Exploring FORTH

Owen Bishop
in collaboration with
Audrey Bishop

Exploring FORTH

Owen Bishop
in collaboration with
Audrey Bishop

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

British Library Cataloguing in Publication Data
Bishop, O. N.
Exploring FORTH
1. FORTH (Computer program language)
I. Title II. Bishop Audrey
001.64'24 Q.A.76

ISBN 0-246-12188-2

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Chapter One
Which FORTH?

There has been an escalating interest in FORTH among micro owners during the past few years. As a result of this, the FORTH language is being made available on an increasing number of popular microcomputers. There are tapes, disks, cartridges and special ROMs, all of which provide FORTH for those micros which normally operate in BASIC. There is even a micro which has FORTH as its resident language. This book is intended to be used with any of these microcomputers, whatever version of FORTH they use.

The reason that this is possible is that FORTH is an easily transportable language. That is to say, you can write a 'program' on one micro, then key it into a different micro with a reasonable chance that it will work first time. The word 'program' was put into quotes in the previous sentence because the idea of 'writing a program' does not apply to FORTH as it does to BASIC and many other languages. As will be explained in more detail later, FORTH is based on a set of words, each of which has a specified action. The writer of a version of FORTH supplies you with a set of a few hundred words. When you 'program' in FORTH, you use these words to define new words of your own. You extend the language originally supplied to you by adding whatever words you need. You can then use the words you have defined to define even more words. The action of some of your words may be most elaborate. Yet everything is done in short, easily understood steps.

With BASIC and many other languages, you put together the statements and functions that are provided by the version of the language, building up line upon line of program. The program consists of a series of instructions telling the computer what to do. If the BASIC of your micro lacks certain statements which you need, you can often write a program line to do what is wanted, though sometimes this is difficult and it is always less satisfactory. For

example, if your BASIC lacks the REPEAT...UNTIL statements, you can manage with GOTO, but the program runs much more slowly.

There is nothing in FORTH which corresponds exactly to a program. It is true that there are the sequences of words used in defining other words, but these are short sequences more like subroutines or procedures than programs. On the other hand, FORTH words differ from subroutines or procedures because there is no 'main program' to jump back to after they have been executed.

(a)

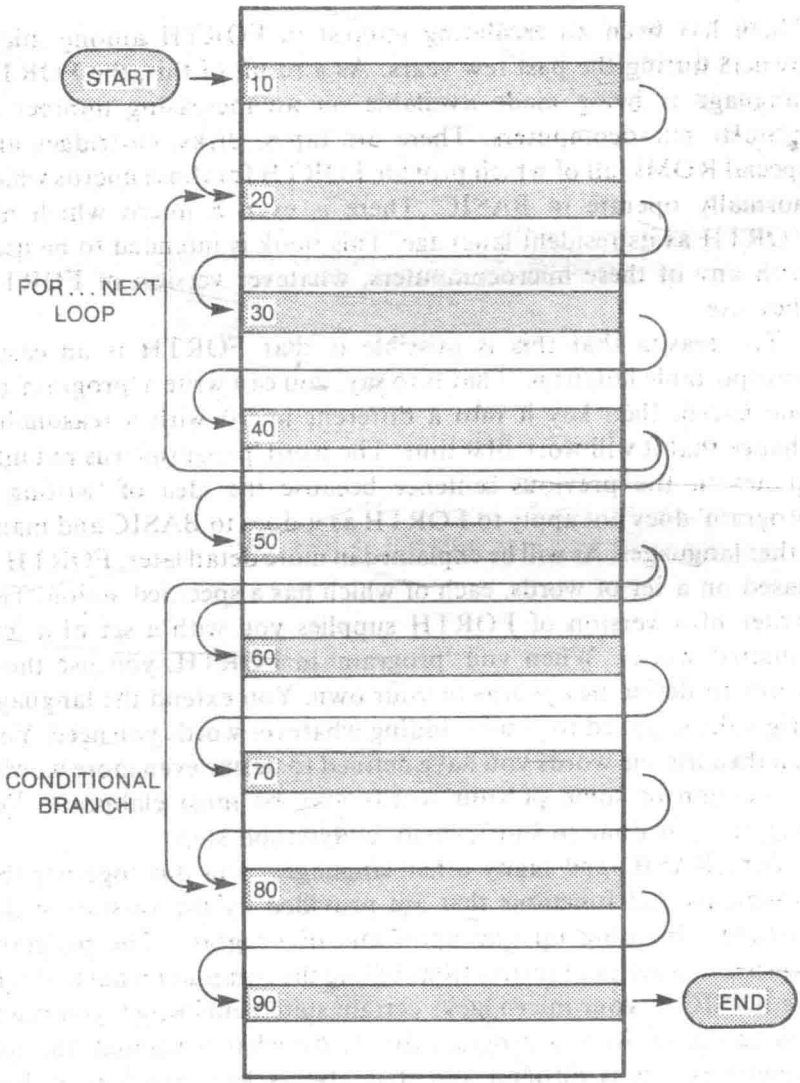

Fig. 1.1. The course of action when running (a) a program in BASIC; (b) an application in FORTH.

Figure 1.1 illustrates the difference between FORTH and a program-based language, such as BASIC. You can see how the action of a program proceeds line-by-line, perhaps with loops and repetitions, from the start of the program to the end. It is a *program* in the true sense; a list of things to be done.

FORTH has no such list. The action moves from one word to another. It threads its way among the words of the language itself. Exactly how this happens is explained later. FORTH is described as a *threaded language*. One word calls upon the actions of others in

(b)

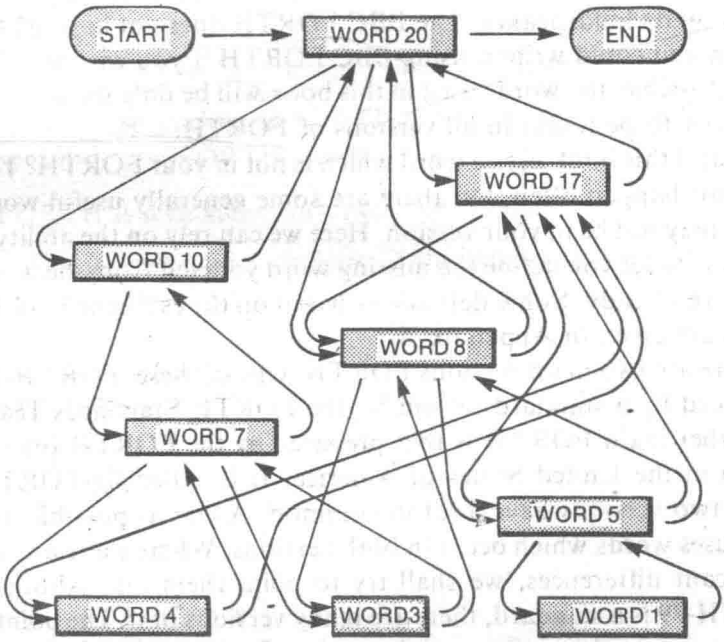

Fig.1.1 (contd)

performing its own actions. There is no 'listing' that you can read from start to finish, to see what the program does. To follow the action you must thread your way from word to word. But, since the definition of each word is short and clearly related to the definitions of other words, this is an easy matter. When working with FORTH, we use or apply the existing words to create new words. For this reason, it is best to speak of an *application* rather than a 'program'.

We will return to this topic in more detail in various parts of the book.

Most of the words that are provided with your version of FORTH will be the same as those provided in other versions and will have the same action. There may be a few additional words which the designer thinks you will find helpful. There will also be words which relate to the special features of your computer and are not present in other versions. For example, the Acornsoft FORTH for the BBC Microcomputer has a word MODE which is used for changing the display mode of the computer. This word does not apply to other computers, such as the Jupiter Ace. But the FORTH of the Jupiter Ace has some words of its own, including BEEP, to make a beeping sound on its loudspeaker. The BBC FORTH does not have BEEP, though you could write it using BBC FORTH if you wished to. As far as possible, the words used in this book will be only those which are likely to be found in all versions of FORTH.

What if this book uses a word which is not in your FORTH? This does not happen often, but there are some generally useful words which may not be in your version. Here we can rely on the ability of FORTH to let you define the missing word yourself from the words you have already. Some definitions based on the essential FORTH words are given in Appendix A.

There are two main versions FORTH. One of these, FORTH-79, is defined by a standard set out by the FORTH Standards Team. The other main FORTH is that prepared by the FORTH Interest Group in the United States of America. It is called fig-FORTH. These two versions have a lot in common. As far as possible, this book uses words which occur in both versions. Where there are any significant differences, we shall try to point them out. Although FORTH-79 is a standard, there are many versions of it. The point is that the standard specifies a minimum set of words and how they are to act. Any FORTH which has this minimum set and which has a number of other standard features can claim to be FORTH-79. The writers of such a version are then free to add any other words, especially words like those mentioned above which cater for the features of a given micro. Provided that special words are avoided, an application can generally be transferred from one micro to another, without any problems. On the other hand, the words which apply to the special features of a micro are usually those which provide the most effective displays or make best use of features such as sound generators. Applications which do not make use of such words are not exploiting the features of the micro to the best

advantage. This dilemma is not the fault of FORTH, but of lack of standardisation in micros, especially with regard to screen format and display routines. This is the point at which you may need to adapt the suggestions given in this book to suit the special features of your computer. Guidance is given wherever possible. By the time you have covered the early chapters of this book, you will feel confident to use the special words of your version of FORTH or to write any other words you need to get the most out of your computer.

To summarise

In this chapter you have learned that:

- Writing applications in FORTH consists of using FORTH words to define new FORTH words of your own.
- FORTH is a flexible, transportable language.
- There are two main versions of the language, FORTH-79 and fig-FORTH.

Chapter Two
Why FORTH?

The increasing popularity of FORTH is due to several factors. One of these, already mentioned, is its *transportability*. This becomes an increasingly important factor as more varieties of microcomputer come on the market. Another factor is its *speed*.

FORTH is fast in two ways. It is claimed that writing an application in FORTH takes only half the time required to write the equivalent program in another high-level language, such as BASIC. A FORTH application takes only one tenth of the time required to write its equivalent in assembler. So here is a way to get your computer into action with the minimum of delay at the writing stage.

Once the application is written, it runs faster too. To check on this, let us see how long it takes the micro to count up to 30000, using BASIC. Here is the program which does it. This took 16 seconds to run on the BBC Microcomputer.

```
>10 FOR J = 1 TO 30000
>20 NEXT
```

Now run FORTH on your computer and key in the following word definition. Type this in exactly as shown below, taking special care to leave all the spaces. FORTH is particular about spaces!

```
: TEST 30000 1 DO LOOP ;
```

When you have finished, press RETURN. The computer responds with the familiar and reassuring 'OK' on the line below. This tells you that it is ready for whatever you want it to do next.

Before doing anything further, consider what you have just typed. The line defines a word named TEST. It was given this name as we want to use it to test how fast the computer counts when using FORTH. The name of the word is followed by two numbers. Comparing this line with the BASIC listing above, shows that these are the values for the end and the beginning of the loop. You will

notice that the number for the end of the loop comes *before* the number for the beginning. This 'back-to-front' habit of FORTH is something we shall see a lot more of. Why it works this way is explained later. It may seem strange at first, but you soon get used to it, just as one soon gets used to driving a car on the opposite side of the road when visiting a foreign country. Then it feels odd when you come back home!

The loop which does the counting begins with the word DO and ends with the word LOOP. In this definition there is nothing between DO and LOOP, just as there was nothing in the BASIC loop given earlier. All the micro is being asked to do is to loop back to DO thirty thousand times.

Now to execute the word TEST. Key in the word TEST by itself on the line below. Get your stop-watch ready, then press RETURN. The 'OK' message appears as soon as the computer has counted to 30000. On the BBC Microcomputer, the test took only two seconds. For this particular operation, FORTH is about eight times faster than BASIC. Comparisons for other operations give different results, depending on what has to be done by the micro, but it seems that the claim that FORTH is up to ten times faster than BASIC is to be believed.

The speed of FORTH makes it ideal for computer games. It also has the advantage that the computer can perform a long series of calculations in a reasonably short time. There are applications in this book which take advantage of the speed of FORTH in both of these ways.

High-level languages are of two main types: *interpreted* and *compiled*. BASIC is an interpreted language. When a program in BASIC is run, the computer goes through it, line by line, working out what the various statements mean. The interpreter program in the ROM of the micro interprets each BASIC statement, calling on machine code routines to perform the necessary actions. The program is interpreted every time it is run. Moreover, if there are lines in a loop which are repeated, say 100 times, then those lines have to be interpreted 100 times each. Interpreting inevitably requires time, which means that BASIC programs run relatively slowly. But interpreters have an advantage. When you are working with an interpreted language, it is easy to stop the program, make small changes in it and then run it again. Programming is relatively quick and easy, and you can instantly see the results of any changes you make.

A compiled language, such as Pascal, has entirely different

features. After you have written the *whole* program, it is compiled into a machine code version which can be stored on tape or disk. After that you use the machine code version. The conversion of the program from a high-level language to machine code is done once and for all. The compiled version of your program runs exceedingly fast, which is a big advantage. The corresponding disadvantage is that it is not possible to make any changes in the program without starting from the beginning and recompiling the new version. This is annoying if there are bugs in the program, as there are almost certain to be at first. It is much more important to get the program right before it is compiled. Some people would say that this is a good thing, for it forces you to work carefully and plan everything in detail before you begin to program. Needless to say, a lot of other people are put off by this strict approach to programming, which is perhaps one reason why Pascal has never become popular with micro owners. Another disadvantage of compiled languages is that they generally require more memory than is present in the average micro.

FORTH is neither interpreted or compiled, in the usual sense of these words. When you *define* a FORTH word, the name of the word is stored away, together with a string of numbers which link your word to words you have used in the definition. Once a word has been defined, it becomes a part of the language. This is roughly equivalent to compiling, but it does not produce machine code. It produces a word which refers to other words already compiled. Some of these words, the *primitives*, perform the more fundamental actions. The primitives have links to routines in machine code which, in effect, will do all the real work when the word is executed.

When a word is *executed*, during the running of an application, FORTH acts more like an interpreter. As each word is executed, several other words may be called into action. Some of these call on primitives which in turn bring various machine code routines into operation. Since the words have already been 'compiled', interpretation is very fast. This gives FORTH its speed. However, since the words of FORTH are each individually 'compiled', we have an extremely flexible and accessible system. You define words one at a time, and can test each word thoroughly before you go on to write the next. If, later, a definition turns out to be unsatisfactory in some way, you can re-define it without having to re-compile the rest of the application. In this and other ways, FORTH allows the user to interact freely with it, one of the advantages of an interpreted language, yet has the speed and compactness of a compiled

language. It combines the advantages of both.

Since the core of FORTH has a relatively limited number of tasks to perform, it is short and requires little memory. The core routines, and essential word definitions of a typical implementation of FORTH require only about 8 kilobytes of memory. In any given application you need add only those words which are required by the application. This keeps memory requirements to a minimum.

The flexibility of FORTH has been mentioned already. The language is a highly structured one, yet you are allowed the freedom to alter its structure to suit your own purposes. We shall see examples of this in later chapters. It is considered by some that a language that can alter itself and extend itself so readily is not really a language at all! This flexibility gives it several advantages in the field of education. Other languages present the user with a rigid system of statements or commands, and fixed ways of doing things. This implies that the user has quite a lot to learn before beginning to use the language. If there are difficult aspects of learning the language, it is not possible to alter the language to make it easier. With FORTH we can make things much easier for the beginner or those with special difficulties. Words can be defined which do very simple and easily understood things. We can give them names which readily make sense to the learner. Words from the user's everyday vocabulary are more likely to convey meaning than words chosen by someone else. The beginner can give any preferred name to the words. One can go further. If the name of an existing FORTH word is confusing to the user, there is no difficulty in re-defining it with an entirely different name. For example, 'duplicate' is a word in English not readily understood by young people, so the FORTH word DUP may not be understood either. Perhaps MAKE-TWO would be better understood in the earlier stages of learning FORTH. To add this to the language, all that is required is:

```
: MAKE-TWO DUP ;
OK
```

From then on MAKE-TWO will have exactly the same action as DUP. In the same way it is easy to adapt the FORTH vocabulary to suit those whose mother tongue is not English.

It must be evident from the above that the writers have a high opinion of FORTH and its potentialities. Turn now to Chapter Three and begin to experience the delights of FORTH for yourself.

To summarise

In this chapter you have found out how to:

- Key in and use FORTH words.
- Define a new FORTH word, using existing FORTH words.
- Use a DO ... LOOP loop.

You have learned that:

- FORTH is fast.
- FORTH combines the advantages of an interpreted language with those of a compiled language.

Chapter Three

Stacking It Up

The stack is at the heart of all operations in FORTH. It is therefore very important to understand what the stack is and how it is used.

The stack is a part of memory specially set aside for holding numbers. It is not a very large part of memory, usually less than 100 bytes. The word 'stack' implies rather more than the related word 'heap'. In a stack, things are arranged in some kind of *order*. There are several ways of thinking about the stack. One of these is to imagine a stack of postcards in a clip-board (Fig. 3.1). Putting them on the clip-board stops the cards from getting out of order. In other words, it prevents the stack from turning into a heap!

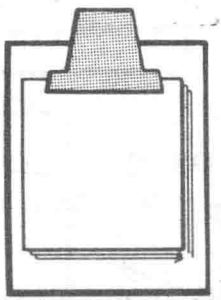

Fig. 3.1. The clip-board ready to demonstrate the stack.

When we look at the stack, we see only the top card. We will refer to this as *top-of-stack*. This particular card is only top-of-stack for as long as the stack remains unaltered. We could add another card to the stack, placing it on top of the stack. Now the original top-of-stack card is covered and the newly added card becomes the top-of-stack. Or we could remove the top-of-stack card and throw it away. Then the newly exposed card which was below it becomes the new top-of-stack.

Note that when we add a card to the stack we always place it on

top. We never try to insert it further down in the stack. Also, when we remove a card from the stack, we always remove the *top* card, never a card from further down the stack. These two rules are an essential part of the way the stack works.

Suppose we begin with an empty clip-board. This can be referred to as an 'empty stack'. It would help at this stage if you were to have an empty clip-board beside the computer. You also need about ten cards (or sheets of paper), and a pencil. If you do not have a clip-board it does not matter; just keep the windows and doors shut so there is no wind to blow the stack away! When you are ready with this equipment, turn on the computer and call up FORTH.

First write the figure 4 on one of the cards (we will refer to them as cards, even if you are really using scraps of paper). Place it in the clip-board. This card is top-of-stack. The value stored at top-of-stack is 4 (Fig. 3.2).

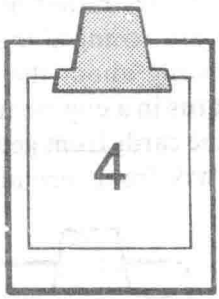

Fig. 3.2. Value 4 at top-of-stack.

When you first run FORTH, its stack is empty, just like the clip-board was. To place 4 at top-of-stack, all you have to do is type:

```
4
```

and press RETURN. You will see the 'OK' prompt on the next screen line, indicating that your instructions have been obeyed and the computer is waiting to be told what to do next.

How do we know that 4 has really been placed at top-of-stack? FORTH has a word which tells the computer to take the top number off the stack and display it. This word is the shortest word possible:

```
.
```

No, it's not a dirty mark on the page, it is a full-stop. When we refer to it, we call it 'dot'. Key in 'dot', then press RETURN. The sequence so far is shown in Fig. 3.3. As you can see, the computer has displayed 4, the number stored at the top-of-stack.

```
4
OK
.
4 OK
```

Fig. 3.3.

Can you get the computer to display it again? Try 'dot' followed by RETURN. What happens next depends on the version of FORTH you are using. On the BBC Microcomputer, for example, you get:

```
.
0 .  ? MSG # 1
```

The words MSG # 1 refer you to error message no. 1. If you look this up in the manual, you will find that it means 'Stack empty'.

On the Jupiter Ace you will get:

```
-18572ERROR 2
```

The first number may be different, but the error message will always be number 2, which means 'Stack empty'.

It seems that the 4 is no longer on the stack. It is the same as if you had taken the card from the clip-board, and pinned it up on the wall for everyone to see (Fig. 3.4). Do this now – take the card from the board and put it where everyone can see it (no need to pin it to the wall!).

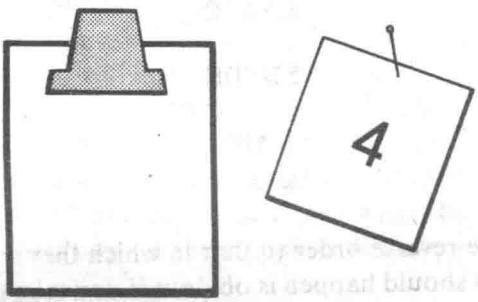

Fig. 3.4. Value 4 displayed, leaving the stack empty.

To sum up so far: we put 4 on the stack. Then we used 'dot'. The action of 'dot' was to take the number from the top-of-stack, leaving the stack empty, and display the number on the screen.

Now put the 4 card back on the board, so it is once more top-of-stack. Next write 55 on another card and place this on top of the 4 card. The 55 card is now top-of-stack. The 4 card has become second-on-stack. Finally write 666 on a third card and put this on top of the 55 card (Fig. 3.5). Remember we always add to the top of