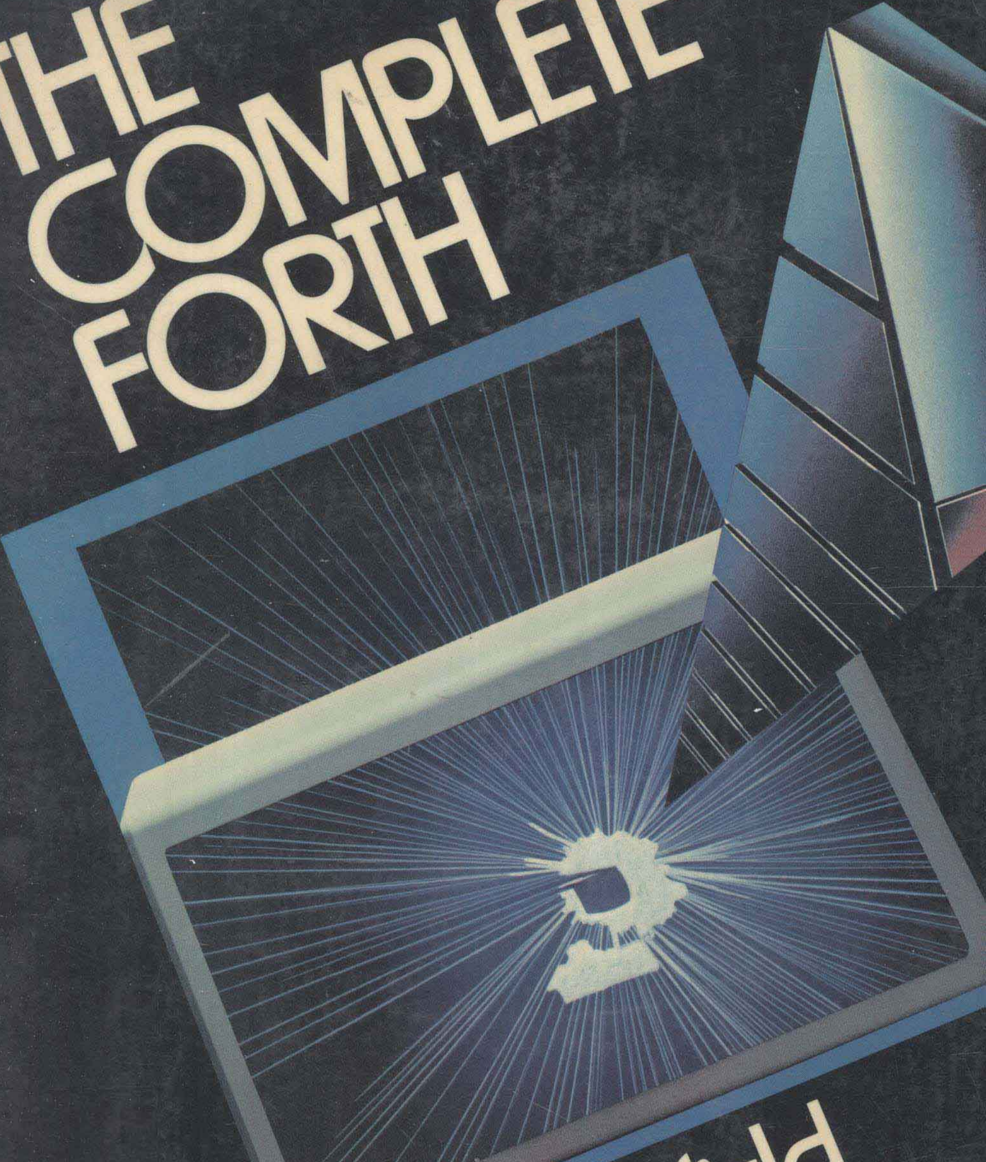
THE COMPLETE FORTH



Alan Winfield

THE COMPLETE FORTH

A.F.T.Winfield

Sigma/Wiley NY.

First published in 1983 by:

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

Library of Congress Cataloging in Publication Data

Winfield, Alan, 1956–
The complete FORTH.

Includes index.
1. FORTH (Computer program language) I. Title.
II. Title: The complete F.O.R.T.H.
QA76.73.F24W55 1983 001.64'24 83-14515
ISBN 0-471-88235-6

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

84 85 10 9 8 7 6 5 4 3 2 1

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.

Introducing FORTH

One of the first things that newcomers to computing discover is that in order to be useful computers must be programmed. For beginners this usually means programming in BASIC, but the computer professional as well as the adventurous hobbyist will probably dabble with a wide range of other languages as well; Pascal, Fortran and Assembler (machine code) to name just a few. FORTH is a relatively recent addition to this programming language spectrum and provides a unique way of programming a computer, which is quite unlike BASIC, or any of the other languages I have mentioned.

How is FORTH Different?

Well, it is the way in which programming in FORTH is achieved by actually extending the language. Let me illustrate what I mean with an example.

Suppose that you have a computer in an office which is to be used for routine office work. A useful program for this computer would be a 'calendar', which will calculate things like the day of the week a particular date falls upon, or the number of days left in the current financial year, or even give an instant display of the calendar for any given month and year.

Such a program could easily be written in, say, BASIC, but would have to be loaded off disk or cassette and RUN whenever it was needed. Then it would probably display a menu to ask the user which function was required. In short, it would require a reasonable amount of typing effort to load and run.

The FORTH programmer approaches the problem in quite a different way. He first decides upon the most convenient way of requesting the required calendar information. For example, to find the day of the week of a particular date, simply typing the date, followed by 'day':

```
1 january 1982 day
```

and ideally FORTH will respond straight away with the reply:

```
Friday
```

To find the number of days left this financial year we could type the date, followed by 'daysleft':

```
1 december 1981 daysleft  30
```

and FORTH has responded with the answer '30'. To request a print out of the calendar for one month we could type:

```
february 1982 month
```

and FORTH would respond by printing:

```
February   : 1982
 Sun Mon Tue Wed Thu Fri Sat
      1   2   3   4   5   6
   7   8   9  10 ... etc.
```

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.

Contents

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.

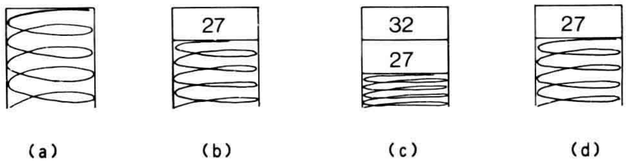


Figure 1.1 The Spring Loaded Stack

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 * .
```

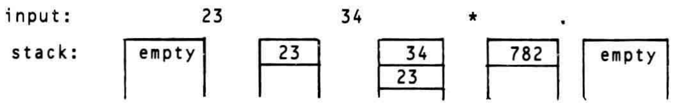


Figure 1.2

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.

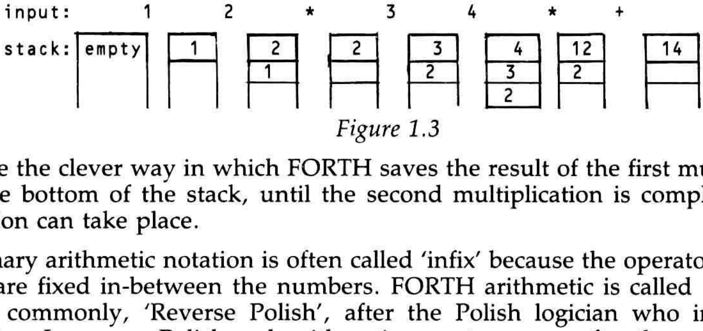


Figure 1.3

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 reasón 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.

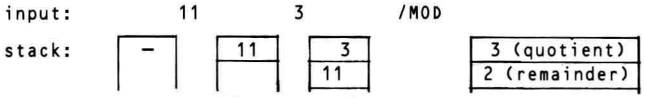


Figure 1.4 The {/MOD} operation

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
```