ADVANCED SPECTRUM FORTH

Don Thomasson



Melbourne House Publishers

SPECTRUM

PART VI: COMPILING 69
<BUILDS . . . DOES> 72
READING THE DICTIONARY 73
MORE COMPILING 74

PART VII: PROGRAMMING TECHNIQUES 77
NON-INTEGER ARITHMETIC 77
A FINAL PROGRAM 79
A Three-Dimensional Noughts and Crosses Program 83

APPENDIX A: THE DICTIONARY 89

INTRODUCTION

FORTH was the brain child of a man called Charles Moore. After trying out various aspects of the concept for some years, he began to fit them together towards the end of the nineteen-sixties, though it was some time before a completely coherent version emerged. Seeing his invention as a 'fourth generation' computer language, but restricted by his equipment to five-letter names, he chose to drop the 'u' and call his concept 'FORTH'.

Since then, the language has developed in various directions, the two mainstream versions being FORTH79 and fig-FORTH. The 'fig' prefix refers to the FORTH Interest Group, of San Carlos, California, who have endeavoured to make the language accessible to as wide a range of users as possible.

That range now extends to users of a number of microcomputers, but each implementation is slightly different from the others, the original definition of fig-FORTH allowing ample scope for individual choice of facilities. This book is specifically based on Spectrum FORTH by Abersoft, which is perhaps the most complete and user-friendly version available. Much of what is said will be equally applicable to other versions of fig-FORTH, and where the other versions omit some of the words described it will be possible to extend them to match the Abersoft standard, though some facilities may be more difficult to provide.

FORTH79 is a different matter. It uses different words, and gives some words different meanings. And while the basic concept is much the same as that of fig-FORTH the implementation is quite different. However, flexibility is an inherent characteristic of both types of FORTH, and this may allow some of the discrepancies to be smoothed out by the provision of new words and revised definitions.

A major problem facing newcomers to FORTH is the size of the 'vocabulary', which is the list of predefined words. This is usually tabulated in the order of the ASCII code representation of the words, but when the vocabulary is displayed in response to the command VLIST (followed by return) the words are in a completely different order. In either

case, finding the right word to do a particular job is not easy. The approach adopted here is to group the most frequently used words according to the kind of action which they call up, so that a given kind of function can be located quickly. In addition, Appendix A provides a full definition of the standard words in the form of a compressed summary of the 'dictionary', the area of store in which the words are defined.

Nevertheless, it would be wrong to expect to be able to start writing FORTH programs immediately. The best approach is to learn the operative words in groups, experimenting with them while you get the feel of the system. Then you can start defining new words of your own with greater confidence, and later begin to define complete programs.

This progressive approach is possible because FORTH can be used at different levels. It will interpret existing single words or a series of words in 'direct' mode, which may be compared with direct mode in BASIC. On the other hand, it will compile new words, creating new dictionary entries for them, and these words can thereafter be used in compiling further definitions. Ultimately, a single word will call up a complete program.

The dictionary entries are formed in such a way that the routine associated with each word can be found very quickly. Whereas a BASIC interpreter has to scan through link tables to find the required entry point, FORTH stores the entry point directly, and no scanning is required. This makes FORTH a good deal faster than BASIC, which is perhaps its principal asset. Another asset is economy of store usage.

To enjoy these advantages, you must be prepared to help the language along, by doing things that would be done for you automatically if you were using BASIC. You must think a little more, keeping a watchful eye on what the program is doing. In return, FORTH will give you a flexibility limited only by the scope of your imagination.

Getting Started

The Abersoft tape takes about 70 seconds to load; and this header should appear on the screen;

48K SPECTRUM fig-FORTH (version)
© Abersoft: 1983

This tells you that your Spectrum has become a FORTH machine.

The first sign of the change is that the flashing cursor shows a letter 'C', rather than 'L'. As most of the standard FORTH words are in upper case lettering, it is more convenient to work in 'capitals mode', though you can always select 'L' mode in the usual way, using CAPS SHIFT and 2.

You will notice that the little beep indicating key depression has disappeared. If it returns, you have dropped out of FORTH into BASIC, for some reason, but you can return to FORTH by GOTO3. You can get into BASIC deliberately by typing MON and ENTER, but you will find that only direct execution is possible, as there is no room to add to the short BASIC program already established.

The keys will no longer produce BASIC words, which are not needed by FORTH. You will have to type everything letter by letter. There is no Extended Mode, but you will find that some symbols normally obtained in that mode, plus a few new ones, can be produced by using SYMBOL SHIFT and certain other keys, e.g:

Key 'Y' gives [
Key 'U' gives]
Key 'A' gives ~
Key 'S' gives |
Key 'D' gives \
Key 'F' gives {
Key 'G' gives }

Graphics are accessible in the usual way.

Once you get out of the habit of using key 'K' for LIST, which is a FORTH word, you will find no serious problems.

A number of familiar functions are retained, as detailed in the section on 'Spectrum Specials', but you will find that they are not quite the same as the BASIC forms. For example, the parameters must come before the command, not after.

An important difference concerns BREAK. In some circumstances, pressing CAPS SHIFT and SPACE together will still stop a program, but an alternative is provided by CAPS SHIFT and 1. This, however, will only work if you have written your program to respond to this input. If you make no provision for BREAK, you may find that the program is stuck in an infinite loop, with no way out. Details of the way to avoid this are given in the section on branching and looping, but you should have no need to worry about them for the time being.

If you decide to take a different kind of break, by loading another sort of program, you will need to reset RAMTOP, otherwise you will get an Out of Memory report. The easiest procedure is to switch the computer off and on again, so that the usual parameters are initialised.

If you are interested in such matters, the initial area occupied by the FORTH program after loading is 5E06 to 8159, with reserved workspace from CB40 upwards. The intervening space is available for your FORTH words, which take the form of extensions of the 'dictionary' that is the fundamental definition of FORTH. The dictionary grows upwards, and the calculation stack grows downwards. If you want to know how much space is left between them, input:

FREE . ENTER

This will display the number of free bytes.

A ZX Printer will work with FORTH. Switch it on by:

1 LINK ENTER

Switch it off by:

0 LINK ENTER

Because FORTH is different in a number of ways from most other languages, the next section will examine its general characteristics. If you already know what they are, you can skip forward with confidence.

The Essentials of FORTH

Used in direct mode, FORTH works like a calculator operating in Reverse Polish Notation. The data which you key in is shown on the screen and stored in an 80-character input buffer, which will hold 2½ lines of displayed data. When the buffer is full, or if you press ENTER to indicate that an input is complete, the system begins to scan the buffer, stopping at the first space character which it finds. The data which has been scanned is then compared with the list of words in the dictionary, which can be seen by:

VLIST ENTER

You can stop the listing by pressing BREAK (CAPS SHIFT/1), as the full list occupies more than the screen area.

If the data matches a defined FORTH word, the action associated with that word is performed. Otherwise, the data is checked to see if it is a valid number. If it is, the number is stored on the calculation stack. If it is neither a FORTH word nor a number, error report 0 is displayed, with the offending word.

Because Reverse Polish Notation is used, with data stored on a 'stack', the FORTH words act on the last numbers to be input. Try this input:

```
4 5 * .
```

Leave a space between each element of the input, remembering that "space" is the standard "delimiter" marking the start and end of each word or number. The response will be to display the number 20. The way this works is that FORTH takes the data up to the first space, in this case the number 4. Because it is a number and not a FORTH word it is stored on the stack. The procedure is repeated for the number 5, and this is stored on the stack on top of 4 . Then FORTH comes across the "*". Because it is a FORTH word and we are in direct mode it is executed. FORTH takes the last number (5) from the stack and the next one down (4) and multiplies them together, putting the result back on the stack. The next FORTH word in the buffer is the dot "." This displays the result.

If we expand on the above and input the following:

```
4 5 * 23 + .
```

Up to the "*" everything is the same as above with the result 20 on the stack. FORTH continues to read the line and stores 23 on the stack on top of 20, and goes on to read the next word. This (+) is a FORTH word and it adds the top two numbers on the stack, 23 and 20, and stores the result, 43, back on the stack. The 'dot' once again displays the result.

The action of the FORTH words in these inputs has been explained in a relatively simple manner that should suffice for the moment, but it will be evident that exact definitions are needed, and these will be supplied later on.

There is no need to enter a complete set of numbers and words on one line. Each could go on a separate line, followed by ENTER, since the input system deals with only one element at a time. If you enter too many characters, overfilling the buffer, you will probably get an error message, because the last word entered may not be complete, but the earlier elements of the line will be processed normally, and as the error report identifies the word to which it refers you will know where to continue the entry.

Not all the words displayed by VLIST can be used in direct mode, and an attempt to use them will produce error 17. These words can only be used in compiling mode, which sets up new dictionary entries defining fresh FORTH words.

Here is an example:

```
: TEST1 * + . ;
```

The colon instructs the system that all input words and numbers up to the subsequent semicolon are to be compiled, the result being identified with the word name TEST1 .

Now input the following:

```
23 4 5 TEST1
```

The result is 43, as in the earlier example. TEST1 performs the multiplication, the addition, and the display function which we previously input as separate words. It would be a good idea to simulate the stack on paper to ensure you get your values and operators in the correct order. If you do this for the above you will see why 23 comes first.

You could define another word:

```
: TEST2 23 4 5 TEST1 ;
```

An input of TEST 2 would display 43. That would be rather pointless, but it illustrates how a FORTH hierarchy is built up. In the end, a complete program is called by input of a single word. That word defines a list of other words to be executed in turn, and some or all of the words in the list may define further lists.

It is interesting to compare this with the action of BASIC, which uses numbered lines containing statements which specify the action to be taken. The numbering is used to indicate the required order in which the statements are to be executed. A well-structured BASIC program, however, can be seen as a number of functional blocks, each composed of a number of lines, though some programs cram so many statements into a line that a single line may represent a block. This makes the program difficult to interpret from the listing.

FORTH, on the other hand, gives each functional block a name, and that name is linked with a name list defining the functions to be carried out by the block. Groups of blocks are similarly linked by further names and lists, until a single name and list brings all the groups together in a single program called by a single name.

Clearly, this is not a process which can be approached too casually. Because a BASIC program can be modified by editing, it is sometimes possible to create a fairly simple routine as you go, correcting errors as they become obvious. With FORTH, it is advisable to plan ahead, deciding on the content of each functional block in advance. This can impose an irksome discipline if you prefer the more casual approach permitted by BASIC, but the speed of execution of the result makes the effort worthwhile.

An important problem for anyone approaching FORTH for the first time is the sheer number of words which are available for use. The only solution is to learn the vocabulary in sections, gradually expanding your understanding of the range of actions which you can handle. That is the basis on which the remainder of this book is planned. If the detailed explanations of some words seem too complex, leave them and come back later, accepting that the words act as described. You will find that you can very soon begin to understand what is happening, even in the most complex functions.

Another problem for newcomers to computing stems from the fact that all numeric data is stored in binary form. Not so many years ago, binary was considered incomprehensible for anyone but a computer expert. After a lecture on the subject, a man got up, scrawled a series of ones and noughts on the blackboard, saying that no one could make sense of that. As he put in the last nought, a small voice from the audience said 'three hundred and eighty four'. The man stared, and laboriously checked the value of his 'incomprehensible' binary number, to find the translation was correct. The small boy who had made the translation looked surprised, and explained that all he had done was to take the first digit, and when another digit was added he doubled what he had and added the new digit. For example:

1	1
1	3
0	6
0	12
0	24
0	48
0	96
0	192
0	384

The number on the blackboard was 110000000, not the most difficult of numbers to translate, but the principle is useful to remember; and it applies to any number base.

For example, in base 3, the number 112020 translates thus:

1	1
1	4
2	14
0	42
2	128
0	384

FORTH performs this process for any number base you like to select, not only for input, but also for output. This can lead to confusion if you fail to appreciate what is happening, so the study of the vocabulary will begin by looking at the way numbers are input, stored, and output.

PART I:
Direct Mode

This part deals with FORTH words which can be input for immediate execution. Some versions of BASIC can be used in this way, in what is then called 'direct' or 'calculator' mode. In a later section the use of compiling to create new words will be examined, and that will prove more convenient for complex operations, but though it can be slow and tedious to work in direct mode it gives a better insight into exactly what is happening.

NUMBER REPRESENTATION

The key to the operation of FORTH is the 'stack'. For those who are unfamiliar with the term, a stack is the 'last-in-first-out' buffer store which can be compared with the old-fashioned filing spike. If you put bills on to the spike as they arrive, the first to come off will be the one which was put on last. FORTH uses two stacks, one for calculation and the other for holding link addresses. They will be called the Calculator Stack (or often just Stack by itself) and the Return Stack, respectively.

In Spectrum FORTH, the Z80 stack is used for the Calculator Stack. The Z80 stack handles data in the form of 16-bit words, even where nothing more than a single ASCII code is involved. That suits FORTH perfectly. Computers using other processors may have stacks that hold 8-bit words, but they have to store two such words for each transfer to the stack. In general, except for some specialised functions, FORTH is unaltered by machine characteristics.

Another point is that FORTH recognises no distinction between data types, which can have disconcerting consequences on occasion. The meaning of data words depends on the way they are treated. For example, type 40000 followed by ENTER. Then type a "dot" (full stop), again followed by ENTER. The dot displays the number from the top of the stack, which is of course the last number stored there. The number displayed is −25536 and not 40000. Where did that come from?

Signed and Unsigned Numbers

A 16-bit binary number can have 65536 different values. In 'pure binary', these values range from 0 to 65535, bit n having the value 2^n, where the least significant bit is bit 0. If, on the other hand, we adopt '2's complement' representation, the value allocated to the most significant bit is -2^{15} instead of 2^{15}. The rest of the bits can represent values from 0 to 32767. If bit 15 is 0, that is the range of the complete number. If bit 15 is 1, then 32768 (2^{15}) is subtracted, making the range −32768 to −1. Since the number is always positive when bit 15 = 0 and negative when bit 15 = 1, the bit is commonly called the 'sign' bit.

A mild word of warning is needed here regarding the number −32768. It behaves a little oddly, being unchanged by negation. In this respect, it can be compared with zero. It is best avoided, the working range being taken as ± 32767.

FORTH allows you to choose between pure binary and 2's complement representation at will. The simple 'dot' (as the full stop character is known in FORTH circles), will display a number on the basis that it is in 2's complement form, while U. will assume that the number is in pure binary. Since the number 40000 is outside the 2's complement range, it was stored as pure binary. Using dot to display it gave the interpretation of 2's complement. Both 40000 and −25536 are represented by the binary number 1001110001000000. Note that they add up to 65536.

Double Numbers

The value range for 16-bit numbers is rather limited, so FORTH provides for the use of 32-bit numbers, which are held as two successive stack items, the upper 16 bits being nearest to the top of the stack. 4,294,867,206 different values can be represented, and 2's complement representation is usual, giving a range of ±2,147,483,647. The anomalous number in this case is −2,147,483,648.

You can enter a double number from the keyboard by including a decimal point somewhere in the number. The position of the point is not directly relevant, but the number of digits to the right of it is noted in the variable DPL for possible future reference.

A point to watch is that a number outside the ±32767 range, entered without a decimal point, can cause problems. The input routine always interprets a numeric input as a double number, but if there is no decimal point the upper half of the result is thrown away. This can cause some mystifying results on occasion.

A double number can be displayed by the word D. , which assumes that the number is double and in 2's complement form.

Number Base

By default, FORTH accepts input numbers and generates output numbers on the basis that decimal notation is intended, converting the

numbers to and from the internal binary representation. If so instructed, however, it will work in any number base you desire, within reason. The word HEX will call up hexadecimal notation, the word DECIMAL will restore decimal working. That, however, is only the beginning.

The word ! instructs FORTH to store a number on the second stack position in a location defined by the top of stack. The phrase n BASE ! will therefore store the number n in the variable BASE, which determines the number base to be used in input and output. HEX is equivalent to 16 BASE ! , and DECIMAL is equivalent to 10 BASE ! , while binary working is obtained by 2 BASE ! . These statements assume that decimal working is effective. If binary has been selected, 10 BASE ! would do nothing, since in binary 10 has the value two! You need 1010 BASE ! .

For a digit value greater than 9, the letters A-Z provide a means of extending the digit range, so it would be possible to work in base 35. Beyond that some odd characters get into the act.

The word @ instructs FORTH to obtain the contents of the location defined by the top of stack and put the result on the stack, so you might think that the current base could be determined by BASE @ . , but the result is always 10. Can you see why? There is a hint above.

When using hexadecimal, the unsigned 'pure binary' convention is to be preferred, as negative hexadecimal numbers are confusing.

Formatted Numbers

One of the virtues of FORTH is an ability to display numbers in a very precise format. The simplest examples involve the words .R for single signed numbers, D.R for double numbers, and U.R for unsigned single numbers.

These words take the number on top of the stack as defining a 'field' of that number of character positions. The next number on the stack is displayed in the right hand end of the field. Whereas a normal numeric output adds a space after the number, these functions do not. It is therefore relatively easy to build up neat tabulations with the numbers 'right justified', that is with their right-hand digits in a vertical line. Other facilities of this kind will emerge in due course.

Meanwhile, we have made a start by defining ten FORTH words and phrases.

For convenience, they may be summarised as follows:

TOS means 'top of stack', 2OS means 'second on stack' and so on.

.	Remove TOS and display it as a signed single number.
U.	Remove TOS and display it as an unsigned single number.
D.	Remove TOS,2OS and display as a signed double number.

n BASE !	Set base n notation
HEX	Set hexadecimal notation
DECIMAL	Set decimal notation
BASE @ .	Read current base
.R	Remove TOS,2OS. Display 2OS as a single signed number at the right hand end of a field of TOS characters.
U.R	Remove TOS,2OS. Display 2OS as a single unsigned number at the right hand end of a field of TOS characters.
D.R	Remove TOS,2OS,3OS. Display 2OS,3OS as a double signed number at the right hand end of a field of TOS characters.

ARITHMETIC PRIMITIVES

The range of FORTH words is built up on a foundation of a number of blocks of machine code. Other words are created by combining or modifying these 'primitives', but the action of these other words can only be understood if the primitives are examined first.

You may be surprised to find that there are only eight arithmetic primitives. Later, we will see that they form the basis for another seventeen derivatives.

It is important to remember that FORTH words act on numbers or other data which is already on the stack. To add 7 and 8, we need:

```
7 8 +
```

Examining what is happening, we begin by typing the characters and spaces, which are stored in the Terminal Input Buffer. The INTERPRET routine takes the entries delimited by spaces in turn. It sees 7 as a valid number, having first considered it as a possible FORTH word, and the numeral 7 goes on the stack. 8 is treated similarly, and it also goes onto the stack. The numeral 8 is 'TOS' and the numeral 7 '2OS'. The plus sign, however, is recognised as a FORTH word, and it is executed. It takes the two top stack items, adds them together, and puts the result back on to the stack. We could now display the result by using 'dot' or 'Udot'.

Incidentally, there is no need to input all three characters on the same line. The effect would be the same if we input:

```
7
8
+
```

This is because FORTH operates on each word or number in turn. Pressing ENTER merely invites INTERPRET to examine what has been

put in the Terminal Input Buffer (TIB for short). If you want to know where the TIB is held, HEX TIB @ U. will tell you. All part of the FORTH service!

A minus sign is also a FORTH word. It removes the two top items from the stack, subtracts TOS from 2OS, and puts the result back as TOS.

This kind of calculation is known as Reverse Polish Notation, and as it has been used in some calculators it may not be completely unfamiliar. It avoids ambiguity, since the operators act in a clearly defined way on clearly defined numbers. Where normal arithmetic notation needs brackets and priority rules to determine its interpretation, Reverse Polish leaves no room for doubt.

On the other hand, it does require that the right numbers are in a position to be operated on at the right time, and this can call for some careful thought and advance planning. A browse through the dictionary definitions in Appendix A will reveal some interesting examples.

The next 'primitive' is U*, which takes away TOS and 2OS and puts their product on the stack as a double number. Then we have U/MOD , which takes away the three top stack items, treating the second and third as a double number to be divided by the first item. The remainder is put on the stack as a single number, and the quotient, again as a single number, is put on top of it.

Putting the remainder on the stack may seem pointless, but it is there for a very good reason. Suppose you have a double number on the stack representing a number of seconds. Adding 60 U/MOD will divide the number by 60, and the quotient will give the number of minutes, the remainder the number of remaining seconds. Try:

```
200. 60 U/MOD . .
```

The decimal point after 200 is needed to set it up as a double number for U/MOD to work on. We can use 'dot' for the output, because the result will be less than 32767, and there will be no confusion over sign. Now try:

```
36484. 60 U/MOD 0 60 U/MOD . . .
```

The zero entry is needed to make the single word result of the first U/MOD into a double word for the second U/MOD to work on. The three figures represent hours, minutes and seconds.

This could be regarded as a calculation to base 60, and the routine for decimal output works on a similar sort of basis to that shown above, except that 60 is replaced by 10.

There are times when a single number needs to be converted to a double number form, perhaps to allow U/MOD or a similar operator to be used. If U/MOD is applied to a single number, it will pick up whatever happens to be next on the stack and work on that. Once again, there can be strange results.

If the single number is unsigned, we can extend it by putting a zero on the stack to form the upper half of the double number. If the single number is signed, and we want to preserve the sign, we must use

S—>D, which puts a zero on the stack if the single number is positive, or FFFFH if the single number is negative. This is called 'propagating the sign bit'.

To add two double numbers together, D+ is required. It removes four items from the stack, treating them as two double numbers, adds them together and puts the result on the stack as a double number.

The last two primitives perform negation, MINUS acting for single numbers and DMINUS for double numbers. The action is to subtract the number from zero. To all intents and purposes, the number seems to stay in its place on the stack, though in fact it is removed and then put back.

These eight primitives may seem rather inadequate, but before we examine the seventeen derivatives we need to consider the stack manipulation operators, without which FORTH would be very restricted indeed.

The operators described in this section may be summarised:

+	Remove TOS,2OS. Place their sum on the stack.
—	Remove TOS,2OS. Place 2OS-TOS on the stack.
U*	Remove TOS,2OS. Place their product on the stack as a double number.
U/MOD	Remove TOS,2OS,3OS. Treating 2OS,3OS as a double number, divide it by TOS. Place the remainder on the stack, then the quotient. All numbers are treated as unsigned.
D+	Remove TOS,2OS,3OS,4OS. Treating them as two double numbers, put their double number sum on the stack.
S—>D	Sign-extend a single number to form a double number. The original number becomes 2OS, the extension TOS.
MINUS	Negate a single number on TOS.
DMINUS	Negate a double number on TOS,2OS.

STACK MANIPULATORS

It will be evident that mathematical and other operations will call for a certain amount of stack shuffling in order to bring the required data to the right position. This is relatively easy when the number of items on the stack is small, but a routine whch will be described later on involves up to 72 items on the stack at the same time, and that is a more difficult proposition.

One of the hardest-worked words in FORTH is DUP, which duplicates the TOS by adding a copy of the original TOS. This is essential when the value of TOS is being checked, as the check would destroy the number if it were not duplicated. A variant is —DUP, which only duplicates if TOS is

non-zero. This is useful where TOS is no longer needed when it reaches zero. There is also 2DUP, which duplicates TOS and 2OS. This is useful for duplicating double numbers, but it can also be used with single numbers. Suppose we want to calculate (4 + 3)*3 + 4. We can use:

	Stack
4 3	4 3
2DUP	4 3 4 3
+	4 3 7
U*	4 21 0
DROP	4 21
+	25

U* generates a double number, so we discard the upper 16-bit entry by DROP, which discards TOS. 2DROP discards TOS and 2OS.

Next we have SWAP, which interchanges TOS and 2OS, and 2SWAP which interchanges TOS/2OS with 3OS/4OS.

OVER is very useful. It adds a new TOS which is a copy of the previous 2OS. 2OVER adds a new TOS and 2OS which are copies of the previous 3OS and 4OS.

Finally, in this group, ROT rotates the top three stack items, bringing 3OS to TOS, 2OS and TOS to 2OS.

When you begin to write full-scale programs, you will find that it is very useful — if not essential — to write down tables like the one above, so that you can keep track of the stack movements.

By so doing, you will soon realise that there is no way to ring the changes to bring deeply-buried items to the top of the stack, but you will find that this problem is eased when we come to the process of compiling new word definitions.

Because of this limitation, it is often convenient to introduce some numbers in the course of a computation. These may be included in the input stream, or they may — as will appear in due course — be called up from constants of variables. Since that uses extra storage space, it is regarded as uneconomical by FORTH purists, but there is no point in striving too hard for perfection.

The manipulators which have been mentioned may be defined as follows:

	Stack Before	**Stack After**
DUP	a	a a
2DUP	a b	a b a b
DROP	a	
2DROP	a b	
SWAP	a b	b a
OVER	a b	a b a
2OVER	a b c d	a b c d a b
ROT	a b c	b c a

Remember that the top of the stack is the right-hand entry. Only the relevant entries are shown. There may be other items 'behind' these, i.e. deeper into the stack and to the left of those shown. These are unaltered by the manipulators.

OTHER MANIPULATORS

At this point, it is necessary to mention some stack and display manipulators which do not fall conveniently under other headings.

First there are three which are not permissible in direct mode, but which are used in producing some of the arithmetic derivatives. Reference has been made in passing to the Return Stack. It is sometimes convenient to remove the top of the calculator stack from immediate action, perhaps to allow access to items further down the stack. These words facilitate this by allowing the top of the calculator stack to be transferred to the Return stack on a temporary basis.

- >R removes TOS and transfers the data to TORS (Top of Return Stack).
- R> performs the opposite function.
- R copies the TORS to TOS, without altering TORS.

These words must be used with care, the Return stack being restored to its original state before a definition is completed, or reference is made to the Return stack contents for other purposes, as in control of loops. (A slip of this sort occurred in the definition of 2OVER in early Abersoft tapes. If you find that HEX 7F4E @ U. gives 6173, you have the bug. It can be cured by HEX 6188 DUP 7F4E ! 7F50!)

The screen manipulators are fairly obvious. CR calls for newline, and CLS clears screen. SPACE outputs a space, while n SPACES outputs n spaces.

In direct mode, ." *string*" outputs the string of characters between the quote characters. It corresponds to the BASIC PRINT "XXXXX".

MORE ARITHMETIC

Armed with the stack manipulators, we can now find out how the extra seventeen operators are created. However, we must first take a brief look at the logic operators, which are also involved.

AND takes TOS and 2OS and compares them bit by bit. Where both have a given bit in a true state, the corresponding bit in the result is set true. Otherwise the bit is set false. The result is put on to the stack.

OR works in a similar way, but puts a true bit in the result where either TOS or 2OS have a true bit in that position.

XOR also works in a similar way, but puts a true bit into the result if the corresponding bits in TOS and 2OS are different. An interesting use is to obtain a result with sign bit set if the signs of TOS and 2OS are different.

Some of the derived arithmetic operators are quite simple. 1+ is equivalent to 1 + , and it adds 1 to TOS. Simlarly, 2+ is equivalent to 2 + , and it adds 2 to TOS.

Next, we have +−, which removes TOS and negates the new TOS if the original TOS was negative. This is implemented by calling MINUS if TOS was less than zero. The double number version is D+−, which removes TOS and negates the double number in 2OS,3OS if TOS was negative.

MIN and MAX come next. They remove TOS and 2OS and discard one or the other, restoring the survivor to TOS. MIN restores whichever was less, MAX restores whichever was greater. In both cases 2DUP creates copies of TOS and 2OS to use as a basis for comparison, and then drops one of the original entries.

Finally, at the first derivation level, we have M/MOD, which resembles U/MOD , except that it leaves a double number result instead of a single number result. This is convenient when working with large numbers. The word is constructed as follows:

	Stack	
	a b c	
>R	a b	c to TORS
0	a b 0	make b a double number
R	a b 0 c	recover c
U/MOD	a d e	divide b by c quotient e. remainder d
R<	a d e c	recover c, clear TORS
SWAP	a d c e	
>R	a d c	e to TORS
U/MOD	f g	a/d divided by c. quotient g. remainder f
R>	f g e	Recover e. TORS clear

Two divisions are performed. The first is 'scaled' by a factor of 65536, because b is really the upper half of a double number. The quotient and remainder are similarly scaled, so it is possible to 'concatenate' a and d to form a valid double number, which is again divided by c. The two quotients g and e can then be 'concatenated' as a double number.

At the second level of derivation, we have ABS and DABS. ABS is implemented by DUP +− . TOS is duplicated, and the copy is removed. If it is negative, the original TOS is negated. Whatever happens, TOS emerges as positive, the absolute value being preserved. DABS similarly uses 2DUP D+− to perform the same process on a double number.

We now come to M/ and M* . M* generates a double number product of two signed single numbers. It is based on U* , with the additions needed to preserve signs.

	Stack	
	a b	
2DUP	a b a b	
XOR	a b c	c is positive if a and b are of the same sign
>R	a b	c to TORS
ABS	a b	b made absolute
SWAP	b a	
ABS	b a	a made absolute
U*	d e	product
R>	d e c	c from TORS. TORS clear.
D+−	d e	negate d/e if c negative.

The XOR function is used to set up a flag indicating the required sign of the result, and this is held on the return stack while the calculation is performed.

M/ is a little more complicated:

	Stack	
	a b c	
OVER	a b c b	
>R	a b c	b to TORS
>R	a b	c to TORS
DABS	a b	double number a b made positive
R	a b c	c copied from TORS
ABS	a b c	and made positive.
U/MOD	d e	a/b divided by c. quotient e, remainder d.
R>	d e c	c from TORS. TORS = b
R	d e c b	b copied from TORS
XOR	d e f	f positive if b and c have same sign.
+−	d e	negate e if f negative
SWAP	e d	
R>	e d b	b from TORS. TORS clear
+−	e d	negate d if b negative.
SWAP	d e	remainder d, quotient e.

The numbers a/b and c, though not the copies of them on the return stack, are made positive, and U/MOD calculates the absolute values of remainder and quotient. The quotient is negated if the signs of b and c differ. The remainder is given the sign of b.

At the next level we have simpler derivatives. */MOD combines multiplication and division. It removes three items from the stack, all as single numbers. 2OS and 3OS are multiplied together, and the double number product is divided by TOS. Signed numbers are assumed throughout.

	Stack	
	a b c	
>R	a b	c to TORS
M*	d e	double number product
R>	d e c	c from TORS. TORS clear
M/	f g	f remainder, g quotient.

/MOD removes TOS and 2OS. Treating them as single numbers, it divides 2OS by TOS and puts the remainder, then the quotient, on the stack. In essence, it is M/ adapted to operate on a single number:

	Stack	
	a b	
>R	a	b to TORS
S—>D	a c	sign extend a to double number form
R>	a c b	b recovered from TORS. TORS clear
M/	d e	a/c divided by b, remainder d, quotient e.

Now, at last, we come to the simple multiplying operator * . It is implemented by M* DROP, the upper half of the double number result being discarded. Similarly, the simple division operator / is implemented by /MODE SWAP DROP, which discards the remainder.

Finally, we come to */ and MOD . The first is implemented by */MOD SWAP DROP , the remainder being discarded. The second is implemented by /MOD DROP , which removes the quotient.

These operators have been examined in detail because the simple definitions can sometimes seem ambiguous, and also because the detailed definitions show how FORTH words can be built up, pyramid fashion, to perform complex actions.

Because the compiled definitions give explicit link addresses for each function which they call, the required functions can be found very quickly, and even a complex pyramid structure of many layers can be executed rapidly, though only the 'primitives' actually perform the necessary work.

In this section, we have examined:

AND	Remove TOS,2OS. Put on the stack a 16-bit AND of the two words.
OR	As AND, but an OR function
XOR	As AND, but an exclusive OR function
1+	Add 1 to TOS

2+	Add 2 to TOS
+−	Remove TOS. If it is negative, negate new TOS
D+−	Remove TOS,2OS. If original TOS is negative, negate new TOS/2OS.
MIN	Drop TOS or 2OS, whichever is larger
MAX	Drop TOS or 2OS, whichever is smaller
M/MOD	Remove TOS,2OS,3OS. Divide 2OS/3OS (as unsigned) by TOS. Put the remainder, then the quotient, on the stack as single signed numbers.
ABS	Give TOS a positive sign if necessary, by negation
DABS	Give TOS/2OS a positive sign if necessary, by negation.
M/	Remove TOS,2OS,3OS. Divide 2OS/3OS (as signed) by TOS. Put the remainder on the stack as a single number with the sign of the dividend, then the quotient with the sign appropriate to the signs of dividend and divisor.
M*	Remove TOS,2OS. Put their product on the stack as a signed double number.
*/MOD	Remove TOS,2OS,3OS. Multiply 2OS by 3OS to form a double number product. Divide this by TOS. All numbers are signed. Put the remainder, then the quotient, on the stack as signed numbers.
/MOD	Remove TOS,2OS. Divide 2OS by TOS and put the remainder on the stack with the sign of the dividend, then the quotient as a single signed number.
*	Remove TOS,2OS. Treating them as single signed numbers, put their signed single number product on the stack.
/	Remove TOS,2OS. Divide 2OS by TOS and put the quotient on the stack as a signed number.
*/	Remove TOS,2OS,3OS. Multiply 2OS by 3OS and divide the double number result by TOS. Put the quotient on the stack.
MOD	Remove TOS,2OS. Divide 2OS by TOS and put the remainder on the stack.

MEMORY ACCESS

Access to memory depends on seven primitives allowing access to bytes, words and double words. If these are used normally, their detailed

action is not important, but if they are used in special ways their implementation can become significant. The primitives are:

C!	Remove TOS,2OS. TOS defines the address of a byte location. The lower byte of 2OS is set in that location.
!	Remove TOS,2OS. TOS defines the address of the first of two consecutive byte locations. The lower byte of 2OS is stored in the first location, the upper byte in the second. These locations are not machine dependent.
2!	Remove TOS,2OS,3OS. TOS defines the address of the first of four consecutive byte locations. The lower byte of 2OS is stored in the first location, the upper byte in the second. The lower byte of 3OS is stored in the third location, the upper byte in the fourth. These locations are not machine dependent.
C@	Remove TOS. TOS defines the address of a byte location, the contents of which are put on the stack. (Upper byte 0)
@	Remove TOS. TOS defines the address of the first of two consecutive byte locations. A word is formed from the contents of the first location as the lower byte and the contents of the second location as the upper byte, and the word is put on the stack.
2@	Remove TOS. TOS defines the address of the first of four consecutive byte locations. Two words are put on the stack. The lower byte of the second word comes from the first location, the upper byte from the second. The lower byte of the first word (which becomes 2OS) comes from the third location, the upper byte from the fourth. These locations are not machine dependent.
+!	Remove TOS,2OS. TOS defines the address of the first of two byte locations. Preserving the conventions set up by ! and @ , the contents of 2OS are added to the contents of the two locations.

? is not a primitive (= @ .) but it fits conveniently here. It prints the contents of a location defined by TOS.

These words may be used with explicit addresses, but it is usually more convenient to use a variable name. Some variables, such as BASE, are defined from the start. Others will be mentioned as they arise.

Calling a variable name puts the asociated address on the stack, so n BASE ! will put the value n into the BASE variable, while BASE @ will put the contents of BASE on the stack. New variables are set up by:

Y VARIABLE name

This sets up a 16-bit location pair, the address of which will be put on the stack in response to name . The variable is given the initial value Y.

For example:

10 VARIABLE PILE

will set up a variable with an initial value of 10 (interpreted according to the current value of BASE) and the variable can be read by PILE @ or reset by PILE ! .

For double number variables, the format is:

YY 2VARIABLE name

Four bytes locations are reserved and set to the double number YY.

There is no standard provision for setting up byte variables, but extended store areas can be set up by using ALLOT. When a new word is set up in the 'dictionary', the necessary entries are made by reference to a pointer DP . ALLOT moves the pointer forward, extending the space available. For example:

X VARIABLE name 6 ALLOT

will set up a two-byte variable, with the necessary arrangement for creating access to it. The dictionary pointer will then move forward six locations, which are thereby reserved for extensions of the variable data. The locations are not cleared. The eight bytes so reserved could be used for a four-word array which could be accessed by:

n name SWAP DUP + +

SWAP brings n to TOS , where it is doubled, the result being added to the address set up by name . The final result will be the address of the lower byte of the nth element of the array. Adding @ would read the contents of the element. It would be equally possible to use the reserved space to hold eight bytes, when the address of the nth byte would be obtained by:

n name +

The correct read and write words for the chosen memory size must always be used, this being an example of the way the flexibility of FORTH places demands on the user.

Constants can also be established. These behave rather differently from variables, in that calling the constant name puts the value of the constant on the stack, not the address of the constant. Since there is no question of writing to constants, who needs to know where they are? The defining formats for single and double number constants are:

X CONSTANT name

XX 2CONSTANT name

One use for a constant is to define an address outside FORTH. The Spectrum operating system defines three bytes called FRAMES, which count in fiftieths of a second to indicate time elapsed since switch-on, with provision for setting the overall contents of the locations to any desired value. The three bytes can be read in combination by:

23672 @ 23674 C@

This sets up a double number.

Rather than relying on memory of the right addresses, it is possible to define the first address as a constant:

23672 CONSTANT FRAMES

The locations can then be read by:

FRAMES DUP @ SWAP 2+ C@

The stack action is:

FRAMES	23672		
DUP	23672	23672	
@	23672	(23672)	Contents of location 23672
SWAP	(23672)	23672	
2+	(23672)	23674	
C@	(23672)	(23674)	Contents of location 23674

This leaves the double number on the stack which is the current value of the Spectrum system variable FRAMES. It can be displayed by using 'Ddot'.

Note how the addresses disappear as they are used, leaving the required result unencumbered.

Access to operating systems outside FORTH can provide useful extensions, but caution is necessary. In the case of the Spectrum FRAMES variable, for example, incorrect values can be read. The variable is updated every 20 mS, and after every 20 minutes 50.72 seconds the lower two bytes reach 65535. At the next update, they revert to zero, and the third byte is incremented. If this happens to occur between the time the lower bytes are read and the time the upper byte is read, a major error will result.

This can be checked by calling FRAMES2 again immediately, when the result should not differ from the previous result by more than one.

However, while bearing this in mind, you may care to work out how to use M/MOD and U/MOD to display elapsed time in hours, minutes and seconds.

In addition to the words used to access particular locations or groups of locations, there are words that deal with wider areas.

FILL removes TOS, 2OS and 3OS. Starting at a location defined by 3OS, 2OS bytes are filled with the code defined in the lower byte of TOS. For example, if a block of store has been defined by:

0 VARIABLE ARRAY n ALLOT

the locations in the block can be set to zero by:

ARRAY n 2+ 0 FILL

ARRAY sets up the start address in 3OS, 2OS is set to n+2, the number of bytes (including the two set up by VARIABLE), and 0 defines the code to be set in each location.

ERASE is effectively 0 FILL , allowing the above to become:

ARRAY n 2+ERASE

Similarly, BLANKS is effectively 32 FILL and fills the area with 'space' codes.

CMOVE takes away TOS as a count, 2OS as a destination address, and 3OS as a source address. It copies the number of bytes defined by TOS from an area running upwards from the source address into an area running up from the destination areas.It should not be used if the source and destination areas overlap, with source below destination, since the source would be corrupted before it was copied.

TOGGLE may be included in this group of words. It removes TOS and 2OS, using 2OS to define a byte location. The contents of the location are XORed with the lower byte of TOS, changing the state of any bit which is true in TOS. Originally created to serve a specific purpose, TOGGLE can be useful in other contexts.

In the Spectrum, for example:

23697 2 TOGGLE

will reverse bit 1 of PFLAG and switch OVER on and off.

Summing up the words which have been discussed:

C!	Remove TOS,2OS. Store low byte of 2OS at TOS
!	Remove TOS,2OS. Store 2OS at TOS
2!	Remove TOS,2OS,3OS. Store 2OS/3OS at TOS.
C@	Remove TOS. Replace by contents of byte location TOS.
@	Remove TOS. Replace by contents of word location TOS.
2@	Remove TOS. Replace by contents of double word location TOS.
?	Remove TOS. Output the contents of word location TOS as a single signed number.
X VARIABLE name.	Set up a two-byte variable with contents X, with location address linked to name.
XX 2VARIABLE name	Set up a four-byte variable with contents XX, with location address linked to name.
ALLOT	Remove TOS. Reserve next TOS bytes in dictionary.
X CONSTANT name	Set up a constant with value X, linked to name.
XX 2CONSTANT name	Set up a double word constant with value value XX, linked to name.

FILL	Remove TOS,2OS,3OS. Set the code in the lower byte of TOS in 2OS locations starting at 3OS.
ERASE	Remove TOS,2OS. Set zero in TOS locations starting at 2OS.
BLANKS	Remove TOS,2OS. Set space codes in TOS locations starting at 2OS.
CMOVE	Remove TOS,2OS,3OS. Copy TOS bytes from an area running upwards from 3OS to an area running upwards from 2OS.
TOGGLE	Remove TOS,2OS. Modify the contents of byte location 2OS by XORing with the lower byte of TOS.

SPECTRUM SPECIALS

In general, FORTH is independent of the machine in which it runs, apart from variations in the fundamental vocabulary, but there are usually some special features that are machine dependent. The functions described here are particular to the Abersoft FORTH, which also has a larger general vocabulary than some other implementations.

First, there are the screen control and graphics facilities, which are very similar to those of Spectrum BASIC, except that the parameters precede the command instead of following it. For functions relating to character position, for example, the line is defined by 2OS, the column by TOS.

Y X AT	sets print position at line Y column X
Y X ATTR	puts on the stack the attribute byte for line Y column X.
Y X SCREEN	puts on the stack the ASCII code for the character at line Y column X, except where the character is user-defined.
	Colour setting is straightforward:
X INK ,	where X = 0 – 9
X PAPER ,	where X = 0 – 9
X BORDER ,	where x = 0 – 7

The numbers have the same significance as in BASIC.

BRIGHT , FLASH , INVERSE and GOVER are enabled by a non-zero TOS and disabled by a zero TOS. Thus 0 BRIGHT inhibits BRIGHT, while 1 BRIGHT enables it. GOVER is the BASIC OVER, renamed for obvious reasons, OVER having a dedicated meaning in FORTH.

For dot graphics, 2OS specifies the X coordinate and TOS the Y coordinate. For example, 20 100 PLOT is equivalent to the BASIC form PLOT 20,100.

DRAW , however, works on an absolute basis, rather than relatively, a line being drawn to a specified position, whatever has gone before.

POINT returns 1 or 0 on TOS according to the state of the specified point on the screen.

A particular feature of the dot graphics commands is that no error report appears if they stray off the screen, but the plotting process continues with limited co-ordinate values, and if you allow an excessive excursion you may have to wait a long time for the trace to reappear.

UDG puts the start address of the user-defined graphics area on the stack.

Finally, there is BLEEP , which is significantly different from BEEP. TOS defines pitch, and 2OS defines duration, but while there is greater flexibility than is directly available in BASIC the system is more difficult to use. Precalculation is necessary to obtain musical scales, on the following basis:

To generate a frequency of F Hz, TOS must be set to:

TOS = (437500/F) − 30

Looking in the opposite direction:

F = 437500/(TOS + 30)

The duration of the note is determined as a number of cycles, so 2OS must be set to F × T , where T is the duration in seconds.

A point to note is that if a very low frequency is selected, with a high duration, the system may appear to hang up, because the 'BEEPER' routine in BASIC goes on and on and on . . .; without the user being able to use BREAK.

BREAK (caps shift and space) will work in some circumstances but an alternative is provided by CAPS SHIFT and 1. This, however, will only work if you have written your program to respond to it. If you make no provision for BREAK you may find your program stuck in an infinite loop.

One other point worth mentioning is that the ZX printer can be used with FORTH. It is switched on by 1 LINK and 0 LINK switches it off.

SUMMARY OF PART I

More than eighty FORTH words have now been defined, but another two hundred odd remain to be mentioned. However, many of these are 'system' words, which you will not need to use directly. Before going further, it is as well to become familiar with the words already defined, by trying out various combinations and observing the results.

Now and then, you may see an error report. If you try to read from the stack when it is empty, you will see error 1. If you use a word which is not in the dictionary, error 0 will appear. After any error the system clears down ready for a fresh start. Other errors are unlikely to arise at this stage, unless you do something rather improbable.

You will find that there is no need to enter all your commands at once. Your input data goes to the Terminal Input Buffer. When you press ENTER, or if you overfill the buffer, the INTERPRET function is called to work out what you have said. It will first try to match a group of characters delimited by spaces with the name of a dictionary entry, and if that fails it will try to interpret the group as a number which is valid in respect of the current value of BASE. If that also fails, it will put up error 0.

INTERPRET works on one word at a time, so if you press ENTER after each word or number the action will be the same. Now and then a complete sequence is desirable.

If you want to clear the screen before a particular action, you need to put CLS in the string of words and numbers which will call the action. Don't forget that the system insists on tacking 'ok' on to the end of an output which completes a string of actions. You may want to put 0 0 AT near the end of the string to put the 'ok' away in the top left corner of the screen.

If you want to find out what is happening to the stack without losing any data, a useful trick is to use:

ROT DUP . ROT DUP . ROT DUP .

This will display 3OS,2OS and TOS, in that order, but will leave the stack unaltered. You should be able to work out why.

What can we do with the words we have examined? Not a great deal, you may say. We can carry out some complex arithmetic, but it is all in integer form, and that may seem limiting. With a little patience, you will find that the limitation need not be serious. A scan through the DRAW routine listed in Appendix A will give you a hint of the possibilities in that direction. The changes in X and Y values are calculated in units of 0.0000152, by using the scaling principle. That will be examined in detail later.

In terms of exploring the possibilities of FORTH, we have as yet barely begun. As a prelude to further progress, we need to look at the way the dictionary works, and the way we can begin to create words of our own.

PART II:
The Dictionary

This part describes the structure of the dictionary, and the way FORTH uses the dictionary to respond to commands. Those who are impatient to go further ahead may skip this section for the moment, but it will help to explain mechanisms involved in the functions to be described.

THE DICTIONARY

Loading the FORTH tape sets up an area of store as the FORTH Dictionary, which contains all the essentials of the system. During initialisation a number of variables are set up outside the dictionary area and stacks and buffers are established, but the dictionary remains the heart of the system.

The name 'dictionary' is precisely appropriate, since it refers to a list of words and their definitions. When you input a FORTH word, the system searches the dictionary for it and executes the defined action. With at least 260 words to check, this takes time, which is only acceptable because the process of entering the word will have taken longer. A more rapid process is needed for executing a program.

The definitions for some words are so complex that they amount to small programs in themselves. The word M/ calls up fifteen other words, and some of these call up yet more words. To allow this to be done quickly, the definitions are set up in the form of link addresses to the definitions for the other words, so that a simple machine code routine can jump directly to the required area. As there is no need to search for any word other than the first, FORTH wastes no time in moving from one process to another. That is why it is so fast.

To make such a system work efficiently, the dictionary entries must be laid out with care. Each entry is built up from four 'fields'.

First, there is the Name field. This begins with a byte called the length byte, which — among other duties — gives the number of letters in the

name being defined. Since the maximum permitted length is 31 letters, it can be defined by bits 0 — 4 of the length byte.

Bit 5 of the length byte is the 'smudge' bit. When this is true, the word will not be recognised as valid, though it will be listed as present if VLIST is called to display the dictionary contents. The bit is set while a new dictionary entry is being created, and cleared when creation is complete, so that incomplete definitions are marked as invalid.

Bit 6 of the length byte is set to indicate 'precedence'. This means that the word will always execute, even during the creation of a new dictionary entry, when the system is said to be in 'compiling' mode and is taking all the other words it finds as being part of the new definition.

Bit 7 of the length byte is always set true.

The word itself, in ASCII code, completes the Name Field, the code for the last letter having bit 7 set to mark the end of the word.

While the dictionary is being searched, each definition has to be examined in turn. To simplify this, the Name Field is immediately followed by the Link Field, which contains the address of the next definition to be examined. The search begins with the most recent entry, which is at the top of the dictionary, and works back down the store, checking a name field, and picking up the link if no match is found. This uses a variable called HERE, which is set from the main dictionary pointer.

The linking process is illustrated in Fig 1, from which it can be seen that the number of locations that have to be scanned is kept to an absolute minimum.

When the required entry is found, the third field comes into play. This is the Code Field, and it contains a link to machine code which must be executed to implement the function. The code itself can be anywhere in the dictionary, but where it covers the complete execution of the function it usually follows immediately after the Code Field. In some circles, words which are entirely executed in machine code are called 'primitives', as they are the basic elements which perform the actual work, being called up in the required sequence by higher level words.

Many words, however, are defined solely by reference to other words, and for these the Code Field links to a routine which interprets the fourth field, the Parameter Field.

The Parameter Field contains a series of links to the Code Fields of other words, some of the links being followed by data necessary to the action of the other words. For example, there is a word LIT , which is an abbreviation of LITERAL. Its function is to put the data which comes next in the Parameter Field on to the stack. Similarly, the word ." is followed in the Parameter Field by text to be output. In this way, the Parameter Field constitutes a complete program definition for the word which it serves.

Some of the more mysterious FORTH words exist mainly to help the interpretation of the dictionary. TRAVERSE , for example, moves the

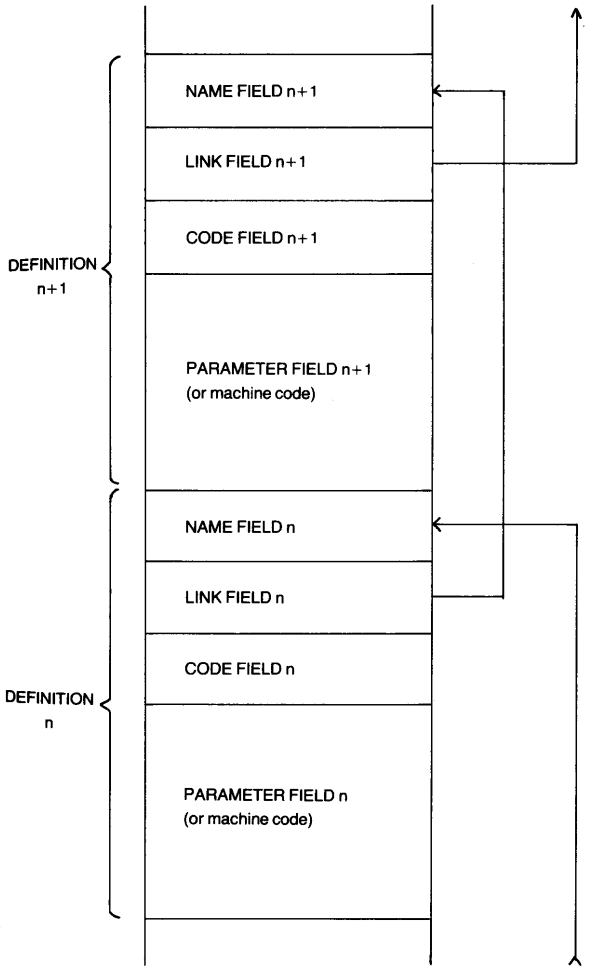


FIG. 1: DICTIONARY FORMAT, SHOWING SCAN PATH

scanning pointer (temporarily held on 2OS) from one end of the name field to the other. Taking TOS and 2OS from the stack, it adds TOS to 2OS repeatedly until a byte with bit 7 true is found. If TOS is 1, TRAVERSE scans forwards. If TOS is −1, the scan is downwards. The resulting pointer value is left on TOS.

TRAVERSE is usually called as part of a process. The word NFA, for example, converts TOS from the address of a Parameter Field to the address of the associated Name Field, and has the form:

```
5 - -1 TRAVERSE
```

Subtracting five from the Parameter Field Address takes TOS back past the Code Field and the Link Field, which contain two bytes each, to the end of the Name Field. The combination −1 TRAVERSE then scans back to the start of the Name Field.

PFA performs the reverse conversion, its form being:

```
1 TRAVERSE 5 +
```

The Name Field is scanned to the end, and the addition of 5 taking TOS to the Parameter Field Address.

Adding a new dictionary entry is simple. If you input:

```
23672 CONSTANT FRAMES
: FRAME FRAMES DUP @ SWAP 2+ C@ ;
```

you will find that you have created a new constant called FRAMES and a new word FRAME that can be used to put the contents of the Spectrum FRAMES variables on to the stack as a double word.

The colon calls up compiling mode, so instead of executing the words which follow it the system sets up a new dictionary entry with the new FRAME, setting the Parameter Field to call up DUP @ SWAP 2+ C@ in sequence. (See Part III for more detail on colon definitions.)

It is now possible to read the variable by calling the single word FRAME, instead of calling up each of the constituent functions in turn. If you wish, you can use the word FRAME in a further definition, since it has exactly the same standing of the rest of the words in the dictionary.

The semicolon terminates the process, returning the system to execution mode.

To find the correct links for the parameter field entries, it is necessary to search through the dictionary for the relevant words, so compilation is relatively slow, but execution of the compiled word will be fast. Once you have defined a word, you can use it in defining others, and you will find that there are a number of standard words which can only be used in 'colon definitions'.

An important point concerns the 'smudge' bit of the length byte in the name field. When the creation of the new entry begins, the bit is set to indicate an invalid definition, and it is not reset until definition is completed. If anything goes wrong, such as the discovery of a non-existent word in the definition, the bit is left set. The word can then be

found by VLIST, which lists the words in the dictionary, but it will not otherwise be recognised.

Incidentally, the smudge bit is manipulated by SMUDGE, which is defined as LATEST 32 TOGGLE. LATEST leaves the address of the most recently created name field, and 32 (= 20H) selects bit 5 as the one to be 'toggled'.

Having defined FRAME, you may like to use it in further definitions:

```
: MINSEC 50 M/MOD ROT DROP 60 U/MOD 0 60 U/MOD . . . ;
: CLOCK FRAME MINSEC ;
```

Taking the second definition first, it calls FRAME, then MINSEC. FRAME puts the contents of the FRAMES variable on the stack as a double number, and MINSEC first divides the number by 50 to give the number of seconds since switch-on. The unwanted remainder is discarded by ROT DROP, and 60 U/MOD then separates the minutes and seconds. As this produces a single number result, 0 is added to the stack to form a double number, and a further 60 U/MOD will generate the hours and minutes as separate numbers. The three figures obtained are output in turn.

Calling the single word CLOCK will therefore output the elapsed time since switch-on in hours, minutes and seconds. If you feel ambitious, you may like to try to define a word that will set the contents of the FRAMES variable in the BASIC area to give real time.

If this examination of the dictionary format has whetted your curiosity, you may like to browse through the summary of the standard dictionary in Appendix A, which gives the definitions of all the standard words. You will find that a variable is entered in much the same way as an executive word, with a special routine to interpret the parameter field, while a constant is again similar in format but again uses a special routine.

Because a single variable takes up at least eight bytes of dictionary space, it is understandable that the ideal approach is to minimise the use of variables. However, with ample space available it is permissible to be a little extravagant.

This excursion into the mechanism of FORTH was not an essential factor in using the language, but it should have given a useful insight into what goes on inside the system. When all the available words are understood, it will be seen that the foregoing description has been very much simplified. Instead of relying on existing words to form a new definition, it is possible to enter machine code routines to execute special tasks. The process of compiling can be halted to allow the system to calculate data for further compilation, then compiling can be resumed.

To make the most of the system, it is best to limit the size of each created definition, rather than attempt to cram everything into a single definition. That way, the validity of each new word can be tested before going further, and overall confusion can be avoided.

Mention of testing naturally leads on to the question of how errors can

be corrected. This will be dealt with in detail at a later stage, but it will suffice for the moment to say that it is always possible to FORGET a defined word. If it is the word most recently defined, that word alone will be erased from the dictionary by the simple process of setting the search start pointers to the name field of the previous definition. If other words have been defined since, all of them will be lost. In either case, however, the relevant data remains unaltered. It is merely ignored by the search processes. This may be compared with the use of OLD after NEW, in some versions of BASIC, to restore a program unintentionally destroyed.

There are many ways to use FORTH. At a simple level, you can rely on the construction of a pyramid of dictionary definitions, with no fancy tricks involved. Growing confidence may lead you to explore more complex techniques, and eventually you may begin to see how the dictionary entries can be modified to make them do what you want. If your ambitions lead you along this path, progress with caution rather than speed, for there are traps for the unwary, but an attempt will be made to provide you with the information you will require.

In this section, we have encountered:

1 TRAVERSE	Convert TOS from name field start address to name field end address.
−1 TRAVERSE	Convert TOS from name field end address to name field start address.
CFA	Convert TOS from parameter field address to code field address.
LFA	Convert TOS from parameter field address to link field address.
PFA	Convert TOS from name address to parameter field address.
NFA	Convert TOS from parameter field address to name field address.
LATEST	Put contents of CURRENT on the stack. (Identifying the name field address of the latest entry in the dictionary.)
SMUDGE	Change the state of the location defined by LATEST by XORing TOS (LOWER BYTE) with the contents of that location.
TOGGLE	Change the contents of the location defined by the address in 2OS by XORing the contents with TOS (lower byte).
FORGET	Change the contents of CURRENT to point to the name field of the dictionary entry identified by the link field of the word which follows FORGET.

PART III:
Colon Definitions

This part explains how new words can be created by use of 'colon definitions', and introduces words related to the branching and looping processes. If the use of FORTH were limited to the execution of words input from the keyboard, it would be a very limited language indeed, but the whole concept rests on the ability to create new entries in the main dictionary, or in a subsidiary dictionary created for a particular purpose. Once a new entry has been made, it can be used to help to define further entries, until a complete program can be defined in terms of a single word.

In compiling new definitions, there are a number of words which are unacceptable for direct execution, and these will be examined in due course. First, however, some examples of the use of created definitions will help to show the possibilities.

EXTENDING THE DICTIONARY

In the section on memory access, we saw that a variable entry could be extended by ALLOT to reserve an area which could hold an array. Access to a given element of the array required a short routine which would work out the address of the element:

```
n name SWAP DUP + +
```

Since n, the number of the required element, and name, the name of the array, are variable factors, we can usefully define:

```
: ACALC SWAP DUP + + ;
```

We can now obtain the required address by the shorter entry:

```
n name ACALC
```

This will have the same affect as the sequence previously defined. We can now go a step further and define:

```
: A! ACALC ! ;
: A@ ACALC @ ;
```

Having defined ACALC , we can now use it to define two words which will write to and read from the array element address. This further compresses the code required. It would have been possible to define A! and A@ directly, without defining ACALC first, but we are dealing with principles here, rather than with practice, and the route adopted demonstrates the point more clearly.

In a program examined later on, we will require a three-element array called PILE which will be set up by:

```
0 VARIABLE PILE 4 ALLOT
```

We can then access element n of the array by:

```
n PILE A@
```

or write to it by:

```
n PILE A!
```

We must be careful not to make n greater than 2, or we will step outside the reserved area. The subscript range, it should be noted, is 0 to 2, not 1 to 3 as in Spectrum BASIC. If we called 3 PILE A! we would overwrite the name field of the subsequent dictionary entry.

Multidimensional arrays are more difficult to handle. The usual rule for calculating an element number n from given subscripts and dimensions is based on the iterative application of:

$n_x = n_{x-1}$ *dimension$_x$ + subscript$_x$

Initially, $n_0 = 0$, so n_1 = subscript$_1$

Then n_2 = subscript$_1$ *dimension$_2$ + subscript$_1$

This process is repeated for each subscript in turn.

For a three-element array, the FORTH sequence would be:

```
S3 S2 S1 D2 * + D3 * +
```

This would put the element number on the stack. We can therefore define:

```
: ELEMENT D2 * + D3 * + ;
```

We could now read from the elements defined by subscripts A1, S2, S3 by calling:

```
S3 S2 S1 ELEMENT ARRAY A@
```

where ARRAY is the name given to the ARRAY.

In these fairly simple examples, you will note that each definition begins with a colon, to instruct the system to create a new dictionary entry in compiling mode, and ends with a semicolon to instruct a return to direct execution mode. The word following the colon is taken as the name of the new entry, and the words that follow are set up in the Parameter Field in terms of their Code Field addresses. Note that the colon and semicolon are not needed when setting up variables and constants.

You need have little fear that you will overfill the available space by creating too many definitions. You can always use FREE . to check how much space is left, and you will find that it shrinks quite slowly. If you want to calculate how many bytes a new definition needs, add up:

- The number of letters in the new word name, plus 1.
- Two bytes for the Link Field.
- Two bytes for the Code Field.
- Two bytes for each word used, including the semicolon but not the colon.
- Four bytes for each numeric value.
- Four bytes for a branch or loop. (See next section)

The word ELEMENT , defined above, takes up 32 bytes.

You should now be in a position to experiment with the construction of some simple sets of words. As you create each new definition, make a note of the way it changes the stack. ACALC removes n and an address, leaving a modified address in exchange. A! removes n, the address, and a single number, while A@ removes n and the address and leaves a new single number. ELEMENT removes three subscripts and leaves n.

If you note these changes, you will find that it is easier to keep track of the overall stack changes, since there is no need to go through each definition word by word to see what is happening.

For the moment, you will be hampered by the fact that your definitions soon scroll off the screen and are lost, but a remedy for this will be demonstrated in Part IV, when the RAM-Disc system is described. This is a storage system in which you can save your definitions and bring them back whenever a query arises or you want to amend them.

BRANCHING AND LOOPING

Exponents of structured programming regard it as meritorious that the FORTH language does not offer a function comparable with the BASIC word GOTO, which they regard as unsanitary. In fact, it would be very difficult to introduce such a word directly, since there would be no way of defining the destination. Inside the dictionary, however, most of the branching and looping functions are implemented by relative jumps, which are at least near relatives to GOTO.

The jump functions are:

BRANCH XXXX , which transfers action to a link address by adding XXXX to the interpretive pointer.

0BRANCH XXXX, which acts in the same way, but only if TOS is zero. Otherwise, it has no effect.

Since XXXX is a 16-bit word, it would theoretically be possible to jump

to any other point in the dictionary, but the actual jump spans used are relatively small.

BRANCH , in isolation, is of little value, and is usually employed to define an alternative action in association with 0BRANCH. In order to appreciate the action of 0BRANCH , we must consider some of the special ways in which TOS can be set to determine whether 0BRANCH acts or not.

First, there are simple arithmetic calculations. If they make TOS zero at the time when 0BRANCH is called, it will act. Otherwise, it will do nothing. In either case, TOS will be removed.

Next come the logic operators, which are not very different:

=	Put a non-zero state on TOS if TOS = 2OS, otherwise a zero state.
<	Put a non-zero state on TOS if 2OS is less than TOS, else zero.
>	Put a non-zero state on TOS if 2OS exceeds TOS, else zero.
U <	As < , but treating the numbers as unsigned.

Each of these operators removes TOS and 2OS, and is itself lost when 0BRANCH acts. It is usual to refer to the item put on TOS by the operator as a true flag (non-zero) or a false flag (zero).

Next there are operators which act on a single value, rather than on the comparison of TOS and 2OS:

0<	Put a non-zero state on TOS if TOS is negative, else zero.
0=	Put a non-zero state on TOS if TOS is zero, else zero.

Both these operators remove the original TOS.

The operator 0= effectively reverses the state of a flag on TOS, and is also called NOT . It may be used after other operators to invert their sense.

A number of other functions leave flags. For example, ?TERMINAL puts a non-zero state on the stack if the BREAK combination (CAPS SHIFT and SPACE or CAPS SHIFT and 1) is called from the keyboard. Other examples will arise from time to time.

CONDITIONED BRANCHING

A deceptively familiar branching format is IF . . . ELSE . . . ENDIF , but its action may seem less familiar. It ocurs in the form:

condition IF action1 ELSE action2 ENDIF action3.

If condition (on TOS) is true, actions 1 and 3 are performed. If condition is false, actions 2 and 3 are performed. The condition flag is removed from the stack, and it is sometimes convenient to create it by using

−DUP , which only duplicates TOS if it is non-zero. The original TOS would then be available for use in action1, but would be discarded if action2 was performed.

The implementation in the dictionary should be straightforward. IF is replaced by 0BRANCH , which performs an onward jump to action2 if condition is zero. Otherwise, a BRANCH at the end of action 1 performs an onward jump to action 3.

Compilation of IF . . . ELSE . . . ENDIF is checked by a reference number which is checked by ?PAIRS . IF sets up 2 , and ELSE requires 2 to be set. ELSE also sets up 2, and ENDIF requires 2 to be set. It is therefore permissible to omit ELSE and action 2 if only one conditional action is requied.

If you feel it is more expressive, you can replace ENDIF by THEN . They mean exactly the same.

The FORTH ' DO loop ' is similar to the BASIC ' FOR loop ', the equivalent forms being:

FOR N = A TO B . . . NEXT	= B+1 A DO . . . LOOP
FOR N = A TO B STEP C . . . NEXT	= B+1 A DO . . . C+LOOP

Note that the limit parameter is B+1, not B. While the iteration index is less than the limit, LOOP returns action to a point immediately after DO , but when the index is equal to the limit the words following LOOP are executed. The Return Stack holds the necessary data.

DO is compiled as (DO) . When (DO) is executed, it transfers TOS to TORS (the initial index value A) and 2OS to 2ORS (the limit value B+1).

LOOP is compiled as (LOOP) . When (LOOP) is executed, TORS is incremented, and the result is compared with 2ORS to see if the limit value has been reached. If it has not, the routine jumps back to a point immediately after (DO) , the jump span being calculated at the time the sequence is compiled, and set in the word following (LOOP) in the Parameter Field.

+LOOP is compiled as (+LOOP) , which acts in much the same way as (LOOP) , except that C (stored in the Parameter Field as a 'literal', a number to be put on TOS) is added to the index instead of unity.

Within a DO loop, the iteration index is normally on TOS , unless it has been covered by >R , in which case the covering item must be removed by R> before (LOOP) is reached. The word R will therefore put TORS on TOS, without disturbing the count. For some unexplained reason, a synonym of R is commonly used: I . The two words execute in precisely the same way, even using the same machine code. By using I , the current index value can be used in calculations within the loop.

The value of 2ORS can be copied to TOS by I' , giving the limit value of the index. They can be useful where the required value is (limit — index).

There is sometimes a need to use the iteration count of an outer DO loop, where two loops are nested. For this, 3ORS is required, and it can be copied to TOS by the word J .

There is a slight catch about this. If a Return Stack value is read within the definition of a word called within a DO loop, a return link has been added to the Return Stack during execution of the definition, and it is necessary to dig a little deeper, using I' instead of I , J instead of I'. For example:

```
: CALC2 I' . ;
: CALC1 10 0 DO CALC2 LOOP ;
```

Now CALC1 will print out the numbers 0 to 9.

So will the listing of CALC1 after defining it as:

```
: CALC1 10 0 DO I . LOOP ;
```

The return link shows where execution should be resumed in the calling sequence Parameter Table.

In BASIC jumping out of a FOR NEXT loop is sometimes permissible, sometimes unwise. To get out of a DO loop prematurely, the word LEAVE is used. This sets the index to be equal to the limit value, terminating the loop at the next LOOP . LEAVE commonly occurs in the format:

```
condition IF LEAVE ENDIF
```

Less familiar to users of older forms of BASIC, but implemented in some recent versions, are the remaining loop formats:

```
BEGIN action UNTIL
```

repeats action until action puts a true state on the stack. A common format to allow manual exit from a loop is:

```
BEGIN action ?TERMINAL UNTIL
```

Pressing CAPS SHIFT and 1 will cause ?TERMINAL to return a true value on TOS, so that the loop drops out.

If no conditonal is set up before UNTIL , the loop may continue for ever. Note that UNTIL removes TOS, being implemented by 0BRANCH , which returns action to just after BEGIN if it finds TOS zero.

You may use END instead of UNTIL if you wish. They mean the same. This is another example of FORTH synonyms, different words which mean the same thing, and are executed by the same code.

A more complex loop structure is provided by:

```
BEGIN action1 WHILE action2 REPEAT action3
```

WHILE removes TOS. If TOS = 0, action1 is followed by action3. Otherwise, action1 and action2 alternate.

The format BEGIN action AGAIN should be used with caution, as there is no way out of the loop other than a call of QUIT , ABORT , or EXIT , executed conditionally within action. You have been using QUIT from the start, as it is the routine which accepts user inputs. It clears the Return Stack, whereas ABORT clears both stacks, and is more drastic. EXIT removes TORS , destroying the link which would otherwise be used to direct further action. What happens then depends on what emerges as

the new TORS. Within a DO loop, it might be the index count, and that would cause mayhem. Experiment with EXIT cautiously, though the worst that can happen is that you have to re-load the FORTH tape and start again . . .

Finally, there is the CASE structure, a special addition to fig-FORTH that is omitted in some implementations. The format is:

```
condition   CASE
            A OF action1 ENDOF
            B OF action2 ENDOF
            C OF action3 ENDOF
            etc.
            ENDCASE
```

If A matches the stated condition, action1 is executed. If B matches the condition, action2 is executed, and so on. The condition must be in the form of a single number (or the equivalent, e.g. an ASCII code). A, B and C can be words or sequences of words producing a similar result. An example will be found near the end of the program given in the last section of this book.

STRINGS AND THINGS

So far, we have only talked about numbers, saying little or nothing about text. It is time to remedy that deficiency.

The word ." compiles a dictionary entry containing the text which follows. For example:

```
: ABC ." Program Number 1 " ;
```

will set up a dictionary entry which will print the text when the word ABC is called. Note that there must be a space between ." and the start of the text, and that the text is completed by a further 'quotes' character.

The same definition can include numerics and other words, allowing such definitions as:

```
: BCD 10 6 AT ." Program No 2 " ;
```

This will position the text on the screen at line 10 column 6.

Note that in the compiled entry ." is replaced by the link to the compiled form (.") and a character count follows. The actual output is performed by TYPE , which outputs TOS characters, taken from an area of store starting at 2OS. (TOS and 2OS are removed.) The character count is set up by WORD which establishes the number of characters and enters the number, with the actual text, in the dictionary. There is usually no need to worry about these inner functions.

FORTH does not have, as a basic provision, any facilities for string slicing. If you want anything like that, you must construct it for yourself. The procedure would be to locate the string on which you wish to

operate, identify the part of the string you require, and extract that part either for immediate use or to form a new entry. An illustration will show the method best.

Define a basic string:

```
: DATE ." JanFebMarAprMayJunJulAugSepOctNovDec " ;
```

The sequence –FIND DATE will set TOS to 1, indicating that the word DATE has been found, 2OS will contain the length byte for the word, and 3OS will contain the parameter field address of the entry. All we need is the address:

```
–FIND DATE DROP DROP 3 + CONSTANT POINTER
```

This will define a constant pointing to the address plus 3, which is the start of the stored text. We have skipped over the two bytes defining ." and the length byte. Note that this sequence is for immediate execution. It is not a colon definition.

To pick out the nth month, we must calculate 3*(n–1) and add it to the constant POINTER. That will give us the address of the first character of the month name. The address is set in the variable OUT :

```
: DATEADD 1 - 3 * POINTER + OUT ! ;
```

To get the address of the nth month, we need n DATEADD , so we can construct:

```
: OUTDATE DATEADD 3 0 DO OUT @ @ EMIT LOOP ;
```

The contents of OUT are incremented by EMIT , which outputs the character put on TOS by OUT @ @ . The whole month name can now be output by n OUTDATE .

The procedure described above may seem rather complicated at first, but it has the advantage of extreme flexibility.

FORTH also provides facilities for printing formatted numerics. The words involved are:

<#	Set up HLD to the address of the text output buffer, as held in PAD. This buffer has no fixed address. It floats 68 bytes above the top of the dictionary.
#	Operates on a double number on TOS/2OS, generating the least significant digit of its representation in the current number base, and leaving the residue on TOS/2OS as a double number. The digit is stored in PAD, using HLD as a decrementing pointer.
#S	Repeat # until the double number is reduced to zero.
SIGN	Used between <# and #> , inserts a minus sign before a converted numeric string if 3OS is negative. 3OS is discarded, but TOS/2OS are undisturbed. They will normally contain the double number being operated on.

#>	Complete formation of a numeric string by dropping the double number and leaving the address of its location and a length byte. TYPE normally follows.
TYPE	Outputs TOS characters taken from store, starting at 2OS.

These words allow a rigid format to be set up. Whereas #S will terminate output when the double number reaches zero, a pattern can be set up using # which will insert trailing zeroes. There is one more provision:

HOLD	Decrement HLD and store TOS in the numeric string as an ASCII character code. For example, 46 HOLD will insert a decimal point.

The full capabilities of these words can best be found by experiment. Note that a decrementing pointer is used to set up the string in the buffer, whereas TYPE uses an incrementing pointer. The least significant digit is generated first, but is output last.

The process can use a DO loop to call # DPL times before the decimal point is inserted, DPL holding the decimal point position. SIGN comes last, so that its result is displayed or printed first.

It is convenient to mention here some of the other words that relate to strings and text output.

COUNT, used with TOS pointing to the length byte of a text entry, puts the actual length byte on TOS and the address of the start of the subsequent text on 2OS. TYPE can then be used to output the text.

. CPU outputs the name of the computer, in case you have forgotten which machine you are using . . .

–TRAILING modifies the length byte of a numeric text string to remove trailing spaces. It requires the original length byte on TOS and the address of the start of the text on 2OS. It leaves the same situation, with the length byte adjusted. It can therefore be interposed between COUNT and TYPE, or may precede TYPE when it is used to output a formatted string.

. ID takes TOS as a name field address and ouputs the name of the relevant definition.

The pure string functions in FORTH are not too extensive, as it is mainly seen as a calculating language, but quite a lot can be done with the facilities provided, given a little thought.

INPUT/OUTPUT

The input of keyboard data for immediate execution or for setting up colon definitions is so simple that it tends to be taken for granted, but quite a lot is done immediately to achieve that simplicity.

The QUIT routine is used for such input, because it is entered

whenever other actions are complete and control is to be returned to the user. It invokes QUERY, which puts the address of the Terminal Input Buffer on the stack, and then the number 80H, which is the size of the buffer in bytes. EXPECT is then called to transfer characters from the keyboard to the Terminal Input Buffer until either Return is pressed or eighty characters have been input. QUERY then zeroes the input pointer IN , so that the data which has been input can be scanned.

This data is in ASCII code form, and needs to be interpreted. Any group of codes beginning and ending in a space may be a FORTH word, and the dictionary is searched to check this. If no match is found, the group may be numeric, and NUMBER is called to check this. NUMBER requires a pointer address on TOS, and it checks for a valid numeric in terms of the current value of BASE, also looking for a decimal point. If the latter is found, DPL is set to the number of digits which follow, otherwise it is set to −1. If the number is not valid, an error is reported.

If DPL is positive, INTERRUPT sets up a double number on the stack, otherwise setting a single number. If the code group is not identified as either a word or a number, INTERPRET reports error.

An interesting point is that the potential word is entered in the dictionary whether it is valid or not, but the dictionary pointer is not advanced until the word has been checked, so the word meanwhile has no real existence, and may later be over-written.

If the word is recognised in direct mode, it is executed, while numbers are put on the stack, so a continuity of action is maintained, even if the input text is supplied in small sections. In compiling mode, compilation proceeds in a similar manner.

While compiled definitions are running, a rather different situation applies. The keyboard is ignored unless an input is specifically invited. This can be done by:

KEY	Put the ASCII code for a key depressed on TOS.
INKEY	As KEY, but if no key pressed put FFH on TOS.

The difference is that KEY waits for a key depression, whereas INKEY does not.

These words are convenient for simple control, for responding Y or N, or for halting action until you are ready to go on. It is possible to translate the ASCII code to a numeric value by using DIGIT, which requires BASE on TOS and the character on 2OS:

```
KEY BASE @ DIGIT
```

This will put the numeric value on the stack, then a true flag if the code is a valid number, or will put a false flag on the stack if the code is invalid. However, the number will not be displayed unless you expand the sequence to:

```
KEY DUP EMIT BASE @ DIGIT
```

It would be possible to extend the sequence further, to accept and combine a number of digits as a complete number, but the combination

QUERY INTERPRET is more convenient.

The output of data to the display has been adequately covered elsewhere, and the use of 1 LINK to switch the printer on and 0 LINK to disable it have also been mentioned. The remaining input/output functions are those relating to the ports:

INP	Removes TOS as a port number, and sets TOS as the byte read from the port.
OUTP	Removes TOS as a port number and 2OS as data. The data is output to the specified port.

A certain amount of care is needed in selecting ports, as some have dedicated uses in the Spectrum system. Bits 0 to 4 of the port address should be high, and bit 7 should be low, which leaves just four usable addresses: 1FH, 3FH, 5FH and 7FH. The limitation applies more particularly to output ports.

ODDMENTS

A number of quite important words have yet to be mentioned. Those associated with the compiling process will be dealt with later, in PART IV, but an attempt will be made here to collect together the other stragglers.

First, there is the word FORTH itself. It calls up the FORTH vocabulary, as distinct from any other set of words which may have been defined, such as the EDITOR vocabulary. Calling the name of a vocabulary switches links within the dictionary so that the required set of words becomes available. To be precise, CONTEXT is reset, this being the pointer to the start point for searches. An attempt to FORGET a word which is not in the currently-selected vocabulary gives error 24, because you might destroy a lot more words than you intend to. In any case, you will not be allowed to FORGET any word which lies beyond the point defined by FENCE, which acts as a protective barrier.

When you do FORGET a word, you discard not only that word, but also every word that comes before it in the dictionary, that is every word that is of more recent origin. This is sometimes used to advantage by compiling the dummy word TASK as the first word in a program. FORGET TASK will then discard the complete program, including any invalid words it contains. Such words cannot be forgotten directly.

The variable STATE determines current working mode, direct execution or compiling, and will generate an error if a word inappropriate to that state is used. The error checking system employs a number of words, a good starting point being ?ERROR, which occurs in the form f n ?ERROR, where n defines the error number and f is a flag which is true if the error is present. If the flag is false, f and n are cleared from the stack and action continues.

If the flag is true, ERROR is called with n on TOS. What happens then depends on the contents of WARNING. If WARNING is negative, (ABORT) is called, ABORT being called in turn. Otherwise, the offending word is displayed, followed by a query, and then MESSAGE is called, still with TOS = n, after which the stack pointer is re-initialised and QUIT follows.

WARNING also controls the action of MESSAGE. If WARNING = 0, the format 'MSG # n' is displayed. If WARNING is greater than zero, the system will display an explicit text message, the text being held in the RAM-disc storage system described in Part IV. Since such text would pre-empt one fifth of the available storage space, it is not normally used in Abersoft FORTH, and it would be completely impractical for other implementations which provide less storage space.

?STACK checks the value of the main stack pointer, reading its contents by SP@ and comparing the result with SO, the initialised value, and with HERE + 80H. If the pointer is below HERE + 80H, there is a risk of conflict with the stored program, and ?ERROR is called with n = 7 and a true flag. If the pointer is above SO , the stack has been over-emptied, and ?ERROR is called with n = 1 and a true flag. If neither state exists, a false flag is set, and though n = 7 ERROR is ineffective. The approach of an error 7 condition can be anticipated by using FREE . to display the amount of free space.

Other error checks are:

?COMP calls error 17 if the system is not in compiling mode.

?EXEC calls error 18 if the system is not in direct execution mode.

?PAIRS calls error 19 if the branching and looping words are not correctly paired in sequence.

?CSP calls error 20 if the stack pointer fails to match the contents of the variable CSP, indicating an incomplete dictionary entry.

?LOADING calls error 22 if the terminal input buffer is in use.

Subsidiary functions and variables associated with the error system are:

!CSP	Sets CSP to current stack pointer contents.
RO	The initialising source for the return stack pointer.
RP@	Reads the return stack pointer to TOS.
RP!	Sets the return stack pointer from RO.
SP!	Sets the calculation stack pointer from 20.

Finally, in this group, WHERE can be called after an error during compiling, whereupon it will pinpoint the source of the error by putting the relevant line on display with an accusing arrow pointing to the offending word. The verdict is not always accurate. If, for example, a colon is missing from the front of a colon definition, the error may only become apparent when the semicolon is reached. Provided this is borne in mind, WHERE provides a useful aid for debugging source code.

Few of the words just described would normally be called by the user. Like many others, they are provided for use inside the FORTH system.

COLD and WARM are a different matter. They allow the system to be re-initialised with and without the loss of dictionary extensions. They can be called as words, but they are also accessible from BASIC, by GOTO 2 for COLD and GOTO 3 for WARM. The BASIC system can be entered by MON .

It is sometimes useful to know how big the dictionary is, since it can be saved on tape in extended form, and SIZE serves this purpose, by putting the number of bytes between ORIGIN and HERE on TOS. ORIGIN is a nominal start point for the program, and n+ORIGIN puts on TOS the address of the nth byte from that start point.

EXECUTE and COMPILE induce entry to the working mode named, by setting STATE to the appropriate value.

NOOP does nothing, except perhaps fill a gap.

That leaves only the fact that 0 , 1 , 2 , 3 are FORTH words in the form of constants, saving any need to work out their value, and n USER provides the address of a location within the User Variable area, which is defined in Appendix A.

The Appendix will also answer most questions about any word which has not been covered as yet, including the EDITOR vocabulary, which contains 28 words. However, the examples which will be found in Part V should clear up a number of possible obscurities.

PART IV:
The RAM Disc

This part discusses the provisions for storing source code, for saving and loading tape copies, and describes the EDITOR vocabulary.

SCREENS

One further major facility is needed to make FORTH fully viable. The input of colon definitions for immediate compilation can be very tedious, especially if frequent alterations are needed. The answer lies in the Screens, which store source code for compilation in a way that allows changes to be made quickly and fairly easily.

Spectrum FORTH provides eleven screens, each containing sixteen lines of 64 characters. This is the standard FORTH screen format, and it is not entirely compatible with the smaller Spectrum display. It leads to the user having to put up with some scrolling and using CAPS and 1 to BREAK, but this is easily manageable.

The input n LIST will display and select screen n, but the inital display will show lines full of queries, because the store area is full of zero bytes. INIT-DISC will fill all the screen areas with space codes. Source code can then be entered, using the EDITOR vocabulary, and can thereafter be compiled by n LOAD, where n is the number of the screen concerned. If --> is entered at the end of the screen, compilation will continue to the next screen. In that way, all ten screens 1 — 10 can be compiled if required.

SAVET will save the entire contents of the RAM disc on tape, and it can be reloaded by LOADT or checked by VERIFY, all these words being variations of (TAPE), the common tape system command. All the tape files have the name DISC, so you must make your own arrangements for identifying one from another.

What may not be immediately obvious is that a number of sets of screens can be loaded in turn, each being compiled while it is present, so that relatively large programs can be set up.

FORTH conventions, some enforced by the characteristics of the language, impose some limitations on the way the screens are used.

Screen 0 is reserved for comments and explanation, and cannot be used for loading. Similarly, line 0 of each screen is used for a heading, and x y INDEX will output the headings of screens x to y. As with all comments, the headings should be enclosed between round brackets, with a space after the open bracket, as it is a FORTH word meaning 'ignore all that follows until a close bracket is found'. The brackets can be used as temporary entries to force the system to start compiling in mid-screen, the open bracket being set at the start of line 0 and the close bracket immediately before the point where compiling is to start. The word ;S will cause compiling to stop, as will a null entry anywhere on the screen.

Other conventions arise from the fact that the screens were originally intended to work as part of a disk system. This meant that the total screen area was effectively as large as a disk, and it was reasonable to allocate two screens to the task of supplying textual error messages. Errors 1 to 15 were supplied by screen 4, and the remainder by screen 5. This facility still exists, and can be brought into action by making WARNING equal to 1 (1 WARNING !). You will have to enter the messages in the screens concerned, but this may be a useful facility while you are getting used to the error numbers. You will lose two screens, but that may not be too catastrophic in the early stages.

For printed record purposes, TRIAD is useful. It will display or print the content of three successive screens, starting at 0, 3, 6 or 9. Thus 5 TRIAD would print the group including screen 5, i.e. screens 3, 4 and 5.

There are a number of words which are provided for use in screen handling, but before examining these it will be best to look at the EDITOR vocabulary, which is essential for setting up screen data.

THE EDITOR

Immediately after loading the FORTH tape, VLIST will display a string of words beginning with UDG , the last entry in the basic FORTH vocabulary. If you define any new words in that vocabulary, they will appear before UDG , being more recent creations. Input EDITOR , and VLIST will now show an additional group of words at the start of the display. These are the words in the special EDITOR vocabulary. The word EDITOR has switched links in the dictionary to bring them into action. The word FORTH will switch links again so that the EDITOR words are no longer accessible.

There are almost too many words in the EDITOR vocabulary, and it is best to get used to them by degrees. First, you will need to select a 'screen' and bring it into action in a cleared state. If you want to select it without clearance, you need n LIST. At any time thereafter, as long as the

EDITOR is enabled, L will also display the screen for line 0.

The simplest method for entering data into an empty screen is the format m P text . This puts the text in line m of the current screen.

If you could guarantee 100% accuracy in typing and total infallibility in the framing of colon definitions, this one command would suffice for setting up source code, but there will inevitably be changes to be made. The first step is to pinpoint the position at which the change is needed, and for that there are useful search facilities, hinging round the cursor position, which is stored in the variable R#.

TOP zeroes the variable, and should be used before a search unless you are sure the cursor is above or to the left of the change point. Suppose you want to change FIRTH into FORTH . If the offending word occurs only once in the screen, TOP X FIRTH will erase it, leaving the cursor at the position the word occupied. C FORTH will then insert the word FORTH in the same place. Remember that C and X , being FORTH words, require a following space before the start of associated text. If you input C without text, you will enter a null code, and that must be removed, since it acts as a terminator. Fortunately, TOP X will not only locate the null, it will replace it with a space.

If the word you want to erase occurs more than once, you can find the first occurrence, without erasure, by using TOP F text , and then N will step forward to the next occurrence. When you have found the right position, the cursor will be at the end of the text, but B will move it back to the start of the text, so that X or C can be used.

TILL will delete all text from the cursor position to the end of the cursor line.

The cursor can be positioned precisely by n M , which moves the cursor n places and displays the resulting position. The value of n can be positive or negative.

In all these moves and changes, the operative text is held in a buffer called PAD, and this can be used to hold lines temporarily in order to shuffle the order of lines on the screen. To move line n of the current screen to PAD you need n H , while n D will move the text and delete the source. On the other hand, n E will delete the line without saving it.

Once a line is in PAD , and you want to insert it betwen two existing lines which are adjacent, n S will move line n and all the lines below it down one line, line 15 being lost, and n R will transfer the text from PAD into the gap. More simply, n I will perform both operations.

In cases where the screen is very full, it is sometimes useful to employ n T , which types a single line and holds it in PAD. The line can then be checked in isolation.

The remaining EDITOR words are mostly sub-functions of those which have been described, and are not normally needed for direct use, but an exception is COPY .

The form a b COPY will copy the contents of screen a to screen b. An extension of this uses a negative screen number, which will copy a

screen to or from the free space between the dictionary and the stack. If free space is limited, it is as well to do a trial run to make sure the area used is not occupied, making a dummy transfer from it.

This facility has the advantage that a screen can be moved from one tape record to another. The record originally containing the screen is loaded, and the screen is transferred to free space and erased from the source screen. The result is saved and verified. The second record is then loaded, and the screen is transferred back to a free screen. Being outside the normal screen area, it will not have been affected by the saving and loading. It can now be saved as part of the second record.

This little trick can be very useful, but is best performed when there are no dictionary extensions and the free space is at its largest extent.

Perhaps n DELETE should be mentioned. It deletes n characters to the left of the cursor position, and is really provided to serve X . The remaining words are MATCH , #LOCATE , #LEAD , #LAG , -MOVE , -TEXT , 1LINE and FIND . All these are primarily internal words, rather than user words.

If the editing system has a fault, it lies in the large number of words provided. With a little practice, you will find that a small subset will serve most purposes.

It will be noted that two words in the EDITOR vocabulary, R and I , are identical with words in the FORTH vocabulary. For direct execution, however, the EDITOR forms are found first when the EDITOR vocabulary is enabled, so these are used.

The user words in the EDITOR vocabulary may be summarised:

B	Move the cursor back by the length of the text held in PAD.
C	Insert the following text at the cursor position, spreading the original text to make room.
D	Remove TOS as a line number of the current screen and delete that line after copying it to PAD.
DELETE	Remove TOS as a number of characters, and delete that number of characters to the left of the cursor.
E	Remove TOS as a line number and erase that line with space codes.
F	Search for match with following text from cursor to end of screen.
FIND	As F , but using text already in PAD, and end with TOP .
H	Remove TOS as line number, and copy that line to PAD.
I	Perform S and R , inserting line from PAD at line TOS .
L	List the currently-selected screen.

M	Add TOS to cursor position and display line.
N	Find next occurrence of match to text in PAD.
P	Put the following text in the line defined by TOS.
R	Replace line identified by TOS with text in PAD.
S	Move line identified by TOS and following lines down one line.
T	Display line and copy to PAD.
TILL	Delete from cursor to end of line.
X	Delete next occurrence of following text.
TOP	Set cursor to zero.
CLEAR	Clear screen identified by TOS.

BEHIND THE SCREENS

To preserve at least nominal compatibility with standard fig-FORTH, Spectrum FORTH includes in its vocabulary a number of words which relate to true disk operation, but are not directly relevant to the RAM disc system.

A buffer area is established between CBE0 and CFFF, and this contains eight buffer areas of nominally 128 byte capacity, four extra bytes being provided for control purposes. In a true disk system, these would be in direct communication with disk, holding records read in and providing data to be read out.

In general, words which relate to these buffers can be ignored, except perhaps for the purpose of adventurous experiment. The words, all defined in detail in Appendix A, are:

Constants:

#BUFF	Number of buffers allocated (8)
B/BUF	Number of bytes per buffer (128)
B/SCR	Number of blocks per screen (8)
C/L	Number of characters per line (64)
FIRST	Address of lowest buffer start (CBE0)
HI	Address of end of screens area (FBFF)
LIMIT	Address of end of buffer area plus 1 (D000)
LO	Address of start of screens are (D000)

Variables:

BLK	Block number being interpreted. If BLK = 0, the TIB is in use as a source.

OFFSET — Can be used to displace the screens area, this being equivalent to moving to a different disc area. Best left at zero.

PREV — Holds the address of the most recently used buffer.

USE — Holds the address of the least recently used buffer.

SCR — Holds the number of the screen in use.

Functions:

+BUF — Removes TOS as the address of the current buffer, and selects the next buffer in sequence, putting the address of the latter on 2OS and a flag on TOS. If the new buffer is that identified by PREV, the flag is false.

.LINE — Removes TOS as a screen number. If that block is held in a buffer, the address of the buffer is returned on TOS. Otherwise the contents of the buffer identified by USE are read to disc (in this case RAM disc) and the block is copied into that buffer, the address of which is returned to TOS.

BLOCK — Removes TOS as a block number. If that block is held in a buffer, the address of the buffer is returned on TOS. Otherwise the contents of the buffer identified by USE are read to disc (in this case RAM disc) and the block is copied into that buffer, the address of which is returned on TOS.

BUFFER — Removes TOS as a block number, and assigns the next buffer to that block, first saving the buffer contents on disk if they have been updated. The address of the buffer is put on TOS. (BLOCK uses BUFFER)

DRO — Set OFFSET = 0

EMPTY-BUFFERS — Set up the control fields of all buffers to the initial state, i.e. as empty.

FLUSH — Write all updated buffers to disk.

LINE — Removes TOS as a line number, and returns the address of the start of the line in the current screen.

R/W — Removed TOS as a flag, 2OS as a block number, and 3OS as an address. If the flag is false, data is written from the buffer to disk (RAM disc). If the flag is true, a read operation is performed.

TEXT — TOS is removed as a delimiter, and the following text is copied to PAD.

UPDATE — The buffer identified by PREV is marked as updated, i.e. its contents have been altered.

PART V:
Simple Programs

In this part, some simple programs will be discussed as examples.

SIMPLE PROGRAMS

Armed now with all the facts needed, we can begin to look at some simple programs. As a start, how about a program to dump store data and display it?

```
SCR   #1
0     (DUMP PROGRAM)
1     : TASK ;
2     : PLINE  CR DUP 5 U.R 8 0 DO
3              DUP C@ 3 .R 1+
4              LOOP ;
5     : PBLOCK CR 16 0 DO PLINE LOOP ;
6     : GETN QUERY INTERPRET ;
7     : DUMP HEX CLS ." Start Address ?"
8            GETN BEGIN PBLOCK CR KEY DROP
9            ?TERMINAL UNTIL ;
```

Set the program up in screen number 1, check and if necessary correct any errors. Then call FORTH 1 LOAD, and thereafter the word DUMP will call the program. In most BASICs, it would be much longer, to obtain the required format and to perform the hexadecimal conversions.

This is not laid out in formal FORTH style, but it will still load perfectly well. Note the use of TASK at the start, so that the program can be forgotten by FORGET TASK.

PLINE performs a newline, then duplicates TOS, which holds the current dump address. The copy is used to display the address in a five-position field (5 U.R) and then an eight-iteration loop is entered which again duplicates the address, reads the contents of the location defined, and displays the result in a three-position field. The address, on TOS again, is incremented, and the loop iterates. The overall result is a single line dump. It is in hexadecimal, because DUMP sets that condition.

PBLOCK repeats PLINE sixteen times, putting in an extra newline first.

GETN puts the required start address on TOS.

DUMP , the overall word for the program, sets HEX working, clears the screen, and puts up the invitation to input a start address. That might equally well have been included in GETN , to show the purpose of that word more clearly. The BEGIN-UNTIL loop then repeats until BREAK (CAPS SHIFT and 1) is pressed, but the action pauses after each block until a key is pressed. Note that the result of KEY is dropped, being unwanted.

Set the program up in screen number 1, check it and if necessary correct any errors. Then call 1 LOAD, and thereafter the word DUMP will call the program. In most BASICs, it would be much longer, to obtain the required format and to perform the hexadecimal conversions.

You may want to see text, rather than hex codes. Very well, we need to change PLINE as follows:

```
: TPLINE CR DUP 5 U.R SPACE 16 0 DO
            DUP C@ 32 MAX DUP 160 > IF 128 — ENDIF
            EMIT 1+ LOOP ;
```

As there is only one character per location dumped, we can put sixteen locations in a line. We don't want codes below 32 to be displayed, which would create mayhem, so we put 32 MAX , so that 32 is taken if the previous TOS was less. Codes above 160 are in the token area, so for these we subtract 128. We use EMIT , to output the ASCII code character instead of the hexadecimal value.

These two simple routines raise a number of important points. As we have changed the name of the definition from PLINE to TPLINE , it will not be called by PBLOCK , and we have to make a new version of PBLOCK called TPBLOCK and a new version of DUMP called TDUMP . GETN need not be duplicated if it had already been loaded.

If we had called the definition PLINE it would have made no difference, because PBLOCK would still refer to the original PLINE , the one which was defined before PBLOCK .

Using the editor, it is fairly simple to copy page 1 to page 2 and alter names as necessary, with the TPLINE definition at the top.

The next point concerns numbers. The loading was done with the system in the DECIMAL state, so the numbers are given in decimal form. HEX is not executed at line 7 of screen 1 , because it is being compiled. It

is worth noting that LIST sets the DECIMAL mode for its own purposes, so it is usually best to use decimal values in source code. If hex values are more convenient, you only need to insert HEX , outside a definition so that it will be executed, not compiled.

What next? Well, a notable absence from the dictionary is any kind of random generator, and most programs need one sooner or later. Here is one possibility:

```
0 VARIABLE SEED
: MODSEED SEED @ 75 U* 75 0 D+
          OVER OVER U< - - 1 - DUP SEED ! ;
: RND MODSEED U* SWAP DROP ;
: RAND -DUP 0= IF 23672 @ THEN SEED ! ;
```

RAND is a randomizer. 0 RAND will set SEED from the low byte of FRAMES . If TOS is non-zero, its value will be set in SEED.

RND is the actual random generator from the user point of view. It is used in the form n RND , which generates a number between 0 and n . MODSEED changes the value of SEED and creates the basic random number between 0 and 1 by which n is multiplied.

The actual working of the words is less important than the result they give. Here are some definitions that allow you to check the randomness:

```
0 VARIABLE ARRAY 62 ALLOT
: AGEN ARRAY SWAP 2 * + ;
: A! AGEN ! ;
: A@ AGEN @ ;
: ZERO 32 0 DO 0 I A! LOOP ;
: SETUP 0 RAND 1000 0 DO 32 RND DUP A@ 1+
SWAP A! LOOP ;
: DISP CR 32 0 DO I A@ 8 .R LOOP ;
: GRAPH CLS 32 0 DO I 8 * 0 PLOT I 8 * I A@
DRAW LOOP ;
: CHECK ZERO SETUP GRAPH ;
: DISCHECK ZERO SETUP DISP ;
```

An array of 64 bytes is set up. ZERO clears the array to zero. SETUP then generates a thousand random numbers in the range 0 to 32 (never actually reaching 32) and adds one to the nth array element every time the number n results. Two options are available for checking the randomness. DISCHECK calls ZERO SETUP DISPLAY , which displays the 32 numbers in the array in four columns. CHECK SHOWS THE RESULT AS A HISTOGRAM. A more stringent check can be made by increasing the size of the DO loop in SETUP.

These simple programs are worth analysing in some detail. You may find that you can improve on them. There are often alternative ways of doing the same thing in FORTH. You may find that the stack

manipulations are worth following through. Try combining the GRAPH and DISP routines to give you a graph with numbers.

Above all, experiment with some routines of your own. If you feel short of ideas, pick out a fairly simple BASIC program and set about converting it to FORTH . If the original is untidy, you will need to sort it out first, because FORTH programs have to be structured. That means they have to be laid out in a coherent way. In particular, the definitions must be in the right order.

In the next section, we will examine the approach to writing a somewhat larger program than those already given.

PLANNING A PROGRAM

The classic 'Towers of Hanoi' problem will be used as a basis for illustrating the way a program is planned. The problem involves a number of discs of different sizes arranged in three piles. Starting with all the discs on pile 0, they must be moved, one disc at a time, to pile 2, no disc ever being placed on top of one smaller than itself.

The overall structure of the program will be roughly:

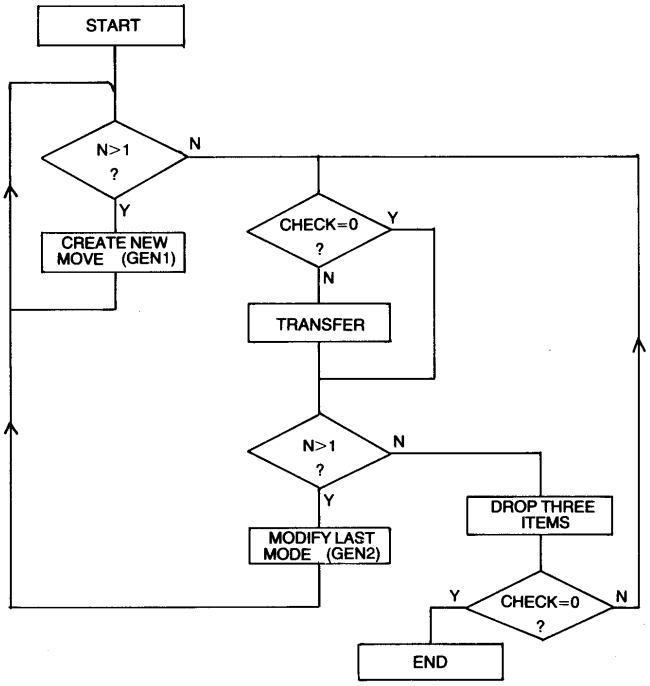
Display the title
Set up initial conditions
Move the discs
Ask if a repeat is wanted
If so, repeat from initialisation.

The first step is to decide the display format. It will be convenient to put the pile centres in the 6th, 17th and 28th columns. For graphic plotting, the horizontal coordinates will be approximately 43, 131, and 219. Approximately, because to get them in the centre of the pile numbers it would be necessary to create special characters for the numbers, since the normal numbers have no centre dot position.

If we make the nth disc 6*n dots wide, we will be able to handle up to twelve discs.

The working method must then be decided. The flow chart given here shows an approach which uses neither recursion nor empirical fiddles, and is defensible on the grounds of simple logic. Each move is specified by three stack items. TOS gives a disc number, and will be shown as n; 2OS holds the number of the pile from which the disc is to be taken, and will be shown as s; 3OS holds the number of the pile to which the disc is to be moved, and will be shown as d.

If n discs are to be moved, the initial top-of-stack items must be 2 0 n , asking for a move of disc n from source 0 to destination 2. That move can only be made if all the other discs are on pile 1, and the GEN1 function calculates 2 1 n-1 as a prior move. That, however, requires 2 0 n−2 as a prior move, and so on. GEN1 takes the original three top stack items d s and n and creates three more which are related to them as follows:



FLOW CHART FOR TOWERS OF HANOI MOVE FUNCTION

The new 3OS is the same as d, the original 3OS.

The new 2OS is calculated as 3 – d – s . Since the total of all the pile numbers is 3, this will identify the pile not involved in the original move.

The new TOS is one less than the original TOS.

When repetition of this process reduces TOS to 1, the actual moving process can begin. If there are five discs, the stack contains:

2 0 5 1 0 4 2 0 3 1 0 2 2 0 1

The 2 0 1 and 1 0 2 moves can be made, putting disc 1 on pile 2 and disc 2 on pile 1 . Before the third move, 2 0 3, can be made, an extra move is needed to clear pile 2. It is calculated by GEN2, which modifies an existing move, rather than adding a new one. The change is:

The new 3OS is calculated by 3 – d – s.

The 2OS is unaltered

TOS is decremented.

This, in the above context, gives 1 2 1 , moving disc 1 from pile 2 to pile 1.

If a move sequence is worked through on this basis, it will be found that the moves generated are those required to make the specified move. Note, however, that n can have two meanings. It may mean disc n, or it may mean n discs. The distinction is unimportant in practice.

GEN1 will obviously be a FORTH word. It will first have to copy 3OS, and a special sub-word will help in that respect:

```
: 3OVER >R OVER R> SWAP ;
```

TOS is passed to TORS while OVER copies 3OS as a new TOS. After the original TOS is restored, SWAP brings the copy of 3OS to 2OS. The original stack d s n becomes d s n d.

GEN1 can then be defined:

```
: GEN1 3OVER 3 SWAP — 3OVER — 3OVER 3OVER 1 — ;
```

Like all new words of any complexity, this should be checked by tabulating the stack changes:

	Stack
	d s n
3OVER	d s n d
3 SWAP —	d s n 3–d
3OVER —	d s n 3–d–s
3OVER	d s n 3–d–s s
3OVER	d s n 3–d–s s n
1 —	d s n 3–d–s s n–1

The three new items have been added without affecting the rest of the stack.

GEN2 may similarly be defined:

```
: GEN2 ROT DUP >R 3 SWAP — ROT —
R> SWAP ROT 1 — ;
```

Checking stack changes:

	Stack	
	d s n	
ROT	s n d	
DUP >R	s n d	TORS = d
3 SWAP —	s n 3–d	
ROT	n 3–d s	
—	n 3–d–s	
R>	n 3–d–s d	
SWAP	n d 3–d–s	
ROT	d 3–d–s n	
1 —	d 3–d–s n–1	

We now need to consider the TRANSFER function. It must erase disc n in its present position, and display it in its new position. In the process, the stack must not be altered, since GEN2 may need the data. If GEN2 is not involved, the three move-defining items will be dropped. It would no doubt be possible to achieve this by use of the stack alone, but it will be simpler to make use of some variables and an array. The variables are DSIZE (half width of the disc in dots), XPOS (vertical centre line of disc as a dot coordinate), and YPOS (vertical coordinate of bottom of disc).

The array is PILE , and it holds the number of discs on each pile. As there is only one array, we will set it up as follows:

```
0 VARIABLE ARRAY PILE 4 ALLOT
: AGET PILE SWAP 2 * + ;
: A! AGET ! ;
: A@ AGET @ ;
```

The address of element P will be put on TOS by P AGET, and the contents of the element can be accessed by P A@ . We can write to the element by X P A! , where X is the data to be written.

Since the numbers involved are small, we could have used a three byte array instead of a three word array, but the point is not too important.

By invoking the BASIC function OVER, we can use the same routine to both draw and erase discs:

```
: DDRAW 1 GOVER 6 0 DO (Draw six horizontal lines)
  XPOS @ DSIZE @ —      (x in PLOT = XPOS–DSIZE)
  YPOS @ I + PLOT       (y in PLOT = YPOS+I)
  XPOS @ DSIZE @ +      (x in DRAW = XPOS+DSIZE)
  YPOS @ I + DRAW       (y in DRAW = YPOS+I)
  LOOP 0 GOVER ;
```

We now need a routine to translate the three top of stack items to the required variable values:

```
: SETUP OVER OVER          (Stack d s n to d s n s n)
  3 * DSIZE !              (DSIZE = 3*n)
  DUP A@                   (Stack d s n s PILE(s))
  1 + 8 * YPOS !           (YPOS = 8*(PILE(s) + 1))
  88 * 43 + XPOS !         (XPOS = s*88 + 43)
  DDRAW ;
```

This would erase the top disc on pile s. To insert the same disc on pile d, e must first adjust the contents of the PILE array to take the move into account, and then call SETUP again with the stack temporarily changed from d s n to d s d n;

```
: ADJUST OVER AGET -1 SWAP +!(Decrement PILE(s))
  3OVER AGET 1 SWAP +! ;     (Increment PILE(d))
: TRANSFER SETUP ADJUST 3OVER
  SWAP SETUP SWAP DROP ;
```

Finally, to complete the flow chart, we need:

```
: CHECK 0 A@ 1 A@ + ;        (Returns 0 when move
                              complete)
```

The word MOVE , covering the flowchart, can now be defined:

```
: MOVE BEGIN CHECK WHILE
       BEGIN DUP 1 > WHILE
         GEN1
       REPEAT
       BEGIN
         CHECK IF TRANSFER ENDIF
         DUP 1 > NOT WHILE
         DROP DROP DROP
       REPEAT GEN2
     REPEAT    ;
```

Compare this with the flowchart.
We now require INIT, which will have the form:

```
: INIT ENQUIRE               (Ask how many discs)
  LINEDRAW                   (Draw the base line)
  PILEDRAW                   (Draw the initial pile)
  WAIT                       (Pause before beginning MOVE)
: ENQUIRE 2 0                (Initial destination and source)
  CLS 10 5 AT                (Clear screen, position text)
  ." How many discs ?"
  QUERY INTERPRET            (Get response)
  2 MAX 12 MIN ;             (Limit range)
: LINEDRAW 1 INK             (Blue line)
  CLS 20 0 AT                (Clear screen, position line)
    ."(5 RVSP) 1 (10 RVSP) 2 (10 RVSP) 3 (4 RVSP)"
```



```
  2 INK ;                    (Red discs)
```

In the above, 10 RVSP means ten reverse video spaces, Graphic 8.

```
: PILEDRAW 3 0 DO 0 I A! LOOP(Zero array elements)
  OVER OVER DUP 0 DO         (Stack d s n to d s n s n)
  0 AGET 1 SWAP +!           (Increment PILE(0))
  I — SETUP DROP OVER        (Draw disc n-I. Restore n)
  LOOP DROP DROP ;           (Discard unwanted items)
```

We need a function to respond to a request to run again:

```
: ASK 0 0 AT . " Again ?" KEY 89 - ;
```

This will return zero if key Y is pressed.
We also need a title function:

```
: TITLE CLS 10 6 AT . "THE TOWERS OF HANOI " WAIT ;
```

All that remains is to define the top level function:

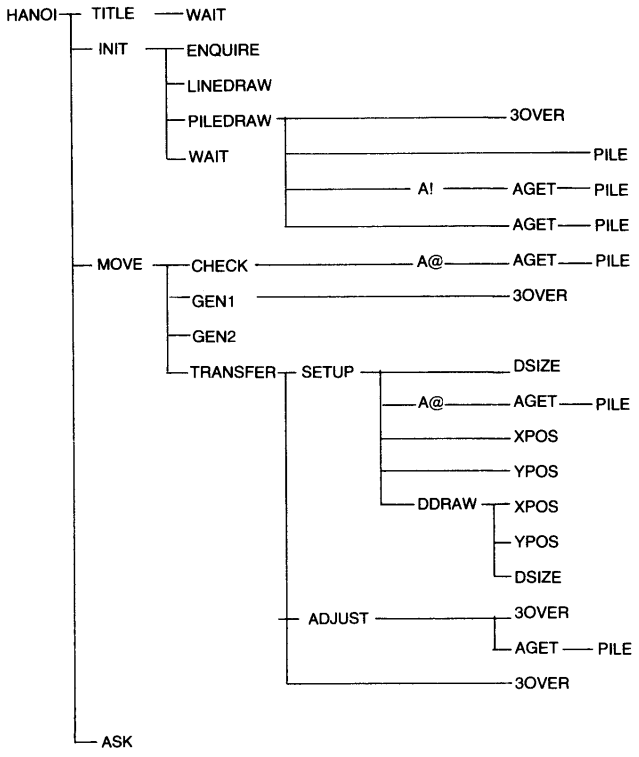
```
: HANOI TITLE BEGIN INIT MOVE 0 INK ASK UNTIL ;
```

We can now construct a hierarchy of functions, showing how each stands in relation to the others, and this will serve as a basis for deciding the order in which they should be compiled:

Definitions in the right hand column must be set up first, then column by column to the left. The order is not completely rigid, providing the necessary priorities are observed. A possible layout on four screens is shown in the attached listing.

This is a moderately complex program, mainly because of the commitment of the stack to the move data. It was chosen as illustrating a wide range of techniques, without becoming too abstruse.

It is always wise to save the screens on tape before actually running such a program. A slight error in entry can cause lockup or some other malfunction, and all stored data may then be lost.




```
SCR # 1
0 ( THE TOWERS OF HANOI: 1)
1 : TASK ;
2 0 VARIABLE DSIZE
3 0 VARIABLE XPOS
4 0 VARIABLE YPOS
5 0 VARIABLE PILE 4 ALLOT
6 : 3OVER >R OVER R> SWAP ;
7 : AGET PILE SWAP 2 * + ;
8 : A! AGET ! ;
9 : A@ AGET @ ;
10 : DDRAW 1 GOVER 6 0 DO
11         XPOS @ DSIZE @ -
12         YPOS @ I + PLOT
13         XPOS @ DSIZE @ +
14         YPOS @ I + DRAW
15         LOOP 0 GOVER ; -->

SCR # 2
0 ( THE TOWERS OF HANOI: 2)
1 : SETUP OVER OVER
2           3 * DSIZE !
3           DUP A@
4           1 + 8 * YPOS !
5           88 * 43 + XPOS !
6           DDRAW ;
7 : ADJUST OVER AGET -1 SWAP + !
8            3OVER AGET 1 SWAP + ! ;
9 : WAIT 20000 0 DO LOOP ;
10 : ENQUIRE 2 0 CLS 10 5 AT
11            ." How many discs?"
12            QUERY INTERPRET
13            2 MAX 12 MIN ;
14 : CHECK 0 A@ 1 A@ + ;
15 -->

SCR # 3
0 ( THE TOWERS OF HANOI: 3)
1 : LINEDRAW 1 INK CLS 20 0 AT
2 ." _____1_________2_________3____" 2 INK
3 : PILEDRAW 3 0 DO I A! LOOP
4    OVER OVER DUP 0 DO
```

```
5     0 AGET 1 SWAP +!
6     I - SETUP DROP OVER
7     LOOP DROP DROP ;
8 : GEN1 3OVER 3 SWAP - 3OVER
9          - 3OVER 3OVER 1 - ;
10 : GEN2 ROT DUP >R 3 SWAP -
11    ROT - R> SWAP ROT 1 - ;
12 : TRANSFER SETUP ADJUST 3OVER
13       SWAP SETUP SWAP DROP ;
14 : INIT ENQUIRE LINEDRAW
15          PILEDRAW WAIT ; -->

SCR # 4
0 ( THE TOWERS OF HANOI: 4)
1 : TITLE CLS 10 0 AT
2 ." THE TOWERS OF HANOI" WAIT ;
3 : MOVE BEGIN CHECK WHILE
4            BEGIN DUP 1 > WHILE
5              GEN1
6            REPEAT
7            BEGIN
8              CHECK IF TRANSFER ENDIF
9              DUP 1 > NOT WHILE
10           DROP DROP DROP
11           REPEAT GEN2
12         REPEAT ;
13 : ASK 0 0 AT ." Again ?" KEY 89 - ;
14 : HANOI TITLE BEGIN INIT
15    MOVE 0 INK ASK UNTIL ;
```


PART VI:
Compiling

This part covers the words which are used in compiling, and explains some of the less obvious ways in which such words can be used.

COMPILING

When the system is in compiling mode, most of the words that it finds in the input stream are incorporated in a new dictionary entry. Each word is located by searching through the dictionary, and the address of its Code Field is added to the new Parameter Field. Some words, however, are treated differently.

Words which have the 'precedence bit' in their length byte set true are not compiled, they are executed. For example, the definition of the word DO is:

```
COMPILE (DO)
HERE
3
```

The first line enters the Code Field address for (DO) in the Parameter Field, and HERE then saves the contents of the dictionary pointer for use later, the contents being held on the stack. Then 3 is put on the stack, this being the check number for a DO — LOOP combination.

The definition of LOOP is:

```
3 ?PAIRS
COMPILE (LOOP)
BACK
```

Another 3 is put on the stack, and ?PAIRS compares this with 2OS, which should contain the 3 left by DO . If it does not, then some error has been made, such as a failure to balance another type of loop format.

If the comparison shows a match, the Code Field address for (LOOP) is added to the new Parameter Field, and then BACK is called. This again

calls HERE to put the contents of the dictionary pointer on the stack, and performs a subtraction. The result is written to the new Parameter Field, being the jump span needed to reach a point immediately after the (DO) entry.

Similar procedures apply with other branching and looping combinations.

Colon and semicolon also have the precedence bit set. Colon first checks that the system is in direct execution mode, since it is not permissible in compiling mode. Then the current stack pointer is saved in CSP as a reference, and CONTEXT is set equal to CURRENT to ensure that the right value of dictionary pointer is used, putting the new word into the selected vocabulary. CREATE then sets up the first two fields of the new entry. The Code Field is set to point to a short machine code routine which establishes conditions for interpretation of the Parameter Field by saving the current Instruction Pointer value on the Return Stack and setting the pointer to a new value, the start of the parameter field. The Instruction Pointer is used to read the link addresses and other data required for execution.

Semicolon reverses the process. It expects to find that the stack pointer has returned to the value saved in CSP by colon, and issues an error report if this is not the case. Then ;S is compiled into the last position in the Parameter Field, the 'smudge' bit is set to the valid state, and direct execution is resumed. When executed, ;S removes the return link address from the Return Stack and sets it in the Instruction Pointer.

Such processes can be forgotten for most purposes, but they can be important if you want to play tricks in setting up new words. Some of the definitions given in Appendix A look simple enough, but if you try to compile them directly you may find that the system objects.

Take, for example, the words formed by square brackets. The colon process uses] to switch to compiling mode, and semicolon uses [to return to direct execution. All these words do is to change the contents of the STATE variable, but they are still very useful, allowing an interlude of direct execution in the middle of a compiling process. This might be used to calculate a critical value that depends on a word compiled earlier.

Colon also uses ;CODE , which compiles (;CODE) . When executed, (;CODE) sets the Code Field of the most recently compiled word to point to a machine code routine which follows ;CODE . In the case of colon, the actual machine code is called by all definitions that have a real Parameter Field. If an assembler were available, it would be entered automatically to set up the code in question. In its absence, the words , and C, can be used. Comma stores TOS in the next dictionary location, while C, stores only the low byte of TOS. By using these two words, it would be possible to set up a new definition directly, but that would be rather a waste of time and effort. However, it is sometimes useful to use the words to set up specific links or data within a Parameter field, or data within an array. In the latter case there is no need to use ALLOT to make

room for the array, as comma and C, both advance the dictionary pointer to the first empty location.

Perhaps the oddest word in FORTH is the one with a Name Field that contains C1 80 . A length of 1 letter, in other words, and that letter is conveyed by 80H — 80H = 0 . A null code is used to terminate screens and buffers, and it will stop interpretation. If a null is accidentally inserted in the middle of a screen, it will be found impossible to interpret the screen beyond that point.

LATEST puts on the stack the Name Field address of the most recently defined dictionary entry. This is used by IMMEDIATE , which sets the precedence bit for that entry, allowing words to be defined that will execute in compiling mode.

On the other hand, [COMPILE] will cause the following word to compile, even if it has a true precedence bit. The possibilities which this opens up are slightly mind-boggling at first. It is possible to visualise words that will set up DO loops or other looping or branching functions. However, the classic example is:

```
: XXXX [COMPILE] FORTH ;
```

Without [COMPILE], the word FORTH would execute when XXXX was compiled, leaving the definition of XXXX as 'no action'. As it stands, the colon definition will be equivalent to FORTH , and executing XXXX will select the FORTH vocabulary. In a sense, [COMPILE] defers the action of the word on which it acts, making it effective at execution time, rather than at compile time.

Next we have two important words which are rarely seen: LITERAL and DLITERAL. They are called when INTERPRET finds a numeral in the input stream. In direct mode, they do nothing, since the numeral is put on the stack and remains there. In compiling mode, LITERAL compiles LIT and then uses comma to transfer the numeral from the stack to the Parameter Field. When the field is interpreted, LIT transfers the numeral back to the stack. DLITERAL acts in a similar way, but compiles LIT , then TOS, LIT again, then 2OS. As the D suggests, it is intended to deal with double numbers.

CREATE sets up the first two fields of a dictionary entry, using WIDTH to check that the length of the name is within valid limits. WIDTH is a variable, normally set to 31.

The word 'tick' is sometimes useful. In the form:

```
' XXXX
```

in direct execution it puts the Parameter Field address of XXXX on the stack. In compiling, it transfers the address as a LIT entry in the new Parameter Field.

At this point, it is necessary to call a halt. The words which are more likely to be of direct use in the compiling process have been described, with one exception, which is complex enough to merit a section of its own.

A fascination and a frustration of FORTH is that there are many slightly esoteric tricks that you can play with it, so many that it would be pointless to try to explain them all. The programs given in this book provide some illustrations, but by no means cover the full range of possibilities. You have to invent your own solutions, since they are probably aimed at solving problems that are not common enough to have generated ready-made answers.

<BUILDS . . . DOES>

It has been said that the <BUILDS . . . DOES> combination is both one of the most powerful facilities in fig-FORTH and also one of the most difficult to explain. It is a tool used to perform complex processes which create other tools. Here is an example:

```
: ARRAY <BUILDS 2 * ALLOT DOES> SWAP 1 —
  2 * + ;
```

This creates a new word ARRAY, which can be used as follows:

```
8 ARRAY HEAP
```

This will create a one-dimensioned array called HEAP with space for nine two-byte words, the 8 specified plus one from the basic variable, as in the simpler methods of array creation already discussed. All this is done by the 2 * ALLOT following <BUILDS , but there is more. If we call 5 HEAP , the words following DOES> will decrement 5 to 4 , double the result, and add the array base address. Thus n HEAP @ will read the nth element of the array, n HEAP ? will display that element, and m n HEAP ! will set the element to m.

The word ARRAY remains available as a tool to establish further arrays of a similar kind. It will be seen that it saves a lot of trouble in relation to the methods described earlier.

The point to grasp is that the words following <BUILDS are used to determine the basic form of the new word, while the words following DOES> determine the process which will be associated with it.

Here is a more complex example which handles two-dimensional arrays and checks the subscripts to ensure that they are within the bounds defined by the given dimensions. Three new words are involved, and compilation is carried out with hexadecimal notation selected for the sake of convenience and clarity.

```
HEX
: OFB . " Out of array bounds. " SP! QUIT ;
: OFBCK >R ROT ROT DUP R> DUP FF AND ROT < IF
OFB ENDIF 100 /
  >R OVER R> DUP ROT < IF OFB ENDIF ;
: ARRAY2 <BUILDS FF AND DUP C, SWAP FF AND
```



```
DUP C, * 2 * ALLOT DOES> DUP @ OFBCK SWAP
1 — * +  + ;
```

With these words defined, an array is created, with dimensions x and y, by:

```
x y ARRAY2 MATRIX
```

The dimensions are stored at the start of the array, and the number of bytes reserved is 2*x*y . This is all determined by the words following <BUILDS . The words following DOES> take effect when MATRIX is executed. Note that x and y are stored by C, and they therefore occupy a byte each. DUP @ puts them both on TOS, while preserving the subscripts behind them, OFBCK is then called to compare the subscripts with x and y , and to report if the subscripts are out of range. Work through the stack changes, but remember that hexadecimal notation is used, so that a division by 100 is really a division by 256, which will bring the upper byte of TOS into the lower byte position.

These examples illustrate the use of <BUILDS . . . DOES> in a particular context, but the general concept should be clear enough. Broadly speaking, <BUILDS establishes a constant, and the subsequent code extends the compilation. DOES> uses a standard segment of machine code to perform a little sleight of hand so that the words which follow it are appended to the entry created by <BUILDS .

There are other ways of creating compiling tools, but the best way to learn about these is to try various combinations and check on the definitions that result.

READING THE DICTIONARY

It is sometimes useful to be able to see how the code you have defined has been set up in the dictionary. For this, a simple program is:

```
: SCAN BEGIN CR
    DUP @ 2+ NFA ID.
    KEY CASE
    68 OF 2+ DUP @ U. 2+ 0 ENDOF
    69 OF 1 ENDOF
    70 OF 2+ 0 ENDOF
    ENDCASE UNTIL ;
: $ -FIND QUERY INTERPRET
    IF DROP CLS SCAN
    ELSE . " NOT FOUND " ENDIF ;
```

Input $ followed, after a space, by the name of the definition you want to examine, and press return twice. The first name will appear at the top of the screen. Press F to get the next name, unless the first is BRANCH , OBRANCH , LIT , (LOOP) or (+LOOP) . All these have data words following, and these and the next word can be obtained by pressing D .

To end, press E . The end of the definition is usually marked by ;S . If you continue after that, all sorts of strange things may happen.

It would be nice, of course, to make the special names call up the appropriate action, so that it was all automatic, but there would still be a problem with (.") , which is followed by text. To continue, it would be necesary to scan past the text, which would not be too difficult. If you want to use this routine a lot, you should be able to work out how to make it more automatic.

In considering jump spans, remember that they represent a number of bytes, and each word or data link occupies two bytes. A branch span of ten will thus go ahead five entries.

One small oddity. It is necessary to load the above programs in DECIMAL , as the three ASCII codes 68, 69 and 70 are in that form. The program, on the other hand, is best run in HEX . How would you ensure that these requirements were met?

MORE COMPILING

The normal compiling system automatically keeps watch on the progress of compilation, and when you try less direct methods you must be prepared to do the same. You need to examine the detailed structure of the words you use, and that is one of the reasons why APPENDIX A has been provided.

In the section of the Appendix covering the EDITOR vocabulary, you will find a slight oddity. The words R and I have been redefined, but subsequent definitions use the earlier meanings. To achieve this sort of trick, it is only necessary to insert the word FORTH at a suitable point, whereupon the FORTH versions of the words will be compiled. The word EDITOR will later ensure that the definition is continued on the basis of EDITOR words already defined.

To set up a special vocabulary, you must first establish the word SPECIAL (or whatever you want to call it) by the entry:

```
VOCABULARY SPECIAL
```

This puts an entry in the FORTH vocabulary. To select the SPECIAL vocabulary, you need:

```
SPECIAL DEFINITIONS
```

Words compiled thereafter will be put into the SPECIAL vocabulary.

Why have a special vocabulary ? One good reason is that too large a vocabulary slows down compiling, and access to a specialised vocabulary can be much faster. You may also want to use the same word for different purposes, as in the EDITOR.

Having compiled a program, you may feel that it is annoying to have to recompile it whenever you want to run it. There is no need for that. You

can save the whole of the extended dictionary, including the original vocabulary, in one operation.

To do this, you first need to know how much to save, and SIZE will provide that information. You then need to alter the locations which hold the initialising data, so that the extension of the dictionary will be taken into account.

The first step is to make sure you are not in a special vocabulary, by putting in:

```
FORTH DEFINITIONS
```

Then, as a matter of personal taste, it seems easiest to set HEX , since we are dealing with addresses and displacements, which are usually more familiar in hexadecimal.

LATEST holds the name field address of the latest dictionary entry, and all searches start from that address, which must be set in 5E4C:

```
LATEST OC +ORIGIN !
```

Next, the dictionary pointer must be copied into 5E5C and 5E5E:

```
HERE 1C +ORIGIN !
HERE 1E +ORIGIN !
```

If we want to have our program protected, we must move FENCE to the same place:

```
HERE FENCE !
```

Finally, we need:

```
' FORTH 8 + 20 +ORIGIN !
```

This sets up the PFA of the FORTH word, plus 8, in 5E60.

The amended program can then be saved, using line 9 of the vestigial BASIC program, suitably altered to take account of SIZE. After verification, the program is ready to be loaded in place of the original FORTH tape, when it should perform as it did before saving.

This technique is particularly useful if you have created your own pet extensions to FORTH for general use.

PART VII:
Programming Techniques

This part deals with some of the less obvious techniques which can be used in writing programs.

NON-INTEGER ARITHMETIC

If you find integer arithmetic too limiting, look at the definition of DRAW in Appendix A (Dictionary Word No. 447). It works in increments of 0.0000152.

The method used is to treat a double number as being scaled by a factor of 1/65536. For example, the word LASTY is read from the BASIC wordspace, and is then converted into a double number by adding a zero lower word — no, not an upper word, a lower word. In integer terms, the result is LASTY * 65536 , and this is stored in the double variable Y1. The scaled version of LASTX is similarly stored in X1.

The required position coordinates are then compared with LASTX and LASTY — in their unmodified form — to determine how many steps will be required to draw the line. It will be either ABS(X — LASTX) or ABS(Y — LASTY) , whichever is greater. The nominal increments in the X and Y values are then calculated by dividing 65536 times the overall changes by the number of steps. M/MOD is used, with alternative routes for positive and negative differences, and the results are stored in INCX and INCY as double numbers.

Since both the current x and y values and the related increments have been multiplied by 65536, it is valid to add the increments to the original values repeatedly, to define the new x and y values, but only the upper bytes of x and y need be read, these being the integer parts of the numbers.

This simple but effective approach can be used where the final result is required in integer form. If decimal places are wanted, rather more complex processes are needed.

Fixed point working is fairly straightforward in concept, but involves some complications in the details. The scheme is that a fixed number of decimal places is adopted, say two. In this case, all numbers will be multiplied by 100, as a scaling factor. When a number is input, a decimal point will normally be included, and the resulting value of DPL will be noted and used to perform any necessary correction. For example, if 23.02 is input, DPL = 2, and there is no correction. With an input of 23.0, on the other hand, a multiplication by ten would be required.

The modified numbers can be added or subtracted without difficulty, since they have the same scaling, but multiply and divide routines must be modified. After two numbers have been multiplied together, the result must be divided by 100, otherwise that multiplier will be applied twice. Before a division, the dividend must be multiplied by 100, or the scaling factor will be lost.

Output of the final result requires only that the decimal point be placed in the correct position.

A wary eye must be kept on possible overflow. A simple multiplication of 10 by 20 becomes a multiplication of 1000 by 2000, and the result before correction is 2,000,000 , corrected to 20,000. Double numbers will clearly be the rule, and even these will run out of steam with an uncorrected result of about 2,000,000,000, giving a corrected value of 200,000.00. If a scaling factor greater than 100 were used, the limitation on maximum value would be increased.

The answer lies in triple words, which in turn call for a whole new family of manipulators and operators.

Alternatively, a form of floating point can be considered. This will inevitably be much slower than integer working, but opens up many possibilities.

A single-precision system can be based on double numbers, with the top eight bits reserved as the exponent and the remaining 24 bits serving as the mantissa. This corresponds to a typical BASIC single precision system. The mantissa is always 'normalised' so that its most significant bit is true, the exponent being decremented when the mantissa is shifted left, and incremented when the mantissa is shifted right. There must be two sign bits, one for the exponent and one for the mantissa. Since the most significant bit of the mantissa is always 1 in truth, it is sometimes used to hold the mantissa sign bit.

For addition and subtraction, it is necessary to match the exponents of the two numbers concerned, by increasing the smaller exponent and decreasing the associated exponent. For multiplication, the exponents are added and the mantissas multiplied together. In one way and another, floating point is complex, and not to be approached light heartedly.

Those who have studied the inner workings of Spectrum BASIC may be able to make use of the floating point routines which it contains, but the approach to that is much too complex to be studied here. It is worth

bearing in mind, however, that the BASIC ROM contains many routines that might be incorporated in FORTH if the right technique can be found.

In particular, access to the exponential and trigonometric functions would be appreciated. It is possible to perform such functions in FORTH, but only in a relatively crude manner. Square roots may be calculated by iterations of:

$$S_{n+1} = \tfrac{1}{2}(X/S_n + S_n)$$

where S_0 is an arbitrary value and S_n is an approximation to the square root of X. Where S_n is too large, it is roughly halved for each iteration. Where it is too small, it becomes much larger. As a rough approximation, k + 2 iterations will usually produce a reasonably accurate result for numbers up to 2^k , if S_0 is made equal to $2^k/4$.

An interesting point arising from this is that trigonometric calculations can sometimes be replaced by a technique based on vectors. The idea is that x and y coordinates can be combined as a single vector R by the expression:

$$R = \sqrt{a^2 + b^2}$$

The effective angle can then be expressed by a/b , and this, with R , specifies the resultant vector completely. Vectors can be added or subtracted by adding or subtracting the a and b values, and it is possible by such means to perform such operations as defining a circle — but non-integer calculations are essential for that.

A FINAL PROGRAM

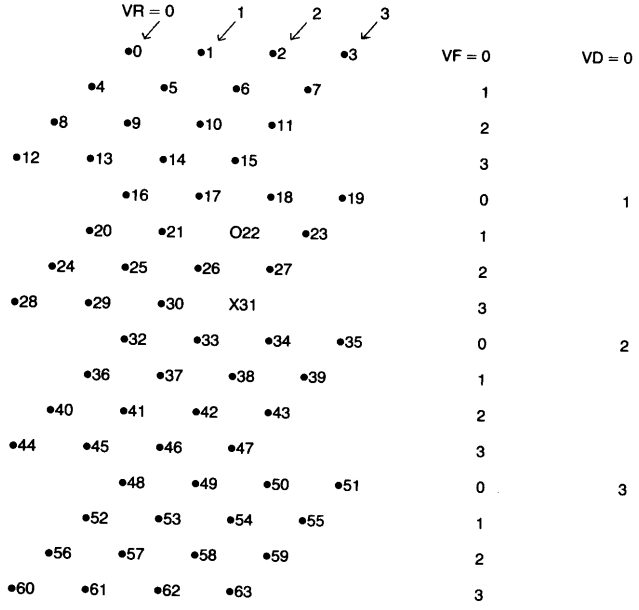
As a final illustration of some assorted FORTH techniques, here is a final fairly large program. It plays three-dimensional noughts and crosses with considerable success, though it can be beaten if you know how, and have enough concentration to keep a wary eye on what the computer is doing.

Candidly, it is a personal favourite, having been written and rewritten for a whole host of computers, including some that were intended for rather more serious duties. In machine code, the necessary decisions are usually made in a fraction of a second, but in BASIC they may take well over a minute. The FORTH version takes about fifteen seconds, which is just about acceptable, and is at least five times faster than BASIC.

The reason why so much time is needed is the sheer magnitude of the calculations involved. Firstly, the program size is cut down by dispensing with the usual table of possible lines within the cubic matrix, and the computer is asked to calculate these lines from set rules. For any given position, either four or seven lines are involved, and there are sixty-four positions. Four entries have to be checked to assess the state of each line.

Once the content of a line has been worked out and expressed by a code number, the priority level for the line must be looked up and added to the total for the position being studied, this being repeated for each line passing through that position. Finally, the highest priority must be found.

The diagram of the display will help to explain some of the variables. The numbers 0 to 3 are used for the coordinates, and some players may prefer to use 1 to 4 , in which case a simple change to the INPROC routine will be needed. The numbers are entered in the order VD , VF , and VR . The position indicated will be marked on the display, and you are given a chance to change your mind before the entry is actually made.



OX 3 Display Format and Coordinates

CH is the position number, used mainly for array reference and for marking the latest entry. It is equal to 16 * VD + 4 * VF + VR .

Next come four status flags, a temporary hold for priority values, and a variable OL which is used in finding the highest priority.

There are four arrays. AE is a byte array holding the 0 and X entries, AP is a word array holding the priorities, AX is a hold for the positions in a line, and AW is the priority index, holding the weightings for each line configuration, showing the relevant priority. Note how AW is set up, rather in the way a BASIC DATA statement is input, but with spaces on either side of the commas.

INIT clears flags and arrays, and DISP puts up the display. POSCAL works out the values of VD , VF , and VR corresponding to the value of CH , and is used by CURSOR , which marks the latest entry as defined by CH , by putting up a reverse video space in OVER mode. As CURSOR always follows DISP they are combined as PLAY .

Two display positions are established by P1 and P2 , and DROP 2 is defined to discard 2OS.

The input routines begin with GETN , which waits for a key depression, displays the relevant character, and attempts to convert it as a number, base ten. Success puts a true flag on TOS, the number on 2OS, and the input code on 3OS. If the input is not numeric, TOS holds a false flag and 2OS holds the code.

INPOS sets display position P1 and calls GETN . If the flag returned on TOS is false, the code then revealed on TOS is checked to see if it is 82 ('R'), which calls for a restart. In this case, BE is set, and TOS = 1 to make the routine drop out at UNTIL . If the input is numeric, the code is dropped by DROP2 . and INPROC is called to obtain the rest of the input, checking its validity and offering a chance for a change of mind before actually making an O entry. If the 1-4 input range is wanted, instead of 0-3, the position calculation in INPROC should be changed to:

ROT 1 − 16 * + 1 − SWAP 1 − 4 * + DUP AE + C@

INPOS repeats until TOS holds 1 when UNTIL is reached.

Next come the routines which calculate priorities. CALC3 is called with array AX set to the positions in a given line. It checks the contents of each position in turn, adding 1 for an 'O' entry, 5 for an 'X' entry. This produces a number characteristic of the line format. For example, a line containing OOX. would be given the number 7 . Looking up the seventh entry in AW gives a priority of zero, because the line is 'dead', containing both O and X entries. A line containing OOOO, on the other hand, would have a number 4 to represent it, and the fourth entry in AW is again 0, because it is too late to worry about priorities. The game is won. If the line contains XXX. its number is 15, and that gives a priority of 896, urging an X entry to win.

In fact, this priority is pre-empted, to save time. If the line number is 15 , the empty position is located and filled, and the BX flag is set, so that the

X win is declared immediately on return. Similarly, a line with a number 4 sets BO to declare an O win.

In general, the priority for the line is added to VP , which accumulates the priorities for all lines passing through a particular point.

CALC2 works out the positions in a line, using data provided by CALC1 . Each time CALC2 is called, there are five numbers on the stack which express the formula for a given line. For example, a vertical line has the formula:

4*VF + VR + 16*n

where n = 0 to 3.

Since the calculating routines are complex and prone to error in setting up the source code, it is useful to define a temporary form of CALC3 as follows:

```
: CALC3 CR 4 0 DO I 2 * AX + ? LOOP ;
```

This should be defined after the real CALC3 but before CALC1. It will display a list of four or seven sets of positions forming lines for each value of CH . This allows a check to be made on the correctness of the line selection system.

An important point about these routines is the use of I' in CALC2 . When this routine is called by CALC , an extra entry is put on the Return Stack, and it is necessary to dig down to 2ORS to find the loop count.

The calculating routines are called by CALCP , which scans through all the CH values from 0 to 63, checking the priority against each value and entering it in array AP. The routine also checks for true states of BO and BX , and if either is found the routine drops out, also dropping through the subsequent SELECT routine.

Since CALC3 sets a zero priority for any occupied location, it might seem that time could be saved by skipping the call to CALCP for such locations. That seems a good idea, but it isn't, because it would mean that an OOOO line would never be scanned, and an O win would never be found . . .

The SELECT routine scans array AP , setting OL to the value read if it is the highest yet found, and setting CH to match. Ideally, the scan should start at a random point, and this is simulated by putting 56*VR on the stack, but a true random number would be better. When the scan is complete, the entry identified by CH is set up. If OL has remained at zero, all lines are 'dead', and BD = 1 to mark the game as drawn.

Finally, the overall OX3 routine has to link all these routines together. There are two main loops, one taken for each game, to include INIT, and an inner loop for each game turn. After PLAY INPOS , a check is made for the end flag BE , and if it is true CALCP and SELECT are skipped. The CASE structure is used to check the four flags, issue reports, and alter the stack values to determine whether the inner or outer loop should be taken.

No FORTH program is ever perfect, and this one is no exception. It has been used to make a number of special points, and you may enjoy attempting to improve it. In that case, spread it out well in the screens, to make changes easy, but don't be too generous. You have 150 lines in ten screens, and as laid out here you would need about 190 lines. You could use two sets of screens, but that could be inconvenient.

A Three-Dimensional Noughts and Crosses Program

```
( OX3 )
: TASK ;
0 VARIABLE VR                          (Left to right coordinate)
0 VARIABLE VF                          (Back to front coordinate)
0 VARIABLE VD                          (Top to bottom coordinate)
0 VARIABLE CH                          (Position number)
0 VARIABLE BO                          (O WIN flag)
0 VARIABLE BD                          (DRAW flag)
0 VARIABLE BE                          (End flag)
0 VARIABLE BX                          (X WIN flag)
0 VARIABLE VP                          (Priority hold)
0 VARIABLE OL                          (Highest priority found)
0 VARIABLE AE 62 ALLOT                 (Play state array)
0 VARIABLE AP 126 ALLOT                (Priority array)
0 VARIABLE AX 6 ALLOT                  (Line definition array)
1 VARIABLE
AW 2 , 16 , 256 , 0 , 4 , 0 , 0 , 0 , 0 , 32 , 0 , 0 , 0 , 0 ,
    896 , 0 , 0 ,0 , 0 , 0 , 0 , 0 ,    (Priority basis)
: INIT AE 64 ERASE AP 128 ERASE
      0 BO ! 0 BD ! 0 BE ! 0 BX ! ;     (Clear arrays, flags)
: DISP CLS 0 4 0 DO                    (Display state of play)
            4 0 DO
              CR 4 1 - 2 * SPACES
              4 0 DO
                DUP AE + C@ DUP
                IF EMIT
                ELSE ." ." DROP
                ENDIF
                SPACE SPACE
              LOOP
            LOOP CR
          LOOP ;
: POSCAL CH @ 16 /MOD VD ! 4 /MOD VF ! VR ! ;
: CURSOR POSCAL VD @ 5 * VF @ + 1+     (Vertical position)
                VR @ 3 * VF @ 2 * - 8
                                       (Horizontal position)
```

```
                AT 1 GOVER 2 INK
                                    (Position cursor)
                ." ■" 0 GOVER 0 INK ;
: PLAY DISP CURSOR ;
: P1 9 18 AT ;                      (Text position 1)
: P2 21 0 AT ;                      (Text position 2)
: DROP2 SWAP DROP ;                 (Drop 2OS)
: GETN KEY DUP EMIT DUP 10 DIGIT ;
                                    (Get a digit)
: INPROC GETN
          IF DROP 2 GETN            (If numeric discard code)
            IF DROP2                (If numeric discard code)
              ROT 16 * + SWAP
              4 * + DUP AE + C@ (Calculate position, check)
            IF P2 ." Position Taken     "
            ELSE CH ! PLAY P2       (If free, set CH, display)
                ." Is that right?   "
                KEY 89 - 0=         (Leave 0 if entry not good)
              ENDIF DUP             (Duplicate flag)
              IF 79 AE CH @ + C!    (If 1 make 0 entry)
                P2 20 SPACES        (and clear message)
              ENDIF
            ELSE 0                  (Not numeric. Leave 0)
            ENDIF
          ELSE 0                    (Not numeric. Leave 0)
          ENDIF ;
: INPOS BEGIN
          P1 GETN
            IF DROP2 INPROC         (If numeric discard code)
            ELSE 82 - 0=            (Else check for 'R')
              IF 1 BE ! 1           (If 'R' set BE. TOS = 1)
              ENDIF 1               (If other letter TOS = 1)
            ENDIF
        UNTIL ;                     (Repeat if TOS = 0)
: CALC3 0 4 0 DO
            I 2 * AX + @            (Read position number)
            AE + C@                 (Read contents)
            79 - 0=                 (Check for 'O')
            IF SWAP 1+ SWAP         (If 'O' increment TOS)
            ENDIF
            88 - 0=                 (Check for 'X')
            IF 5 +                  (If 'X' add 5)
            ENDIF
          LOOP                      (Line key number on TOS)
```



```
            AE CH @ + C@ 0=         (Is position CH free?)
            IF DUP 2 * AW + @ VP + ! (If so, add priority to VP)
            ENDIF
            DUP 4 - 0=              (Check for OOOO line)
            IF 1 BO !               (If OOOO set BO)
            ENDIF
            15 - 0=                 (Check for XXX. line)
            IF 8 0 DO               (If XXX. identify free position)
              I AX + @ AE + C@ 0=
              IF AX I + @ CH !      (If free set CH)
                99 CH @ AE + C!     (and enter 'X')
              ENDIF
              2+LOOP
              1 BX !                (Set BX)
            ENDIF ;
: CALC2 VD @ * SWAP                 (Calculate position)
          VF @ * + SWAP
          VR @ * + SWAP
          I' * + +
          AX I' + 2 * + ! ;
: CALC1 POSCAL
          4 0 DO 0 1 0 4 16 CALC2 LOOP CALC3
          4 0 DO 0 4 1 0 16 CALC2 LOOP CALC3
          4 0 DO 0 16 1 4 0 CALC2 LOOP CALC3
          VF @ VR @ - 0=
          IF 4 0 DO 0 5 0 0 16 CALC2 LOOP CALC3
            VD @ VF @ - 0=
            IF 4 0 DO 0 21 0 0 0 CALC2 LOOP CALC3
            ENDIF
          ENDIF
          VD @ VR @ - 0=
          IF 4 0 DO 17 0 4 0 CALC2 LOOP CALC3
            VD @ VF @ + 3 - 0=
            IF 4 0 DO 12 13 0 0 0 CALC2 LOOP CALC3
            ENDIF
          ENDIF
          VD @ VF @ - 0=
          IF 4 0 DO 0 20 1 0 0 CALC2 LOOP CALC3
            VD @ VR @ + 3 - 0=
            IF 4 0 DO 3 19 0 0 0 CALC2 LOOP CALC3
            ENDIF
          ENDIF
          VD @ VR @ + 3 - 0=
          IF 4 0 DO 3 15 0 4 0 CALC2 LOOP CALC3
            VF @ VR @ - 0=
```

```
            IF 4 0 DO 15 11 0 0 0 CALC2 LOOP CALC3
            ENDIF
          ENDIF
          VF @ VR @ + 3 - 0=
          IF 4 0 DO 3 3 0 0 16 CALC2 LOOP CALC3
          ENDIF
          VD @ VF @ + 3 - 0=
          IF 4 0 DO 12 12 1 0 0 CALC2 LOOP CALC3
          ENDIF ;
: CALCP 1 0 CH ! 0 VP !                  (TOS = 1,CH = 0,VP = 0)
          BEGIN
            CALC1 BO @ 0=                (Calculate priority)
            IF VP @ AP CH @ 2 * + !      (If BO = 0 set priority in AP)
              0 VP !                     (Zero VP)
              BX @                       (If BX = 1 change flag to)
              IF DROP 0 64               (1 and set endcount.)
              ELSE CH @ DUP 1 + CH!      (Increment CH, leave copy)
              ENDIF
              64 - 0=                    (If CH = 64 leave 1)
          ELSE DROP 0 1                  (If BO = 1 change stack)
          ENDIF                          (If TOS = 0 repeat, else exit)
        UNTIL ;                          (with flag for SELECT)
: SELECT BEGIN WHILE                     (Drop through if TOS = 0)
             0 OL !                      (Zero 'highest priority')
             VR @ 56 *                   (Quasi random number)
             64 0 DO
               2+ 128 MOD                (Advance pointer on stack)
               DUP AP + @ OL @ >         (Compare AP(n) with OL)
               IF DUP AP + @ OL !        (If AP(n) greater, OL = AP(n))
                 DUP 2 / CH !            (and set CH from TOS)
               ENDIF
             LOOP
             88 CH @ AE + C!             (Set chosen 'X' entry)
             DROP 0                      (Ensure no repeat)
          REPEAT
          OL @ 0=
          IF 1 BD !                      (If OL =0 set BD)
          ENDIF ;
: OX3 BEGIN INIT                         (Game loop point)
        BEGIN PLAY INPOS                 (Make play)
          BE @ 0=                        (Check e..d flag)
          IF CALCP SELECT                (If BE = 0, find X move)
          ENDIF
          0 0 P2                         (CASE condition = 0)
```



```
            CASE BE @ 0= OF
                   DROP 1 ENDOF          (If BE = 1 TOS = 1)
                   BD @ 0= OF
                   ." Drawn Game       " (If BD = 1 report draw)
                   DROP 1 ENDOF          (and TOS = 1)
                   BO @ 0= OF
                   ." 0 Wins           " (If BO = 1 report 0 Win)
                   DROP 1 ENDOF          (and TOS = 1)
                   BX @ 0= OF
                   PLAY ." X Wins      " (If BX = 1 report)
                   DROP 1 ENDOF          (with display. TOS = 1)
            ENDCASE
          UNTIL
          P1 ." Another run ? " KEY 89 -
        UNTIL ;
```

APPENDIX A:
The Dictionary

The contents of the Abersoft FORTH dictionary are defined in condensed form. Machine code sections are expressed in pseudo-code related to Z80 processor functions. Parameter Fields are translated from Code Field addresses to word names. A particular entry can be located through reference to the link list in the main index.

The Z80 registers are used as follows:

BC holds the Instruction Pointer (IP), which gives the address of the next link to be implemented.

DE is used to hold data to be put on the stack where more than one word is to be pushed. (The FORTH W register) It also holds incremental values and jump spans.

HL is used to hold data to be put on the stack, its contents being pushed after the contents of DE where two words are stacked.

IX holds the user area pointer.

SP is the stack pointer.

These are the special uses of the registers. All registers may be called into service for other purposes as necessary.

The dictionary definition is not easy to follow, because there are so many cross-references, but this is unavoidable. Examination of some of the more complex functions will often suggest useful formats for new words.

It is useful to remember that a word must be defined before it can be used in a further definition, so the constituents of a definition will always occur before the definition itself.

The FORTH program begins with a number of variables and constants. While these are not strictly part of the dictionary, they are needed to understand how some dictionary functions operate.

5E06	S0	Initial Stack Pointer Contents
5E08	R0	Initial Return Stack Pointer Contents
5E0A	TIB	Address of Terminal Input Buffer
5E0C	WIDTH	Maximum word length
5E0E	WARNING	Message control flag
5E10	FENCE	Dictionary protection limit
5E12	DP	Dictionary Pointer
5E14	VOC-LINK	Vocabulary search start address
5E16	BLK	Block number
5E18	IN	Text buffer pointer
5E1A	OUT	Output pointer
5E1C	SCR	Last used screen
5E1E	OFFSET	Block offset
5E20	CONTEXT	Dictionary start address
5E22	CURRENT	Dictionary start address
5E24	STATE	System mode
5E26	BASE	Number representation base
5E28	DPL	Decimal point location
5E2A	FLD	Unused
5E2C	CSP	Stack pointer hold
5E2E	R#	Editing cursor, etc.
5E30	HLD	Address of last character in output
5E32		
5E34		
5E36		
5E38		
5E3A		
5E3C		
5E3E		
5E40	00	
5E41	C3 B0 6D	Entry jump to COLD at 6DB0
5E44	00	
5E45	C3 93 6D	Entry jump to WARM at 6D93
5E48	01 01	
5E4A	00 0E	
5E4C	49 81	
5E4E	0C 00	
5E50	66 5E	
5E52	CB40	Initial value for 5E06
5E54	CBE0	5E08
5E56	CB40	5E0A
5E58	001F	5E0C
5E5A	0000	5E0E
5E5C	8159	5E10
5E5E	8159	5E12
5E60	77AC	5E14

5E62		
5E64		
5E66	5E00	Source for IX
5E68		Return stack pointer

The dictionary proper now begins. To cover possible modifications, the entry points are defined by link numbers rather than by addresses, but the actual addresses can be determined quite simply with the aid of −FIND . Entries marked * execute even in compile mode.

1 PUSHDE — Push the contents of DE on to the stack

2 PUSHHL — Push the contents of HL on to the stack

3 NEXT 1 — HL = BC. BC = BC + 2

4 NEXT2 — HL = (HL). Go to (HL).

Where, as above, the entries are closely spaced, they run on to the subsequent entry. PUSHDE and PUSHHL are used on return from other modules to set data on to the calculation stack, and then the interpretive pointer IP (held in BC) is used to look up the next link, to which the routine jumps. IP is incremented to point to the link for the next action. This is the central control of the whole FORTH system.

5 LIT — HL = (BC). BC = BC + 2. Go to PUSHHL.

The word stored in the location pair following the LIT link is read into HL and pushed on to the calculation stack. LIT, in effect, identifies the word as data, not a link.

6 EXECUTE — POP HL. Go to NEXT2

TOS is popped into HL as a link to a routine to be executed.

7 BRANCH — HL = BC. DE = (HL). HL = HL+DE. BC = HL. Go to NEXT 1.

The word stored in the location pair following BRANCH is added to the IP, interpretation being resumed at the resulting address.

8 0BRANCH — POP HL. If HL = 0 go to BRANCH. Else BC = BC + 2. Go to NEXT1.

This is the conditional branch, performed only if TOS = 0. Otherwise, the branch span word is skipped by advancing the IP.

9 (LOOP) — DE = 1

10 — HL = (RSP). (HL) = (HL) + DE. DE = (HL) HL = HL + 2. If D was negative, go to 11. Else set sign flag on DE − (HL). Go to 12

11 — Set sign flag on (HL) − DE.

12 — If negative go to BRANCH. Else HL = HL + 2. (RSP) = HL BC = BC + 2. Go to NEXT1.

This is the compiled form of LOOP. HL reads the Return Stack Pointer, and DE is added to TORS. 2ORS is then compared with

TORS, taking the sign of DE into account. (See (+LOOP) below.) If the index in TORS has neither reached nor passed the limit value in 2ORS, BRANCH is entered to loop back to immedately after (DO), the necessary jump span being calculated at compile time. Otherwise the RSP is set back two words, effectively discarding the two relevant entries. The IP skips the branch span data, and NEXT1 continues the action.

13 (+LOOP) POP DE. Go to 10.

Instead of being set to 1, as for (LOOP), DE is set from TOS, and (LOOP) follows.

14 (DO) (RSP) = (RSP) − 4. POP DE. HL = (RSP)
(HL) = DE. POP DE. HL = HL + 2
(HL) = DE. Go to NEXT 1.

This is the compiled form of DO. Two new entries are added to the Return Stack. The new TORS is set from TOS (initial index value) and the new 2ORS is set from 2OS (limit index value).

15 I HL = (RSP). DE = (HL). PUSH DE. Go to NEXT1.

The contents of TORS are copied to TOS. Within a DO loop, this transfers to the current index value.

16 DIGIT POP HL. POP DE. A = E −48
If A is negative go to 18.
Else if A is less than 10 go to 17.
A = A − 7.
If A is less than 10 go to 18.

17 If A is greater than L go to 18.
E = A. HL = 1. Go to PUSHDE.

18 L = H. GOTO PUSHHL.

The number base from TOS is popped into HL. The ASCII code in 2OS is popped into DE. The code is converted to a numeric value. If that value is negative, greater than number base or otherwise invalid, 18 is reached. A zero is pushed as TOS. (It is assumed that the number base does not exceed 255, so H = 0). Otherwise, the number is pushed as 2OS, with TOS = 1 to indicate successful conversion.

19 (FIND) POP DE.

20 POP HL. PUSH HL.
If (DE) XOR (HL) AND 3FH ≠ 0 go to 26.

21 INC HL. INC DE.
If (DE) XOR (HL) ≠ 0 go to 25.
If bit 7 of (DE) XOR (HL) = 0 go to 21.
HL = DE + 5

22 Exchange (SP) and HL

23 DEC DE.

24 A = (DE). If bit 7 of A = 0 goto 23.
E = A. D = 0. HL = 1. Go to PUSHDE.

25 If (DE) XOR (HL) = 1 go to 27.

26 INC DE. A = (DE). If bit 7 of A = 0 go to 26.

27 INC DE. Exchange DE and HL. DE = (HL)
If DE ≠ 0 goto 20.
Else POP HL. HL = 0. Go to PUSHHL.

This is a search routine looking for a match to text pointed to by 2OS, the search starting at TOS. The reference text is a word name, and the routine searches through the dictionary name fields. The search pointer is popped into DE, and the reference text pointer is popped into HL and then restored. If the codes (length bytes) pointed to are unequal in respect of bits 0-5 , the routine jumps on to 26. (Note that the SMUDGE bit is not taken into account.)

Otherwise, the pointers are advanced. The routine loops through 21 until either a mismatch is found or bit 7 of (DE) is found, marking the end of the dictionary name. In the latter case HL = DE + 5 , so that HL points to the Parameter Field Address, and this is put on to the stack in exchange for the reference text pointer. DE is then repeatedly decremented until the length byte of the dictionary entry is found, and the length byte is then pushed on to the calculation stack, followed by a true flag.

If a character mismatch is found, 25 is reached. If bit 7 of the dictionary name had bit 7 set, the routine skips on to 27. Otherwise, and if a length byte mismatch brings the routine to 26, DE is repeatedly incremented until (DE) has bit 7 set. One more increment sets DE pointing to the Link Field Address. The link is read, and if it is not zero the routine loops back to 20. If the link is zero, the end of the dictionary has been reached, and TOS is replaced by a zero (false) flag to show that no match was found.

28 ENCLOSE POP DE. POP HL. PUSH HL.
A = E. D = A. E = FFH. DEC HL.

29 INC HL. INC E. If (HL) = A go to 29
D = 0. PUSH DE. D = A. A = (HL)
If A ≠ 0 go to 30.
Else D = 0. INC E. PUSH DE.DEC E. PUSH DE.
Go to NEXT1.

30 A = D. INC HL. INC E. If A = (HL) go to 31.
If (HL) ≠ 0 go to 30.
D = 0. PUSH DE. PUSH DE. Go to NEXT1.

31 D = 0. PUSH DE. INC E. PUSH DE. Go to NEXT1.

This routine scans text to locate delimiters. The delimiter is popped from TOS into DE, the start address for the scan is popped from

2OS into HL, and restored. The delimiter code is copied into A and D. E is set to − 1 and HL is set back. A loop through 29 then seeks the first non-delimiter character, advancing HL and E. When the loop drops out, D = 0 and DE is pushed. (Offset to first non-delimiter character in 3OS) If (HL) = 0, E + 1 is pushed (offset to next delimiter in 2OS) and then E is pushed (offset to first character not included in TOS).

If (HL) ≠ 0, a loop through 30 scans the text until a delimiter is found (go to 31) or (HL) = 0. In the first case the values pushed are E and E+1. In the second case E is pushed twice.

32 EMIT — Machine code at 274 is called
OUT +! OUT is incremented.

33 KEY — Machine code at 266 is called.

For details, see the machine code routines.

34 ?TERMINAL — HL = 0. Machine code at 263 is called.

35 CR — Machine code at 276 is called.

36 CMOVE — HL = BC. POP BC. POP DE. Exchange (SP),HL
If BC = 0 go to 37
Else LDIR. POP BC.

37 — Go to NEXT1.

LDIR performs the copy action, but BC is needed to hold the length parameter, so the IP passes to HL, then to the stack, and back to BC when LDIR has been executed.

38 U* — POP DE. POP HL. PUSH BC.
B = H. A = L. CALL 39
PUSH HL. H = A. A = B. B = H. CALL 39.
POP DE. C = D. HL = HL + BC. A = A + carry.
D = L. L = H. H = A. POP BC. PUSH DE.
Go to PUSHHL.

39 — HL = 0. C = 8

40 — HL = 2*HL. RL A. If no carry go to 41.
HL = HL + DE. A = A + carry.

41 — DEC C. IF C ≠ 0 go to 40.
RETURN.

The subroutine at 39 multiplies A by DE, with the result in HL. It is first used to multiply the lower byte of 2OS by TOS, then to multiply the upper byte of 2OS by TOS. The two results are combined as a double number, the lower half pushed from DE, the upper half from HL. Note that here, again, the IP is saved on the stack to make BC available for other use.

42 U/MOD — HL = 4 + SP. E = (HL). (HL) = C
INC HL. D = (HL). (HL) = B. POP BC. POP HL.

If BC is greater than HL go to 43.
Else HL = FFFFH. DE = FFFFH. Go to 48.

43 — A = 16

44 — HL = 2*HL RL A. Exchange DE and HL.
HL = 2*HL. If no carry go to 45.
INC DE. AND A.

45 — Exchange DE and HL. RR A. PUSH AF.
If no carry go to 46.
A = A AND L. HL = HL − BC − carry
Go to 47.

46 — HL = HL − BC. If no carry go to 47.
HL = HL + BC. DEC DE

47 — INC DE. POP AF. DEC A. If A ≠ 0 go to 44.

48 — POP BC. PUSH HL. PUSH DE. Go to NEXT1

When data is pushed on to the Z80 stack, the stack pointer is left pointing to the last-entered byte. Adding four to the SP therefore points to the high byte of 3OS. This is taken into DE, and replaced by the IP from BC. TOS is taken into BC, and 2OS into HL. The routine divides 2OS/3OS by TOS. If TOS is not greater than 2OS, FFFFH is pushed on to the stack twice. Otherwise, an iterative division routine is performed on a restoring basis, which leaves a true remainder. This is pushed from HL, the result from DE.

49 AND — POP DE. POP HL. HL = HL AND DE.
Go to PUSHHL.

50 OR — POP DE. POP HL. HL = HL OR DE.
Go to PUSHHL.

51 XOR — POP DE. POP HL. HL = HL XOR DE.
Go to PUSHHL.

52 SP@ — HL = SP. Go to PUSHHL

53 SP! — DE = (IX + 6). Exchange DE,HL. SP = HL.
Go to NEXT1

IX points to 5E40 (See COLD) and is used to access the variable area.

54 RP@ — HL = (RSP). Go to PUSHHL.

55 RP! — DE = (IX + 8). Exchange DE,HL. (RSP) = HL.
Go to NEXT1.

56 ;S — HL = (RSP). BC = (HL). HL = HL + 2.
(RSP) = HL. Go to NEXT1.

This key function terminates interpretation of a colon definition or a screen. The IP is recovered from the Return Stack.

57 LEAVE — HL = (RSP). DE = (HL). HL = HL + 2.
(HL) = DE. Go to NEXT1.

Used within a DO LOOP, LEAVE copies the current index value to

the limit entry so that the loop will drop out at the next LOOP.

58 >R — POP DE. HL = (RSP). (HL) = DE. HL = HL − 2
(RSP) = HL. Go to NEXT1

59 R> — HL = (RSP). DE = (HL). HL = HL + 2
(RSP) = HL. PUSHDE. Go to NEXT1.

These two transfer data between TOS and TORS, adjusting the stacks. The next function copies TORS to TOS without changing TORS.

60 R — Go to 15

R is identical in action to I.

61 0= — POP HL. If HL = 0 then HL = 1, else HL = 0.
Go to PUSHHL.

62 0< — POP HL. HL = 2*HL. If no carry HL = 0,
else HL = 1. Go to PUSHHL.

63 + — POP DE. POP HL. HL = HL + DE.
Go to PUSHHL.

64 D+ — HL = SP + 6. DE = (HL). (HL) = BC.
POP BC. POP HL. HL = HL + DE.
Exchange DE,HL. POP HL.
HL = HL + BC + carry.
POP BC. PUSH DE. Go to PUSHHL.

The IP is saved on the stack in place of 4OS. Two separate additions are used to add two double numbers together.

65 MINUS — POP DE. HL = 0 − DE. Go to PUSHHL.

66 DMINUS — POP HL. POP DE. DE = 0 − DE.
HL = 0 − HL − carry. PUSH DE. Go to PUSHHL.

67 OVER — POP DE. POP HL. PUSH HL. Go to PUSHDE.

68 DROP — POP HL. Go to NEXT1.

69 SWAP — POP HL. Exchange (SP) and HL. Go to PUSHHL.

70 DUP — POP HL. PUSH HL. Go to PUSHHL.

71 2DUP — POP HL. POP DE. PUSH DE. PUSH HL.
Go to PUSHDE.

72 +! — POP HL. POP DE. (HL) = (HL) + E. INC HL.
(HL) = (HL) + D + carry. Go to NEXT1.

73 TOGGLE — POP DE. POP HL. (HL) = (HL) XOR E
Go to NEXT1.

74 @ — POP HL. DE = (HL). PUSH DE. Go to NEXT1.

75 C@ — POP HL. L = (HL). H = 0. Go to PUSHHL.

76 2@ — POP HL. HL = HL + 2. DE = (HL). PUSH DE.
HL = HL − 2. DE = (HL). PUSH DE. Go to NEXT1.

77 ! — POP HL. POP DE. (HL) = DE. Go to NEXT1.

78 C! — POP HL. POP DE. (HL) = E. Go to NEXT1.

79 2! — POP HL. POP DE. (HL) = DE. POP DE.
HL = HL + 2. (HL) = DE. Go to NEXT1.

*80 : — ?EXEC — Error if not executing
!CSP — Save stack pointer in CSP
CURRENT @
CONTEXT ! — CONTEXT = CURRENT
CREATE — Create a dictionary heading.
] — Resume compilation
(;CODE) — Point to following code. (81)

81 — HL = (RSP). (HL) = BC.
HL = HL − 2. (RSP) = HL.
INC DE. BC = DE.

This is another key routine. Execution of colon performs the checks shown above, then sets up a link to 81. The IP then interprets the words which follow, setting up links or data words to represent them. This forms the parameter field of the new dictionary entry. The process is terminated by semicolon, below:

*82 ; — ?CSP — Error if SP ≠ CSP.
COMPILE — Compile link into dictionary.
;S
SMUDGE — Restore the SMUDGE bit.
[— Suspend compilation.

Note that colon and semi-colon arise only in an execution context, colon switching to compilation and semi-colon restoring execution.

83 NOOP — NOOP does nothing, except to go to NEXT1.

84 CONSTANT — CREATE — Create a dictionary entry.
SMUDGE — Toggle the SMUDGE bit.
, — (Comma) Store TOS in dictionary.
(;CODE) — Point to following code.

85 — INC DE. Exchange DE,HL.
DE = (HL) PUSH DE.
Go to NEXT1.

CONSTANT creates a dictionary entry which refers to the code at 85. This picks up the following word and puts it on TOS.

86 VARIABLE — CONSTANT — Create a constant entry.
(;CODE) — Point to following code.

87 — INC DE. PUSH DE. Go to NEXT1.

The code at 87 puts the address of a variable on the stack.

88 USER — CONSTANT — Create a constant entry.
(;CODE) — Point to following code.

89 — HL = IX + DE. Go to PUSHHL.

A variable in the general workspace is referenced, using IX and a single byte displacement picked up in E. The address of the variable goes on TOS.

90	0		Constant 0 to TOS
91	1		Constant 1 to TOS
92	2		Constant 2 to TOS
93	3		Constant 3 to TOS
94	BL		Constant 20H (Space code) to TOS
95	C/L		Constant 40H (Characters per line) to TOS
96	FIRST		Constant CBE0H (1st buffer start) to TOS
97	LIMIT		Constant D000H (Last buffer end) to TOS
98	B/BUF		Constant 8OH (Bytes per buffer) to TOS
99	B/SCR		Constant 8 (Blocks per screen) to TOS

The above use the code at 85.

100	+ORIGIN	LIT 5E40 +	Add 5E40 (nominal origin) to TOS
101	SO		Address 5E06 to TOS
102	RO		Address 5E08 to TOS
103	TIB		Address 5E0A to TOS
105	WIDTH		Address 5E0C to TOS
106	WARNING		Address 5E0E to TOS
107	FENCE		Address 5E10 to TOS
108	DP		Address 5E12 to TOS
109	VOC-LINK		Address 5E14 to TOS
110	BLK		Address 5E16 to TOS
111	IN		Address 5E18 to TOS
112	OUT		Address 5E1A to TOS
113	SCR		Address 5E1C to TOS
114	OFFSET		Address 5E1E to TOS
115	CONTEXT		Address 5E20 to TOS
116	CURRENT		Address 5E22 to TOS
117	STATE		Address 5E24 to TOS
118	BASE		Address 5E26 to TOS
119	DPL		Address 5E28 to TOS
120	FLD		Address 5E2A to TOS

121	CSP		Address 5E2C to TOS
122	R#		Address 5E2E to TOS
123	HLD		Address 5E30 to TOS

The above use the code at 89.

124	1+	1 +	Add 1 to TOS
125	2+	2 +	Add 2 to TOS
126	HERE	DP @	Read dictionary pointer to TOS.
127	ALLOT	DP +!	Add TOS to dictionary pointer.
128	,	HERE!2ALLOT	Store TOS at HERE, add 2 to DP. (Comma)
129	C,	HERE C! 1 ALLOT	Store byte of TOS at HERE, DP = DP + 2.
130	−		POP DE. POP HL. HL = HL − DE. Go to PUSHHL.
131	=	− 0=	If TOS = 2OS, TOS = 1 else 0.
132	<		POP DE. POP HL. If bit 7 of D XOR H = 0 then HL = HL − DE (signs differ) If H is positive then HL = 0 else 1. Go to PUSHHL.
133	U<	2DUP XOR 0<	If signs of TOS,2OS differ set true flag.
		0BRANCH 000C	If false flag go to 134.
		DROP	Discard TOS
		0< 0=	Set true flag if new TOS positive.
		BRANCH 0006	End.
134		− 0<	Set true flag if TOS exceeds 2OS.
135	>	SWAP <	Exchange TOS,2OS and perform reverse function.
136	ROT		POP DE. POP HL. Exchange (SP), HL.
137	SPACE	BL EMIT	Output a space.
138	−DUP	DUP 0BRANCH	Duplicate TOS
		0004	End if TOS = 0
		DUP	Duplicate TOS
139	TRAVERSE	SWAP	Stack: n address. (n = ∓ 1)
140		OVER + LIT 007F	Stack: n address+n
		OVER	Stack: n address+n 7FH address+n
		C@ <	Compare (address+n) with 7FH.

```
                0BRANCH
                FFF0            If (address+n) not greater than
                                7FH go to 140.
                SWAP DROP       Else discard n, leaving
                                address+n.
```

This scans through a dictionary name field in a direction determined by the sign of n, until a byte with bit 7 true is found.

```
141 LATEST      CURRENT @ @     TOS = (CURRENT)
142 LFA         LIT 0004 -      Subtract 4 from TOS.
143 CFA         2 -             Subtract 2 from TOS.
144 NFA         LIT 0005 -      Subtract 5 from TOS.
                LIT FFFF        TOS = - 1.
                TRAVERSE        Scan back through name field.
145 PFA         1 TRAVERSE      Scan forward through name field.
                LIT 0005 +      Add 5 to TOS.
146 !CSP        SP@             Read stack pointer
                CSP!            Write to CSP.
147 ?ERROR      SWAP            Stack: n flag
                0BRANCH
                0008            If flag zero go to 148.
                ERROR           Report error
                BRANCH 0004     End
148             DROP            Discard error number.
149 ?COMP       STATE @ 0=      True flag on TOS if STATE = 0
                LIT 0011        Error number 17
                ?ERROR          Report error if flag true.
150 ?EXEC       STATE @         False flag if STATE = 0
                LIT 0012        Error number 18
                ?ERROR          Report error if flag true.
151 ?PAIRS      -               False flag if TOS = 2OS
                LIT 0013        Error number 19
                ?ERROR          Report error if flag true.
152 ?CSP        SP@ CSP @ -     Compare SP with CSP.
                                False flag if equal.
                LIT 0014        Error number 20
                ?ERROR          Report error if flag true.
153 ?LOADING    BLK @ 0=        True flag on TOS if TIB in use.
                LIT 0016        Error number 22
                ?ERROR          Report error if flag true.
154 COMPILE     ?COMP           Error if not compiling.
                R>              TORS to TOS
                DUP 2+          Stack: TORS TORS+2
                >R              TOS to TORS
```



```
                @ ,             Store (TORS) at HERE
*155 [          0 STATE!        Set STATE = 0
156 ]           LIT 00C0
                STATE !         Set state = 192
157 SMUDGE      LATEST
                LIT 0020
                TOGGLE          Toggle bit 5 of LATEST.
158 HEX         LIT 0010
                BASE !          Set BASE = 16
159 DECIMAL     LIT 000A
                BASE !          Set BASE = 10
160 (;CODE)     R>              TORS to TOS
                LATEST
                PFA CFA !       Write TORS to LATEST
                                code field.
*161 ;CODE      ?CSP            Check that SP = CSP
                COMPILE
                (;CODE)         Compile (;CODE)
                [               Set STATE = 0
                SMUDGE          Toggle bit 5 of LATEST
162 <BUILDS     0 CONSTANT      Constant entry established.
163 DOES>       R>              TORS to TOS
                LATEST PFA !    Copy to parameter field of
                                LATEST
                (;CODE)         Link to following code.

164                             HL = (RSP) - 2. (HL) = BC.
                                (RSP) = HL INC DE.
                                Exchange DE,HL. BC = (HL)
                                Go to PUSHHL.
```

The IP is put on the return stack and re-initialised from (DE+1), while DE + 1 goes on to TOS.

```
165 COUNT       DUP 1+          Stack: addr addr+1
                SWAP C@         Stack: addr+1 (addr)
```

The stack initially holds the address of the length byte of a text string. The address of the start of the text is put on 2OS with the length byte on TOS.

```
166 TYPE        -DUP            Duplicate TOS if non-zero
                0BRANCH
                0018            Branch to 168 if TOS = 0
                OVER +          Stack: addr addr+1length
                SWAP            Stack: addr+length addr
                (DO)
```

167		I C@	Read (I) to TOS
		EMIT	Print TOS
		(LOOP) FFFB	Loop to 167
		BRANCH 0004	End
168		DROP	Discard address
169	−TRAILING	DUP 0	Stack: addr length length 0
		(DO)	
170		OVER OVER	Stack: addr length addr length
		+ 1 −	Stack: addr length addr+length−1
		C@ BL −	Compare (addr+length−1) with space code
		0BRANCH	
		0008	Go to 171 if match found.
		LEAVE	Set loop exit condition.
		BRANCH 0006	Go to 172
171		1 −	Decrement length
172		(LOOP) FFEO	Loop to 170.

A text string is scanned backwards, the length byte being decremented for each space code found, until a non-space code appears.

173	(.")	R	TORS copied to TOS
		COUNT	Adjust address and length
		DUP 1+	Stack: addr length length+1
		R> + >R	TORS = TORS + length + 1
		TYPE	Output string.

TORS is updated and the string defined is output.

*174	."	LIT 0022	Code for " (delimiter)
		STATE @	TOS = STATE
		0BRANCH	
		0014	If executing go to 175.
		COMPILE (.")	
		WORD	Store text at HERE
		HERE C@	Read length byte
		1+ ALLOT	Reserve length + 1 bytes.
		BRANCH 000A	End
175		WORD	Store text at HERE
		HERE	
		COUNT TYPE	Output text

This word acts quite differently in execute mode and compile mode. In compile it sets up (.") with the text. In execute it outputs the text.

176	EXPECT	OVER + OVER	Stack: addr addr+limit addr
		(DO)	

177		KEY DUP	Stack: addr input input
		LIT 000E	
		+ORIGIN @	TOS = (5E4E)
		=	Zero flag if TOS ≠ input
		0BRANCH	
		002A	If zero flag go to 179.
		DROP DUP I =	Zero flag if I ≠ addr.
		DUP	Duplicate flag
		R>2 − + >R	TORS = TORS − 2 + flag
		0BRANCH	
		000A	If flag zero go to 178
		NOOP NOOP	Delay ?
178		BRANCH 0008	Code for cursor left.
		BRANCH 0028	Go to 182
179		DUP	Stack: addr input input
		LIT 000D	Newline code
		=	Zero flag if input ≠ newline.
		0BRANCH	
		000E	If zero flag go to 180
		LEAVE	Terminate loop
		DROP BL 0	Stack: addr space-code, zero.
		BRANCH 0004	Go to 181
180		DUP	Stack: addr input input
181		I C!	Store input in (I)
		0 I 1+ !	Zero next location.
182		EMIT	Output TOS
183		(LOOP) FF9C	Loop to 177
		DROP	Clear stack, end.
184	QUERY	TIB @	Terminal input buffer address on TOS.
		LIT 0050	Limit length (80 bytes)
		EXPECT	See above
		0 IN !	Zero IN.
*185	(See note)	BLK @	Block number to TOS
		0BRANCH	
		002A	Go to 187 if TIB in use
		1 BLK +!	Increment block number
		0 IN !	Zero IN
		BLK @	Block number to TOS
		B/SCR 1 −	Blocks per screen − 1 to TOS
		AND 0=	Zero flag if (TOS AND 2OS) = 0
		0BRANCH	
		0008	If zero flag go to 186
		?EXEC	Error if not executing
		R> DROP	Discard TORS

```
186                 BRANCH 0006  End
187                 R>DROP       Discard TORS
```

The name field for this entry contains only CIH, 80H, indicating zero code. It switches blocks on a null code.

```
188 FILL                         HL = BC. POP DE. POP BC.
                                 Exchange (SP),HL
                                 Exchange DE,HL
189                              If BC = 0 go to 190
                                 (DE) = L. INC DE. DEC BC.
                                 Go to 189
190                              POP BC. Go to NEXT1
```

DE, then HL, hold the character to be used. The number of entries is held in BC, while HL, then DE, holds the start address, which is incremented after each entry.

```
191 ERASE       0 FILL           See above.
192 BLANKS      BL FILL          See above.
193 HOLD        LIT -1 HLD +!    Decrement HLD
                HLD @ C!         Store TOS in (HLD)
194 PAD         HERE
                LIT 0044 +       TOS = HERE + 68
```

The PAD buffer floats above the dictionary at a distance of 68 bytes.

```
195 WORD        BLK @            TOS = block number
                0BRANCH
                000C             If TIB in use go to 196
                BLK @            TOS = block number
                BLOCK            TOS = block address
                BRANCH 0006      Go to 197
196                 TIB @        TOS = TIB ADDRESS
197                 IN @ +       Add IN
                    SWAP         Stack: addr delimiter
                    ENCLOSE      Stack: addr offset1 offset2 offset3
                    HERE LIT 0022
                    BLANKS       Set up 34 space code entries.
                    IN +!        Add offset3 to IN
                    OVER         Stack: addr offset1 offset2 offset1
                    - >R
                    R HERE C!    TORS = offset2 - offset1
                                 (HERE) = TORS
                    + HERE 1+ R> Stack: addr+offset1
                                 HERE+1 TORS
                    CMOVE        Copy TORS bytes from
                                 address+offset1 to HERE+1
```



```
198 (NUMBER)    1+ DUP           Stack: X/Y addr+1 addr+1
                >R               TORS = addr+1
                C@               TOS = (addr+1)
                                 (ASCII numeric code)
                BASE @           TOS = BASE
                DIGIT            Convert ASCII code to number.
                0BRANCH
                002C             If invalid go to 200
                SWAP BASE @      Stack: X number Y BASE
                U*               Stack: X number Y*BASE
                DROP             Discard upper word of double
                                 number product.
                ROT BASE
                @ U*             Stack: number Y*BASE X*BASE
                D+               Stack: X*BASE + (Y*BASE +
                                 number)
                DPL @ 1+         TOS = DPL + 1
                0BRANCH
                0008             If DPL = -1 then go to 199
                1 DPL +!         Increment DPL (To count
                                 characters after point)
199             R>               Recover address from TORS.
                BRANCH FFC6      Go to 198
200             R>               Clear TORS
```

This routine is normally called by NUMBER, which sets X/Y as zero. The process merits close scrutiny.

```
201 NUMBER      0 0 ROT DUP      Stack: 0 0 addr addr
                1+ C@            Stack: 0 0 addr (addr+1)
                LIT 002D         Code for minus sign
                =                Zero flag if (addr+1) ≠ minus
                                 sign code
                DUP              Duplicate flag
                >R               Flag to TORS
                +                Stack: 0 0 addr+flag (Step past
                                 sign code)
                LIT -1
202             DPL !            Set DPL
                (NUMBER)         See above
                DUP C@           Stack: number/number
                                 addr (addr)
                BL -             Compare with space code
                0BRANCH
                0016             If space code go to 203
                DUP C@           Stack: number/number
                                 addr (addr)
```

```
                LIT 002E -      Compare with decimal point code
                0 ?ERROR        Error if not decimal point
                0               Set DPL = 0 if decimal point.
                BRANCH FFDC     Go to 202
203             DROP R>         Recover sign flag from TORS
                0BRANCH
                0004            If flag = 0 then end.
                DMINUS          Negate result.
```

NUMBER always produces a double number result. The upper byte is only discarded by INTERPRET if there is no decimal point.

```
204 -FIND       BL              Space code to TOS
                WORD            Copy name to HERE
                HERE            Set up reference pointer
                CONTEXT @ @     Set up search pointer
                (FIND)          Search for word
                DUP             Stack: PFA length flag flag.
                0=              Reverse flag
                0BRANCH
                000A            End if match found
                DROP            Discard flag
                HERE            Set up reference pointer
                LATEST          Set search pointer to LATEST
                (FIND)          Search again.
```

If (FIND) fails, only the flag is left on the stack, so there is no need to discard the length byte and PFA. A search is then made starting at LATEST instead of (CONTEXT).

```
205 (ABORT)     ABORT
```

This word allows for special user versions of (ABORT)

```
206 ERROR       WARNING
                @ 0<            True flag if WARNING negative
                0BRANCH
                0004            If WARNING not negative
                                go to 207
                (ABORT)
207             HERE
                COUNT TYPE      Output offending word.
                (.") "? "       Output query, space.
                MESSAGE         Output error number or text
                SP!             Reset SP
                BLK @ -DUP      Block number, duplicated if
                                non-zero.
                0BRANCH
                0008            If TIB in use go to 208
                IN @ SWAP       Stack: IN block
```



```
208             QUIT            Return to user control.
209 ID.         PAD LIT 0020
                LIT 005F FILL   Fill the PAD buffer with
                                32 5F codes.
                DUP PFA LFA     Stack: NFA LFA
                OVER -          Stack: NFA LFA-NFA
                PAD SWAP        Stack: NFA PAD LFA-NFA
                CMOVE           Copy LFA-NFA bytes from
                                NFA to PAD
                PAD COUNT       Check copy length
                LIT 001F AND    Limit to 31 bytes.
                2DUP            Stack: addr length addr length
                + 1 - DUP       Stack: addr length
                                addr+length-1
                @               TOS = last letter codes
                LIT FF7F AND    Remove bit 7
                SWAP !          Restore modified bytes
                TYPE SPACE      Output word, add space.
210 CREATE      -FIND           Search for word name
                                and enter at HERE
                0BRANCH
                0010            Go to 211 if unique
                DROP            Discard length
                NFA ID.         Output word name
                LIT 0004        Message 4
                MESSAGE         Output number or message
                SPACE           Add a space
211             HERE DUP C@     Stack: HERE (HERE)
                WIDTH @ MIN     Limit length to 31 characters
                1+ ALLOT        Allot one more
                DUP             Stack: HERE HERE
                LIT 00A0
                TOGGLE          Toggle bits 5 and 7 of length byte
                HERE 1 -        HERE now modified by ALLOT
                LIT 0080
                TOGGLE          Toggle bit 7 of last letter
                LATEST ,        Set up link field
                CURRENT @ !     CURRENT is read as an address.
                                (CURRENT) = HERE
                HERE 2+ ,       Set up code field
*212 [COMPILE]  -FIND           Search for word name and enter
                                at HERE
                0=              TOS = 0 if found.
                0 ?ERROR        Error 0 if not found.
                DROP            Discard length byte
```

```
                CFA ,           Set up code field
*213 LITERAL    STATE @
                0BRANCH
                0008            If executing then end.
                COMPILE LIT ,   Set up LIT entry.
*214 DLITERAL   STATE @
                0BRANCH
                0008            If executing then end.
                SWAP
                LITERAL
                LITERAL
215 ?STACK      SP@ SO @        Stack: SP SO
                SWAP            Stack: SO SP
                U<              True flag if SP exceeds SO
                1 ?ERROR        Error 1 if true flag
                SP@ HERE
                LIT 0080 +      Stack: SP HERE+128
                U<              True flag if HERE+128 is
                                the greater
                LIT 0007
                ?ERROR          Error 7 if true flag.
216 INTERPRET   -FIND           Search for word name and
                                enter it at HERE
                0BRANCH
                001E            If not found go to 219.
                STATE @ <       True flag if length is
                                less than state.
                0BRANCH
                000A            If false go to 217
                CFA ,           Enter CFA at here
                BRANCH 0006     Go to 218
217             CFA EXECUTE     Execute indicated function.
218             ?STACK          Check stack bounds
                BRANCH 001C     Go to 222
219             HERE
                NUMBER          Interpret word as number
                DPL @ 1+        TOS = DPL + 1
                0BRANCH
                0008            If DPL = -1 go to 220
                DLITERAL        Set up a double number
                BRANCH 0006     Go to 221
220             DROP            Discard upper byte of
                                double number.
                LITERAL         Set up a single number.
```



```
221             ?STACK          Check stack bounds
222             BRANCH FFC2     Go to 216
```

The comparison of the length byte with state implements the special characteristics of some words. Note that if a word name happens also to be a valid number in the current BASE, it will not be possible to input that number, as it will always be interpreted as a word first.

```
223 IMMEDIATE   LATEST
                LIT 0040
                TOGGLE          Toggle bit 6 of LATEST
224 VOCABULARY  <BUILDS         Set up a constant entry.
                LIT A081 ,      Store A081 at HERE
                CURRENT @
                CFA ,           Store CFA of CURRENT
                                at HERE
                HERE
                VOC-LINK @ ,    Store VOC-LINK at HERE
                VOC-LINK !      Store previous HERE at
                                VOC-LINK
                DOES>
225             2+ CONTEXT !    Set CONTEXT to CURRENT + 2
*226 FORTH                      Execute code at 164
                                Execute 225
227 (space)                     Zero code field.
228 DEFINITIONS CONTEXT @
                CURRENT !       CURRENT = CONTEXT
*229 (          LIT 0029        Delimiter is close bracket.
                WORD            Set up comment but ignore.
230 QUIT        0 BLK !         Set block = 0 (TIB in use)
                [               STATE = 0
231             RP!             Initialise RSP
                CR QUERY        Newline. Get input text
                INTERPRET       Interpret text
                STATE @ 0=      True flag if executing.
                0BRANCH
                0007            If false flag go to 232
                ." "ok"         Output "ok"
232             BRANCH FFE7     Go to 231
233 ABORT       SP!             Initialise SP
                DECIMAL         Select decimal representation.
                ?STACK          Check stack bounds
                CLS CR          Clear screen, newline
                .CPU            Output "48K Spectrum"
```

```
                (.") text       Output "fig-FORTH 1.1A"
                CR              Newline
                (.") text       Output "(C) Abersoft: 1983
                CR              Newline
                FORTH           Select FORTH vocabulary
                DEFINITIONS     CURRENT = CONTEXT
                QUIT            Return control to user.
234 Entry from  WARM start      BC = address of link to 236
                                IX = (5E66) = 5E00
                                HL = (5E52) = CB40
                                SP = HL
                                Go to NEXT1 (Thence to 236)
235                             Link to CFA for WARM
236 WARM        EMPTY-
                BUFFERS         Mark buffers as empty
                ABORT           See above.
237 Entry from  COLD start      (FLAGS2) = 8 (BASIC variable)
                                (hold) = 0 (See 266)
                                BC = address of link to 267
                                IX = (5E66) = 5E00
                                HL = (5E52) = CB40
                                SP = HL
                                Go to NEXT1
238                             Link to CFA for COLD
239 COLD        EMPTY-
                BUFFERS         Mark buffers as empty
                LIT CBE0 USE !  Set next buffer = CBE0
                LIT CBE0
                PREV !          Set last buffer = CBE0
                DR0             OFFSET = 0
                LIT 5E52
                LIT 5E66 @
                LIT 0006 +
                LIT 0010        Stack: 5E52 (5E66)+6 16
                CMOVE           Copy 16 bytes, starting with
                                5E52 to 5E06
                LIT 5E4C @      TOS = 8149
                LIT 6CF8 !      (6CF8) = 8149 (Link field of 227)
                ABORT
240 S -> D                      POP DE. HL = 0. A = D AND 80H
                                If A ≠ 0 then DEC HL.
                                Go to PUSHDE
241 +-          0<              If TOS negative true flag
                0BRANCH
                0004            If false flag then end
```



```
                MINUS           Negate TOS
242 D+-         0<              True flag if TOS negative
                0BRANCH
                0004            If false flag then end.
                DMINUS          Negate double word.
243 ABS         DUP +-          If TOS negative then negate
244 DABS        DUP D+-         If TOS negative then negate
                                double word.
245 MIN         2DUP >          If 2OS greater than TOS then
                                true flag.
                0BRANCH
                0004            If false flag go to 246
                SWAP
246             DROP            Discard the greater.
247 MAX         2DUP <          If 2OS less than TOS then
                                true flag.
                0BRANCH
                0004            If false flag to to 248
                SWAP
248             DROP            Discard the lesser.
249 M*          2DUP XOR > R    TORS = TOS XOR 2OS
                                (Stack preserved)
                ABS SWAP
                ABS             Make TOS,2OS positive
                U*              Product of TOS*2OS
                R> D+-          Negate result if signs of
                                TOS,2OS differed.
250 M/          OVER >R >R      Stack: X Y TORS = Z 2ORS = Y
                DABS            Convert double number
                                to positive
                R ABS           Convert Z to positive
                U/MOD           Stack: rem quot.
                R> R XOR        Negative if signs of XY and
                                Z differ
                +-              If negative then negate quotient.
                SWAP
                R> +-           Negate remainder if Z negative
                SWAP
251 *           M* DROP         Discard upper half of product
252 /MOD        >R S -> D R>    Sign extend 2OS
                M/              As above.
253 /           /MOD SWAP
                DROP            Discard remainder
```

254	MOD	/MOD DROP	Discard quotient
255	*/MOD	>R	TOS to TORS
		M*	Product
		R>	Restore original TOS
		M/	Remainder Quotient
256	*/	*/MOD SWAP	
		DROP	Discard remainder
257	M/MOD	>R 0 R	Make 2OS double
		U/MOD	Remainder quotient1
		R> SWAP >R	Remainder divisor
			TORS hold quotient1
		U/MOD R>	Remainder quotient2 quotient1
258	(LINE)	>R	Stack: Line TORS: Screen
		LIT 0040	
		B/BUF	Bytes per buffer
		*/MOD	Line*64/bytes per buffer.
			(Rem and quot)
		R> B/SCR *	Rem quot Screen*B/SCR
		+	Rem quot+Screen*B/SCR
		BLOCK +	Add block address
		LIT 0040	
259	.LINE	(LINE)	See above
		−TRAILING	Discard trailing spaces
		TYPE	Output
260	MESSAGE	WARNING @	
		0BRANCH 001E	If WARNING = 0 go to 262
		−DUP	Duplicate if non-zero
		0BRANCH 0014	If message 0 then go to 261
		LIT 0004	Screen 4
		OFFSET @	Block offset
		B/SCR /	Divide by blocks per screen
		−	Subtract from screen number
		.LINE	Output message
		SPACE	Add a space
261		BRANCH 000D	End
262		(.") text	Output "MSG # "
		.	Output number.

The following code is used by ?TERMINAL (see 34) BASIC routines are involved.

263 PUSH BC. PUSH DE. CALL 1F54. HL = 0
If carry go to 264.
Else INC L. Go to 265

264 If (5C08) = 7 then INC HL.

265 POP DE. POP BC. Go to PUSHHL.

The following code is used by KEY. (See 33) BASIC routines are involved.

266 PUSH BC, A=2. CALL 1601. A=12. RST10 A = 1. RST10.

267 (5C08) = 0

268 A = (hold). RST10. A=8. RST10
If (5C08) = 0 then go to 268
If (5C08) ≠ 6 then go to 271
HL = 5C6A. (HL) = (HL) XOR 8

269 HL = hold. If bit 3 of A = 1 then go to 270
(HL) = 4CH. Go to 267

270 (HL) = 43H. Go to 267

271 If A ≠ 0FH then go to 273
A = 2. HL = 5C41.
(HL) = (HL) XOR A
If (HL) = 0 then go to 272
A = (5C6A). Go to 269

272 A = 47H. (hold) = A. Go to 267

In BASIC, 5C08 is LASTK 5C41 is MODE. 5C6A is FLAGS2

The routine from 273 onwards is mainly concerned with recoding certain keys.

273 C6 (AND) becomes 5B [
C5 (OR) becomes 5D]
E2 (STOP) becomes 7E ~
C3 (NOT) becomes 7C |
CD (STEP) becomes 5C /
CC (TO) becomes 7B {
CB (THEN) becomes 7D }
If A is greater than A5H (after the above conversions) go to 267 (i.e. ignore input) Else L = A.
H = 0. A = 12. RST10. A = 0 RST10. A = 20. RST10. A = 8. RST10. POP BC Go to PUSHHL.

The following code is used by EMIT (See 32). Once again, BASIC routines are involved.

274 PUSH BC. PUSH HL. A = 2. CALL 1601 POP HL. PUSH HL.
A = 1. RST10. A = (hold2).
If A = 0 go to 275

```
                                CALL 1601. POP HL. PUSH HL.
                                A = L. RST10
275                             POP HL. A = FFH. (5C8C) = A.
                                POP BC. Go to NEXT1
```

5C8C in BASIC is SCRCT (Scroll count). Setting it to FFH ensures permanent scrolling. The contents of hold2 control the printer (see 337).

The following code is used by CR (See 35).

```
276                             PUSH BC. A=2. CALL 1601. A =
                                2DH. RST 10
                                A = (hold2). If A = 0
                                then go to 277.
                                CALL 1601. A = 0D. RST10.
277                             POP BC. (5C8C) = FFH.
                                Go to NEXT1.
278 USE                         TOS = (Oldest block buffer)
279 PREV                        TOS = (Last block buffer)
280 #BUFF                       TOS = (Number of disc buffers)
281 +BUF          LIT 0084 +    Stack: Addr+132
                 DUP LIMIT =   Stack: Addr+132 true flag if
                               Addr+132 = LIMIT
                 0BRANCH 0006  If false flag go to 282
                 DROP FIRST    Substitute FIRST
282              DUP PREV @ -  Compare with PREV.
                               (Leave addr flag)
283 UPDATE       PREV @ @      TOS = contents of PREV
                 LIT 8000 OR
                 PREV @ !      Set bit 15 of contents of PREV.
284 EMPTY-BUFFERS
                 FIRST LIMIT
                 OVER -        Stack: FIRST LIMIT-FIRST
                 ERASE         Clear buffer area to zero.
                 LIMIT FIRST
                 (DO)          From First to LIMIT - 1;
285              LIT 7FFF I !  Set 7FFF at (I)
                 LIT 0084      Step 132
                 (+LOOP) FFF2  Loop to 285
286 DRO          0 OFFSET !    Set OFFSET to 0
287 BUFFER       USE @ DUP >R  Stack: (USE) TORS: (USE)
288              +BUF          Advance to next buffer.
                 0BRANCH FFFC  If PREV go to 288
                 USE !         Mark as in use
```



```
                 R @ 0<        If last buffer updated (negative)
                               set true flag.
                 0BRANCH 0014  If false flag go to 289.
                 R 2+ R @      (USE) = (USE) + 2. ((USE))
                 LIT 7FFF AND  Zero msb
                 0 R/W         Writer buffer to disc.
289              R !           Set (USE) as written
                 R PREV !      Set PREV = USE
                 R> 2+         Leave (USE) + 2
290 BLOCK        OFFSET @ +    Stack: blockno + offset
                 >R            Transfer to TORS
                 PREV @ DUP    Stack: PREV PREV
                 @             Stack: PREV (PREV)
                 R - DUP       STACK: PREV (PREV)-TORS
                 +             Stack: PREV+(PREV)-TORS
291              0BRANCH 0034  If zero go to 293
                 +BUF 0=       Move to next buffer. True if PREV
                 0BRANCH 0014  Go to 292 if false.
                 DROP          Discard buffer address
                 R BUFFER      See above (Blockno+offset
                               provides data)
                 DUP           Duplicate address
                 R 1 R/W       Read into last buffer
                 2 -           Modify address
292              DUP @ R -     Compare buffer data with TORS
                 DUP +         Double TOS
                 0=            Reverse flag state
                 0BRANCH FFD6  If false flag go to 291
                 DUP PREV !    Set PREV
293              R> DROP       Discard TORS
                 2 +           Advance address
294 LO                         TOS = D000 (Start of RAM-disc)
295 HI                         TOS = FBFF (End of RAM-disc)
296 R/W          >R            TORS = direction flag
                               (0=write,1=read)
                 B/BUF *       Stack: addr block*128
                 LO +          Stack: addr block*128 + D000
                 DUP HI >      True flag if limit exceeded
                 LIT 0006      Error 6
                 ?ERROR        Report error if true flag
                 R>            Clear TORS. Direction flag
                               on TOS
                 0BRANCH 0004  Go to 297 if false
                 SWAP
```

```
297                B/BUF
                   CMOVE          Copy a buffer-full as instructed.
298  FLUSH         #BUFF
                   1+
                   0 (DO)
299                BUFFER DROP
                   (LOOP) FFF8)   Loop to 299
300  LOAD          DUP 0 =        True flag if match
                   LIT 0009       Error number 9
                   ?ERROR         Report error if true
                                  (load from screen 0)
                   BLK @          Read block number
                   >R             Block no to TORS
                   IN @ >R        IN to TORS
                   0 IN !         IN = 0
                   B/SCR *        Screen*B/SCR
                   BLK !          Write to BLK
                   INTERPRET
                   R> IN !        Restore IN
                   R> BLK !       Restore BLK
*301 -->           ?LOADING       Error if not loading
                   0 IN !         IN = 0
                   B/SCR BLK @    B/SCR BLK
                   OVER           B/SCR BLK B/SCR
                   MOD            BLK rem of BLK/B/SCR
                   - BLK +!       BLK = BLK + BLK - rem
*302 (tick)        -FIND 0—       Search for name. Reverse flag
                   0 ?ERROR       Error 0 if not found.
                   DROP           Discard length byte
                   LITERAL
303  BACK          HERE - ,       Store addr - HERE
*304 BEGIN         ?COMP          Error if not compiling
                   HERE 1         Identifies loop point.
                                  Sets PAIRS ref = 1
*305 ENDIF         ?COMP          Error if not compiling
                   2 ?PAIRS       Error if PAIRS ref not 2
                   HERE OVER -    Stack: a HERE-a
                   SWAP !         Write link span.
*306 THEN          ENDIF          The words mean the same.
*307 DO            COMPILE (DO)
                   HERE 3         Identify loop point.
                                  PAIRS ref = 3
308  LOOP          3 ?PAIRS       Error if PAIRS ref not 3
                   COMPILE (LOOP)
```



```
                   BACK           Calculate backward link span.
*309 +LOOP         3 ?PAIRS       Error if PAIRS ref not 3
                   COMPILE (+LOOP)
                   BACK           Calculate backward link span.
*310 UNTIL         1 ?PAIRS       Error if PAIRS ref not 1
                   COMPILE 0BRANCH
                   BACK           Calculate backward link span
*311 END           UNTIL          The words mean the same
*312 AGAIN         1 ?PAIRS       Error if PAIRS ref not 1
                   COMPILE BRANCH
                   BACK           Calculate backward link span.
*313 REPEAT        >R >R          TOS, 2OS to return stack
                   AGAIN
                   R> R>          TOS,2OS restored
                   2 -            Convert PAIRS ref from 4 to 2.
                                  (See WHILE)
                   ENDIF
*314 IF            COMPILE 0BRANCH
                   HERE
                   0 ,            Reserve space
                   2              PAIRS ref = 2
*315 ELSE          2 ?PAIRS       Error if PAIRS ref not 2.
                   COMPILE BRANCH
                   HERE
                   0 ,
                   SWAP
                   2 ENDIF 2      PAIRS ref = 2
*316 WHILE         IF 2+          Execute IF, then increase PAIRS
                                  ref by 2.
317  SPACES        0 MAX          Ensure positive value
                   -DUP           Duplicate if non-zero
                   0BRANCH 000C   End if TOS = 0
                   0 (DO)
318                SPACE          Output space.
                   (LOOP) FFFC    Loop to 318
319  <#            PAD HLD !      Set HLD = PAD
320  #>            DROP DROP      Discard balance of number
                   HLD @          Read HLD
                   PAD OVER -     Stack: HLD PAD-HLD
321  SIGN          ROT            Bring sign to TOS
                   0<             True if TOS less than 0
                   0BRANCH 0008   If TOS = 0 then end.
                   LIT 002D       Minus sign code
```

```
                HOLD            Store at HLD
322  #          BASE @          Stack: XY BASE
                M/MOD           Stack: rem double-quotient
                ROT             Stack: double-quotient rem.
                LIT 0009 OVER   Stack: double-quotient rem
                                9 rem.
                <               TOS true if rem exceeds 9
                0BRANCH 0008    Go to 323 if false.
                LIT 0007 +      Add 7 to rem
323             LIT 0030 +      Add 48 to form ASCII code
                HOLD            Store at HLD
324  #S         #               See above
                OVER OVER       Stack: double-number
                                double-number
                OR 0=           True flag if number = 0
                0BRANCH FFF4    If false go to 324
325  D.R        >R SWAP OVER    Stack: Y X Y (XY double number)
                                TORS = n
                DABS            Make XY positive
                <# #S           Set up number in PAD
                SIGN #>         Add sign
                R> OVER -       Stack: addr count n-count
                SPACES          Output n-count spaces
                TYPE            Output number
```

Note that the number is reduced to zero by #S, so the sign has to be determined from the copy of Y.

```
326  .R         >R              n to TORS
                S-->D           Sign extend to double number.
                R>              Restore n
                D.R             Above
327  D.         0               Call for a zero-length field.
                D.R             See above
                SPACE           Add a space
328  .          S-->D           Sign extend to double number
                D.              See above
329  ?          @ .             Print contents of address
                                specified on TOS
330  U.         0               Form double number, unsigned
                D.              See above
331  VLIST      LIT 0080
                OUT !           OUT = 128
                CONTEXT @ @     Contents of CONTEXT on TOS
332             OUT @
                LIT 001F
```



```
                LIT 0008        Stack: (CONTEXT) OUT 31 8
                - >             True flag if OUT exceeds 23
                0BRANCH 000A    If false flag go to 333
                CR 0 OUT !      Newline. OUT = 0
333             DUP             Duplicate pointer
                ID.             Output name
                PFA LFA @       Read link field
                DUP 0=          True flag if zero
                ?TERMINAL       True flag if BREAK
                OR              Either true flag effective
                0BRANCH FFD0    If false flag loop to 332
                DROP            Drop last address
334  LIST       DECIMAL
                CR              Newline
                DUP SCR !       Set chosen screen
                (.") text       Output 'SCR # '
                .               Output screen number
                LIT 0010
                0 (DO)
335             CR              Newline
                I               Line number
                LIT 0003 .R     Print in a three-space field
                SPACE           Add a space
                I SCR @
                .LINE           Output stored line
                ?TERMINAL       True if BREAK
                0BRANCH 0004    If false go to 336
                LEAVE
336             (LOOP) FFE2     Go to 335
                CR              Newline
337  LINK                       POP HL. A = L. If A ≠ 0 then A = 3
                                (hold2) = A. Go to NEXT1
338  CLS                        PUSH BC. A=2. CALL 1601.
                                CALL 0D6B A=2. CALL 1601.
                                POP BC. Go to NEXT1
339  .CPU       (.") text       Output "48K SPECTRUM "
```

At this point, an area is reserved for tape headers of the form:

```
03 44 49 53 43 20 20 20 20 20 20 FF 2B 00 D0 20 20
   D  I  S  C
03 44 49 53 43 20 20 20 20 20 20 FF 2B 00 D0 20 20
   D  I  S  C
```

```
340  (TAPE)                     POP HL. PUSH BC. PUSH IX.
                                A = L. HL = D000. IX = (start of
                                header area). A = (5C72).
```

```
                        CALL 075A. POP IX. POP BC.
                        Go to NEXT1
341 MON                 A = (5CB0). If A = 0 then RST08
                        Else go to NEXT1
342 TEXT        HERE
                C/L 1+          Stack: HERE C/L+1
                BLANKS          Set up a line of spaces at HERE
                WORD            Set string at HERE
                HERE PAD C/L
                1+ CMOVE        Copy complete line to PAD
343 LINE        DUP             Stack: Line Line
                LIT FFF0 AND    Limit to 15
                LIT 0017        Error number 23
                ?ERROR          Error if non-zero
                SCR @           Read current screen
                (LINE)          Set up line address and account
                DROP            Drop length count.
344 LOADT       1 (TAPE)
345 SAVET       FLUSH 0 (TAPE)
346 VERIFY      2 (TAPE)
347 2DROP       DROP DROP
348 2SWAP       ROT >R          Stack from a b c d to a c d
                                b on TORS
                ROT R>          Stack: c d a b
349 SIZE        HERE
                0 +ORIGIN -     TOS = HERE - ORIGIN
350 FREE        SP@ HERE -      TOS = SP - HERE
351 FORGET      CURRENT @
                CONTEXT @ -     TOS = CURRENT - CONTEXT
                LIT 0018        Error number 24
                ?ERROR          Report error if non-zero
                '               TOS = PFA of word to be
                                forgotten.
                DUP             Duplicate
                FENCE @ U<      True flag if PFA below FENCE
                LIT 0015        Error number 21
                ?ERROR          Error if TOS true.
                DUP NFA         Duplicate PFA and substitute
                                NFA for copy.
                DUP !           DP = NFA
                LFA @           Convert PFA to LFA and read link.
                CURRENT @ !     Set link at location defined
                                by CURRENT.
```


Note that no actual erasure occurs. Because the pointers have been reset, new entries will over-write the debris.

```
352 INDEX       CLS             Clear screen
                1+ SWAP         Reverse parameters,
                                incrementing end.
                (DO)
353             CR              Newline
                13 .R           Output screen number in a
                                three-space field.
                SPACE           Add a space
                0 I .LINE       Output line 0 of screen I
                ?TERMINAL       Set true if BREAK
                0BRANCH 0004    Go to 354 if no BREAK.
                LEAVE           Terminate loop
354             (LOOP) FFE6     Loop to 353
355 TRIAD       CLS             Clear screen
                3 /             Divide screen number by 3.
                3 *             Multiply by 3. (Result x)
                3 OVER +        Stack: X X+3
                SWAP            Stack: X+3 X
                (DO)
356             CR              Newline
                I LIST          List screen I
                ?TERMINAL       TOS = 1 if BREAK
                0BRANCH 0004    If no BREAK go to 357
                LEAVE           Terminate loop
357             (LOOP) FFF0     Loop to 356.
*358 EDITOR                     Execute code at 164
                                Execute 225
359 (space)                     Links to last item in EDITOR
                                vocabulary. Calls 227
360 WHERE       DUP             Stack: IN BLK BLK
                B/SCR /         Stack: IN BLK BLK+B/SCR
                DUP SCR !       Set SCR from TOS.
                                Stack unchanged.
                (.") text       Output "SCR # "
                DECIMAL .       Output screen number.
                SWAP C/L        Stack: BLK IN C/L
                /MOD            Stack: BLK rem quotient
                C/L *           Stack: BLK rem line
                ROT             Stack: rem line BLK
                BLOCK +         Stack: rem line+addr
                CR              Newline
                C/L TYPE        Type line.
```

```
                CR              Newline (Stack: rem)
                HERE C@ -       Stack: rem - (HERE)
                SPACES          Output TOS spaces
                LIT 005E        Code for upward arrow
                EMIT            Output character
                EDITOR          Select EDITOR vocabulary
                QUIT            Return control to user.
```

This very useful routine may be called following an error to find the location of the error. It sets EDITOR mode and clears the stacks.

This is the start of the EDITOR vocabulary. This item links on to the start of the FORTH vocabulary via entry 227. In VLIST, with the EDITOR selected, the EDITOR vocabulary appears first, followed by the FORTH vocabulary.

```
361 #LOCATE     R# @            Read cursor location
                C/L /MOD        Stack: column line
362 #LEAD       #LOCATE         See above
                LINE            Stack: column addr
                SWAP            Stack: addr column
363 #LAG        #LEAD           See above
                DUP             Stack: addr column column
                >R              TOS to TORS
                +               Stack: addr+column
                C/L R>          Stack: addr+column C/L column
                -               Stack: addr+column
                                C/L-column
364 -MOVE       LINE            Stack: addr1 addr2
                                (TOS = address of line)
                C/L             Stack: addr1 addr2 C/L
                CMOVE           Copy one line from addr1 to addr2
                UPDATE
365 H           LINE PAD 1 +    Stack: Line-addr PAD+1
                C/L DUP         Stack: Line-addr PAD+1 C/L C/L
                PAD C!          Stack: 1st PAD location set to C/L
                CMOVE           Copy one line from Line-addr
                                to PAD.
366 E           LINE C/L        Stack: addr C/L
                BLANKS          Fill line with spaces
                UPDATE
367 S           DUP 1 -         Stack: Line Line-1
                LIT 000E
                (DO)
368             I LINE          Address of line I
                I 1+            I+1
                -MOVE           Copy line I to line I+1
```



```
                LIT -1
                (+LOOP) FFF0    Go to 368
                E               Erase line
369 D           DUP             Stack: Line Line
                H               Copy line to PAD
                LIT 000F
                DUP ROT         Stack: 15 15 Line
                (DO)
370             I 1+ LINE       Address of line I+1
                I
                -MOVE           Copy line I+1 to line I
                (LOOP) FFF4     Go to 370
                E               Erase line 15.
371 M           R# +!           Add TOS to cursor position.
                CR SPACE        Newline, space
                #LEAD TYPE      Output text to cursor
                LIT 005F        Code for underline
                EMIT            output code
                #LAG TYPE       Output text from cursor to
                                end of line.
                #LOCATE .       Output line number
                DROP            Discard column.
372 T           DUP             Stack: Line Line
                C/L *           Stack: Line Line*C/L
                R# !            Set cursor position to line start.
                DUP             Stack = Line Line
                H               Copy line to PAD
                O M             Output line. (cursor at left end)
373 L           SCR @           Read current screen number
                LIST            List that screen
                O M             Display cursor line.

374 R           PAD 1+ SWAP     Stack: PAD+ Line
                -MOVE           Copy line from PAD to
                                specified line.

375 P           1 TEXT          Set text in PAD. (Delimiter is 1)
                R               Copy text to specified line.

376 I           DUP             Stack: Line Line
                S               Spread at Line
                R               Copy line from PAD
```

Note that R and I are not unique. While the EDITOR is effective, the FORTH meanings are replaced. However, the original R is used by several EDITOR definitions.

```
377 TOP       0 R# !          Set cursor position to zero.
378 CLEAR     SCR !           Set screen from TOS
              LIT 0010 0      Sixteen iterations
              (DO)
379           I E             Erase line I
              (LOOP) FFFA     Go to 379
380 COPY      B/SCR *         Stack: SCR1 SCR1*B/SCR
              OFFSET @ +      Add OFFSET
              SWAP            Stack: SCR2*B/SCR+OFFSET
                              SCR1
              B/SCR *         Stack: SCR2*B/SCR+OFFSET
                              SCR1*B/SCR (= A B)
              B/SCR OVER +    Stack: A B B+B/SCR
              SWAP            Stack: A B+B/SCR B
              (DO)
381           DUP             Stack: A A
              I BLOCK         Address of block I
              2 - !           Write A to address-2
              1+              Stack: A+1
              UPDATE
              (LOOP) FFEE     Loop to 381
              DROP            Discard A
              FLUSH           Copy updated buffers to
                              RAM-disc.
382 -TEXT     SWAP            Stack: Addr1 Addr2 Count
              -DUP            Duplicate TOS if non-zero
              0BRANCH 002A    If zero go to 386
              OVER +          Stack: Addr1 Addr2
                              Addr2+Count
              SWAP            Stack: Addr1 Addr2+Count
                              Addr2
              (DO)
383           DUP             Stack: Addr Addr
              C@ I C@ -       Stack: Addr (Addr)-(I)
              0BRANCH 000A    If zero go to 384
              0=              True flag if zero
              LEAVE
              BRANCH 0004     Go to 385
384           1+              Increment address
385           (LOOP) FFE6     Loop to 383
              BRANCH 0006     End
386           DROP 0=         Discard address. Set false flag.
387 MATCH     >R >R           Stack: Addr1 Count1.
                              (Addr2 Count2 on RS)
```



```
              2DUP
              R> R>           Stack: Addr1 Count1 Addr1
                              Count1 Addr2 Count2
              2SWAP           Stack: Addr1 Count1 Addr2
                              Count2 Addr1 Count1
              OVER +          Stack: Ad1 Ct1 Ad2 Ct2 Ad1
                              Ad1+Ct1
              SWAP            Stack: Ad1 Ct1 Ad2 Ct2
                              Ad1+Ct1 Ad1
              (DO)
388           2DUP            Stack: Ad1 Ct1 Ad2 Ct2 Ad2 Ct2
              I -TEXT         Stack: Ad1 Ct1 Ad2 Ct2 flag
              0BRANCH 001A    If false go to 389
              >R 2DROP R>     Discard 2OS,3OS
              - I SWAP -
              0 SWAP 0 0
              LEAVE           Terminate loop
389           (LOOP) FFDC     Loop to 388
              2 DROP SWAP
              0=              Reverse flag
              SWAP
390 1LINE     #LAG PAD        Stack: cursor toEOL PAD
              COUNT           Stack: cursor toEOL PAD count
              MATCH
              R# +!           Update cursor position.
391 FIND      LIT 03FF
              R# @ <          True if cursor position less than
                              03FF
              0BRANCH 0012    If zero go to 392
              TOP             Zero cursor position.
              PAD HERE
              C/L 1+
              CMOVE           Copy from PAD to HERE,
                              C/L+1 bytes
              0 ERROR         Report error 0
392           1LINE           See above
              0BRANCH FFDE    If zero go to 391
394 DELETE    >R #LAG         Stack: Cursor toEOL n to TORS.
              + R -           Stack: Cursor toEOL-n
              #LAG            Stack: Cursor toEOL-n
                              Cursor toEOL
              R MINUS         Add to stack -n
              R# +!           Add -n to cursor position
              #LEAD +
              SWAP
```

```
                CMOVE
                R> BLANKS
                UPDATE
395 N           FIND            Search
                O M             Display
396 F           1 TEXT          Text to PAD
                N               Find next
398 B           PAD C@          Read length from PAD
                MINUS           Negate
                M               Add to cursor position.
399 X           1 TEXT          Text to PAD
                FIND            Locate
                PAD C@          Read length from PAD
                DELETE          Delete last n characters
                O M             Display line
400 TILL        #LEAD +         Line+column
                1 TEXT          Text to PAD
                1LINE
                0=              Reverse flag
                0 ?ERROR        Error if true
                #LEAD +         Line+column
                SWAP -
                DELETE
                0 M             Display line
401 C           1 TEXT          Text to PAD
                PAD COUNT       Modify references
                #LAG
                ROT OVER        Stack a b c to b c a c
                MIN >R
                R R# +!         Add TORS to cursor position.
                R - >R
                DUP HERE R
                CMOVE
                HERE #LEAD
                + R>
                CMOVE
                R>
                CMOVE
                UPDATE
                0 M             Display line
```

This completes the EDITOR vocabulary. When the vocabulary is enabled, dictionary searches start with the C entry. The NEXT entry below links on to WHERE (360)

```
402             NEXT            Constant returning address of
                                link 3
```



```
403             PUSHHL          Constant returning address of
                                link 2
404             PUSHDE          Constant returning address of
                                link 1
405 INP                         POP HL. PUSH BC. BC=HL IN
                                A, (C). POP BC H = 0. L = A.
                                PUSH HL. Go to NEXT1
406 OUTP                        POP HL. POP DE. PUSH BC.
                                BC = HL A = E. OUT (C),A.
                                POP BC. Go to NEXT1
407 SCREEN                      POP HL. POP DE. PUSH BC.
                                PUSH IX. C = E. B = L.
                                CALL 2538. CALL 2BF1. A =
                                (DE). H = 0. L = A. POP IX.
                                POP BC. PUSH HL. Go to NEXT1.
408 AT          ABS DUP         Stack: line col col
                LIT 001F >
                0BRANCH 0008    If col not greater than 31 go to 409
                2DROP           Clear stack
                BRANCH 0022     End.
409             SWAP ABS DUP    Stack: col line line
                LIT 0015 >
                0BRANCH 0008    If line not greater than 21
                                go to 410
                2DROP           Clear stack
                BRANCH 000C     End
410             LIT 0016
                EMIT            Output 22
                EMIT            Output line
                EMIT            Output col
411 BORDER                      POP HL. PUSH BC. A = L.
                                CALL 2297. POP BC.
                                Go to NEXT1
412 BLEEP                       POP HL. POP DE. PUSH BC.
                                PUSH IX. CALL 03B5. POP IX.
                                POP BC. Go to NEXT1.
413 PAPER       ABS DUP
                LIT 0009 >
                0BRANCH 0008    If parameter not greater than 9
                                go to 414
                DROP            Clear stack
                BRANCH 0088     End
414             DUP
                LIT 0009 =
```

```
                0BRANCH 001A If parameter not 9 then go to 415
                LIT 5C91     Address of PFLAG
                C@           Read PFLAG
                LIT 0080 OR  Set bit 3
                LIT 5C91 C!  Write to PFLAG
                DROP         Discard parameter.
                BRANCH 0064  End
415             DUP
                LIT 0008 =
                0BRANCH 001A If parameter not 8 then go to 416
                LIT 5C8E     Address of MASKP
                C@           Read MASKP
                LIT 0038 OR  Set bits 3, 4, 5
                LIT 5C8E     Address of MASKP
                DROP         Discard parameter.
                BRANCH 0040  End
416             LIT 0008 *   Multiply parameter by 8
                LIT 5C8D C@  Read ATTRP
                LIT 00C7 AND Zero bits 3, 4, 5
                OR           OR parameter
                LIT 5C8D C!  Write to ATTRP
                LIT 5C91 C@  Read PFLAG
                LIT 007F AND Zero bit 7
                LIT 5C91 !   Write to PFLAG
                LIT 5C8E C@  Read MASKP
                LIT 00C7 AND Zero bits 3, 4, 5
                LIT 5C8E !   Write to MASKP
417 ATTR                     POP HL. POP DE. PUSH BC.
                             PUSH IX. C = E. B = L.
                             CALL 2583. CALL 1E94. H = 0.
                             L = A. POP IX. POP BC.
                             PUSH HL. Go to NEXT1
418 POINT                    POP HL. POP DE. PUSH BC.
                             PUSH IX. C = E. B = L. A = L.
                             If A is not less than HOH then
                             A = AFH, B = A. CALL 22CE.
                             CALL 1E94. H = 0. L = A.
                             POP IX. POP BC. PUSH HL.
                             Go to NEXT1.
419 INK         ABS DUP
                LIT 0009 >
                0BRANCH 0008 If parameter not greater than 9
                             go to 420
                DROP         Discard parameter.
                BRANCH 0082  End
```



```
420             DUP
                LIT 0009 =
                0BRANCH 001A If parameter not 9 then
                             go to 421.
                LIT 5C91 C@  Read PFLAG
                LIT 0020 OR  Set bit 5
                LIT 5C91 C!  Write to PFLAG
                DROP         Discard parameter
                BRANCH 005E  End
421             DUP
                LIT 0008 =
                0BRANCH 001A If parameter not 8 then
                             go to 422
                LIT 5C8E C@  Read MASKP
                LIT 0007 OR  Set bits 0, 1, 2.
                LIT 5C8E C!  Write to MASKP
                DROP         Discard parameter
                BRANCH 003A  End
422             LIT 5C8D C@  Read ATTRP
                LIT 00F8 AND Zero bits 0, 1, 2.
                OR           OR parameter.
                LIT 5C8D C!  Write to ATTRP
                LIT 5C91 C@  Read PFLAG
                LIT 00DF AND Zero bit 5
                LIT 5C91 !   Write to PFLAG (Word write)
                LIT 5C8E C@  Read MASKP
                LIT 00F8 AND Zero bits 0, 1, 2.
                LIT 5C8E !   Write to MASKP (Word write)
423 FLASH       0BRANCH 0018 If parameter 0 then go to 424
                LIT 5C8D C@  Read ATTRP
                LIT 0080 OR  Set bit 7
                LIT 5C8D !   Write to ATTRP (Word write)
                BRANCH 0014  End
424             LIT 5C8D C@  Read ATTRP
                LIT 007F AND Zero bit 7
                LIT 5C8D !   Write to ATTRP (Word write)
425 BRIGHT      0BRANCH 0018 If parameter 0 then go to 426
                LIT 5C8D C@  Read ATTRP
                LIT 0040 OR  Set bit 6
                LIT 5C8D !   Write to ATTRP (Word write)
                BRANCH 0014  End
426             LIT 5C8D C@  Read ATTRP
                LIT 00BF AND Zero bit 6
                LIT 5C8D !   Write to ATTRP (Word write)
```

```
427 GOVER       0BRANCH 0016 If parameter 0 go to 428
                LIT 5C91 C@  Read PFLAG
                2 OR         Set bit 1
                LIT 5C91 !   Write to PFLAG
                BRANCH 9914  End
428             LIT 5C91 C@  Read PFLAG
                LIT 00FD AND Zero bit 1
                LIT 5C91 !   Write to PFLAG (Word write)
429 INVERSE     0BRANCH 0018 If parameter 0 go to 430
                LIT 5C91 C@  Read PFLAG
                LIT 0008 OR  Set bit 3
                LIT 5C91 !   Write to PFLAG
                BRANCH 0014  End
430             LIT 5C91 C@  Read PFLAG
                LIT 00F7 AND Zero bit 3
                LIT 5C91 !   Write to PFLAG (Word write)
431 NOT         0=           Synonyms.
432 I'                       HL = (RSP) + 2. DE = (HL).
                             PUSH DE. Go to NEXT1.
433 J                        HL = (RSP) + 4. DE = (HL).
                             PUSH DE. Go to NEXT1.
434 2CONSTANT   CREATE
                SMUDGE
                HERE 2!      Write double number.
                LIT 0004
                ALLOT        Reserve four more bytes.
                (;CODE)      Link to following code.
435                          INC DE. Exchange DE,HL.
                             HL = HL + 2 DE = (HL).
                             PUSH DE. HL = HL - 2.
                             DE = (HL). PUSH DE.
                             Go to NEXT1.
436 2VARIABLE   2CONSTANT
                (;CODE)      Link to following code.
437                          INC DE. PUSH DE. Go to NEXT1.
438 U.R         >R 0 R>      Make 2OS double number
                             (unsigned)
                D.R
```


Additional pages for ADVANCED SPECTRUM FORTH

```
439 2OVER       2SWAP        Stack: a b c d becomes c d a b
                2DUP         Stack: c d a b a b
                >R >R        a b to return stack. Stack c d a b
                2SWAP        Stack: a b c d
                R> R>        Restore a b
```

In early issues, this contained an error, in that the last two words were >R instead of R>. To correct the error, locations 7F4E and 7D50 need to be changed from 6173 to 6188.

```
440 EXIT        >R DROP      Discard TORS.
```

This word needs to be used with extreme care.

```
441 PLOT                     POP HL. POP DE. PUSH BC.
                             PUSH IX. C = E. B = L. If H or D
                             is non-zero go to 442.
                             If L exceeds AFH go to 442.
                             CALL 22DF
442                          POP IX. POP BC. Go to NEXT1.
443 X1                       A double variable used by DRAW
444 Y1                       A double variable used by DRAW
445 INCX                     A double variable used by DRAW
446 INCY                     A double variable used by DRAW
447 DRAW        LIT 5C7E C@  Read LASTY
                DUP 0 SWAP   Stack: x y LASTY 0 LASTY
                Y1 2!        Store 65536*LASTY in Y1.
                - DUP ABS    Stack: x y-LASTY
                             y-LASTY(abs) x
                ROT          Stack: y-LASTY
                             y-LASTY(abs) x
                LIT 5C7D C@  Read LASTX
                DUP 0 SWAP   Stack: y-LASTY y-LASTY(abs)
                             x LASTX 0 LASTX
                X1 2!        Store 65536*LASTX in X1
                - DUP ABS    Stack: y-LASTY y-LASTY(abs)
                             x-LASTX x-LASTX(abs)
                ROT MAX      Stack: y-LASTY x-LASTX
                             (greater of absolutes)
```

```
        >R DUP 0<       Stack: y-LASTY x-LASTX flag
        0BRANCH 0012    If x-LASTX positive go to 448
        ABS 0 SWAP      Stack: y-LASTY 0
                        x-LASTX(abs)
        R M/MOD         Divide 65536*(x-LASTX) by
                        greater of absolutes.
        DMINUS          Negate quotient
        BRANCH 000A     Go to 449. Stack: y-LASTY
                        rem double-quotient.
448     0 SWAP R        Stack: y-LASTY 0 x-LASTX
        M/MOD           Divide 65536*x-LASTX by
                        greater of absolutes.
449     INCX 2!         Store quotient in INCX
        DROP            Discard remainder
        DUP 0<          Stack: y-LASTY flag
        0BRANCH 0012    If y-LASTY positive go to 450
        ABS 0 SWAP      Stack: 0 y-LASTY(abs)
        R M/MOD         Divide 65536*(y-LASTY(abs))
                        by greater of absolutes.
        DMINUS          Negate quotient
        BRANCH 000A     Go to 451.
450     0 SWAP R        Stack: 0 y-LASTY
        M/MOD           Divide 65536*(y-LASTY) by
                        greater of absolutes.
451     INCY 2!         Store quotient in INCY
        DROP            Discard remainder
        R> 1+ 0
        (DO)
452     X1 @            Read upper half of X1
        Y1 @            Read upper half of Y1
        PLOT
        X1 2@           Read X1
        INCX 2@         Read INCX
        D+ X1 2!        Store double sum in X1
        Y1 2@           Read Y1
        INCY 2@         Read INCY
        D+ Y1 2!        Store double sum in Y1
        (LOOP) FFD8     Loop to 452
*453 CASE   ?COMP       Error if not compiling
        CSP @           Read SP hold
        !CSP            Set SP hold from SP
        LIT 0004        PAIRS reference
*454 OF LIT 0004
        ?PAIRS          Check pairing
```



```
        COMPILE OVER
        COMPILE =
        COMPILE 0BRANCH
        HERE 0 ,
        COMPILE DROP
        LIT 0005        PAIRS reference
*455 ENDOF  LIT 0005
        ?PAIRS          Check pairing
        COMPILE BRANCH
        HERE 0 ,
        SWAP 2
        ENDIF
        LIT 0004        PAIRS reference
*456 ENDCASE    LIT 0004
        ?PAIRS          Check pairing
        COMPILE DROP
457     SP@             Read SP
        CSP @           Read SP hold
        = 0=
        0BRANCH 000A    If SP = SP hold go to 458
        2 ENDIF
        BRANCH FFEC     Go to 457
458     CSP !           Set SP hold from TOS
459 INKEY               PUSH BC. CALL 028E. C = 0.
                        If A ≠ 0 go to 460.
                        Else call 031E. If no carry go
                        to 460. DEC D. E = A. CALL 0333.
460                     L = A. H = 0. POP BC.
                        Go to NEXT1.
461 INIT-DISC   LIT D000
        LIT FF2B
        BLANKS          Clear D000 – FF2B to
                        space codes
462 UDG     LIT 5C7B @  Read UDG
```

INDEX

The first number against each word is a link in Appendix A.

FORTH Vocabulary

!	77	21,24	.R	326	11,12
!CSP	146	46	/	253	19,20
#	322	42	/MOD	252	19,20
#>	320	43	0	90	47
#BUFF	280	53	1	91	47
#S	324	42	2	92	47
'	302	71	3	93	47
(	229	50	0<	62	38
(.")	173	41	0=	61	38
(;CODE)	160	70	0BRANCH	8	37
(+LOOP)	13	73	1+	124	17,19
(ABORT)	205	46	2+	125	17,20
(DO)	14	69	2!	79	21,24
(FIND)	19	—	2@	76	21,24
(LINE)	258	—	2CONSTANT	434	22
(LOOP)	9	69	2DRCP	347	15
(NUMBER)	198	—	2DUP	71	15,18
(TAPE)	340	49	2OVER	439	15,16
*	251	19,20	2SWAP	348	15
*/	256	19,20	2VARIABLE	438	22,24
*/MOD	255	19,20	:	80	32,35
+	63	4,5	;	82	32,36
+!	72	21	;CODE	161	70
+-	241	17,20	;S	56	70
+BUF	281	54	<	132	38
+LOOP	309	39	<#	319	42
+ORIGIN	100	47	<BUILDS	162	72
,	128	—	=	131	38
—	130	13,14	>	135	38
—>	301	49	>R	58	16
—DUP	138	39	?	329	21,24
—FIND	204	42	?COMP	149	46
—TRAILING	169	43	?CSP	152	46
.	328	9,11	?ERROR	147	45
."	174	41	?EXEC	150	46
.CPU	339	43	?LOADING	153	46
.LINE	259	54	?PAIRS	151	46

HEX	158	11,12
HI	295	53
HLD	123	43
HOLD	193	43
I	15	39
I'	432	39
ID.	209	43
IF	314	38
IMMEDIATE	223	71
IN	111	43
INDEX	352	50
INIT—DISC	461	49
INK	419	25
INKEY	459	44
INP	405	45
INTERPRET	216	12
INVERSE	429	25
J	433	39
KEY	33	44
LATEST	141	84,34
LEAVE	57	40
LFA	142	32,34
LIMIT	97	53
LINE	343	54
LINK	337	3
LIST	334	59,49,50
LIT	5	—
LITERAL	213	30,71
LO	294	53
LOAD	300	49
LOADT	344	49
LOOP	308	39
M*	249	18,19,20
M/	250	18,19,20
M/MOD	257	17,20
MAX	247	17,20
MESSAGE	260	46
MIN	245	17,20
MINUS	65	14
MOD	254	19,20
MON	341	2
NEXT	402	—
NEXT1	3	—
NEXT2	4	—
NFA	144	32,34
NOOP	83	47
NOT	431	38
NUMBER	201	43
OF	454	41
OFFSET	114	54
OR	50	16,19
OUT	112	42
OUTP	406	45
OVER	67	15,16
PAD	194	51
PAPER	413	25
PFA	145	32,34
PLOT	441	26
POINT	418	26
PREV	279	54
PUSHDE	1404	—
PUSHHL	2403	—
QUERY	184	44
QUIT	230	50,53,40
R	60	16
R#	122	51
R/W	296	55
R>	59	16
RO	102	46
REPEAT	313	40
ROT	136	15
RP@	54	46
RP!	55	46
S—>D	240	14
SO	101	46
SAVET	345	49
SCR	112	54
SCREEN	407	25
SIGN	321	42
SIZE	349	47
SMUDGE	157	33,34
SP!	53	46
SP@	52	46
SPACE	137	16
?STACK	215	46
?TERMINAL	34	38
@	74	21,24
ABORT	233	40
ABS	243	17,20
AGAIN	312	39
ALLOT	127	22,24
AND	49	16,19
AT	408	25
ATTR	417	25
B/BUF	98	53
B/SCR	99	53
BACK	303	69
BASE	118	11,12
BEGIN	304	39
BL	94	—
BLANKS	129	24,25
BLEEP	412	26
BLK	110	53
BLOCK	290	54
BRANCH	7	37
BRIGHT	425	25
BORDER	409	25
BUFFER	287	54
C!	78	21,24
C/L	95	53
C,	129	70
C@	78	21,24
CASE	453	41
CFA	143	32,34
CLS	338	16
CMOVE	36	24,25
COLD	239	47
COMPILE	154	47
CONSTANT	84	22,24
CONTEXT	115	45
COUNT	165	43
CR	35	16
CREATE	210	70
CSP	121	46
CURRENT	116	70
D+	64	14
D+-	242	17,20
D.	327	10,11
D.R	325	11,12
DABS	244	17,20
DECIMAL	159	11,12
DEFINITIONS	228	50
DIGIT	16	44
DLITERAL	214	71
DMINUS	66	14
DO	307	39
DOES>	163	72
DP	108	—
DPL	119	10,43,44
DRAW	447	26
DRO	286	54
DROP	68	15
DUP	70	14
EDITOR	358	45,49
ELSE	315	38
EMIT	32	42
EMPTY BUFFERS	284	54
ENCLOSE	28	—
END	311	40
ENDCASE	456	41
ENDIF	305	39
ENDOF	455	41
ERASE	191	24,25
ERROR	206	45
EXECUTE	6	47
EXIT	440	40
EXPECT	176	44
FENCE	107	45,75
FILL	188	23,25
FIRST	96	53
FLASH	423	25
FLD	120	—
FLUSH	298	54
FORGET	351	34,45
FORTH	226	45
FREE	350	3,46
GOVER	427	25
HERE	126	46

SPACES	317	16	USE	278	54
STATE	117	45	USER	88	47
SWAP	69	15,17	VARIABLE	86	21,24
TEXT	342	55	VERIFY	346	49
THEN	306	39	VLIST	331	33,50
TIB	103	13	VOC-LINK	109	—
TOGGLE	73	24,25,34	VOCABULARY	224	74
TRAVERSE	139	32,34	WARM	236	47
TRIAD	355	50	WARNING	106	46,50
TYPE	166	43	WHERE	360	46
U<	133	38	WHILE	316	40
U*	38	13,14	WIDTH	105	71
U.	330	10,11	WORD	195	41
U.R	438	11,12	X	185	71
U/MOD	42	13,14	XOR	51	20,18
UDG	462	26	[	155	70
UNTIL	310	40	[COMPILE]	212	71
UPDATE	283	55	]	156	70

EDITOR Vocabulary

#LAG	363	52	FIND	391	52
#LEAD	362	52	H	365	51
#LOCATE	361	52	I	376	51
—MOVE	364	52	L	373	51
—TEXT	382	52	M	371	51
1LINE	390	52	MATCH	387	52
B	398	51	N	395	51
C	401	51	P	375	51
CLEAR	378	51	R	374	51
COPY	380	51	S	367	51
D	369	51	T	372	51
DELETE	394	52	TILL	372	51
E	366	51	TOP	377	51
F	396	51	X	399	51

ADVANCED SPECTRUM FORTH
MELBOURNE HOUSE REGISTRATION CARD

Please fill out this page and return it promptly in order that we may keep you informed of new software and special offers that arise. Simply fill in and indicate the correct address on the reverse side.

Name .....

Address .....

..... Code .....

Which computer do you own? .....

Where did you learn of this product?

☐ Magazine. If so, which one? .....

☐ Through a friend

☐ Saw it in a Retail Shop

☐ Other. Please specify .....

Which magazines do you purchase?

Regularly: .....

Occasionally: .....

What Age are you?

☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ Over 25

We are continually writing new material and would appreciate receiving your comments on our product.

How would you rate this software?

☐ Excellent ☐ Value for money
☐ Good ☐ Priced right
☐ Poor ☐ Overpriced

Please tell us what books/software you would like to see produced for your SPECTRUM.

_____

_____

_____

PUT THIS IN A STAMPED ENVELOPE AND SEND TO:

In the United Kingdom return page to:
Melbourne House (Publishers) Ltd., Melbourne House, Church Yard, Tring, Hertfordshire, HP23 5LU

In Australia & New Zealand return page to:
Melbourne House (Australia) Pty. Ltd., Suite 4, 75 Palmerston Crescent, South Melbourne, Victoria, 3205.

ADVANCED SPECTRUM FORTH is an essential aid to anyone who wishes to discover FORTH's true potential on their Spectrum, providing the bridge from beginner status to advanced FORTH programmer, using examples and detailed explanations.

The popularity of FORTH as an alternative language, combining the ease of a high level language and the speed of machine code, has led to it being available to the Spectrum user.

However, to really utilise the power and flexibility of FORTH, a lot more in-depth knowledge of the language is required than just knowing and being able to use the pre-defined FORTH words.

This book is designed to provide details and to illustrate the techniques that will allow the FORTH programmer, who is already familiar with FORTH, to realise the full potential of FORTH as a language.

Apart from an in-depth look at Abersoft's FIG-FORTH this book deals with specific programming techniques and FORTH program design considerations, using examples.

This book contains sections which provide a close look at how FORTH handles arithmetic, how to manipulate strings and arrays, how to define new dictionary definitions, and a chapter which covers the peculiarities of FORTH on the Spectrum.

Aimed specifically at those programmers who know the essentials of FORTH and dealing exclusively with Abersoft FORTH for the Spectrum, this book provides those details about the Spectrum and about Abersoft FORTH that more general books cannot provide.

£8·95

ISBN 0-86161-142-X

ADVANCED SPECTRUM FORTH

Don Thomasson

Melbourne
House
Publishers



SPECTRUM

ADVANCED SPECTRUM FORTH is an essential aid to anyone who wishes to discover FORTH's true potential on their Spectrum, providing the bridge from beginner status to advanced FORTH programmer, using examples and detailed explanations.

The popularity of FORTH as an alternative language, combining the ease of a high level language and the speed of machine code, has led to it being available to the Spectrum user.

However, to really utilise the power and flexibility of FORTH, a lot more in-depth knowledge of the language is required than just knowing and being able to use the pre-defined FORTH words.

This book is designed to provide details and to illustrate the techniques that will allow the FORTH programmer, who is already familiar with FORTH, to realise the full potential of FORTH as a language.

Apart from an in-depth look at Abersoft's FIG-FORTH this book deals with specific programming techniques and FORTH program design considerations, using examples.

This book contains sections which provide a close look at how FORTH handles arithmetic, how to manipulate strings and arrays, how to define new dictionary definitions, and a chapter which covers the peculiarities of FORTH on the Spectrum.

Aimed specifically at those programmers who know the essentials of FORTH and dealing exclusively with Abersoft FORTH for the Spectrum, this book provides those details about the Spectrum and about Abersoft FORTH that more general books cannot provide.

£8·95

Melbourne
House
Publishers

ISBN 0-86161-142-X

9 780861 611423

ADVANCED SPECTRUM FORTH

Don Thomasson

Melbourne
House
Publishers

ADVANCED SPECTRUM FORTH

Don Thomasson



ADVANCED SPECTRUM FORTH is an essential aid to anyone who wishes to discover FORTH's true potential on their Spectrum, providing the bridge from beginner status to advanced FORTH programmer, using examples and detailed explanations.

The popularity of FORTH as an alternative language, combining the ease of a high level language and the speed of machine code, has led to it being available to the Spectrum user.

However, to really utilise the power and flexibility of FORTH, a lot more in-depth knowledge of the language is required than just knowing and being able to use the pre-defined FORTH words.

This book is designed to provide details and to illustrate the techniques that will allow the FORTH programmer, who is already familiar with FORTH, to realise the full potential of FORTH as a language.

Apart from an in-depth look at Abersoft's FIG-FORTH this book deals with specific programming techniques and FORTH program design considerations, using examples.

This book contains sections which provide a close look at how FORTH handles arithmetic, how to manipulate strings and arrays, how to define new dictionary definitions, and a chapter which covers the peculiarities of FORTH on the Spectrum.

Aimed specifically at those programmers who know the essentials of FORTH and dealing exclusively with Abersoft FORTH for the Spectrum, this book provides those details about the Spectrum and about Abersoft FORTH that more general books cannot provide.

£8·95

ISBN 0-86161-142-X

9 780861 611423

Melbourne House Publishers

ADVANCED SPECTRUM FORTH

Don Thomasson

Melbourne House Publishers

ADVANCED SPECTRUM FORTH

Don Thomasson



Published in the United Kingdom by:
Melbourne House (Publishers) Ltd.,
Church Yard,
Tring, Hertfordshire HP23 5LU.
ISBN 0 86161 142 X

Published in Australia by:
Melbourne House (Australia) Pty. Ltd.,
Suite 4, 75 Palmerston Crescent,
South Melbourne, Victoria, 3205.

© 1984 by Don Thomasson.

All rights reserved. This book is copyright. No part of this book may be copied or stored by any means whatsoever whether mechanical or electronic, except for private or study use as defined in the Copyright Act. All enquiries should be addressed to the publishers.

Printed in Hong Kong by Colorcraft Ltd.

The terms Sinclair, ZX, ZX80, ZX81, ZX Spectrum, ZX Microdrive, ZX Interface, ZX Net Microdrive, Microdrive Cartridge, ZX Printer and ZX Power Supply are all Trade marks of Sinclair Research Limited.

DCBA9876543210

CONTENTS

INTRODUCTION 1
Getting Started 2
THE ESSENTIALS OF FORTH 4

PART I: DIRECT MODE 9
NUMBER REPRESENTATION 9
Signed and Unsigned Numbers 10
Double Numbers 10
Number Base 10
Formatted Numbers 11
ARITHMETIC PRIMITIVES 12
STACK MANIPULATORS 14
OTHER MANIPULATORS 16
MORE ARITHMETIC 16
MEMORY ACCESS 20
SPECTRUM SPECIALS 25
SUMMARY OF PART 1 26

PART II: THE DICTIONARY 29

PART III: COLON DEFINITIONS 35
EXTENDING THE DICTIONARY 35
BRANCHING AND LOOPING 37
CONDITIONAL BRANCHING 38
STRINGS AND THINGS 41
INPUT/OUTPUT 43
ODDMENTS 45

PART IV: THE RAM DISC 49
SCREENS 49
THE EDITOR 50
BEHIND THE SCREENS 53

PART V: SIMPLE PROGRAMS 57
PLANNING A PROGRAM 60