the user must type in three replies on one line, each separated by commas. Using the powerful combination of {QUERY} and {WORD}, we may devise a definition to achieve this, and place each of the three replies in a separate 'string', as follows:

```
CREATE 1reply 10 ALLOT ok              ( define reply strings )
CREATE 2reply 10 ALLOT ok
CREATE 3reply 10 ALLOT ok

: GETREPLIES
        1reply 10 32 FILL              ( clear each string buffer )
        2reply 10 32 FILL
        3reply 10 32 FILL
        CR ." ?"                       ( newline and prompt )
        QUERY                          ( get input from user )
        44 WORD                        ( fetch first word )
        COUNT 1reply SWAP CMOVE        ( move into 1reply )
        44 WORD                        ( fetch second word )
        COUNT 2reply SWAP CMOVE        ( move into 2reply )
        1 WORD                         ( fetch third word )
        COUNT 3reply SWAP CMOVE ; ok   ( into 3reply )

GETREPLIES
?One,Two,Three ok

1reply 10 TYPE One       ok            ( print each reply )
2reply 10 TYPE Two       ok
3reply 10 TYPE Three     ok
```

In practice {GETREPLIES} would be incorporated into other definitions to make up the whole program. The delimiter character used for the first two replies is 44, the ASCII value for a comma, but for the third word the delimiter is 1, which has the effect of causing all of the remaining characters up to the end of the line to be copied

by {WORD}. Notice also that the three different reply strings are given names that differ in the first character rather than the last. This is as a precaution against some FORTH systems in which only the first three or four characters are saved in the new dictionary entry, in which case reply1, reply2 and reply3 would be indistinguishable.

Two additional features of {QUERY} are worth noting. The first is that since {QUERY} uses the terminal input buffer, anything previously in the buffer will be overwritten, including FORTH input. Thus in

```
GETREPLIES  ." This will not work"
```

{GETREPLIES} will execute correctly, but anything following it will not execute. The second is that if {BLK} (described in chapter 6) is non-zero then {QUERY} will attempt to fetch input from a disk or cassette block. This may be used to advantage should we require string input from a disk block at run time.

7.4 Number Bases

A feature that is common to all number input-output operations, which we have not yet exploited, is that the 'base' or 'radix' may be altered from its usual default of base 10 (decimal). Whenever a number is being input, or output, its base is determined by the current value of the system variable {BASE}. Radix conversion is therefore very easy in FORTH, and requires only the definition of words to alter the value of {BASE}. Two examples are:

```
DECIMAL ok
: HEX  16 BASE !  ; ok
: BIN   2 BASE !  ; ok
```

The new words defined above, {HEX} and {BIN}, will have the effect when executed of altering the radix for number input and output to base 16 (hexadecimal), or base 2 (binary) respectively. Notice the precaution of typing DECIMAL before compiling HEX and BIN, to ensure that the numbers 16 and 2 really are treated as decimal numbers. {DECIMAL} is a pre-defined word which sets the value of {BASE} to its normal value of 10, for decimal number input and output.

Using {HEX} and {BIN} we may perform radix conversion, for example:

```
DECIMAL ok                ( check we're in decimal )
16 HEX . 10 ok            ( 16 decimal = 10 hex )

3FF DECIMAL . 1023 ok     ( 3FF hex = 1023 decimal )

9 BIN . 1001 ok           ( 9 decimal = 1001 binary )

10101 DECIMAL . 21 ok     ( 10101 binary = 21 decimal )
```

The unsigned print operation {U.} is more useful than {.} for the conversion of negative numbers:

```
 -1 HEX U. FFFF ok        ( -1 decimal = FFFF hex )

FFFF DECIMAL U. 65535 ok
```

The last example is perfectly correct, since the signed number -1, and the unsigned number 65535 (decimal) are both represented internally by the same 16 bit number!

It is not unusual, while using different number bases, to lose track of exactly which

base you are in. Unfortunately, typing:

```
BASE @ . 10 ok
```

does not help at all, because it will always produce the result 10 whatever base we happen to be in! In HEX, for example, BASE equals 16 (decimal), but this will be printed as 10 (hexadecimal). We can overcome this problem with a special definition:

```
: ?BASE
        BASE @          ( fetch current base )
        DUP             ( duplicate it )
        DECIMAL .       ( print it in decimal )
        BASE !          ( and restore the old base )
; ok
```

which may be used at any time as follows:

```
HEX ?BASE 16 ok
BIN ?BASE 2 ok
DECIMAL ?BASE 10 ok
```

A novel, but often very useful, feature of the unlimited range of number bases in FORTH, is that short strings can be treated as numbers, in base 36. Any string from "A" to "ZZZ" can be represented as an unsigned single length number in base 36, which means that 1, 2 or 3 character string comparisons could be performed by ordinary arithmetic operations. If we go to double precision, then the useful range extends to 6 character strings. As an example, here is an array of short strings defined as if it were a number array:

```
DECIMAL ok
: BASE36  36 BASE ! ; ok

BASE36 ok
CREATE DAYS  SUN , MON , TUE , WED , THU , FRI , SAT , ok

DECIMAL ok
: .DAY  ( print day of week numbered 0-7 )
        BASE @ SWAP             ( save current base )
        2 * DAYS + @            ( fetch the day )
        BASE36 U.               ( print it )
        BASE ! ; ok             ( restore old base )

3 .DAY WED ok
6 .DAY SAT ok
```

7.5 Alternative number input

In the majority of FORTH programs number input is achieved using the stack. The values required by a program are pushed onto the stack before the program is 'run' (by quoting its name), and the program takes them off the stack during execution. There are occasions, however, when a more conventional type of number input is needed, where a program halts and waits for a number to be typed into the keyboard before continuing. (Like the BASIC 'INPUT' statement, when used for numerical input.)

Although FORTH does not provide a 'numerical INPUT' type of operation in the basic dictionary, we can easily define one ourselves using the buffered input of

section 7.3, and the word {CONVERT}. {CONVERT} will convert an ASCII string stored in memory into a double precision number on the stack. All we need to do is get a line of input from the keyboard into memory, using {EXPECT} or {QUERY}, and then {CONVERT} the string into a number, as follows:

```
: INPUT
        0 0                         ( double zero onto stack )
        CR ." ?"                    ( print a prompt character )
        QUERY                       ( get a line of input )
        1 WORD                      ( copy all of it to word buffer )
        CONVERT                     ( convert to a number )
        DROP DROP ; ok              ( tidy up the stack )
```

The detailed stack effect of {CONVERT} is as follows:

```
CONVERT          (d1 addr1 → d2 addr2)
```

The string whose length byte is pointed to by addr1 is converted to a double precision number, which is added to d1 and left on the stack as d2. addr2 contains the address of the first non-convertible character. The double zero in the definition for {INPUT} is the initial value d1, and the final {DROP DROP} clears off addr2, and the top half of d2 to leave a single precision result on the stack.[4] (See chapter 8 for a detailed description of how double precision numbers are stored on the stack.)

To use {INPUT} simply include it in a program wherever number input from the keyboard is required. {INPUT} will wait for a number to be typed in (terminated by 'return'), and leave the number on top of the stack:

```
INPUT    ( → n)          Input n from the keyboard
```

Here are some examples of the use of {INPUT}:

```
INPUT
?1234 ok                            ( type in "1234" return )
. 1234 ok                           ( print using "dot" )

BASE36 ok                           ( go into base 36 )
INPUT
?YES ok                             ( type in "yes" return )
U. YES ok                           ( print using "u-dot" )

DECIMAL ok                          ( back into decimal! )
```

The final example shows that {CONVERT} also uses {BASE} to determine the base of the string being converted, altering {BASE} will allow numbers in bases other than decimal to be {INPUT}.

7.6 Summary

The following new words have been introduced in this chapter:

Character input-output:

```
EMIT                (char → )
```

[4]Some FORTH systems have a slightly different word with the same function called {NUMBER}, whose stack effect is (addr→d). To modify {INPUT} to use {NUMBER} instead of {CONVERT} remove the initial {0 0}, and one of the final {DROP}s.

Print the character whose ASCII value is on the stack.

`SPACE` `( → )`

Print a single space.

`SPACES` `(n → )`

Print n spaces. Do nothing if n is zero or negative.

`TYPE` `(addr n → )`

Print the n characters stored at addr upwards. Do nothing if n is zero or negative.

`COUNT` `(addr → addr+1 n)`

Fetch the character count of the string pointed to by addr. Add one to addr to point to the actual start of the string and leave the address and character count on the stack in TYPE form. Range of n is 0..255.

`-TRAILING` `(addr n1 → addr n2)` `"dash-trailing"`

Reduce the character count of the string starting at addr from n1 to n2 to exclude trailing spaces. n1 must be positive.

`EXPECT` `(addr n → )`

Transfer characters from the keyboard into memory starting at addr until either n has been received or 'return' pressed. Do nothing if n is zero or negative. One or two nulls (zero bytes) are added to the end of the string in memory.

`QUERY` `( → )`

Transfer characters from the keyboard into the terminal input buffer until either 80 characters have been received or 'return' pressed. WORD may then be used to process this text if >IN and BLK are set to zero.

`WORD` `(char → addr)`

Copy characters from the terminal input buffer into the word buffer starting with the first non-delimiter character until the next delimiter char, or until the input stream is exhausted. The character count byte is left at the head of the string, pointed to by addr. If the input stream was empty when WORD is called, then leave a zero count byte.

Number input-output:

`BASE` `( → addr)`

System variable containing the current number base for all number input-output operations.

`DECIMAL` `( → )`

Set the input-output number base to ten (decimal).

`CONVERT` `(d1 addr1 → d2 addr2)`

Convert the string starting at addr1+1 into a double number in the current base, adding this to d1 to leave the result d2. addr2 is the address of the first

invalid character (according to the base).

Memory Operations:

`CMOVE` `(addr1 addr2 n → )` `"c-move"`

Move n bytes from addr1 upwards to addr2 upwards. Do nothing if n is zero or negative.

`FILL` `(addr n byte → )`

Fill memory from addr to addr+n with the value byte. Do nothing if n is zero or negative.

Miscellaneous:

`HERE` `( → addr)`

Leave address of next available dictionary location.

`>IN` `( → addr)` `"to-in"`

System variable containing offset into the current input stream.

8
Double Precision and beyond

This chapter describes a number of interesting 'number' topics including double and mixed precision arithmetic, and formatted number printing. The chapter goes on to show how a useful 'fixed point decimal arithmetic' vocabulary could be developed.

Any references to the FORTH-79 extension word set (i.e. double precision stack manipulation etc.), are accompanied by definitions for these words for the benefit of those whose systems do not include them.[1]

8.1 Double Precision Numbers

In chapter 1, I mentioned that FORTH has the facility for double precision (32 bit) arithmetic, thereby giving a much extended range of numbers. To be more specific, signed double precision numbers can have values within the range:

```
-2,147,483,648 to 2,147,483,647 (decimal)
```

and unsigned double numbers can have values within the range:

```
0 to 4,294,967,295 (decimal).
```

FORTH employs a simple and elegant method for telling the difference between single and double precision numbers in the input stream; if a decimal point[2] appears anywhere in a number, then the number will be interpreted as a double precision number and either pushed onto the stack, or compiled if it is within a colon definition. Typing, for example:

```
DECIMAL ok
1000000. ok
```

will have the effect of pushing the double precision number 'one million' onto the stack. To pop the number off the stack and print it we may use the double precision print operation {D.}[3], as follows:

```
D. 1000000 ok
```

The decimal point used to tell FORTH that "this is a double number", when inputting, is not printed by {D.}; although we could use the number formatting which I describe later in this chapter to include the decimal point, if required. Further, the decimal point in the input number has no significance beyond simply indicating that the number should be read as double precision. For example in:

[1]Note that you can easily check if your system conforms to the FORTH-79 standard, just type {79-STANDARD}.

[2]Some systems will recognise a double number if it contains any of the characters "," "." "/" "-" or ":".

[3]If your FORTH system does not have {D.}, then use the definition at the end of section 8.4 to define it.

```
-0.010 D. -10 ok
-10.   D. -10 ok
```

both -0.010 and -10 are interpreted as the double number 'minus ten'. Most FORTH-79 standard systems do, however, place a count of the number of digits after the decimal point in a variable {DPL}, so that -0.010 in the example above would have resulted in DPL=3, and -10. would set DPL=0. I will show later how to use this extra information to build fixed point arithmetic operations.

A double precision number takes up two 'cells' of the stack, with the upper 16 bit half of the number uppermost on the stack. Thus, if we print a double number using two single number print operations, we will get the following for a small double number:

```
100. ok
. 0 ok      ( print upper half )
. 100 ok    ( print lower half )
```

But if we try the same with a large double number the result will not be meaningful.[4] For example:

```
1000000. ok
. 15 ok
. 16960 ok
```

The FORTH-79 standard specifies two signed double precision arithmetic operations, "d-add" and "d-negate", and one double length comparison "d-less-than":

```
D+      (d1 d2 → dsum) Add double numbers to give a double
                              precision result.
DNEGATE (d → -d)       Reverse the sign of the double number.
D<      (d1 d2 → flag) Flag set true if d1 less than d2.
```

In the stack descriptions here, d,d1 etc. simply indicate signed double numbers, each taking up two stack cells.

We may use {D+} like {+}, but with double numbers. To add, for example, one million and two million, type:

```
1000000. 2000000. D+ D. 3000000 ok
```

{DNEGATE} may be used to define a 'double-subtract' operation, should we require it:

```
: D-  DNEGATE D+ ; ok
2000000000. 1. D- D. 1999999999 ok
```

to subtract 1 from two thousand million! Similarly any number of additional double number operations could be defined for a special application. Here is a selection:

```
: 2DUP   OVER OVER ;     ( duplicate double number )
: 2DROP  DROP DROP ;     ( drop double number )
: D0<    SWAP DROP 0< ;  ( test for negative double number )
: D0=    OR 0= ;         ( test for double zero )
: D=     D- D0= ;        ( test for equal double numbers ).
: DABS   DUP 0< IF  DNEGATE  THEN ;  ( make double number positive )
```

[4]We can explain this result as follows:
15 * 65536 = 983040, add 16960 and we have 1000000 !

8.2 Mixed Precision

The FORTH-79 standard specifies four 'mixed precision' arithmetic operations. That is, operations involving a mixture of double and single numbers:

`*/`	`(n1 n2 n3 → quot)`	Multiply n1 by n2 to give a double precision intermediate result. Then divide this by n3 to give a single precision quotient.
`*/MOD`	`(n1 n2 n3 → rem quot)`	As above, but leave a single precision remainder as well.
`U*`	`(un1 un2 → udprod)`	Multiply two unsigned single numbers to give an unsigned double result.
`U/MOD`	`(ud un → urem uquot)`	Divide the double number ud, by the single number un, leaving single precision remainder and quotient. All unsigned.

In the stack notation used here un represents an unsigned single length (16 bit) number and ud an unsigned double length (32 bit) number.

The first two operations are included primarily to avoid overflow problems in calculations involving multiplication then division. As an example, suppose we need to calculate six-sevenths of a set of numbers. We could type:

```
: Frac  6 * 7 / ; ok
100 Frac . 85 ok
10000 Frac . -790 ok    ( wrong !! )
```

The first result, of 85, is perfectly correct. The second result is completely wrong. Its negative sign is a good indication of that. The reason for the error is that the result of multiplying 10000 by 6 is 60000, which is greater than the largest single precision number FORTH can handle (32767). Redefining `{Frac}` to use `{*/}` will, however, overcome this difficulty since the intermediate result of the multiplication will be held as a double length number. 60000 is, of course, well within the double number range:

```
: Frac  6 7 */ ; ok
10000 Frac . 8571 ok
```

This new definition of `{Frac}` will correctly calculate six sevenths of any single length number.

Another possible application of "times-divide" `{*/}`, is to represent decimal numbers in calculations. Take, for example, 'pi' whose value is approximately:

```
3.1416
```

We can define an operation which multiplies by pi, as follows:

```
: *pi  31416 10000 */ ; ok
```

and use this to define `{AREA}`:

```
: AREA DUP * *pi ; ok      ( radius squared times pi )
45 AREA . 6361 ok
```

The remaining two mixed precision operations are the 'primitive' operations used to define all other FORTH multiplication and division operations, (including those

just described). Using them we may define further operations not present in a standard system.

As an example, suppose we need a double precision multiplication, which will multiply the two double numbers on top of the stack, and leave a double result. To achieve this we can multiply the upper and lower halves of each double number separately, using {U*}, and combine the partial products to produce the 32 bit result. Figure 8.1 shows how the operation works like a long multiplication:

```
                    a       b
                    c       d     *
          ------------------------
                  d*bu    d*bl
          d*au    d*al       0     +
          c*bu    c*bl       0     +
c*au      c*al       0       0     +
-------------------------------------
 p         q         r       s
```

Figure 8.1 Double length long multiplication

This diagram shows how to multiply the 32 bit number ab, (a is the upper half, b the lower half), by the 32 bit number cd, (c is the upper half, d the lower half), to produce the 64 bit result pqrs. Each of the four multiplications involved produces a 32 bit result, whose upper half is indicated by u, lower half by l. If we should only require a 32 bit result, then only the first three multiplications are needed and pq need not be calculated.

By far the easiest way of developing a double multiply on the basis of the algorithm just described, is to use four variables to hold the 16 bit halves, a,b,c and d, as follows:

```
VARIABLE a ok  ( top number, upper half )
VARIABLE b ok  ( top number, lower half )
VARIABLE c ok  ( second number, upper half )
VARIABLE d ok  ( second number, lower half )
: D*  a !  b !  c !  d !
      d @  b @  U*
      d @  a @  U*  DROP +
      c @  b @  U*  DROP +  ;  ok

6000. 12000. D* D. 72000000 ok
-5000004. 2. D* D. -10000008 ok
```

While this solution is not as fast as if it had been written using no variables but a great deal of stack manipulation instead, it has the advantage of being easy to write and understand! Notice also that because we are, in effect, truncating the result to give only the lower 32 bits the sign of the result is automatically correct.

It is worth noting, before leaving this topic, that without much extra effort we could develop a 32 by 32 bit multiply, giving a 64 bit result, and then use this operation in turn to develop even greater precision should we require it.

8.3 The Return stack for High Speed Definitions

In a footnote in chapter 5 I mentioned that the advanced FORTH programmer may

use the return stack inside colon definitions as an extra 'pair of hands'. FORTH provides three operations which allow access to the return stack, (pronounced "to-r", "r-from" and "r-fetch"):

`>R`	`(n → )`	Pop n off the normal stack and push onto the return stack.
`R>`	`( → n)`	Pop n off the return stack and push back onto the normal stack.
`R@`	`( → n)`	Copy n off the return stack and push onto the normal stack. (The same as {I} in most systems.)

These operations prove extremely useful, but they must be used with caution! In particular note that:

i. A colon definition must have *no* overall effect on the return stack.
ii. The return stack may be used inside DO loops, provided that the index and limit values held on the return stack are unaffected. (Unless that is the intention, as in {LEAVE}!)

As an example, suppose we need to define a word to add one to the fourth item down on the stack, without affecting the top three items. Using {>R} we can move the top three numbers over to the return stack temporarily to expose the fourth number for the addition. Then with the {R>} operation move the top three numbers back onto the normal stack:

```
: fourth+1   >R >R >R  1+  R> R> R> ; ok

10 20 30 40 fourth+1 . . . . 40 30 20 11 ok
```

The overall effect on the normal stack will be as follows:

```
fourth+1          (n1 n2 n3 n4 → n1+1 n2 n3 n4)
```

but the overall effect on the return stack is to leave it unaffected. We can see this by noticing that the definition of {fourth+1} contains the same number of {>R} words as {R>} words.

We could have avoided the use of the return stack altogether, in the example above, by using {ROLL}:

```
: fourth+1   4 ROLL  1+  4 ROLL 4 ROLL 4 ROLL ;
```

but this is not only longer but very much slower, since {ROLL} is quite a complex operation. If {Fourth+1} is to be executed often, then the faster solution using the return stack is obviously preferable.

Here are two more double number stack words which might be useful. Both utilise the return stack for temporary storage during execution:

```
: 2SWAP  >R ROT ROT R> ROT ROT ;   ( swap top two double numbers )
: 2OVER  2SWAP 2DUP >R >R 2SWAP R> R> ;  (duplicate second double
                                          number on top )
```

(A definition for {2DUP} was given in section 8.1.)

In addition a mixed precision divide is often useful:

```
: M/MOD  >R 0 R@  U/MOD  R> SWAP >R  U/MOD  R>  ;
```

Stack effect: (ud un → unrem udquot)

8.4 Formatted Number Output

In many real applications the type of numerical output produced by the printing operation {.} would not be adequate. For a professional looking computer output, numbers really need to be printed in meaningful formats. For example, dates as 20/01/82, or prices as $49.99. FORTH does provide a set of operations for building 'specialised' number print formats like these examples, in which the format may be specified in a neat and readable way.

Here are the formatting words summarised. (We do not need to know the detailed stack effects to use these words and so, for clarity, I shall postpone the stack descriptions of these words until the summary at the end of the chapter):

```
<#      Start a new formatted number string.
#       Insert the next digit of the number being printed
          into the formatted number string.
#S      Insert all remaining significant digits of the number
          into the formatted number string.
HOLD    Insert the character on the stack into the formatted
          number string.
SIGN    Insert a minus sign into the formatted number string
          if appropriate.
#>      Terminate the formatted number string ready for printing.
```

None of these words actually causes anything to be printed out, their effect is only to prepare a number, digit by digit, ready for printing. The standard string printing operation {TYPE}, which we came across in the last chapter, is used to type out the string after it has been built by a combination of the above operations.

The best way of seeing how number formatting works is with an example, so here is a definition for a 'price printing' operation:

```
DECIMAL ok
: .$    <#  # #  46 HOLD  #S  36 HOLD  #>  TYPE SPACE ; ok

1234. .$  $12.34 ok
```

The sequence of operations during execution of {.$}, for the double number 1234, break down as follows:

i. {<#} initialises a special character buffer (which is in fact the PAD downwards), ready to receive characters.

ii. The first {#} converts the *last digit* of the number (4), into ASCII in the current base (decimal), and inserts this into the formatted number string character buffer.

iii. The second {#} converts the next digit (3) and inserts this into the buffer.

iv. The phrase {46 HOLD} puts the value 46 into the character buffer. This is the ASCII code for a decimal point.

v. {#S} converts all remaining significant digits of the number (2 then 1), placing them into the character buffer.

vi. {36 HOLD} puts the ASCII value for $ into the buffer.

vii. {#>} terminates the completed string and leaves an address and character count on the stack ready for {TYPE}.

viii. Finally, {TYPE SPACE} prints out the finished text string in the correct order, starting with the last character inserted ($). Then a space is printed.

The important features to notice are that the formatted number string is built backwards, starting with the lowest digit, and that the formatting operations are designed to operate upon double precision numbers. This last feature is particularly useful since it means that we have up to ten decimal digits available for special number formats, whereas the five digits of single precision are often not enough.

An additional point is that the double precision number must be unsigned for conversion. If we wish to print negative, as well as positive numbers, then before the initial {<#} negative numbers must be converted to positive (with {DABS}), and the fact recorded ready for {SIGN}. The easiest way to do this is with the phrase {SWAP OVER DABS} before the {<#} word. We may, for example, redefine our {.$} format using this technique, to cover debits as well as credits!:

```
: .$    SWAP OVER DABS
        <#  # #  46 HOLD  #S  36 HOLD  SIGN  #>
        TYPE SPACE ; ok
-12345. .$ -$123.45 ok
```

The effect of {SWAP OVER DABS} on the (signed) double number on top of the stack will be as follows:

```
SWAP    (dlow dhigh → dhigh dlow)       Swap high and low halves of d.

OVER    (dhigh dlow → dhigh dlow dhigh)  Duplicate the high order
                                          part on top of the stack.
DABS    (dhigh d → dhigh ud)            Convert d to unsigned.
```

Later in the definition the word {SIGN} tests dhigh for negative (dhigh will have the same sign as the original signed double number d). {SIGN} then inserts a negative sign if dhigh is negative or does nothing if it is positive.

One final improvement to our {.$} format would be to print right justified, in a field width supplied on the stack. This could be particularly useful if our application required us to print columns of figures, where it would be important for pounds and pence to be vertically aligned:

```
: .$    >R                          ( save field width on rstack )
        SWAP OVER DABS              ( adjust for negative numbers )
        <#  # #  46 HOLD  #S  36 HOLD  #>  ( build format )
        R> OVER - SPACES            ( print leading spaces )
        TYPE SPACE ; ok             ( and type number )

-0.01 CR 10 .$          ( print in field width of 10 )
   -$0.01 ok

123.45 CR 10 .$
  $123.45 ok
```

Here we are simply using the character count supplied by {#>} to calculate the number of leading spaces to print, which we do before the final {TYPE}.

To close this section, here is a selection of format definitions which might be useful, including the date printing format mentioned at the start of the section.

```
: .DATE   <#  # #  47 HOLD  # #  47 HOLD  # #  47 HOLD  #>  TYPE ;
```

```
( Print double number d with one trailing space )
: D.      SWAP OVER DABS          ( d → )
          <# #S SIGN #>
          TYPE SPACE ;

( Print double number d right justified in field width n )
: D.R     >R                      ( d n → )
          SWAP OVER DABS
          <# #S SIGN #>
          R> OVER - SPACES
          TYPE ;

( Print single number n1 right justified in field width n2 )
: .R      >R  S->D  R>  D.R  ;  ( n1 n2 → )
```

The word {S->D} used in the final definition here has the effect of converting a single length number to double length. If your FORTH system doesn't already have this word it can easily be defined as follows:

```
: S->D    DUP 0< IF  -1  ELSE  0  THEN  ;
```

8.5 Fixed Point Arithmetic

We have now covered all of the techniques necessary to be able to develop a useful fixed point arithmetic 'package'. First, we must recap on what exactly we mean by a 'fixed-point' number; it is one in which the decimal point has a fixed position within the number, even while it is being used for calculations in the computer. Bearing this in mind, it seems likely that our standard double precision (integer) arithmetic operations will work equally well for fixed point numbers. The thing that makes them fixed point numbers is simply the way we input them and print them out.

Suppose that we decide to 'fix' the decimal point four digits to the left. Our number range is then:

```
+/-0.0001 to +/-99999.9999
```

These numbers are perfectly acceptable input to FORTH. The number 0.0001 will be represented internally as just 1, and the number 99999.9999 will be represented as 999999999. We could then type:

```
99999.9999 -0.0001 D+ D. 999999998 ok
```

to give a result representing the fixed point number 99999.9998.

Using the number formatting techniques covered in the last section we can easily define a special fixed-point print operation, to use instead of {D.}, as follows:

```
: F.      SWAP OVER DABS
          <#  # # # #  46 HOLD  #S  SIGN  #>
          TYPE SPACE ; ok

0.0005 0.0100 D+ F. 0.0105 ok
100.0000 0.0001 D- F. 99.9999 ok
```

and we now appear to have our fixed-point addition, and subtraction, but with one fatal flaw; we must always type in all four digits after the decimal point. If we do not then some wrong answers are likely to occur, for example:

```
100.1 0.25 D+ F. 0.1026 ok
```

which is nonsense!

What is required here is an operation to 'fix' the input numbers so that whatever is typed in will be represented internally (on the stack) as we really intended. The number 100.8, for example, needs to be 'fixed' into 1008000 to be consistent with the chosen fixed point notation. The scaling factor clearly depends on how many digits were typed after the decimal point in the input number; information available from the variable {DPL} (see section 8.1). {DPL} can be used to calculate how many times the input number must be multiplied by 10 in order to fix it. Here is a definition that will do the job:

```
: FIX   DPL @ 0< IF                    ( if number was single )
                  S->D  0 DPL !        ( convert to double )
                THEN
        DPL @ DUP 4 < IF
                    4 SWAP DO
                              10. D*   ( perform scaling )
                            LOOP
                  ELSE
                  4 > IF
                        ." Out of range" 2DROP
                      THEN
                  THEN ;
```

This definition has a number of additional refinements. One is that it will 'fix' single precision numbers (which have the effect of setting DPL to -1). The first IF clause converts these into double numbers with no digits after the decimal point (DPL=0), so that they will be correctly scaled by the following program. Another refinement is that numbers with more that four places of decimals will produce the error message "Out of range", but not corrupt any other numbers on the stack.

We can now perform some useful calculations, for example:

```
0.04 FIX ok
0.1 FIX D+ ok
0.567 FIX D+ ok
0.0001 FIX D+ ok
10 FIX D+ ok
F. 10.7071 ok
```

We could now, if necessary, extend this package to include other operations. Multiplication, for example, is performed perfectly correctly by our {D*} operation of section 8.2, except that it yields a result 10000 times too large (because of the position of the decimal point). To get round this problem simply requires a routine to divide by 10000:

```
: /10000  DUP >R          ( sign to return stack )
          DABS            ( make unsigned )
          10000 M/MOD     ( mixed precision divide )
          R>              ( adjust sign of quotient )
          0< IF DNEGATE THEN
          ROT DROP ;      ( and lose remainder )
```

coupled with {D*}, to give a fixed-point multiply:

```
: F*      D* /10000 ;

0.456 FIX  20 FIX  F*  F.  9.1200 ok
-0.05 FIX 0.6 FIX  F*  F.  -0.0300 ok
```

Of course, {F*} will not work correctly for very large numbers, where the result of {D*} exceeds the double number range. To overcome this would require a double "times-divide" operator {D*/}, with a 64 bit intermediate result.

To conclude this section it is worth noting that because our fixed-point arithmetic involves only whole-number calculations internally, it is very fast, certainly much faster than equivalent operations using floating-point arithmetic.

8.6 Summary

The following FORTH-79 Standard words have been covered in this chapter:

Stack Manipulation:

`>R` `(n → )` `"to-r"`

Move n onto the return stack for temporary storage. Every >R must have a corresponding R> in the same control structure nesting level of a colon definition.

`R>` `( → n)` `"r-from"`

Move n from the return stack to the data (normal) stack.

`R@` `( → n)` `"r-fetch"`

Copy the number on top of the return stack onto the data stack.

Comparison:

`D<` `(d1 d2 → flag)` `"d-less-than"`

True if d1 is less than d2.

Arithmetic:

`D+` `(d1 d2 → dsum)` `"d-plus"`

Add double precision numbers.

`DNEGATE` `(d → -d)` `"d-negate"`

Two's complement double number (reverse its sign).

`*/` `(n1 n2 n3 → nquot)` `"times-divide"`

Multiply n1 by n2, then divide by n3, leaving the quotient. The product of n1 and n2 is calculated as a 32 bit double precision number.

`*/MOD` `(n1 n2 n3 → nrem nquot)` `"times-divide-mod"`

As */ but leave the remainder as well. The remainder has the same sign as n1.

`U*` `(un1 un2 → udprod)` `"u-times"`

Multiply unsigned single numbers to give an unsigned double precision product.

`U/MOD` `(ud un → urem uquot)` `"u-divide"`

Divide double number by single, giving remainder and quotient. All

unsigned.

Formatted Number Output:

`<#` `( → )` `"less-sharp"`

Initialise a formatted number conversion.

`#` `(ud1 → ud2)` `"sharp"`

Generate from the unsigned double number ud1, the next ASCII character and add it to the formatted number string. ud2 is the quotient after dividing ud1 by BASE, ready for the next digit to be generated. Use between <# and #>.

`#S` `(ud → 0 0)` `"sharp-s"`

Convert all remaining significant digits of ud adding each to the formatted number string. Leave a double zero. If ud was initially zero add a single zero to the output string. Use between <# and #>.

`HOLD` `(char → )`

Insert char into the formatted number string. Use between <# and #>.

`SIGN` `(n ud → ud)`

Insert an ASCII minus sign "-" into the formatted number string if n is negative. Use between <# and #>.

`#>` `(ud → addr n)` `"sharp-greater"`

End formatted number conversion. Drop ud and leave the address and character count of the formatted string ready for TYPE.

Miscellaneous

`79-STANDARD` `( → )`

Verify that system conforms to the FORTH-79 standard.

9
Extending FORTH

One of the most remarkable features of FORTH is the ability to define new 'defining words'. Recall that the effect of a defining word (such as {VARIABLE} or {:}), is to create a new dictionary entry. Defining a new defining word means specifying a new type of dictionary entry and its action when executed. In real terms this gives us the ability to define completely new data structures such as 'string' variables, multi-dimensioned arrays, or even data structures consisting of mixtures of different data types. In addition, we may define new 'compiling words' which may be, for example, new control structures.

9.1 Defining new Defining words

We have already seen how to set up arrays using the words {CREATE} and {ALLOT}, followed by a special colon definition to calculate the address of the required array element. To recap the technique, here it is again for a 10 element array named X:

```
CREATE x 20 ALLOT       ( define the array )
: X  x SWAP 2 * + ;     ( calculate address )
```

While this technique is perfectly satisfactory, it does have the drawback that for each new array needed, these two lines of FORTH (or something very similar) have to repeated over again.

We could really do with an entirely new defining word {ARRAY}, with the same overall effect as the two lines of FORTH above, but repeatable for different array names. We can indeed create such a defining word, with a special colon definition, and the word {DOES>}, as follows:[1]

```
: ARRAY
        CREATE  2 * ALLOT      ( create new dictionary entry )
        DOES>   SWAP 2 * + ;   ( runtime action )
```

To define a ten element array named X, we then simply type:

```
10 ARRAY X
```

and for another twenty element array Y, type:

```
20 ARRAY Y
```

To reference an element in the array, precede its name by the number of the array element (counting from zero). For example:

```
4 X ?     ( print 5th entry in X )
15 Y ?    ( print 16th entry in Y )
```

The operation of {ARRAY} may not seem too obvious, so let us examine it step by step. When {ARRAY} is executed, the first thing to happen is that {CREATE} generates a new dictionary entry, whose name will be the next word in the input stream, (immediately after the word {ARRAY}); "X" and "Y" in the example above. The

[1]Note that FORTH-78 and earlier standards use the word {<BUILDS} instead of {CREATE} in defining word definitions. If your system is one of these, then simply substitute {<BUILDS} for {CREATE} in the examples in this chapter, and they should work correctly.

number on top of the stack, (immediately preceding the word {ARRAY}), is then multiplied by 2 ready for {ALLOT}, which will reserve the required amount of space for the new array. The next word in the colon definition is {DOES>}; a special word only ever used when defining new defining words. {DOES>} has the crucial effect of specifying what new words defined by the new defining word will do when executed.

If we had not included the phrase {DOES> SWAP 2 * +} in the definition of {ARRAY} above, then any new words defined by {ARRAY} would have the same effect as words defined by {CREATE}, that of pushing the address of the first entry in the space allotted. But because we have included the DOES> phrase, the words following {DOES>} will be executed with this address on the stack. Thus typing:

```
4 X
```

will cause the words {SWAP 2 * +} to be executed with the address of the first element of X on top of the stack, and 4 second on the stack. The result of this will be to leave the address of the 5th element (numbered from zero) on top of the stack; exactly the same effect as the special colon definition in the example at the beginning of this section.

The important point to notice here, is that the words after {DOES>} are not executed when {ARRAY} is executed, but when words defined by {ARRAY} are themselves executed.

The overall structure of a colon definition to define new defining words is summarised here:

```
: new_defining_word
        CREATE  ( compile time words )
        DOES>   ( run time words ) ;
```

When the new defining word is used to compile a new dictionary entry, the 'compile time' words are executed; when this new dictionary entry is itself executed the 'run time' words are executed.

To conclude this section, here is an interesting selection of 'standard' defining word definitions:

```
: VARIABLE  CREATE 2 ALLOT ;        ( single length variable )
: CONSTANT  CREATE ,  DOES> @ ;     ( single length constant )
```

Most systems actually define {VARIABLE} and {CONSTANT} as machine code primitives but we could, for example, redefine {VARIABLE} so that an initial value is supplied when the variable is defined (as in earlier FORTH standards):

```
: VARIABLE  CREATE , ;
```

Providing that your system has the word {C,} which is similar to {,} except that it compiles only the lower 8 bits (byte) of the number on the stack into the dictionary entry, then you can define byte length variables and constants:

```
: CVARIABLE CREATE 1 ALLOT ;        ( byte variable )
: CCONSTANT CREATE C,  DOES> C@ ;   ( byte constant )
```

Finally, double precision variables and constants:

```
: 2VARIABLE CREATE 4 ALLOT ;        ( double length variable )
: 2CONSTANT CREATE , ,              ( double length constant )
```

```
        DOES>  DUP 2+ @ SWAP @ ;
```

An associated pair of double number store and fetch operations can easily be defined:

```
: D!  DUP >R !  R> 2+  !  ;          ( d addr → )
: D@  DUP @ >R  2+  @ R>  ;          ( addr → d )
```

9.2 The last word on ARRAYs

There are a number of useful enhancements that can be incorporated very easily into the new defining word {ARRAY} of the last section. One is to check for 'index out of range'; another is to number the array elements from 1 rather than from 0. A new definition for {ARRAY} with these enhancements is as follows:

```
0 : ARRAY
1         CREATE  DUP ,                 ( store array size )
2                 2 * ALLOT             ( and reserve space )
3         DOES>
4                 SWAP 1- SWAP          ( make index 0 upwards )
5                 OVER OVER             ( duplicate index and addr )
6                 @ U< NOT IF           ( test for out of range )
7                           ." Array range error"
8                           QUIT
9                         THEN
10                2 +                   ( skip array size )
11                SWAP 2 * + ;          ( calculate address )
```

This new version of {ARRAY} is used just like the old one, for example:

```
20 ARRAY table ok        ( define a twenty element array )

0 table Array range error        ( index out of bounds! )
21 table Array range error

-1 5 table ! ok          ( set the fifth element to -1 )
```

The compile time action has been modified slightly, to save the array size in the dictionary entry {DUP ,} so that this can be used at runtime to check for index out of bounds.

The runtime action splits into three distinct sections. The first line (after {DOES>}), simply reduces the index, which is second on the stack, by one {SWAP 1- SWAP}. This means that if we specify element 1 of the array we actually get the 0th element.

Lines 5 to 9 inclusive perform the range checking. An interesting feature is the use of the unsigned comparison word {U<}. This will ensure that negative index values will also fail the test, since if negative numbers are treated as unsigned they appear as large positive numbers. The use of {U<} thus avoids two separate signed comparison operations.

Line 10 adjusts the address on top of the stack to skip over the stored array size, and line 11 calculates the address of the required element, as in the earlier definition of {ARRAY}.

The runtime code (lines 4-11) will, of course, execute every time an array defined by {ARRAY} is referenced, with a speed penalty because of the range checking. Since the range checking is only required while a FORTH program is under development, it

is common practice to remove it after the application is fully debugged. The reLOADed program will then run much faster. For example, the {ARRAY} definition above would reduce to:

```
: ARRAY CREATE
                2 * ALLOT
        DOES>
                SWAP 1- 2 * + ;
```

To conclude this section here is a definition for {2DARRAY}, a new defining word to generate 2 dimensional arrays:

```
: 2DARRAY       CREATE
                        DUP ,           ( save second index )
                        * 2 * ALLOT     ( reserve array space )
                DOES>
                        ROT             ( fetch i1 to top of stack )
                        OVER @ *        ( multiply by stored index )
                        ROT +           ( add i2 )
                        2 * + 2 + ;     ( calculate address of i1,i2 )
```

i1 and i2 refer to the two indices needed to pick out an element of the array (element i1,i2), as illustrated here:

```
4 4 2DARRAY square      ( define a 4 by 4 array 'square' )
-1  0 0 square !        ( set element 0,0 equal to -1 )
```

This array contains 16 elements, numbered 0,0 through to 3,3. A good way to access each element in turn is with nested DO loops, as follows:

```
: printsquare                           ( print whole of square )
        4 0 DO                          ( step from 0 to 3 )
              4 0 DO                    ( ditto )
                    J I square .        ( print element I,J )
                 LOOP CR
            LOOP ;
```

9.3 A STRING variable

An obvious candidate for a new defining word is a STRING variable. This would overcome the limitation of the string handling described in chapter 7, of having to use a fixed buffer area for all string operations, and would also allow the development of powerful string matching and comparison operations similar to those in BASIC.

In order to devise a {STRING} defining word, we must first consider what string variables will contain. Character strings of course! But in addition it would be helpful if a string variable contained its maximum length, and its actual length. Figure 9.1 illustrates how a six character string variable would store the four character string "fred".

The maximum length byte will be set up when the string variable is defined and will be used to check for 'string overflow'. The actual length byte will facilitate string manipulation, for example, the string printing word {TYPE} can use this information.

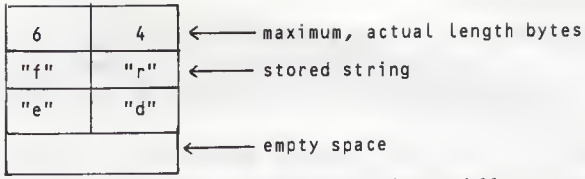


Figure 9.1 The proposed string variable structure

Here is a definition for {STRING} along these lines:

```
: STRING
        CREATE
                DUP C,          ( store maximum length )
                0 C,            ( set actual length to zero )
                ALLOT           ( and reserve string space )
        DOES>
                2+              ( start address of string )
                DUP 1- C@ ;     ( actual length byte )
```

To define a string variable A$, with space for a 20 character string, write:

```
20 STRING A$
```

The string variable A$ will leave two values on the stack, the start address of the string second, and the character count on top. This is exactly what is required by the standard string printing word {TYPE}, and so to print A$, we write:

```
A$ TYPE
```

None of this is particularly helpful until we can input strings into string variables, and for this we really need two operations; {INPUT$} to input strings from the keyboard for interactive question and answer type of programs, and {PUT$} to set up strings inside programs. These two definitions turn out to be almost identical and so I omit detailed comments from the second:

```
: INPUT$
        DROP 1-                 ( address of count byte )
        DUP 1- C@               ( maximum length byte )
        CR ." ? " QUERY         ( get string from keyboard )
        1 WORD                  ( to HERE )
        HERE C@                 ( actual length of string )
        < IF                    ( maximum less than actual? )
             ." String too big"     ( error message )
             DROP QUIT              ( clear stack and exit )
          THEN
        HERE DUP C@ 1+          ( address and bytes to move )
        ROT SWAP CMOVE  ;       ( into the string variable )

: PUT$
        DROP 1-
        DUP 1- C@
        36 WORD                 ( get string following PUT$ )
        HERE C@
        < IF
             ." String too big"
             DROP QUIT
          THEN
        HERE DUP C@ 1+
```

```
        ROT SWAP CMOVE ;
```

The delimiter character supplied to {WORD} in {PUT$} is the ASCII value for the character "$". Strings input using {PUT$} must therefore be terminated by "$", and can include embedded spaces. Here are some examples of the use of {INPUT$} and {PUT$}:

```
20 STRING A$ ok                    ( define 20 character string A$ )

A$ INPUT$                          ( input from keyboard )
? test string ok                   ( terminated by 'return' )

A$ TYPE test stringok              ( print A$ )

A$ INPUT$
? this string is too long String too big

A$ PUT$ another test$ ok           ( input from input stream )

A$ TYPE another testok             ( print A$ )
```

An operation to compare two strings is a useful addition to our string handling vocabulary, and might be defined as follows:

```
: -MATCH  OVER OVER                ( duplicate length and addr2 )
          + SWAP DO                ( loop thru chars )
                    DROP 1+ DUP 1- C@  ( get char from str1 )
                    I C@ - DUP IF      ( not equal? )
                                  DUP ABS / LEAVE  ( flag )
                               THEN
                 LOOP SWAP DROP ;
```

This operation "not-match" will compare two equal length strings, and has the stack effect:

```
-MATCH          (addr1 n addr2 → flag)
```

The two strings, both of length n, starting at addr1 and addr2 are compared, leaving a flag value which is 'false' if the strings match, 'true' and positive if string1>string2, or 'true' and negative if string1<string2. This may be included in a {$=} definition as follows:

```
: $=  ROT OVER = IF                ( string same length? )
                 SWAP -MATCH NOT   ( attempt match then )
              ELSE
                 DROP DROP DROP 0  ( else false )
              THEN ;
```

An interesting application of {$=} which, incidentally, gives another example of GOTOless programming in FORTH, is in a colon definition to ask the user if he wishes to continue or not. The equivalent in BASIC is a familiar construction in games programs etc.:

```
10 PRINT "Do you want to continue (yes/no)";
20 INPUT A$
30 IF A$="no" THEN END
40 IF A$="yes" THEN 60
50 GOTO 10
60 ....
```

The same thing in FORTH is:

```
10 STRING ANSWER$
3 STRING YES$    YES$ PUT$ yes$
2 STRING NO$     NO$ PUT$ no$
: CONTINUE?
        BEGIN
                ." Do you want to continue (yes/no) "
                ANSWER$ INPUT$          ( get reply )
                ANSWER$ NO$ $= IF QUIT THEN
                ANSWER$ YES$ $=
        UNTIL ;         ( loop until the answer is yes )
```

Inserting the word {CONTINUE?} will cause a program to halt, and ask the question "Do you want to continue (yes/no)?". If the answer typed into the keyboard is "no", then the program will quit. If the answer is "yes", then the program continues. If the answer was neither "yes" or "no", then the question will be repeated.

9.4 Self Modifying Data structures

A remarkable consequence of FORTH's ability to define new defining words is that we may build 'intelligent' data structures; for example, arrays that automatically maintain averages, or lists that re-order themselves whenever any entry is altered.

To take the first of these examples, suppose we have a 10 element array 'readings', defined using the word {ARRAY} of section 9.2. To compute the arithmetic average of the contents of this array requires adding together all 10 entries and dividing by 10. A special definition could easily be written to do this as follows:

```
: average               ( take average of array 'readings' )
        0               ( result=0 )
        11 1 DO         ( step 1 to 10 )
                I readings @ +  ( add up each element )
            LOOP
        10 / ;          ( and divide by 10 )
```

If our FORTH application needed us to calculate an average like this often and for many different arrays then, to simplify the overall program, we should define a new defining word {*ARRAY} with the averaging function built into the DOES> part of the definition:

```
: *ARRAY                ( 'special' array with running average )
        CREATE
                DUP ,           ( save array size )
                0 ,             ( set 'average' to zero )
                0 DO            ( step thru elements )
                     0 ,                ( defining and zeroing )
                  LOOP
        DOES>
                DUP DUP @       ( get array size )
                SWAP 4 +        ( point to start of array )
                OVER 0 SWAP
                     0 DO       ( step thru array )
                          OVER @ +      ( add up )
                          SWAP 2+ SWAP  ( bump up pointer )
                       LOOP
                SWAP DROP SWAP /        ( divide by array size )
                OVER 2+ !       ( store average in element 0 )
                2+ SWAP 2 * + ; ( calculate address )
```

Arrays defined by {*ARRAY} may be used just like those defined by {ARRAY}, for example:

```
10 *ARRAY readings ok              ( one set of readings )

10  1 readings ! ok                ( readings(1)=10 )
20  2 readings ! ok                ( readings(2)=20 )
1000 10 readings ! ok              ( readings(10)=1000 )

2 readings ? 20 ok                 ( print contents of readings(2) )
```

Which is exactly how we would expect a 10 element array, with entries numbered from 1 to 10 to behave. But typing:

```
0 readings ? 103 ok
```

will print the average of the values currently contained in the array ((10+20+1000)/10 = 103). This average will be calculated afresh every time the name of the array {readings} is quoted and will always be true however many times we might have altered the values stored in the array.
For example:

```
870 10 readings ! ok               ( alter readings(10) to 870 )
50 6 readings ! ok                 ( set readings(6) to 50 )

0 readings ? 95 ok                 ( new average is 95 )
```

and, of course, *all* arrays defined by {*ARRAY} will have this function built in!

9.5 A Closer Look at the Dictionary

All dictionary entries share the same basic internal structure. In FORTH terminology each part of a dictionary entry is known as a 'field' and every dictionary entry has four distinct fields; the name field, link field, code field and parameter field. Figure 9.2 shows the field structure of a dictionary entry named "EXAMPLE" – it doesn't matter whether {EXAMPLE} is a variable, constant or colon definition – the structure is the same in each case.

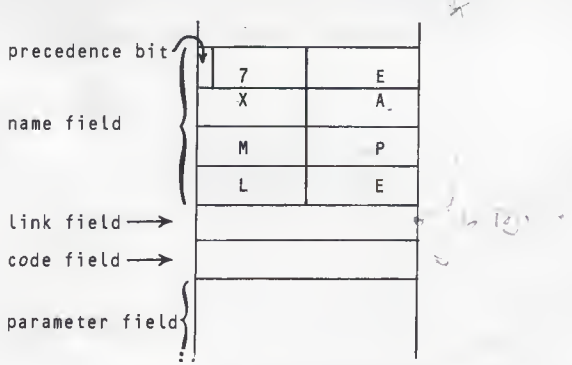


Figure 9.2 The Field structure of a dictionary entry

The name field contains the character count of the original <name> of the definition, followed by the characters of the name stored as ASCII bytes. The length of the name field thus depends on the length of the <name> and FORTH-79

specifies a 32 byte maximum name field, so that only the first 31 characters of a longer <name> will be stored.[2]
The character count takes up only the lower 7 bits of the first byte in the name field, which implies an absolute maximum of 127 characters in the original defined <name>. The top bit is the 'precedence bit' of which I will say more later in this section.

The link field is a fixed length 16 bit cell containing the address of the name field of the previous dictionary entry. FORTH uses this to quickly search backwards through the dictionary when looking for a word. (The FORTH dictionary is what computer scientists would call a 'linked list').

The code field is another fixed length 16 bit cell, and contains the 'code pointer'. This is the address of the 'run-time' code which is executed when the dictionary entry is executed. It is the code pointer that distinguishes between variable, constant or colon definition since the 'action' of the run-time code is different in each case.

The following table summarises the action of the run-time code for each of the four basic types of dictionary entry:

Defining word	*Action of run-time code*
VARIABLE or CREATE	Push the start address of the parameter field onto the stack. (Parameter field address.)
CONSTANT	Fetch the contents of the first cell in the parameter field, and push onto the stack.
:	Execute the words of the colon definition by the addresses stored in the parameter field.

The run-time code described above is normally defined in machine-code for speed, but we can devise our own run-time code in words defined by {CREATE} using {DOES>}, as shown earlier this chapter.

The final field in the dictionary entry, the parameter field, can contain almost anything. In the case of:

```
CREATE null ok
```

the parameter field is empty and of zero length (but the parameter field address will still be returned by {null}). In dictionary entries generated by {VARIABLE} or {CONSTANT} the parameter field consists of one 16 bit cell, containing the value of the variable, or constant. The parameter field may be extended in any of the above three cases by using {ALLOT} {,} or {C,}.

In a colon definition, the parameter field contains a list of code field addresses, one for each word in the body of the colon definition. To illustrate this figure 9.3 details the dictionary entries generated by the following:

[2]Many FORTH systems limit the name field to 4 bytes, which means that although names may be longer, the first three characters of equal length names must be unique to avoid ambiguity. Check your system documentation for more details on this.

```
VARIABLE X
: XSQ   X @ DUP * X ! ;   ( Square X )
```

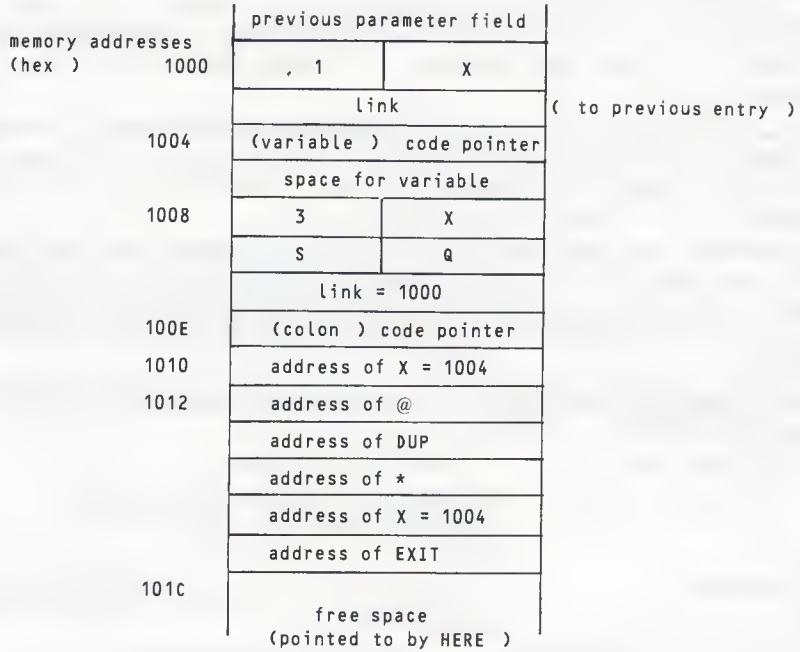


Figure 9.3 Two Dictionary entries illustrated

The action of the colon run-time code (which is often referred to as the 'address interpreter'), is to fetch and jump to each address in the parameter field in turn. As an example, when FORTH executes {XSQ} it jumps to the address interpreter pointed to by the code field (address 100E). The address interpreter then:

i. Fetches the first address in the parameter field, which is the code field address of {X}.

ii. Pushes the address of the next cell in the parameter field (1012) onto the return stack.

iii. Jumps to {X}.

When the run-time code for the variable {X} has finished, the address interpreter pops the value 1012 off the return stack, to fetch the next code field address, for the word {@}, and the above three steps are repeated.

The final entry in the parameter field of {XSQ} was generated by the terminating semi-colon and is the word {EXIT}. {EXIT} has the effect when executed of returning us to the next higher level of execution (by popping an address off the top of the return stack and passing it to the address interpreter). This means that if we have included {XSQ} in another colon definition, for example:

```
: TEST   XSQ X @ ;
```

when {TEST} executes, {XSQ} will be executed with the address of the next word in {TEST}, which is {X}, on the return stack. When the final {EXIT} of {XSQ} is executed, the address interpreter will return correctly to the next word in {TEST}. It is this use of the return stack that enables us to 'nest' colon definitions in this way.

We can use the word {EXIT} inside colon definitions (but not DO loops!) in order to prematurely return to the next higher level of definition, but this breaks the structured nature of FORTH and its use is not recommended.

A question you may well be asking now is "What happens if I have typed {XSQ} directly so that as soon as it has finished, control is returned to the keyboard and 'ok' printed – how does {EXIT} achieve this?". The answer may come as a surprise, but a FORTH system is always executing a colon definition even while waiting for typed input! {EXIT} returns us back into this outer program which looks, in outline, like this:

```
: QUIT  BEGIN
                ( Clear the return stack )
                ( Fetch a line of input using QUERY )
                ( Execute the input using EXECUTE )
                ." ok" CR
        0 UNTIL ;          ( Loop forever )
```

(This is virtually the same {QUIT} that we met in chapter 5.8!)

9.6 Defining new Compiling words

Armed with an understanding of the FORTH dictionary structure we can now go on to define new 'compiling words', but first let us review some compiler and dictionary handling words.

The two words {FIND} and {'} (pronounced "tick") both return information about a particular dictionary entry. FIND <name> will return, on the stack, the code field address of the dictionary entry for <name> (or zero if the word is not found in the dictionary). Thus typing:

```
FIND TEST . 12345 ok ( print code field address of TEST )
```

will tell us if {TEST} is in the dictionary. The word {EXECUTE} will execute the word whose code field address is on the stack, so that:

```
FIND TEST EXECUTE
```

is exactly the same as typing simply:

```
TEST
```

A rather more useful application of {EXECUTE} is to execute 'indirectly' that is, using a code field address stored in a table. We shall see an example later in this section.

{' <name>} will return the parameter field address of the dictionary entry for <name> if it exists. Thus:

```
10 CONSTANT A
1  ' A  !
```

enables us to change the value of a constant.

If {'} is used inside a colon definition it has the unusual effect of using the very next word in the definition as the <name>, not the next word in the input stream. For

example:

```
: TESTPFA  ' TEST ; ok
TESTPFA . 12347 ok      ( parameter field address of TEST )
```

The 'precedence bit', mentioned in figure 9.2 of the last section, has the effect of determining whether a word is compiled or executed when it occurs within a colon definition. Words that have the precedence bit set are called 'immediate' words, and are executed while the colon definition containing them is being compiled. The word {IMMEDIATE} sets the precedence bit of the most recently defined word. For example in:

```
: PRINTNOW  CR ." Compiling.." CR ; IMMEDIATE ok
```

the word {PRINTNOW} will always be executed, and will not generate any compiled code:

```
: TEST  PRINTNOW CR ." Running.." CR ;
Compiling..
ok
TEST
Running..
ok
```

One of the effects of the defining word {:} is to switch FORTH into 'compile' mode, and the terminating {;} switches FORTH back into 'execute' mode. The system variable {STATE} tells us which of these two modes FORTH is in at any particular time, for example:

```
: MODE?  STATE @ IF ." Compiling " ELSE ." Executing " THEN ; ok
IMMEDIATE ok

MODE? Executing ok              ( Execution mode )
: TEST 4 + MODE? ; Compiling ok ( Compile mode )
```

The word {MODE?}, being immediate, will not generate any compiled code.

The two words {[} "left-bracket" and {]} "right-bracket" have the effect of switching FORTH into 'execute' or into 'compile' mode respectively. Thus in:

```
: TEST ." Print later "  [ ." Print now " ] ; Print now ok
TEST Print later ok
```

The FORTH enclosed by the square brackets is executed during the compilation of {TEST}, and generates no compiled code. A rather more useful application is to do 'compile time' arithmetic. For example in:

```
1024 CONSTANT 1K ok
: ADD_5K  [ 1K 5 * ] LITERAL + ; ok
```

The definition of {ADD_5K} is identical to:

```
: ADD_5K  5120 + ; ok
```

but much more readable. The word {LITERAL} has the effect of compiling the number on top of the stack (in FORTH terminology compiled numbers are called 'literals'). Thus:

```
: TEN  [ 10 ] LITERAL ; ok
```

and:

```
: TEN  10 ; ok
```

are identical, and produce the same compiled code.[3]
An interesting application of right-bracket is to generate a table of code field addresses. The word {EXECUTE} may then be used to execute one of the words pointed to by the table, as follows:

```
: ZERO  ." zero " ; ok                    ( Some example words )
: ONE   ." one  " ; ok
: TWO   ." two  " ; ok

CREATE VECTORS ] ZERO ONE TWO [ ok        ( create vector table )
: GOVECTOR  2 * VECTORS + @ EXECUTE ; ok

0 GOVECTOR zero ok
2 GOVECTOR two  ok
```

The word {COMPILE} may be used to define new words which have both a run-time and a compile-time action. The structure of the definition of such a 'compiling word' is as follows:

```
: run_time_action   ....    ;

: compiling_word    COMPILE  run_time_action
                    .. compile time words .. ; IMMEDIATE
```

Since the new compiling word is an immediate it will be executed when it occurs within a colon definition. The word {COMPILE} will compile the code field address for {run_time_action}, and the compile time words will be executed right away.

The word {LITERAL} is an example of a compiling word, and could be defined as follows:

```
: (LITERAL)     R>        ( address of number )
                DUP       ( duplicate it )
                2+ >R     ( point to next cell in code field )
                @ ;       ( get number to stack )

: LITERAL       COMPILE (LITERAL)
                ,         ( store number in dictionary )
                ; IMMEDIATE
```

At compile-time the number on top of the stack is stored in the next cell in the dictionary by {,}. When the run-time code {(LITERAL)} is executed, at run-time, the value on top of the return stack will point to the next cell in the code field, which contains the number. {(LITERAL)} fetches the number onto the stack, and adds two to the address on top of the return stack so that the address interpreter will skip over the cell containing the number.

The looping and conditional structures are also examples of compiling words which modify the address on top of the return stack, at run-time, on order to force the address interpreter to continue from elsewhere in the program. To illustrate this, here is a definition for a new looping structure STEP .. DOWN:
At compile-time {STEP} pushes the current dictionary address supplied by {HERE} onto the stack, and the corresponding {DOWN} compiles this address into the code field using {,}. The run-time code {(DOWN)} places this branch address on the return

[3]Literals actually generate two entries in the parameter field. The first points to the run-time code for literals, and the second cell contains the number. The run-time code will have the effect, when executed, of fetching the number and pushing it onto the stack.

```
: STEP  COMPILE >R              ( loop counter to return stack )
        HERE ;  IMMEDIATE       ( place HERE on return stack )

: (DOWN)  R>                    ( fetch return address )
          R>                    ( and loop counter )
          1- DUP IF             ( decrement loop counter )
                    >R             ( save new value )
                    @ >R           ( and branch address )
                  ELSE          ( end of loop )
                    DROP           ( drop loop counter )
                    2+ >R          ( and skip over branch address )
                  THEN ;

: DOWN  COMPILE (DOWN)          ( compile run-time code )
        , ; IMMEDIATE           ( save HERE from STEP in dictionary )
```

stack {@ >R} when the loop should be repeated, or skips over it {2+ >R} at the end of the loop. The loop counter is held on the return stack and copied off the normal stack by {STEP} at run-time using {>R}. For example:

```
: TEST 10 STEP  I .  DOWN ; ok
TEST 10 9 8 7 6 5 4 3 2 1 ok
```

Finally, the word {[COMPILE]} should be mentioned. This is used to override the precedence bit in immediate words, so that they may be included in colon definitions and executed at run-time, not compile-time. As as example, suppose we need to "tick" the next word in the input stream. Using [COMPILE] we could write:

```
: .PFA  [COMPILE] ' . ; ok ( print parameter field addr )
.PFA TEST 12347 ok
```

9.7 Summary

The following new words have been introduced in this chapter:

Defining words:

`DOES>` (→) "does"

Defines the start of the run-time action of a new defining word. Used in the form:

```
:defining-word ... CREATE ... DOES> ... ;
```

And then:

```
defining-word <name>
```

When <name> is executed the words between DOES> and ; are executed with the parameter field address of <name> on the stack.

Dictionary words:

(→ addr) "tick"

When used in the form ' <name> leaves the parameter field address of the dictionary entry for <name> on the stack. If used in a colon definition the address is compiled into the dictionary as a literal. Error if <name> is not found in the dictionary.

`FIND` (→ addr)

Return the code field address of the next word in the input stream, or zero if the word is not found in the dictionary.

Compiler words:

`IMMEDIATE` `( → )`

Mark the most recently defined dictionary entry as a word that will be executed even when it occurs within a colon definition.

`LITERAL` `(n → )`

If compiling compile n into the dictionary as a 16 bit literal, which will leave n on the stack when later executed.

`STATE` `( → addr)`

System variable indicating the current state of the system. A non-zero value indicates compilation is occurring.

`[` `( → )` `"left-bracket"`

End compilation so that subsequent text is executed.

`]` `( → )` `"right-bracket"`

Set compilation mode so that subsequent text is compiled.

`COMPILE` `( → )`

When a word containing COMPILE executes the code field address following COMPILE is copied (compiled) into the dictionary.

`[COMPILE]` `( → )` `"bracket-compile"`

When used in the form [COMPILE] <name> the word <name> is compiled even if it is an immediate.

`EXECUTE` `(n → )`

Execute the word whose code field address is on the stack.

`EXIT` `( → )`

When included within a colon-definition EXIT has the same effect as `;`. May not be used within a DO loop.

10
FORTH Finale

In this concluding chapter I will put some of the techniques described so far into practice in two complete FORTH programs. I shall outline each program from initial conception, through development and debugging, to finished 'vocabulary'. The first is the calendar vocabulary mentioned in the introduction, and the second is an interactive 'video game'. These are chosen both because they are interesting programs in their own right, and because they each illustrate a particular type of programming problem; the calendar program is largely mathematical, and the video game relies heavily on high speed graphics. Neither program requires disk handling and will run equally well on a cassette based system.

10.1 A Calendar Vocabulary

A useful set of 'calendar' words were first proposed in the introduction to this book. They are:

```
day       (day month year → )     Print the day of the week
                                  that the specified date falls
                                  upon.
month     (month year → )         Print a calendar for the
                                  month specified.
year      (year → )               Print a whole year calendar.
daysleft  (day month year → )     Print the number of days
                                  remaining until the current
                                  year end.
```

Having specified the end result we must now develop a strategy for arriving at this result, in other words, a logical set of sub-definitions which will ultimately build the final definitions as specified here. The key operation at the heart of a calendar program in any language is usually to calculate the weekday of January the first, for any year. Fortunately there is a well known method, called Zeller's congruence, for calculating this, which we can easily use to define a word `{jan1st}`. Logically the next operation needed will be to calculate how many days into the year any given day and month is, which we can call `{daynumber}`. The combination of `{jan1st}` and `{daynumber}` should then allow us to define the first word in our calendar vocabulary fairly easily...

10.1.1 Zeller's Congruence

Blocks 100 and 101 (See section 10.1.4 for full listings).

The following rather complicated formula will calculate the day of the week of the first day in any year, as a number from 0-6 (Sunday to Saturday respectively), and is good for any year from 1582 to 4902 inclusive! In addition it automatically takes care of leap years. The formula is expressed as it would be written in floating point BASIC, with the year in the variable Y, and the resulting day in D.

```
A = INT((Y-1)/100)
B = Y-1-100*A
```

```
D = 799+B+INT(B/4)+INT(A/4)-2*A
D = D MOD 7
```

Translating this directly into FORTH, with the same variable names, gives the following definition:

```
VARIABLE Y  ( year )
VARIABLE a  ( temporary variables )
VARIABLE b
: jan1st  Y @ 1 - 100 /  a !
          Y @ 1 - 100 a @ * -  b !
          799 b @ + b @ 4 / + a @ 4 / + 2 a @ * -
          7 MOD  ;
```

The calculation is simplified slightly by the fact that FORTH division is automatically integer division. {jan1st} could, of course, be written without the temporary variables a and b, and instead use the stack to retain these intermediate values until they are needed. Most FORTH programmers would, however, feel that the extra complexity (in writing and debugging) is not warranted – especially since {jan1st} is not a time critical operation, and will probably only need to be calculated once for every calendar operation.

Entering this into a new disk block (or the equivalent on a cassette based FORTH system), and LOADing the block will enable us to test {jan1st}, as follows:

```
1982 Y ! ok
jan1st . 5 ok
```

Where the result 5 indicates 'Friday' – and a glance at a 1982 calendar will confirm that January the 1st 1982 was indeed a Friday. Now check that {jan1st} hasn't left any unwanted values on the stack:

```
. 0 STACK EMPTY
```

A useful check to apply to any new definition.

At this stage we realise that before long a word which prints out the day of the week as "Sunday", "Monday" etc. will be needed, and since such a word will enable us to test {jan1st} very easily, we may as well define the word next:

```
: "days" ." Sunday    " ." Monday    " ." Tuesday   " ." Wednesday"
         ." Thursday " ." Friday    " ." Saturday " ;
: printday  12 *  ' "days"  +  3 +  9 TYPE ;
```

The word {"days"} simply contains a list of equal length strings. Each string will take up exactly 12 bytes of the parameter field; the first two bytes contain the code pointer for {."}, the third is the count byte and the remaining 9 bytes contain the string itself. {printday} needs a number on top of the stack between 0 and 6, which is multiplied by 12 and added to the parameter field address returned by the phrase {' "days"}. Adding 3 to skip the code pointer and byte count leaves the address of the required string, ready for {9 TYPE}.

A simple 'diagnostic' definition will now enable us to test both {jan1st} and {printday} rigorously:

```
: test1 1985 1980 DO
                    I Y !
                    ." Jan 1st " I . SPACE
                    jan1st
```

```
                ."  - " printday CR
            LOOP ;
```

Which should produce the result:

```
Jan 1st 1980 - Tuesday
Jan 1st 1981 - Thursday
Jan 1st 1982 - Friday
Jan 1st 1983 - Saturday
Jan 1st 1984 - Sunday
```

The technique of interspersing 'diagnostic' definitions with the major definitions under development is well worthwhile, and aids debugging considerably!

10.1.2 Daynumber and day

Blocks 102 and 103.

The word {daynumber} must calculate the number of days in the year up to a given day and month, so that, for example the 2nd of February is the 33rd day in the year (31 days in January + 2). The best approach here is to first set up an array of constants representing the number of days in each month, using the technique described in chapter 3.5, and since only small values are involved a byte array is appropriate:

```
CREATE dpmtable   ( days per month )
  31 C,  28 C,  31 C,  30 C,  31 C,  30 C,
  31 C,  31 C,  30 C,  31 C,  30 C,  31 C,

: dpm  dpmtable + C@ ;
```

We can now write {daynumber} very easily:

```
VARIABLE D  ( day )
VARIABLE M  ( month )
: daynumber  0                          ( initial value )
             12 0 DO
                      M @ I = IF
                                 D @ + LEAVE
                               ELSE
                                 I dpm +
                               THEN
                  LOOP ;
```

{daynumber} will return, on the stack, the number of days up to the day and month given by the variables D and M respectively. It works by simply looping through the months from January (0) to December (11). When the month specified by the variable {M} is reached, then the day in {D} is added to the daynumber so far, and the loop exited using {LEAVE}, else the number of days in the month returned from the table by {dpm} is added to the daynumber, and the loop repeats.

The first of the required calendar vocabulary words may now be easily written as:

```
: day  Y !  M !  D !
       jan1st daynumber + 1- 7 MOD
       printday ;
```

The day returned by {jan1st} for the specified year {Y}, is added to the daynumber, and 1 subtracted to make the total run from zero. The phrase {7 MOD} leaves a value

from 0 to 6, representing the actual day of the week, which is printed by {printday}. For example:

```
31 11 1981 day Thursday ok     ( 31st december 1981 )
1 0 1982 day Friday   ok       ( 1st january 1982 )
```

The months run from 0 for january, through to 11 for december.

Although Zeller's congruence allows for leap years, the words {dpm} and hence {daynumber} do not. Additionally, the definition for {day} does not check to see if the date specified actually did, or will, exist. These two necessary enhancements, together with constant definitions for month names, are straightforward, and are shown in the full block listings for the calendar vocabulary in section 10.1.4.

10.1.3 Month, year and daysleft

Blocks 104 and 105.

The major complexity in the definition of {month} is that of printing layout, since essentially all that is required is to print the numbers from 1 up to the number of days in the month, in their correct 'day' columns, as follows:

```
Sun Mon Tue Wed Thu Fri Sat
                        1   2
  3   4   5   6   7   8   9
etc..
```

We need to calculate the day of the week that the 1st of the month falls upon, then print spaces until under this 'day' column. Once there we count the days of the month, inserting newlines after each 'Sat' column has been filled. The easiest way of keeping track of where we are, in the current line, is with a 'character counter' variable, which is incremented whenever numbers, or spaces, are printed.

A definition for {month} along these lines is as follows:

```
0 VARIABLE chars
1 : month  Y !  M !  1 D !
2          SPACE ." Sun Mon Tue Wed Thu Fri Sat"
3          jan1st daynumber + 1- 7 MOD
4          4 * DUP  SPACES  chars !
5          M @ dpm 1+  1  DO
6                             I 4 .R     4 chars +!
7                             chars @ 24 > IF CR 0 chars ! THEN
8                          LOOP ;
```

The phrase in line 3 is identical to that used in {day} and has the effect of returning the weekday of the first of the month, since the day variable {D} has been set to 1 (on line 1). Line 4 then uses this weekday to print spaces up to the required column, and saves the number of spaces printed in {chars}. Line 5 sets up the DO loop to count from 1 up to the number of days in the month {M}. The phrase {I 4 .R}[1] on line 6 prints each date right justified in its column, and {4 chars +!} increments the character counter, which is checked on line 7 to see if a {CR} is needed.

[1]If you don't have {.R} a definition appears in chapter 8.4.

The version of {month} in block 104, in section 10.1.4, is slightly improved, with range checking using {datecheck}, and a heading printed by {monthprint}. In addition {dpm} takes account of leap years – see block 102. A sample run of {month} is as follows:

```
february 1982 month
february   : 1982
 Sun Mon Tue Wed Thu Fri Sat
       1   2   3   4   5   6
   7   8   9  10  11  12  13
  14  15  16  17  18  19  20
  21  22  23  24  25  26  27
  28

ok
```

The definition of {year} is now very straightforward, as shown in the block 104 listing.

The final word in the calendar vocabulary {daysleft} poses only one slightly awkward problem, which is how to calculate the number of days until the year end (which, in the business world, can be any calendar date), if the next year end is in the following year. The problem is that of leap years (again!); the word {daynumber} as defined in block 103 copes with leap years, but to calculate the number of days left in the current year (till december 31st) we must subtract the daynumber from 365 or 366 as appropriate. The definition of {daysleft} is shown in block 105 and is used as follows:

```
1 june yearend ok          ( initialise yearend )
1 march 1982 daysleft 92 ok
2 june 1982 daysleft 364 ok
```

10.1.4 The Calendar Vocabulary blocks listed

```
100 LIST

 0 ( Calendar Vocabulary, Zeller's congruence )
 1 DECIMAL
 2 FORTH DEFINITIONS
 3 VOCABULARY calendar
 4 calendar DEFINITIONS
 5
 6 VARIABLE Y  VARIABLE M  VARIABLE D       ( Year, Month, Day )
 7
 8 VARIABLE a  VARIABLE b          ( work variables for jan1st )
 9 : jan1st        ( return the day, 0-6, of jan 1st in year Y )
10          Y @ 1 - 100 / a !
11          Y @ 1 - 100  a @ * - b !
12          799 b @ +  b @ 4 / +  a @ 4 / +  2 b @ * -
13          7 MOD ;                                  ( → n )
14
15 101 LOAD  102 LOAD  103 LOAD  104 LOAD  105 LOAD

101 LIST

 0 ( Calendar Vocabulary, string printing )
```

```
 1 : "days"                                ( weekday string table )
 2     ." Sunday    " ." Monday    " ." Tuesday   " ." Wednesday"
 3     ." Thursday  " ." Friday    " ." Saturday  " ;
 4 : printday                                ( print weekday 0-6 )
 5     12 *  ' "days"  +  3 +  9 TYPE ;                  ( n → )
 6
 7 : "months"                              ( month string table )
 8     ." January   " ." February  " ." March     " ." April     "
 9     ." May       " ." June      " ." July      " ." August    "
10     ." September" ." October   " ." November  " ." December  " ;
11 : printmonth                             ( print month 0-11 )
12     12 *  ' "months"  +  3 +  9 TYPE ;                 ( n → )
13
14
15
```

102 LIST

```
 0 ( Calendar Vocabulary, date checking words )
 1 CREATE dpmtable                    ( table of days per month )
 2     31 C,  28 C,  31 C,  30 C,  31 C,  30 C,
 3     31 C,  31 C,  30 C,  31 C,  30 C,  31 C,
 4 : leap?  Y @ 4   MOD 0=                ( is year Y a leap year )
 5          Y @ 100 MOD 0= NOT AND
 6          Y @ 400 MOD 0= OR ;                          ( → flag )
 7 : dpm    DUP dpmtable + C@     ( return no of days per month )
 8          SWAP 1 = leap? AND      ( add 1 if Feb and leap yr )
 9          IF 1+ THEN ;                               ( n1 → n2 )
10 ( Check date within range, all return 'true' if NOT )
11 : Ycheck Y @ DUP 1582 < SWAP 4902 > OR ;             ( → flag )
12 : Mcheck M @ 12 U< NOT ;                             ( → flag )
13 : Dcheck D @ 1 -  M @ dpm  U< NOT ;                  ( → flag )
14 : datecheck   Ycheck Mcheck Dcheck OR OR
15               IF ." Date error" ABORT THEN ;
```

103 LIST

```
 0 ( Calendar Vocabulary, daynumber and day )   : C CONSTANT ;
 1 0 C january    1 C february   2 C march      3 C april
 2 4 C may        5 C june       6 C july       7 C august
 3 8 C september  9 C october   10 C november  11 C december
 4
 5 : daynumber  0 12 0 DO    ( calculate days up to date D/M/Y )
 6                       M @ I = IF      ( loop through months )
 7                                 D @ + LEAVE     ( until M=I )
 8                               ELSE
 9                                 I dpm +   ( accumulate days )
10                               THEN
11                     LOOP ;                             ( → n )
12 ( calculate day of week of date D/M/Y, 0-6 )
13 : D/M/Y jan1st daynumber + 1- 7 MOD ;                 ( → n )
14 : day   Y !  M !  D !  datecheck  ( print day of date given )
15         D/M/Y  printday ;                          ( d m y → )
```

104 LIST

```
 0 ( Calendar Vocabulary, month and year )
 1 VARIABLES chars                        ( character counter )
```

```
 2 : month  Y ! M !  1 D !  datecheck  ( print specified month )
 3          CR M @ printmonth SPACE ." :" Y @ .      ( heading )
 4          CR SPACE ." Sun Mon Tue Wed Thu Fri Sat" CR
 5          D/M/Y                  ( calculate 1st day of month )
 6          4 * DUP SPACES  chars !          ( go to day column )
 7          M @ dpm 1+  1  DO        ( step thru days in month )
 8                            I 4 .R  4 chars +!
 9                            chars @ 24 > IF  CR 0 chars ! THEN
10                         LOOP CR CR ;                ( m y → )
11
12 : year                              ( print whole year calendar )
13          12 0 DO                            ( loop thru months )
14                 I OVER month
15               LOOP DROP  ;                              ( y → )

105 LIST

 0 ( Calendar Vocabulary, yearend and daysleft )
 1 VARIABLE Mend  VARIABLE Dend          ( current end of year )
 2 : yearend                           ( intialise end of year )
 3          OVER OVER 1 = SWAP 29 = AND         ( 29th of Feb? )
 4          IF  ." You can't be serious!" ABORT  THEN
 5          Mend !  Dend ! ;                              ( d m → )
 6 : daysinY                          ( How many days in year Y )
 7          leap? IF  366  ELSE  365  THEN ;             ( → n )
 8 : daysleft                      ( Number of days up to yearend )
 9          Y !  M !  D !            datecheck  daynumber
10          Mend @ M !  Dend @ D !  datecheck  daynumber
11          OVER OVER > NOT IF ( specified date BEFORE yearend?)
12                             SWAP - .
13                          ELSE daysinY SWAP -
14                               1 Y +! datecheck daynumber + .
15                          THEN ;                  ( d m y → )
```

10.2 A Video Game Vocabulary

The game I have chosen to illustrate this vocabulary is 'Solo squash' (which is actually the player versus the machine!). The player controls a bat, which may be moved from left to right along the bottom line of the video display. The machine 'serves' a ball, with a random direction and speed from the top line of the display moving downwards. If the bat intercepts the ball then it bounces back up the screen, and positive points are scored, otherwise points are lost and the machine serves a new ball. Whenever the ball hits the top or sides of the screen, or the middle of the bat, then it 'bounces' in a perfectly elastic fashion. If the player catches the ball with the side of the bat, then it deflects randomly.

In order to 'structure' the vocabulary effectively it is helpful to plan the final word, `{squash}`, right at the start so that we can predict which intermediate words will need to be developed. The easiest way to set down the algorithm is with FORTH comment as follows:

```
: squash
        ( clear the screen and print the score )
        ( generate a new ball )
        BEGIN
```

```
                ( plot the ball and bat )
                ( check keyboard and move the bat if necessary )
                ( move the ball, bouncing if necessary )
                IF      ( the ball is on the baseline )
                  IF    ( and it hit the bat )
                     ( deflect the ball upwards )
                     ( increment the score )
                  ELSE  ( the bat missed the ball )
                     ( serve a new ball )
                     ( decrement the score )
                  THEN
                  ( print the new score )
                THEN
        0 UNTIL ; ( and loop )
```

All we have to do now is write definitions to 'fill in' the comment!

Because of its speed FORTH lends itself particularly well to programs of this type; in fact, FORTH games programs often have to be slowed down in order to make them possible for anyone with less than lightning reflexes! Another interesting point is that many 'arcade' video games are currently written in FORTH.

10.2.1 The Ball handling routines

Blocks 110 and 111 (See section 10.2.4 for full listings).

At the heart of any video game is 'graphics' handling; the ability to directly alter the contents of the screen without the normal constraints of output one line at a time. Any computer system with a memory mapped display, in which a single 'pixel' may be directly addressed and altered can be used. Figure 10.1 illustrates the display screen as defined throughout this vocabulary.

Figure 10.1 The videogame vocabulary display

A 'pixel' is a single picture element, and may be a 'dot' in a high resolution system, or a 'block of dots' in a low resolution (chunky graphics) system. Providing that the video memory addressing starts at the top left hand corner, and runs contiguously toward high memory, finishing in the bottom right corner, then we can define a word to plot a pixel in the position (x,y) as follows:

```
HEX    FOOO CONSTANT vdust       ( start of video memory )
DECIMAL 64 CONSTANT width        ( width of screen )
        16 CONSTANT height       ( height of screen )
: coord width * + vdust + ;      ( x y → addr )
: plot  coord C! ;               ( char x y → )
```

This example defines `{plot}` for a 16 line by 64 character 'chunky' graphics system. If the graphics character has the code 192 (decimal), then it may be plotted at

position 5,5 by the phrase:

```
192  5 5  plot
```

In systems with a high resolution video display {plot} will probably be defined to light up the pixel addressed, and an additional word {unplot} will switch off the pixel, in which case the phrase {char x y plot} in the videogame vocabulary will be replaced by {x y plot}, or if char is 32 ("space") by {x y unplot}. Since there is no standard for high resolution video displays I have assumed, throughout the videogame vocabulary, a low resolution 'chunky graphics' display.

As the final video game will involve mostly 'ball' handling (moving the ball, or finding out where it is), and these functions will be spread out over a number of colon definitions, it seems sensible to hold the current ball position coordinates in a pair of variables, {x} and {y}. In addition, {x} and {y} must never go outside the bounds 0 to xmax and 0 to ymax respectively, and so a definition {xyplot} which includes boundary checking will be useful:

```
width 1- CONSTANT xmax      ( set up xmax and ymax )
height 1- CONSTANT ymax
VARIABLE x  VARIABLE y      ( ball coordinates )
: xycheck   x @  0     MAX   x !    ( force x to zero, if less )
            x @  xmax  MIN   x !    ( or to xmax if greater )
            y @  0     MAX   y !    ( force y to zero, if less )
            y @  ymax  MIN   y ! ;  ( or to ymax if greater )
: xyplot    xycheck  x @ y @ plot ; ( char → )
```

Whenever {x} or {y} fall outside their respective ranges {xycheck} will 'force' them onto a boundary. Notice the use of {MAX} and {MIN} to avoid complex IF structures.

The ball is moved by the word {xystep}, which adds the values in the variables {xstep} and {ystep} to {x} and {y}. 'Bouncing' off a side is achieved simply by NEGATEing {xstep} or {ystep} as appropriate for the side that the ball has hit. The definitions {xleft}, {xright}, {ytop} and {ybottom} check to see if the ball has hit the left side, right side, top or bottom, and 'bounce' the ball if it has. The test definition {pattern} in block 111 will test these functions by drawing a line which bounces when it hits a side, as illustrated by figure 10.2.

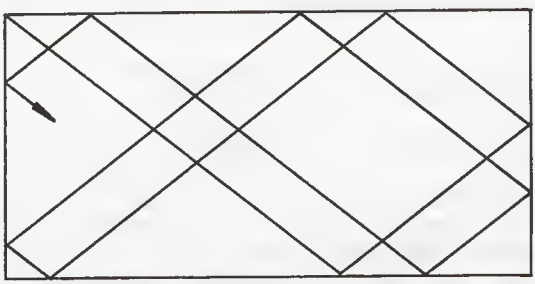


Figure 10.2 A test {pattern}

Type {n pattern} to plot n points. Altering the initial values of {x}, {y}, {xstep} and {ystep} will produce some very interesting patterns!

10.2.2 Bat handling

Block 112

The word {plotbat} will draw a bat three pixels wide on the bottom line of the display, for example:

```
32 bat !
163 plotbat
```

will draw the bat, in our 16 line by 64 character example screen, with the chunky graphics character 163 at positions (31,15), (32,15) and (33,15). The bat is moved to the left by {bat-1} and to the right by {bat+1}, which again will prevent the bat from going off the screen.

The word {movbat} expects an ASCII code from the keyboard on top of the stack; if it is the code for "z" then the ball is moved to the left, if it is "/" then the ball is moved to the right, or if it is neither then the word {ABORT} halts execution. The word {putbat} brings together the bat handling words:

```
32 CONSTANT "blank"             ( the space char )
163 CONSTANT "bat"              ( the bat char )
: putbat
        ?TERMINAL               ( key pressed? )
        ?DUP                    ( duplicate if so )
        IF
                "blank" plotbat ( erase the old bat )
                movbat          ( move it left or right )
                "bat" plotbat   ( and plot new bat )
        THEN ;
```

The word {?TERMINAL} is not a FORTH-79 standard word, but most systems do include it. {?TERMINAL} checks to see if a key has been pressed and returns its ASCII code on the stack if so, or zero otherwise. The word {putbat} therefore has no effect if a key has not been pressed.

On computer systems with a 'joystick' device, {movbat} and {putbat} could be re-defined to make use of this, resulting in a much more 'professional' game.

The following 'test' definition is useful:

```
: testbat
        clrs                    ( clear screen )
        "bat" plotbat           ( plot bat )
        BEGIN
                putbat          ( move it? )
        0 UNTIL ;               ( loop till ABORTed )
```

Typing {testbat} will clear the display and draw a bat, which may be moved with the "z" and "/" keys. Hitting any other key will ABORT the program, and return control to the keyboard, (note that any game program using {putbat} will have this ABORT facility built in). If your keyboard has a 'repeat' key try holding it down while hitting "z" or "/".

10.2.3 The Squash game

Blocks 113 and 114

The preliminary definitions in block 113 are mostly random 'ball' or 'bounce'

selection routines. They all rely upon the routine {random}, which is a standard pseudo random number generator:

```
VARIABLE rnd
1234 rnd !              ( initialise seed )
: random rnd @          ( fetch seed )
         1021 * 41 +    ( 1021*seed + 41 )
         DUP rnd ! ;    ( save new seed )
```

Every time the word {random} is executed a new random number is placed on the stack, within the range -32768 to 32767, excluding 0. The sequence of random numbers repeats itself every 65535 numbers, but that should not be a problem here! Most of the time we only require a limited range of random numbers and the word {>rand} will achieve this:

```
: >rand  random U* SWAP DROP ;
```

Writing {n >rand} will result in a random number in the range 0 to n-1.

The word {newball} selects a value for {x} between 0 and 63, and sets {y} to zero, so that the new ball starts somewhere on the top line of the display. In addition {xstep} is set to 2,1,-2 or -2, and {ystep} to 1 or 2, to give the ball a random downward direction.

{deflect} selects only values of {xstep} and {ystep} thus allowing the ball to be deflected randomly, in an upward direction, when it hits the bat.

The remaining words in block 113 are preliminary definitions to simplify the final definition of {squash}. The words {hitmiddle} and {hitlr} return the flag 'true' if the ball hit the bat in the middle, or on the left or right sides, respectively. {printscore} updates the score display by printing a carriage return, but not a line feed, {13 EMIT}, to overwrite the last score printout.

Now that we have all of the ingredients, a definition for {squash} is very straightforward and can be written by following the algorithm outlined in section 10.2:

```
: squash
   clrs                          ( clear screen )
   0 score !  printscore         ( zero score and print it )
   newball                       ( generate a new ball )
   BEGIN
     "ball" xyplot               ( plot ball )
     "bat"  plotbat              ( and bat )
     skill @ 0 DO
                putbat           ( move the bat? )
             LOOP
     "blank" xyplot              ( erase the ball )
     xystep                      ( move it )
     xright xleft ytop           ( bounce off top and sides )
     y @ ymax > IF               ( ball on baseline or below it? )
         hitmiddle IF            ( hit the bat middle? )
                      10 score +!       ( +10 points )
                      yreverse          ( and bounce )
                   ELSE
                      hitlr IF    ( hit left or right )
                               20 score +!      ( +20 points )
```

```
                        deflect          ( deflect ball )
                     ELSE  ( bat has missed ball )
                        -5 score +!      ( -5 points.)
                        newball          ( serve newball )
                     THEN
                  THEN
                  printscore   ( update score )
               THEN
 0 UNTIL ;                    ( loop till ABORTed )
```

There is only one 'improvement' to the original algorithm, which is that `{putbat}` is inside a loop which repeats `{skill}` times. This has two effects, one is that it slows the game down to a manageable speed, the other is that for each ball position the keyboard is checked `{skill}` times for the "z" or "/" bat movement keys. With a skill value of 400 the game is quite easy, 300 is moderately difficult, and 200 hairaising! The object is, therefore, to achieve a positive score with as low a `{skill}` rating as possible.

10.2.4 The Videogame Vocabulary blocks listed

```
110 LIST

 0 ( Video game vocabulary, basic plotting routines )
 1 FORTH DEFINITIONS                ( set up a new vocabulary )
 2 VOCABULARY videogames      videogames DEFINITIONS
 3 HEX   FOOO CONSTANT vdust      ( system dependent constants )
 4 DECIMAL 80 CONSTANT width  25 CONSTANT height
 5 : coord  width * + vdust + ;                ( x y → addr )
 6 : plot   coord C! ;                         ( char x y → )
 7 width  1 - CONSTANT xmax    height 1 - CONSTANT ymax
 8 VARIABLE x  VARIABLE y      ( x runs 0-xmax,  y runs 0-ymax )
 9 : xycheck  x @     0 MAX  x !                    ( check )
10           x @  xmax MIN  x !                   ( screen )
11           y @     0 MAX  y !                ( boundaries )
12           y @  ymax MIN  y ! ;
13 ( plot char at position x,y with boundary checks )
14 : xyplot   xycheck  x @ y @  plot ;              ( char → )
15 111 LOAD  112 LOAD  113 LOAD  114 LOAD      ( load the rest )

111 LIST

 0 ( Videogame vocabulary, bouncing ball routines )
 1 VARIABLE   xstep  VARIABLE ystep       ( change in position )
 2 : xystep   xstep @ x +!  ystep @ y +!  ;   ( change x and y )
 3 : xreverse xstep @ NEGATE xstep ! ; ( reverse direction of x)
 4 : yreverse ystep @ NEGATE ystep ! ;                ( and y )
 5 : xleft    x @     0< IF xreverse THEN ;  ( check for edges )
 6 : xright   x @ xmax > IF xreverse THEN ;       ( and BOUNCE )
 7 : ytop     y @     0< IF yreverse THEN ;
 8 : ybottom  y @ ymax > IF yreverse THEN ;
 9 : clrs     12 EMIT ; ( clear screen )    192 CONSTANT "ball"
10 ( Test program, bounce around the screen )
11 0 x !  0 y !  1 xstep !  1 ystep !      ( start in top left )
12 : pattern     clrs 0 DO                      ( clear screen )
13                       "ball" xyplot        ( plot the ball )
14                       xystep xleft xright ytop ybottom
```

```
15                      LOOP ;                                  ( n → )
```

```
112 LIST

 0 ( Videogame vocabulary, bat handling routines )
 1 VARIABLE bat   width 2 / bat !   ( bat position on baseline )
 2 : plotbat  DUP bat @ 1- ymax plot ( plot bat on bottom line )
 3            DUP bat @    ymax plot
 4                bat @ 1+ ymax plot ;                ( char → )
 5 : bat+1    bat @ xmax 1-  LITERAL < IF  1 bat +!  THEN ;
 6 : bat-1    bat @                  1 > IF -1 bat +!  THEN ;
 7 : movbat   DUP 122 = IF  DROP bat-1  ELSE  ( "z" = bat left )
 8            DUP  47 = IF  DROP bat+1  ELSE ( "/" = bat right )
 9                          ABORT  THEN THEN ;    ( else abort )
10 32 CONSTANT "blank"  163 CONSTANT "bat"
11 : putbat   ?TERMINAL ?DUP IF       ( move bat if keypressed )
12                          "blank" plotbat movbat "bat" plotbat
13                           THEN ;
14 ( Test bat handling )
15 : testbat  clrs "bat" plotbat  BEGIN putbat 0 UNTIL ;
```

```
113 LIST

 0 ( Videogame vocabulary, squash prelimimaries )
 1 VARIABLE rnd   1234 rnd !              ( random number seed )
 2 : random  rnd @  1021 * 41 +  DUP rnd ! ;            ( → n )
 3 : >rand   random U* SWAP DROP ;                      ( → n )
 4 ( set xstep to either -2, -1, 1 or 2 )
 5 : rxstep     4 >rand 1- DUP 0> NOT IF  1-  THEN  xstep ! ;
 6 ( serve a random new ball )
 7 : newball    64 >rand x !        0 y !        ( set x and y )
 8              2 >rand 1+ ystep !  rxstep ;    ( ystep, xstep )
 9 ( deflect the ball off the bat at random )
10 : deflect    2 >rand 2- ystep !  rxstep ;
12 : hitmiddle  bat @ x @ = ;                          ( → flag )
13 : hitlr      bat @ 1- x @ =  bat @ 1+ x @ = OR ;  ( → flag )
14 VARIABLE skill  300 skill !   VARIABLE score  0 score !
15 : printscore  13 EMIT  ." Score - " score @ 5 .R ;
```

```
114 LIST

 0 ( Videogame vocabulary, solo squash )
 1 : squash
 2    clrs  0 score !  printscore          ( initialise screen )
 3    newball                              ( set up a new ball )
 4    BEGIN
 5          "ball" xyplot  "bat" plotbat   ( plot ball and bat )
 6          skill @ 0 DO putbat LOOP      ( delay and move bat )
 7          "blank" xyplot  xystep xright xleft ytop    ( move )
 8          y @ ymax > IF                ( has ball hit baseline )
 9                       hitmiddle IF 10 score +! yreverse ELSE
10                       hitlr     IF  5 score +! deflect  ELSE
11                                   -5 score +! newball
12                                 THEN THEN
13                       printscore         ( put up new score )
14                     THEN
15  0 UNTIL ;                ( loop till we breakout of putbat )
```

Bibliography

The following is a list of books, articles and papers on the language FORTH.

1. Moore, C.H. and Rather, E.D., "The FORTH program for spectral line observing", Proc. IEEE, vol 61, September 1973.

2. Moore, C.H. and Rather, E.D., "FORTH: A new way to program a minicomputer", Astron. Astrophys. suppl. 15, 1974.

3. James, J.S., "FORTH on microcomputers", Dr. Dobbs, no 26.

4. Rather, E., Brodie, L., Rosenberg, C., "Using FORTH, FORTH-79 standard edition", FORTH Inc., 1979.

5. Moore, C.H., "The Evolution of FORTH, an unusual language", Byte, August 1980, pp76-92.

6. The FORTH Standards Team, "FORTH-79", 1980, distributed by the FORTH Interest Group, P.O. Box 1105, San Carlos, CA 94070, USA.

7. Fritzon, R., "Write your own pseudo-FORTH compiler", Micro-computing, Feb 1981, pp76-92 and Mar 1981, pp44-57.

8. Loeliger, R.G. "Threaded interpretive languages", Byte Books, 1981.

9. Katzan, H., "Invitation to FORTH", Petrocelli, 1981.

10. Brodie, L., "Starting FORTH", FORTH Inc., 1981.

11. Knecht, K., "Introduction to FORTH", Sams, 1982.

Answers to problems in Chapters 1-5

Chapter 1

1.

```
1  2  +   3  4  -   *
10  100  9  /  +   5  +
2  3  4  5  6  +  *  *  *
```

2.

```
(20+10)/(20-10)
1+2+3+4
20-(1*2)
```

3.

```
            100       -200      ABS       MAX
empty         100       -200      200       200
                          100      100
```

overall stack effect: (→ 200)

```
          -10000         0       MIN      NEGATE
empty       -10000        0      -10000     10000
                        -10000
```

overall stack effect: (→ 10000)

```
            1         2         SWAP      OVER
empty         1         2         1         2
                        1         2         1
                                            2
```

overall stack effect: (→ 2 1 2)

```
            10        DUP       DUP       *         *
empty         10        10        10        100       1000
                        10        10         10
                                  10
```

overall stack effect: (→ 1000)

```
            10        20        30        40        3         PICK      +
empty         10        20        30        40        3         20        60
                        10        20        30        40        40        30
                                  10        20        30        30        20
                                            10        20        20        10
                                                      10        10
```

overall stack effect: (→ 10 20 30 60)

4.
A good way of duplicating the top two numbers on the stack is by using the two operations {OVER OVER}. For example:

```
input:          OVER       OVER

stack:           20         10         20
                 10         20         10
                            10         20
                                       10
```

So to find the sum, difference, product and quotient of the same two numbers, the following sequence will work:

```
10 20 ok
OVER OVER + . 30 ok
OVER OVER - . -10 ok
OVER OVER * . 200 ok
OVER OVER / . 0 ok
```

In case the final result seems odd, don't forget that in integer division 10/20 = 0 remainder 10.

Chapter 2

1.

```
10 CONSTANT ten
ten 4 * 1 + CONSTANT fred
```

2.

```
VARIABLE XYZ          -100 XYZ !
VARIABLE A            XYZ @  fred  - A !
```

3.
The first solution that comes to mind is:

```
1  X @  +  X @ X @ *  +  X !
```

But a shorter way of squaring a variable is to use {DUP}:

```
X @ DUP *
```

Additionally the original expression looks like:

```
LET X = X + .....
```

therefore we can use the special operation {+!}:

```
X @ DUP *  1 +  X  +!
```

for a minimal solution?

4.

```
a x @ DUP * *    b x @ *  +  c +
```

5.
If your computer should have an output device 'memory mapped' into memory location 1000, say, then typing:

```
1000 CONSTANT device ok
```

will allow you to send output to the device by typing:

```
1 device ! ok
```

as if the 'device' were a variable. Caution – don't try this with the value '1000', you might corrupt an important memory location.

Chapter 3

1.

```
: triple  3 * ;
```

If a faster solution is required (at the expense of dictionary space) try the following:

```
: triple  DUP DUP + + ;
```

This is faster because 'addition' is much faster than 'multiplication'.

2.

```
: newpage
          12 EMIT          ( print form feed )
          ." Page - "      ( page heading )
          . CR  ;          ( and page number )
```

3.

```
CREATE array  -10 , 1 , 10 , 1000 ,
: array  2 * array + ;
```

Notice that by using the same name for the array address calculating definition, as for the array itself, we effectively 'hide' the array so that it is only accessible through its address calculation routine.

4.

```
: doublearray
          0 array @   2 *    0 array !   ( double element 0 )
          1 array @   2 *    1 array !   ( double element 1 )
          2 array @   2 *    2 array !   ( double element 2 )
          3 array @   2 *    3 array !   ( double element 3 ) ;
```

With a DO loop (see Chapter 5) we can achieve the same with a more compact definition:

```
: doublearray
          4 0 DO
                I array @  2 *   I array !
            LOOP ;
```

4.

```
word      stack effect
DUP       (n1 n2 → n1 n2 n2)      Duplicate top value
*         (n1 n2 n2 → n1 n3)      n3:=n2*n2
SWAP      (n1 n3 → n3 n1)
DUP       (n3 n1 → n3 n1 n1)      Duplicate again
*         (n3 n1 n1 → n3 n4)      n4:=n1*n1
+         (n3 n4 → n5)            n5:=n3+n4
```

Therefore the overall effect of {example} is to square each of the two values on the stack, and add the results:

```
example   (n1 n2 → n5)            n5:=n1*n1 + n2*n2
```

Chapter 4

1.

```
word      stack effect
1         ( → 1)
2         (1 → 1 2)
>         (1 2 → 0)          1 is not greater than 2, so flag
                             is set to 'false'
-4        ( → -4)
0<        (-4 → 1)           -4 is less than 0 so result is 'true'

5         ( → 5)
0>        (5 → 1)            'true'
NOT       (1 → 0)            'false'
```

2.

```
: SIGN  DUP  0> IF
                  ." positive"
               ELSE
               DUP 0=  IF
                          ." zero"
                       ELSE
                          ." negative"
                       THEN
               THEN  ;     ( number is preserved )
```

3.

```
1101101 XOR 1010001 = 0111100
1010 OR 101 = 1111

word      stack effect
4         ( → 4)
5         ( → 4 5)
=         ( 4 5 → 0)         4 does not equal 5
2         (0 → 0 2)
3         (0 2 → 0 2 3)
<         (0 2 3 → 0 1)      2 is less than 3
OR        (0 1 → 1)          result is 1
```

4.

```
A @ 2 =  B @ 2 =  AND NOT  IF  4 A ! THEN
```

5.

{0=} will have exactly the same effect as {NOT} upon a flag value, but {NOT} will not have the same effect as {0=} upon a number.

6.

The phrase {OVER OVER} in ex1 suggests that there should be two values on the stack

initially (see Chapter 1, question 4 above).

```
word      stack effect
OVER      (n1 n2 → n1 n2 n1)
OVER      (n1 n2 n1 → n1 n2 n1 n2)
>         (n1 n2 n1 n2 → n1 n2 flag)    true if n1>n2
IF        (n1 n2 flag → n1 n2)
  SWAP      (n1 n2 → n2 n1)             only SWAP if n1>n2
THEN
DROP      (n2 n1 → n2)                  if n1>n2
          (n1 n2 → n1)                  if not
```

Thus the overall effect is to leave the lesser of the two values on the stack – i.e. the same as {MIN}.

The initial {DUP} in ex2 requires one stack value.

```
word      stack effect
DUP       (n → n n)
IF        (n n → n)
  DUP       (n → n n)             duplicate only if non-zero
THEN
```

Thus the overall stack effect is to duplicate the number on top of the stack only if it is non-zero – i.e. the same as {?DUP}.

Chapter 5

1.

```
: stars
          CR                          ( initial newline )
          DUP 0 DO                    ( set up outer loop )
                DUP 0 DO              ( and inner loop with same limit )
                          ." *"           ( print a star )
                     LOOP
                     CR               ( newline )
                LOOP ;
```

2.

```
: sumall
          0                           ( accumulator to zero )
          SWAP 1+                     ( add 1 to limit )
          ROT                         ( index to top of stack )
          DO
              I +                        ( add up numbers )
          LOOP ;                      ( leave sum on stack )
```

3.

```
: delay 1000 0 DO LOOP ;              (delay approx 1 second )
: countdown
          0 SWAP                      ( swap index and limit values )
          DO
             I .                         ( print countdown )
             delay                       ( and delay )
     -1   +LOOP
          ." We have liftoff!!" ;
```

4.

```
ex1  0 3 6 9 12 15 ok
ex2  10 9 8 7 6 5 4 3 2 1 0 ok
ex3  5 10 15 20 25 ... 95 100 ok
```

5.

Decide initially on the input of parameters, i.e.

```
0 20 3  divisible
```

to print all numbers between 0 and 20 inclusive which are exactly divisible by 3.

```
: divisible
          ROT ROT              ( get index and limit to top of stack )
          SWAP 1+ SWAP         ( add 1 to limit )
          DO                   ( and loop )
            DUP                  ( duplicate divisor )
            I                    ( number to test )
            SWAP MOD             ( divide for remainder )
            0= IF                ( remainder = 0? )
                  I .            (number is divisible if so )
               THEN
          LOOP DROP ;          ( clear stack )

0 20 3  divisible 3 6 9 12 15 18 ok
```

6.

```
: DUMP
          BEGIN
               CR
               8 0 DO                        ( loop through 8 lines )
                     DUP 6 .R SPACE          ( print address )
                     8 0 DO
                           DUP C@ 3 .R SPACE 1+  ( print bytes )
                        LOOP
                     CR
                  LOOP
          KEY 32 -                           ( get a keypress )
          UNTIL                              ( loop if space )
          DROP ;                             ( else exit )
```

(See Chapter 8.4 for a definition of `.R`.)

Glossary of FORTH terminology

The intention of this glossary is to explain any terminology used, but not necessarily explained, in the main text of the book.

Address
A 16 bit value which represents the address of a byte in memory.

ASCII
'American Standard Code for Information Interchange' – the code used by the majority of computers for representing characters as byte values. The following table lists each character together with its byte value in decimal, and hexadecimal.

Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex
NUL	0	00	SPACE	32	20	@	64	40	'	96	60
SOH	1	01	!	33	21	A	65	41	a	97	61
STX	2	02	"	34	22	B	66	42	b	98	62
ETX	3	03	#	35	23	C	67	43	c	99	63
EOT	4	04	$	36	24	D	68	44	d	100	64
ENQ	5	05	%	37	25	E	69	45	e	101	65
ACK	6	06	&	38	26	F	70	46	f	102	66
BEL	7	07	'	39	27	G	71	47	g	103	67
BS	8	08	(	40	28	H	72	48	h	104	68
HT	9	09	)	41	29	I	73	49	i	105	69
LF	10	0A	*	42	2A	J	74	4A	j	106	6A
VT	11	0B	+	43	2B	K	75	4B	k	107	6B
FF	12	0C	,	44	2C	L	76	4C	l	108	6C
CR	13	0D	–	45	2D	M	77	4D	m	109	6D
SO	14	0E	.	46	2E	N	78	4E	n	110	6E
SI	15	0F	/	47	2F	O	79	4F	o	111	6F
DLE	16	10	0	48	30	P	80	50	p	112	70
DC1	17	11	1	49	31	Q	81	51	q	113	71
DC2	18	12	2	50	32	R	82	52	r	114	72
DC3	19	13	3	51	33	S	83	53	s	115	73
DC4	20	14	4	52	34	T	84	54	t	116	74
NAK	21	15	5	53	35	U	85	55	u	117	75
SYN	22	16	6	54	36	V	86	56	v	118	76
ETB	23	17	7	55	37	W	87	57	w	119	77
CAN	24	18	8	56	38	X	88	58	x	120	78
EM	25	19	9	57	39	Y	89	59	y	121	79
SUB	26	1A	:	58	3A	Z	90	5A	z	122	7A
ESC	27	1B	;	59	3B	[	91	5B	}	123	7B
FS	28	1C	<	60	3C	\	92	5C	!	124	7C
GS	29	1D	=	61	3D	]	93	5D	{	125	7D
RS	30	1E	>	62	3E	↑	94	5E	~	126	7E
US	31	1F	?	63	3F	_	95	5F	DEL	127	7F

All characters in the left hand column (and DEL) are 'control characters'. Very few computers implement all of these, but the common ones are:

```
BEL = ring the 'bell'
BS  = 'backspace' one character
HT  = 'horizontal tab' – move to the start of the next column
LF  = 'line feed'
FF  = 'form feed' – clear the screen and start a new page
CR  = 'carriage return' – move to the start of the line
```

Assembler
Assembler is the basic machine language of the microprocessor at the heart of any microcomputer system. Assembler is normally expressed symbolically using a set of mnemonic instructions specified by the manufacturer of the microprocessor, thus, for example, 8080 assembler is quite different to 6502 assembler even though microcomputers using these devices might run the same dialect of BASIC or FORTH. Advanced FORTH systems allow symbolic assembler to be embedded into FORTH applications using an ASSEMBLER vocabulary – specially written for the microprocessor running the FORTH system. Definitions incorporating assembler are known as CODE definitions and have a structure similar to a Colon Definition. For example, a CODE definition of a word to double the number on the stack might be written for an 8080-based machine as follows:

```
CODE  DOUBLE              ( new definition called DOUBLE )
          POPHL  CALL,    ( fetch top of stack to HL )
          H      DAD,     ( add HL to itself )
          PUSHHL JMP,     ( and push back result )
END-CODE
```

Notice that the 8080 assembler instructions are written operand first, mnemonic second. Code definitions are used to speed up time critical parts of a program, but suffer the disadvantage that they require the programmer to understand the assembly language of his microprocessor and the resulting FORTH applications may not run on other systems.

Binary
Base two.

Boolean
A numerical value representing one of the two logical states 'true' or 'false'. Also known as a 'flag'. In FORTH any 16 bit number may be treated as a Boolean, in which case non-zero values are 'true', zero is 'false'.

Byte
An 8 bit value. FORTH normally handles 16 bit numbers on the stack, thus byte values are represented as 16 bit numbers with the top 8 bits set to zero.

Character
A 7 bit value which represents a character in the ASCII standard. When contained in a larger number the upper bit(s) are set to zero.

Compiling word
A word which, when included inside a colon definition, has both a compile-time action and a run-time action. Examples of compiling words are {IF} {ELSE} {THEN} {DO} {LOOP} {+LOOP} {BEGIN} {UNTIL} {WHILE} and {LITERAL}.

Data stack
Same as 'Normal stack'.

Defining word
A word that, when executed, creates a new dictionary entry. The next word in the input stream is taken as the name of the new dictionary entry. Examples of defining words are {:} {CREATE} {VARIABLE} {CONSTANT} and {VOCABULARY}.

Dictionary
The structure which contains all word definitions including both 'system'

(predefined) words and user defined words, in compiled form in memory. Individual dictionary entries are named and are referenced by name.

Fixed-point number
The technique which is usually adopted by the FORTH programmer for representing decimal or 'real' numbers by asuming a fixed position decimal point. For example, if the decimal point is fixed two places to the left then a 1 on the stack represents the fixed-point number 0.01, or 1234 represents 12.34.

Flag
Same as Boolean.

Hexadecimal
Base sixteen, using the digits 0 to 9 and A, B, C, D, E, F.

Immediate
A word which will execute rather than compile, during the compilation of a colon definition. Immediate words are `{'}` `{(}` `{."}` `{;}` `{DOES>}` `{FORTH}` `{[}` and `{[COMPILE]}` together with all compiling words.

Infix
The term used by computer scientists to describe the normal convention for writing arithmetic expressions in which the operators are fixed in between the numbers. For example 5 multiplied by 10 is written 5 * 10.

Input stream
The sequence of characters currently being interpreted. These may come from either the keyboard (through the terminal input buffer) or from disk or cassette (through a block buffer). The values of `{>IN}` and `{BLK}` determine which of these is the current input stream.

Integer
The term used by computer scientists to refer to a 'whole' number (i.e. -1, 27, -342), as distinct from a decimal or 'real' number (i.e. 33.42, -0.047).

Literal
In FORTH terminology a 'literal' is a number appearing inside a colon definition which represents only the number itself (that is the number has not been defined as the name of a high level definition).

Normal stack
Also known as 'data stack', 'parameter stack' or simply 'stack'. A last-in, first-out buffer to contain 16 bit values. This stack is used for arithmetic and general purposes; most FORTH operations pop input values off the stack, and push results back onto the stack. Stack values may represent any number type, see 'Number'.

Number
FORTH has operations to manipulate the following number 'types':

type	*range*
Bit	0 or 1
Character (char)	0 ... 127
Byte (byte)	0 ... 255
Number (n)	−32,786 ... 32,767
Unsigned number (un)	0 ... 65,535
Double number (d)	−2,147,483,648 ... 2,147,483,647

```
Unsigned double number (ud)   0 ... 4,294,967,295
```

(The abbreviation in brackets is that used in the shorthand stack notation used throughout this book).

Double numbers are represented on the stack as two 16 bit values, with the upper 16 bit half above the lower 16 bit half. All other number types are represented on the stack as a single 16 bit value with high order bits set to zero when representing character or byte.

Parameter stack
Same as 'Normal stack'.

Pop
The operation of retrieving a number from the top of the stack.

Postfix
The same as 'Reverse Polish Notation', in which arithmetic expressions are written with the operators after the numbers on which they operate. For example 5 multiplied by 10 is written as 5 10 *. Postfix expressions may be directly evaluated using a stack, and all FORTH arithmetic must be written using postfix notation.

Push
The operation of saving a number on a stack.

RAM
Random Access Memory. Semiconductor memory which may be both read from and written into (changed), as distinct from ROM (Read Only Memory) which cannot be written into by a program. FORTH systems normally run in RAM.

Return stack
A stack reserved primarily for holding the 'return addresses' of words currently being executed.

Reverse Polish Notation
Same as 'Postfix' notation.

Stack
A special buffer for storing numbers such that the last number to be stored (pushed) will be retrieved (popped) first. FORTH maintains two stacks, the 'normal stack' and the 'return stack'.

String
The term given to a list of byte values in memory which represent, in ASCII, a number of characters of text.

Two's complement arithmetic
FORTH represents signed single length and double length numbers using two's complement notation. Thus the topmost bit indicates the sign of the number; 0 means positive and 1 means negative. Taking single length (16 bit) numbers as examples,

```
0000000000000000
```

is, expressed in binary, the smallest positive number (decimal 0), and

```
0111111111111111
```

is the largest positive number (decimal 32,767). Whereas

```
1000000000000000
```

is the lowest negative number (decimal -32,768), and

```
1111111111111111
```

represents the decimal value -1.

To calculate the binary value of a negative number in two's complement notation take its magnitude (absolute value) in binary, invert each bit, and finally add one. For example,

```
decimal +2  =  0000000000000010
inverted    =  1111111111111101
add one     =  1111111111111110
```

which is the two's complement representation of decimal -2, in binary.

The same procedure is used to convert a negative binary number into its decimal equivalent, for example,

```
in binary,      1111111111111000
inverted    =   0000000000000111
add one     =   0000000000001000  =  decimal +8
```

and so the original binary number represented the decimal value -8.

Two's complement arithmetic is useful because it allows us to subtract by adding. For example, to subtract decimal 8 from decimal 2, add the two's complement representations of -8 and +2,

```
decimal +2  =  0000000000000010
decimal −8  =  1111111111111000
sum         =  1111111111111010
```

which represents the correct result decimal -6. Notice that the sign of the result is automatically correct.

Vocabulary
A named subset of the dictionary. A number of different vocabularies may co-exist in the dictionary, all linked into the primary FORTH vocabulary.

Word
In FORTH terminology a WORD is any sequence of characters in the input stream delimited by 'space' characters on either side. This is different to the normal computer terminology of 'word=16 bit binary number'. In FORTH a 16 bit quantity is always referred to as a NUMBER, ADDRESS or sometimes a CELL.

INDEX

This index lists the complete FORTH-79 standard word set, together with the recommended pronunciation where it is not obvious, and the number of the page containing the full formal description of the word. An informal description, with examples, will in most cases be found in the preceding chapter.

! *"store"* 17
"sharp" 85
#> *"sharp-greater"* 85
#S *"sharp-s"* 85
' *"tick"* 100
(*"paren"* 29
* *"times"* 9
*/ *"times-divide"* 84
*/MOD *"times-divide-mod"* 84
+ *"plus"* 9
+! *"plus-store"* 18
+LOOP *"plus loop"* 49
, *"comma"* 29
- *"minus"* 9
-TRAILING *"dash-trailing"* 73
. *"dot"* 10
." *"dot-quote"* 29
/ *"divide"* 9
/MOD *"divide-mod"* 9
0= *"zero-equals"* 37
0> *"zero-greater"* 37
1+ *"one-plus"* 28
1- *"one-minus"* 28
2+ *"two-plus"* 28
2- *"two-minus"* 28
79-STANDARD 85
: *"colon"* 29
; *"semi-colon"* 29
< *"less-than"* 37
<# *"less-sharp"* 85
= *"equals"* 37
> *"greater-than"* 37
>IN *"to-in"* 74
>R *"to-r"* 84
? *"question-mark"* 18
?DUP *"query-dup"* 37
@ *"fetch"* 17
ABORT 50
ABS *"absolute"* 9
ALLOT 29
AND 38
BASE 73
BEGIN 49
BLK *"b-l-k"* 63

BLOCK 62
BUFFER 63
C! *"c-store"* 18
C@ *"c-fetch"* 18
CMOVE *"c-move"* 74
COMPILE 101
CONSTANT 18
CONTEXT 64
CONVERT 73
COUNT 73
CR *"c-r"* 10
CREATE 29
CURRENT 64
D+ *"d-plus"* 84
D< *"d-less-than"* 84
DECIMAL 73
DEFINITIONS 64
DEPTH 28
DNEGATE *"d-negate"* 84
DO 49
DOES> *"does"* 99
DROP 8
DUP *"dupe"* 8
ELSE 38
EMIT 72
EMPTY-BUFFERS 63
EXECUTE 101
EXIT 101
EXPECT 73
FILL 74
FIND 100
FORGET 18
FORTH 64
HERE 74
HOLD 85
I 49
IF 38
IMMEDIATE 101
J 49
KEY 50
LEAVE 49
LIST 62
LITERAL 101
LOAD 62
LOOP 49
MAX *"max"* 9
MIN *"min"* 9
MOD *"mod"* 9
MOVE 63
NEGATE 9

NOT 37
OR 38
OVER 9
PAD 63
PICK 9
QUERY 73
QUIT 50
R> *"r-from"* 84
R@ *"r-fetch"* 84
REPEAT 50
ROLL 9
ROT *"rote"* 9
SAVE-BUFFERS 63
SCR *"s-c-r"* 62
SIGN 85
SPACE 73
SPACES 73
STATE 101
SWAP 9
THEN 38
TYPE 73
U* *"u-times"* 84
U. *"u-dot"* 10
U/MOD *"u-divide-mod"* 84
U< *"u-less-than"* 37
UNTIL 50
UPDATE 63
VARIABLE 18
VOCABULARY 63
WHILE 50
WORD 73
XOR *"x-or"* 38
[*"left-bracket"* 101
[COMPILE] *"bracket-compile"* 101
] *"right-bracket"* 101

FORTH-79 Handy Reference

Stack notation: (normal stack before → normal stack after)

Operand key: n,n1 .. 16 bit value
d,d1 .. 32 bit value
addr .. 16 bit address
byte .. 16 bit value whose lower 8 bits only are set or used by the operation
char .. 16 bit value whose lower 7 bits only are set or used by the operation, representing an ASCII character
flag .. 16 bit value representing a Boolean flag, a zero value = 'false', a non-zero value = 'true'
u .. the prefix denoting an unsigned number

Stack Manipulation:

DUP	(n → n n)	Duplicate top of stack
DROP	(n →)	Lose top of stack
SWAP	(n1 n2 → n2 n1)	Reverse top two stack items
OVER	(n1 n2 → n1 n2 n1)	Duplicate second item on top
ROT	(n1 n2 n3 → n2 n3 n1)	Rotate third item to top
PICK	(n1 → n2)	Duplicate n1th item on top of stack
ROLL	(n →)	Rotate nth item to top
?DUP	(n → n (n))	Duplicate only if non-zero
DEPTH	(→ n)	Count number of items on stack
>R	(n →)	Move top item to return stack
R>	(→ n)	Retrieve item from return stack
R@	(→ n)	Copy top of return stack

Comparison:

<	(n1 n2 → flag)	True if n1 less than n2
=	(n1 n2 → flag)	True if n1 equals n2
>	(n1 n2 → flag)	True if n1 greater than n2
0<	(n → flag)	True if n negative
0=	(n → flag)	True if n is zero
0>	(n → flag)	True if n greater than zero
D<	(d1 d2 → flag)	True if d1 less than d2
U<	(un1 un2 → flag)	Compare as unsigned integers
NOT	(flag → -flag)	Reverse truth value

Arithmetic and logical:

+	(n1 n2 → sum)	Add
–	(n1 n2 → diff)	Subtract (n1-n2)
*	(n1 n2 → prod)	Multiply
/	(n1 n2 → quot)	Divide (n1/n2), quotient rounded toward zero
MOD	(n1 n2 → rem)	Remainder from (n1/n2), rem has sign of n1
/MOD	(n1 n2 → rem quot)	Divide with remainder and quotient
1+	(n → n+1)	Add 1
1–	(n → n-1)	Subtract 1
2+	(n → n+2)	Add 2
2–	(n → n-2)	Subtract 2
D+	(d1 d2 → dsum)	Double precision add
*/	(n1 n2 n3 → quot)	(n1*n2/n3) with double precision intermediate
*/MOD	(n1 n2 n3 → rem quot)	As */ with remainder and quotient
U*	(un1 un2 → ud)	Multiply with double result, all unsigned
U/MOD	(ud un → urem uquot)	Divide double number by single, all unsigned)
MAX	(n1 n2 → max)	Compare n1 with n2 and leave the greater
MIN	(n1 n2 → min)	Compare n1 with n2 and leave the lesser
ABS	(n → |n|)	Absolute value
NEGATE	(n → -n)	Change sign (2's complement)
DNEGATE	(d → -d)	Change sign of double number
AND	(n1 n2 → and)	Bitwise logical AND
OR	(n1 n2 → or)	Bitwise logical OR
XOR	(n1 n2 → xor)	Bitwise logical XOR

Memory:

@	(addr → n)	Replace address by number at address
!	(n addr →)	Store n at address
C@	(addr → byte)	Replace address by byte at address
C!	(byte addr →)	Store byte at address
?	(addr →)	Print the number stored at address
+!	(n addr →)	Add n into the number stored at address
MOVE	(addr1 addr2 n →)	Move n numbers starting at addr1 to addr2
CMOVE	(addr1 addr2 n →)	Move n bytes starting at addr1 to addr2
FILL	(addr n byte →)	Fill n bytes of memory starting at addr

Control Structures:

DO	(end+1 start →)	Set up DO .. LOOP or +LOOP given index range
LOOP	(→)	Add one to index, exit loop when index>end
+LOOP	(n →)	Add n to index, exit loop when index>end for n>0, or when index=end for n<0
I	(→ n)	Place current loop index value onto stack
J	(→ n)	Place index value for next outer loop onto stack
LEAVE	(→)	Force DO loop termination
IF	(flag →)	Construction: IF ..true.. THEN
ELSE	(→)	or IF ..true.. ELSE ..false.. THEN,
THEN	(→)	execute true or false words according to flag value at IF
BEGIN	(→)	Mark the start of an UNTIL loop or a WHILE loop
UNTIL	(flag →)	In BEGIN .. UNTIL, loop until flag is true
WHILE	(flag →)	In BEGIN .. WHILE .. REPEAT construct,
REPEAT	(→)	loop while flag true at WHILE
EXIT	(→)	Prematurely exit this colon definition
EXECUTE	(addr →)	Execute word whose compilation address is at addr

Character input-output:

CR	(→)	Print carriage return and line feed
EMIT	(char →)	Print character
SPACE	(→)	Print one space
SPACES	(n →)	Print n spaces
." text"	(→)	Print text delimited by "
TYPE	(addr n →)	Print the string of n characters at address
COUNT	(addr → addr+1 n)	Fetch count byte n and point to string
-TRAILING	(addr n1 → addr n2)	Reduce character count by trailing spaces
KEY	(→ char)	Read a single character from the keyboard
EXPECT	(addr n →)	Read n characters (or until return) from keyboard into memory at address
QUERY	(→)	Read 80 characters (or until return) from keyboard into input buffer
WORD	(char → addr)	Read next word from input stream using char as delimiter. Leave address of length byte

Number input-output:

BASE	(→ addr)	System variable containing current base
DECIMAL	(→)	Set base to decimal
.	(n →)	Print n with one trailing space
U.	(un →)	Print unsigned with one trailing space
CONVERT	(d1 addr1 → d2 addr2)	Convert string at addr1+1 to double number, add into d1 leaving result d2
<#	(→)	Begin a formatted number conversion
#	(ud1 → ud2)	Convert next digit of ud1 and HOLD it
#S	(ud → 0 0)	Convert and HOLD all remaining significant digits
HOLD	(char →)	Insert character into formatted string
SIGN	(n ud → ud)	HOLD minus sign only if n is negative
#>	(ud → addr n)	Drop ud and prepare string for TYPE

Mass storage input-output:

LIST	(n →)	List block n and set SCR to n
LOAD	(n →)	Interpret block n, then resume normal input
SCR	(→ addr)	System variable containing listed block number
BLOCK	(n → addr)	Leave address of block n, reading block off storage if necessary
UPDATE	(→)	Mark last block accessed as updated
BUFFER	(n → addr)	Assign a free buffer to block n, leaving its address
SAVE-BUFFERS	(→)	Write all updated blocks to storage
EMPTY-BUFFERS	(→)	Mark all buffers as empty

Defining words:

: <name>	(→)	Begin colon definition of <name>
;	(→)	End colon definition
VARIABLE <name>	(→)	Define variable <name>,
<name>	(→ addr)	returns its address when executed
CONSTANT <name>	(→ n)	Define constant <name> with value n,
<name>	(→ n)	returns its value when executed
VOCABULARY <name>	(→)	Define a vocabulary <name>, becomes CONTEXT vocabulary when executed
CREATE <name>	(→)	Create an empty dictionary entry <name>,
<name>	(→ addr)	returns parameter field address when executed
DOES>	(→ addr)	Used in defining new defining-words

Vocabularies:

CONTEXT	(→ addr)	Variable pointing to vocabulary, for word searches
CURRENT	(→ addr)	Variable pointing to vocabulary for new definitions
FORTH	(→)	Set CONTEXT to the main FORTH vocabulary
DEFINITIONS	(→)	Set CURRENT vocabulary to CONTEXT
' <name>	(→ addr)	Find address of <name> in dictionary
FIND	(→ addr)	Find compilation address of next word in input stream
FORGET <name>	(→)	Forget all definitions back to <name>

Compiler:

,	(n →)	Compile n into the dictionary
ALLOT	(n →)	Add n bytes to the parameter field of the most recently defined word
IMMEDIATE	(→)	Mark most recently defined word as immediate
LITERAL	(n →)	Compile n as a literal
STATE	(→ addr)	System variable is non-zero during compilation
[	(→)	Stop compiling input stream and start executing
]	(→)	Stop executing input stream and start compiling
COMPILE	(→)	Compile the address of the following word
[COMPILE]	(→)	Compile the following word, even if immediate

Miscellaneous:

(	(→)	Begin comment, terminate by)
HERE	(→ addr)	Address of next available dictionary location
PAD	(→ addr)	Address of a 64 byte scratchpad area
>IN	(→ addr)	Variable containing offset into input buffer
BLK	(→ addr)	Variable containing block currently being LOADed
ABORT	(→)	Clear all stacks and return control to keyboard
QUIT	(→)	As ABORT but leave normal stack intact
79-STANDARD	(→)	Verify that system conforms to FORTH-79

Based on material produced by the FORTH Standards Team, P.O. Box 1105, San Carlos, CA 94070, USA.

about this book

FORTH is a new, unusual and exciting computer language. Originally developed to control telescopes, it has since been applied in many diverse fields including the animation sequences for 'Star Wars'.

FORTH is a compact and fast language: faster than BASIC yet more flexible. It is more than **just** a language: it is a programming language, editor, assembler and disk operating system all rolled into one. In short, a complete 'environment'. This book describes the standard dialect of FORTH, together with numerous examples, exercises and complete programs. Read it — you'll never use BASIC again!

Alan Winfield is a lecturer in the Department of Electronic Engineering at the University of Hull. He specialises in Computer Languages for engineering and has recently written a complete FORTH compiler.

Sigma Technical Press,
5 Alton Road,
Wilmslow,
Cheshire SK9 5DY,
United Kingdom.

£6.95

ISBN : 0 905104 22 6