BASIC MANUAL

Contents

Introduction - intro 28/7/80

This manual explains how to use Acorn's 4k integer BASIC interpreter. The manual is arranged in two sections. If you have never programmed before you should read the BASIC section starting from Chapter 1; but be warned that you are setting off on an adventure which may require some changes of attitude towards computers. The only way to learn the art of programming is by practice, and so every section of this manual includes many example programs which illustrate the concepts being explained. These should be typed in and tried out, even if at first you do not fully understand how they work. By the end of chapter 4 you will be able to write your own programs for many different types of problem, and you may wish to stop there. The subsequent chapters, 5 to 12, deal with progressively more advanced features of Acorn BASIC.

If you have already programmed in BASIC you may prefer to turn to the reference section; this contain a complete summary of all the BASIC statements, functions, commands and operators. You will be pleased to discover a number of extensions in Acorn BASIC that are not found in other BASICs.

Acknowledgements

The preparation of this manual would not have been possible without the continuous assistance of everyone at Acorn. In particular thanks are due to David Johnson-Davies for writing chapter outlines and writing many of the chapters; to Laurence Hardwick for testing the example programs and writing sections about the teletext VDU and to the many people who provided comments on previous drafts of the manual.

The following example programs were provided by Roger Wilson: Tower of Hanoi, Eight Queens, Prime Numbers, Arbitrary Precision Powers, Day of Week, Lissajou patterns and Renumber; and the following by Nick Toop: Simultaneous Equations and Encoder/Decoder.

The manual was prepared and edited on Acorn System Threes and System Fours, and the final artwork was prepared using the Acorn Text Processing Package.

1.0 START HERE

The BASIC interpreter occupies memory between addresses #C000 and #D000, and it should be entered at #C2B2. It requires one of the Acorn Operating Systems to handle the various devices attached to the computer. If you have BASIC in ROM, you should enter it by typing

```
GO C2B2
```

if you have a DOS. With a COS you can type

```
BASIC
```

and the COS will enter the interpreter at #C2B2. If you have BASIC on cassette, you can enter

```
RUN "BASIC"
```

to your COS to get it to load BASIC and enter it at #C2B2. If you have BASIC on disk, you can enter one of

```
RUN "BASIC"
RUN BASIC
BASIC
```

to get BASIC loaded and enter it at #C2B2. If you wish to save BASIC on cassette or disk, a suitable command is

```
SAVE "BASIC" C000 D000 C2B2
```

for either COS or DOS. The interpreter uses addresses in page 0 up to #0060, all of page 1 and between #0240 and #03FF for working values. When initially entered it expects RAM at #3000 for text to be stored in, but this can be changed.

When you first enter BASIC at #C2B2, it will just type a '>' and wait. The '>' sign is called BASIC's 'prompt'. It indicates that the interpreter is waiting for something to be typed in, a command, perhaps, or a program.

1.1 What BASIC Can Do

Acorn BASIC understands the following special words and symbols:

Commands

LIST, LOAD, NEW

Functions

ABS, BGET, CH, COUNT, EXT, FIN, FOUT, GET, LEN, PTR, RND, TOP

Connectives

AND, OR, STEP, THEN, TO.

Statements

BPUT, DO, END, FOR, GOSUB, GOTO, IF, INPUT, LET, LINK, NEXT, PRINT, PUT, REM, RETURN, RUN, SAVE, SGET, SHUT, SPUT, UNTIL

Operators

!, #, $, %, &, *, +, -, /, :, <, =, >, ?, |, <>, <=, >=

These words and symbols will be explained over the course of the next 12 chapters; for the moment just observe that many of these words have an obvious meaning; for example, try typing:

```
PRINT "HELLO"
```

after the '>' prompt sign. Note that the quotation marks are obtained by holding down the SHIFT key and typing the '2' key. Now type RETURN to indicate that the line is finished, and BASIC will do just that:

```
HELLO>
```

To perform calculations you just need to type PRINT followed by the expression you want to evaluate. For example, try:

```
PRINT 7+6*2
```

When you type RETURN the answer will be printed out. You can try typing anything you like, but any words not on the above lists will probably cause an error. For example, try typing:

```
HELLO
```

after BASIC's '>' prompt. BASIC will reply by printing:

```
Error  94
```

which means that HELLO is not one of the statements or commands that the ATOM understands.

1.2 A Demonstration

Here is a quick demonstration of some more complicated things that BASIC can do. No attempt is made here to explain how these examples work; for that you will have to read the rest of this manual.

You can make BASIC do a lot of typing with very little effort; try entering:

```
DO PRINT "Acorn BASIC "; UNTIL 0
```

Note the difference between the 'O' of DO, which is the letter 'O' and the '0' at the end of the statement, which is the digit '0' on the top row of the keyboard. You will have to type the ESC (escape) key which is at the top left of the keyboard, to stop this program.

Now try typing in the following line:

```
DO PRINT $RND&3+8,$8,"*"; UNTIL 0
```

You will need to use the SHIFT key to get some of the special symbols Again, you will have to type ESC to stop this program. If you want t let BASIC greet people, try:

```
DO INPUT'"Who are you "$TOP;PRINT''"Hello "$TOP'';UNTIL 0
```

Or you may want to impress by generating incomprehensible numbers:

```
@=20;DO PRINT RND;UNTIL 0
```

You may question the usefulness of these examples, but they illustrate the wide range of different tasks that BASIC is capable o These 'programs' all fitted on two lines of the display; to see wh you will be able to do with a longer program take a look at the ma examples later on in this manual.

1.3 The Keyboard

BASIC makes special use of some of the keys on the keyboard, which are very similar to the way the operating system uses them. There are, however, some extensions, so they are all listed here.

1.3.1 RETURN

The RETURN key is a signal to the computer that you have finished typing in a line of characters. The cursor will move to the start of the next line, and the computer may respond to what you have typed by typing out a reply.

1.3.2 Control - CTRL X

Typing X while holding down the CTRL key will cancel the whole of the current line and put the cursor at the start of the next new line.

1.3.3 ESC

The ESC key will stop the interpreter, and return it to the command mode where it will type a '>' again. ESC should get you out of all simple looping programs, but there are some situations when the BREAK key must be pressed to reset the computer. If you need to press BREAK, you can immediately re-enter the interpreter at the 'warm start' of #C2C2. If you enter at the 'cold start', you can recover your text by typing:

```
?#3001=0;END
```

1.3.4 General Characters

BASIC keywords (those given above) must be specified in upper case. Any other characters may be typed in either case, and in this manual will usually be given in lower case.

1.4 Storing Text

Any line typed after BASIC's '>' prompt which starts with a number is not executed, but stored as text in memory. Any type of input can be stored in this way; it could be the text of a document, a program, or data for a program. This section shows how to enter a piece of text, which can then be stored, edited, or output to a printer. The same method is used for entering a program.

The line must start with a line number, which can be any number within the range 1 to 32767, and there is no need to use consecutive line numbers for consecutive lines; indeed, it is wise to choose line numbers spaced by about 10 as you will soon realise. After the line number you should type the line of text. For example, enter the following:

```
10In Xanadu did Kubla Khan
20A stately pleasure-dome decree,
30Where Alph, the sacred river, ran
40Down to a sunless sea.
```

Remember to type RETURN at the end of each line. Each line can consist of up to 64 characters; if you try to type more than 64 characters BASIC will refuse to proceed until you have deleted some characters.

The reason for spacing line numbers somewhat apart is that it is then a simple matter to insert new lines between existing lines. For example, to insert a line before line 40, type:

```
36Through caverns measureless to man
```

The computer will sort the lines into the right order, according t
their line numbers, irrespective of the order in which you entere
them.

1.5 Commands

Commands typed in after the '>' prompt, without a preceding li
number, and followed by RETURN, are executed immediately by BAS
rather than being stored in its memory. For example, now type t
command:

```
LIST
```

This will cause the stored text to be typed out:

```
10 In Xanadu did Kubla Khan
20 A stately pleasure-dome decree,
30 Where Alph, the sacred river, ran
36 Through caverns measureless to man
40 Down to a sunless sea.
```

Note that BASIC has printed out a space after each line number, ev
though you did not type one in. This is to make the output look nice
There are several options available with the LIST command:

```
LIST 10       will list line 10 only
LIST 20,40    will list lines 20 to 40 inclusive
LIST 20,      will list line 20 onwards
LIST ,30      will list up to line 30.
```

A listing can be stopped by typing ESC (escape).

1.6 Editing

One powerful feature of BASIC's text and program storage is th
stored lines can be modified very simply by typing the same li
number followed by the new version. For example, to change line 20
the text just type:

```
20New line two
```

and try listing the program again to see the effect.
To delete a line simply type the line number followed by RETURN

1.6.1 Screen Editing

To change a line with just a minor error in it, it is covenient to
the screen editor function provided by the operating system. There
five control keys which do the following jobs:

```
CTRL-A   move cursor left
CTRL-S   move cursor right
CTRL-Z   move cursor down
CTRL-W   move cursor up
CTRL-Q   read character at cursor.
```

For example, suppose we wanted to edit a piece of stored text. Fi
the text is listed as shown:

```
>LIST
   10 PIECE OF TEXT MATERIAL
>_
```

After listing the program the cursor is positioned after the prompt, as shown. First move the cursor vertically upwards, using the CTRL-W function, until it is opposite the line we wish to edit:

```
>LIST
_ 10 PIECE OF TEXT MATERIAL
>
```

Now use the CTRL-Q function to read the line number:

```
>LIST
  10_PIECE OF TEXT MATERIAL
>
```

Now skip over the space which BASIC printed out with a CTRL-S and use CTRL-Q to read the characters which we want to keep:

```
>LIST
  10 PIECE OF_TEXT MATERIAL
>
```

Now type in the correction to the line:

```
>LIST
  10 PIECE OF CAKE_MATERIAL
>
```

As no more of the old line is required the return key is pressed, and the program may be listed again to verify that the editing gave the correct result.

The CTRL-S function may be used to omit parts of the old line that are no longer required, and CTRL-A may be used to backspace the cursor in order to make room for inserting extra characters in the line.

If you change your mind while editing a line, type CTRL-X (cancel) and the old line will be unchanged.

1.7 Other Commands

Some other useful commands are described here:

NEW will clear the stored text so that a new piece of text can be typed in. It should always be typed before entering a new piece of text.

?#3001=0;END can be typed after typing NEW or suffering a reset to retrieve the text previously in memory. Note that you should only do this if there is already text in memory.

1.8 Errors

By now you will probably have been greeted by the message:

Error X

where X is the error code number. There are two possible reasons for errors:

1. You typed something, probably a command, that BASIC was not expecting or could not interpret

2. BASIC was commanded to do something that it could not do.

For example, typing 'ABC' followed by a RETURN will give the error message:

Error 94

which is probably the most common error; it means that 'ABC' was not a legal command.

Remember that it is impossible to cause physical damage to your Acorn computer, whatever you type at the keyboard. The worst you can do is to lose the stored text, and even that is extremely unlikely. Most errors are really warnings, and a complete explanation of all the error codes is given in Chapter 18.

1.9 Saving Text or Programs

Having entered some stored text into the computer's memory, this section will show how to save this text, and load it back at a later time.

Text and programs are saved as memory images using the operating system's memory to file transfer. The BASIC interpreter provides all the addresses necessary for the operating system; all you have to do is provide the file's name. File names can be anything containing up to 16 characters for the current Cassette Operating System and up to 7 characters for the current Disk Operating System. Guaranteed suitable names are "FRED", "22/4/80", etc.

1.9.1 SAVE

First check that the stored text is still there by typing LIST. To save the stored text to tape, type:

```
SAVE "EXAMPLE"
```

where "EXAMPLE" is the file name chosen for illustration. Type RETURN, and the operating system will manage the rest of the file transfer in its normal fashion. When the '>' prompt reappears, you are back in the BASIC interpreter, and can proceed with your task. A word of warning: if you are using the DOS, then no wait until completion has been performed by the interpreter in order for it to get control back to you as quickly as possible, so you should not destroy the text until the disk transfer is over.

1.9.2 Operating System - *

The * command allows you to call any of the normal operating system commands directly from BASIC, or even a BASIC program.

```
*CAT
```

will give you a catalogue in the normal way.

1.9.3 LOAD

The LOAD statement is complementary to the SAVE statement, it will reload the text into BASIC's current text space. To load EXAMPLE type:

```
LOAD "EXAMPLE"
```

and then typing:

```
LIST
```

will give a listing of the text that was previously saved. Note that any BASIC text is position independant and may be loaded back into any text area.

1.9.4 Errors when Using the OS

If an error is found when using the operating system, the operating system will print its own description of the fault, and then cause an

error, in order to return to whatever program had called it in a fashion indicating an error. With BASIC, this will cause the interpreter to print an Error message, with a code that has been provided by the operating system. These error codes will be consistent for any one particular operating system, but are not consistent over the range of Acorn operating systems.

2.0 CALCULATING IN BASIC

BASIC was invented in 1964 at Dartmouth College, New Hampshire, and it stands for Beginner's All-purpose Symbolic Instruction Code. This chapter introduces some of the facilities available in the BASIC language.

The BASIC language consists of 'statements', 'operators' and 'functions'. The 'statements' are words like PRINT and INPUT which tell the computer what you want to do; they are followed by the things you want the computer to operate on.

The 'operators' are special symbols such as the mathematical signs '+' and '-' meaning 'add' and 'subtract'.

The 'functions' are words like the statements, but they have a numerical value; for example, RND is a function which has a random value.

2.1 PRINT

This is by far the most useful BASIC statement; it enables programs to print out the results of their calculations.

Try typing:

```
PRINT 7+3
```

BASIC will print:

```
10>
```

The '>' prompt reappears immediately after the answer, 10, is printed out. This is the best way to use BASIC as a simple calculator; type PRINT followed by the expression you want to evaluate.

Try the effect of the following:

```
PRINT 7-3
PRINT 7*3
PRINT 7/3
```

You will discover that '*' means multiply; it is the standard multiply symbol on all computers. Also '/' means divide, but you may be surprised that the answer to 7/3 is given as 2, not 2 and 1/2. Acorn BASIC only deals in whole numbers, or integers, so the remainder after the division is lost. The remainder can be obtained by typing:

```
PRINT 7%3
```

The '%' operator means 'give remainder of division'.

More complex expressions are evaluated according to the standard rules of mathematics, so the expression:

```
PRINT 2+3*4-5
```

has the result 9. Multiplications and divisions are performed first, followed by additions and subtractions. Round brackets can be used to make sure that operations are performed in the correct order; anything enclosed in brackets is evaluated first. Thus the above expression could also be written:

```
PRINT (2+(3*4))-5
```

There is no limit to the complexity of expressions that BASIC can evaluate, provided that they can be typed into the input line buffer. You will notice that BASIC calculates extremely rapidly. Try typing:

```
PRINT 9*9*9*9*9*9*9*9
```

Acorn BASIC can calculate with numbers between about 2000 million and -2000 million, which gives an accuracy of between nine and ten digits. Furthermore, because whole numbers are used, all calculations in this range are exact.

2.1.1 Printing Several Things

You can print the results of several calculations in one PRINT statement by separating them with commas:

```
PRINT 7, 7*7, 7*7*7, 7*7*7*7
```

which will print out:

```
    7   49  343 2401
```

Note that each number is printed out on the right-hand side of a column five characters wide. This ensures that when large numbers of results are printed out they will be in neat columns on the screen.

2.1.2 Printing Strings

PRINT can also be used to print out words, or, indeed, any required group of characters. Arbitrary groups of characters are referred to simply as 'strings', and to identify the characters as a string they are enclosed in double quotes. For example:

```
PRINT "The result"
```

will cause:

```
The result>
```

to be printed out. The characters in quotes are copied faithfully, exactly as they appear in the PRINT statement. Thus you could type:

```
PRINT "55*66=", 55*66
```

where the expression inside quotes is a string just like any other. This would print out:

```
55*66=  3630>
```

2.2 Variables - A to Z

You will probably be familiar with the use of letters, such as X and N, to denote unknown quantities. E.g.: "the nth degree", "X marks the spot", etc. In Acorn BASIC any letter of the alphabet, A to Z, may be used to denote an unknown quantity, and these are called 'variables'. The equals sign '=' is used to assign a particular value to a variable. For example, typing:

```
X=6
```

will assign the value 6 to X. Now try:

```
PRINT X
```

and, as expected, the value of X will be printed out. Note the difference between this and:

```
PRINT "X"
```

The assignment statement 'X=6' should be read as 'X becomes 6' because it denotes an operation which changes the value of X, rather than a

statement of fact about X. The following statement:

```
X=X+1
```

is perfectly reasonable, and adds 1 to the previous value of X. In words, the new value of X is to become the old value of X plus one.

Now that we can use variables to stand for numbers, they can also be used in expressions. For example, to print the first four powers of 12 we can type:

```
T=12
PRINT T, T*T, T*T*T, T*T*T*T
```

2.3 Getting the Right Answer

Suppose you wanted to calculate half of 777. You might type:

```
PRINT 777/2
```

and you would get the answer 388. Then, to get the remainder, you would type:

```
PRINT 777%2
```

and the answer will be 1. So the exact answer is 388 and one half.

Suppose, however, you decided to try:

```
PRINT 1/2*777
```

thinking it would give 'a half times 777', you would be surprised to get the answer 0. The reason lies in the fact that the calculation is worked out from left to right in several stages, and at every stage only the whole-number part of the result is kept. First 1/2 is calculated, and the result is 0 because the remainder is not saved. Then this is multiplied by 777 to give 0!

Fortunately there is a simple rule to avoid problems like this:

Do Divisions Last!

The division operation is the only one that can cause a loss of accuracy; all the other operations are exact. By doing divisions last the loss of accuracy is minimised.

Applying this rule to the previous example, the division by two should be done last:

```
PRINT (1*777)/2
```

which is obviously the same as what was written earlier.

2.3.1 Fixed-Point Calculations

An alternative way to find half of 777 is to imagine the decimal point moved one place to the right, and write:

```
PRINT 7770/2
```

The result will then be 3885, or, with the decimal point moved back to the correct place, 388.5. For example, in an accounting program you would use numbers to represent pence, rather than pounds. You could then work with sums of up to 20 million pounds. Results could be printed out as follows:

```
PRINT R/100, " POUNDS", R%100, " PENCE"
```

2.4 Print Field Size - '@'

Numbers are normally printed out right-justified in a field of five character spaces. If the number needs more than five spaces the field size will be exceeded, and the number will be printed in full without any extra spaces. Note that the minus sign is included in the field

size for negative numbers.

It is sometimes convenient to alter the size of the print field. The variable '@' determines this size, and can be altered for other field widths. For example:

```
@=20
```

will print all numbers in the right of a field of 20 spaces. This value will ensure that all numbers are printed inside the field, since the maximum number of characters possible is 11.

The value of '@' can be zero, in which case no extra spaces will be inserted before the numbers.

2.5 Printing a New Line

A single quote in a PRINT statement will cause a return to the start of the next line. Thus:

```
PRINT "A" ' "c" ' "o" ' "r" ' "n" '
```

will print out:

```
A
c
o
r
n
>
```

This is an improvement over most other versions of BASIC, which would require five separate PRINT statements for this example.

2.6 Multiple-Statement Lines - ';'

Acorn BASIC allows any number of statements to be strung together on each line provided they are separated by semicolons. For example the following line:

```
A=1;B=2;C=3;PRINT A,B,C'
```

will print:

```
 1    2    3
```

2.7 Hexadecimal Numbers

Numbers can also be represented in a notation called 'hexadecimal' which is especially useful for representing addresses in the computer. Hexadecimal notation is explained in section 11.2; all that needs to be mentioned here is that hexadecimal notation is just an alternative way of writing numbers which makes use of the digits 0 to 9 and the letters A to F. The '#' symbol, called 'hash', is used to introduce a hexadecimal number. Thus #E9 is a perfectly good hexadecimal number (nothing to do with the variable E).

```
PRINT #8000
```

will print:

```
32768>
```

The PRINT statement prints the number out in decimal. A number can be printed in hexadecimal by prefixing it with an '&' (ampersand) in the PRINT statement. Thus:

```
PRINT &32768
```

will print:

```
8000>
```

2.8 Logical Operations

In addition to the arithmetic operations already described, Acorn BASIC provides three operations called 'logical operations': '&' (AND), '|' (OR), and ':' (Exclusive-OR). These are all operations between two numbers, so there is no danger of confusing this use of '&' with its use to specify printing in hex as covered in the previous section. These are especially useful when controlling external devices from a BASIC program.

The following table gives the results of these three operations for the numbers 0 and 1:

Operands A	Operands B	A & B	A | B	A : B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Try typing the following:

```
PRINT 0 & 1
PRINT 1 | 1
PRINT 1 : 1
```

and verify that the results agree with the table.

2.9 Peeking and Poking

Many BASICs have PEEK and POKE functions which do the following:

PEEK looks at the contents of a place in memory, or memory location.

POKE changes the contents of a memory location.

The '?' operator, called 'query', is used for poking and peeking in Acorn BASIC and it provides a more elegant mechanism than the two functions provided in other BASICs.

The contents of some memory location whose address is A is given by typing:

```
PRINT ?A
```

For example, to look at the contents of location #C000 type:

```
PRINT ?#C000
```

and the result will be 60 (this is the first location in the BASIC interpreter).

To change the contents of a location whose address is A to 13 just type:

```
?A=13
```

For example, try changing location 0 to 30

```
?0=30
```

and check it

```
PRINT ?0
   30
```

3.0 WRITING A PROGRAM

The first step in writing a program, whatever language it will eventually be programmed in, is to express your problem in terms of simple steps that the computer can understand.

An Acorn Computer could be put to an immense number of different uses; anything from solving mathematical problems, controlling external equipment, playing games, accounting and book-keeping, waveform processing, document preparation ... etc. The list is endless. Obviously all these applications cannot be included in a computer's repertoire of operations. Instead what is provided is a versatile set of more fundamental operations and functions which, in combination, can be used to solve such problems.

It is therefore up to you to become familiar with the fundamental operations that are provided, and learn how to solve problems by combining these operations into programs.

Programming is rather like trying to explain to a novice cook, who understands little more than the meanings of the operations 'stir', 'boil', etc., how to bake a cake. The recipe corresponds to the program; it consists of a list of simple operations such as 'stir', 'bake', with certain objects such as 'flour', 'eggs':

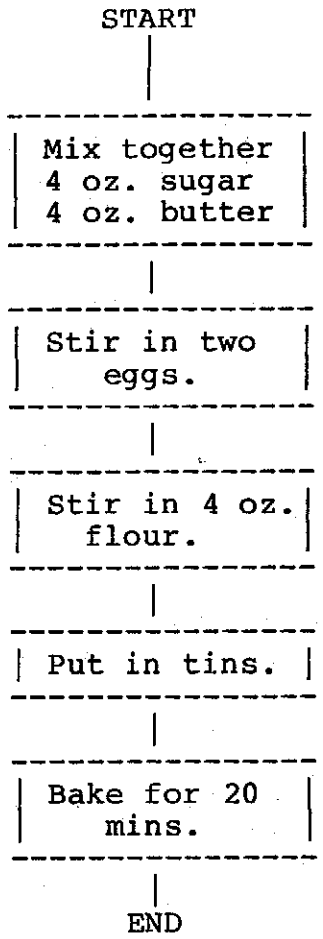
```
   Recipe 1. Sponge Cake
1. Mix together 4 oz. sugar and 4 oz. butter.
2. Stir in 2 eggs.
3. Stir in 4 oz. flour.
4. Put into tins.
5. Bake for 20 mins. at Mark 4.
6. Remove from oven and eat.
7. END
```

The recipe is written so that, provided all the ingredients are already to hand, the cook can follow each command in turn without having to look ahead and worry about what is to come.

Similarly, a computer only executes one operation at a time, and cannot look ahead at what is to come.

3.1 Flowcharts

Before writing a program it is a good idea to draw a 'flowchart' indicating the operations required, and the order in which they should be performed. The generally accepted standard is for operations to be drawn inside rectangular boxes, with lines linking these boxes to show the flow of control. A simple flowchart for the program to bake a cake might be drawn as follows:

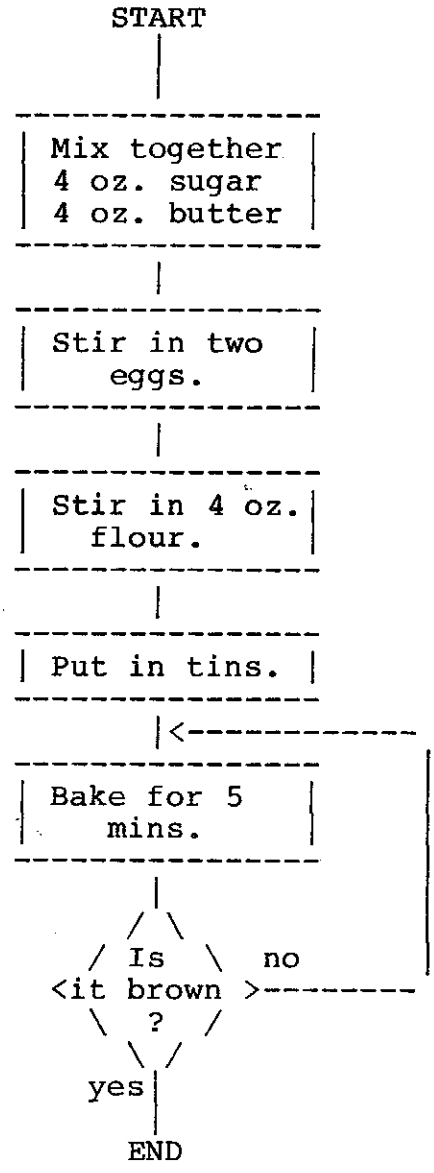


3.2 Decisions

Many recipes do not just contain a sequence of steps to be performed, but contain conditions under which several different courses of action should be taken. For example, for a perfect cake line 5 would be better written:

5. Bake until golden brown.

It would then be necessary to open the oven door every five minutes and make a decision about the colour of the cake. Decisions are represented in flowcharts by diamond-shaped boxes, with multiple exits labelled with the possible outcomes of the decision. The new flowchart would then be:



The action 'bake for 5 mins.' is repeatedly performed until the test 'is it brown?' gives the answer 'yes'. Of course the program would go dramatically wrong if the oven were not switched on; the program would remain trapped in a loop.

With these two simple concepts, the action and the decision, almost anything can be flowcharted. Part of the trick in flowcharting programs is to decide how much detail to put into the flowchart. For example, in the cake program it would be possible to add the test 'is butter and sugar mixed?' and if not, loop back to the operation 'mix butter and sugar'. Usually flowcharts should be kept as short as possible so that the logic of the program is not obscured by a lot of unnecessary fine detail.

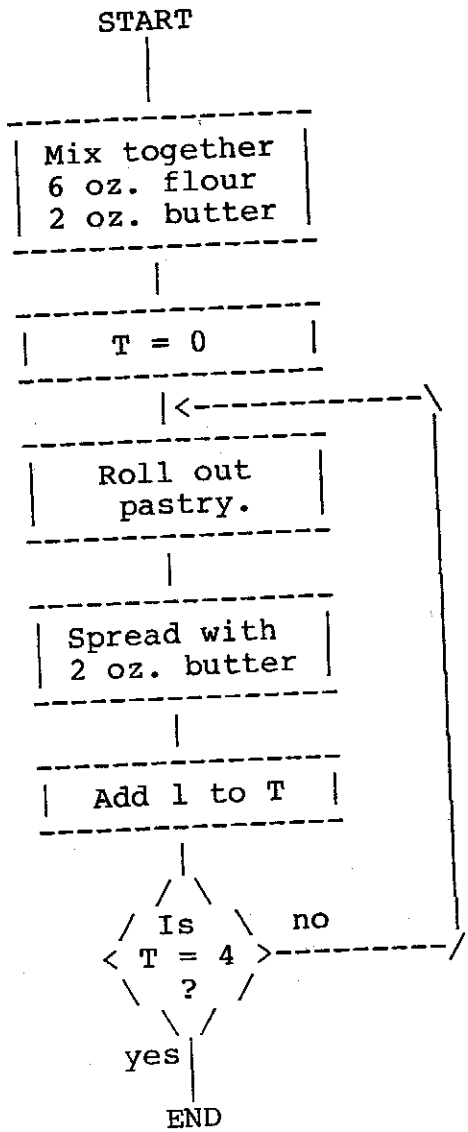
3.3 Counting

Recipes sometimes require a particular series of operations to be performed a fixed number of times. The following recipe for puff pastry illustrates this:

Recipe 2. Puff Pastry
1. Mix 6 oz. flour with 2 oz. butter.
2. Roll out pastry.
3. Spread with 2 oz. butter.

4. Fold in half.
5. Repeat steps 2 to 4 a further 3 times.
6. END

In this recipe the cook has to perform operations a total of 4 times. A cook would probably keep a mental note of how many times he has performed these operations; for the sake of the flowchart it is convenient to give the number of operations a label, such as T. The flowchart of the puff pastry recipe would then be:



The loop consisting of statements 2 to 4 is performed 4 times; t
test at the end gives the answer 'no' for T=1, 2, and 3, and t
answer 'yes' for T=4.

To perform an operation several times in a program an identic
method can be used; a counter, such as T, is used to count the numl
of operations and the counter is tested each time to determine whet
enough operations have been completed.

3.4 Subroutines

A recipe may include a reference to another recipe. For example
typical recipe for apple tart might be as follows:

Recipe 3. Apple Tart

1. Peel and core 6 cooking apples.
2. Make pastry as in recipe 2.
3. Line tart tin with pastry.
4. Put in apple.
5. Bake for 40 mins. mark 4.
6. END

To perform step 2 it is necessary to insert a marker in the book at the place of the original recipe, find the new recipe and follow it, and then return to the original recipe and carry on at the next statement.

In computer programming a reference to a separate routine is termed a 'subroutine call'. The main recipe, for apple tart, is the main routine; one of its statements calls the recipe for puff pastry, the subroutine. Note that the subroutine could be referred to many times throughout the recipe book; in the recipe for steak and kidney pie, for example. One reason for giving it separately is to save space; otherwise it would have to be reproduced for every recipe that needed it.

Note that in order not to lose his place, the cook needed a marker to insert in the recipe book so that he should know where to return to at the end of the subroutine. When using a program the computer keeps a record of where you were when you call a subroutine, and returns you there automatically at the end of the subroutine. In other respects, the process of executing a subroutine on a computer is just like this analogy.

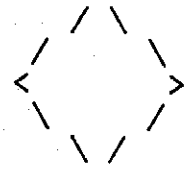
3.5 Planning a Program

Before writing a program it is a good idea to express the problem in one of the forms used in this chapter:

1. As a list of numbered steps describing, in words, exactly what to do at each step

2.As a flowchart using the following symbols:

for actions



for decisions

START start of program

END end of program

Having done this, the job of writing the program will be made easier, and the task of making it work reliably will be reduced.

4.0 WRITING A BASIC PROGRAM

Commands and statements typed after BASIC's prompt are executed immediately, as we have seen in Chapter 2. However if you start the line with a number, the line is not executed but stored as text in the computer's memory.

4.1 RUN

First type 'NEW' to clear the text area. Then try typing in the following:

```
10PRINT "A PROGRAM!"'
20END
```

When these lines have been typed in you can list the text by typing LIST. Now type:

```
RUN
```

The stored text will be executed, one statement at a time, starting with the lowest-numbered statement, and the message 'A PROGRAM!' will be printed out. The text you entered formed a 'program', and the program was executed, statement by statement, when you typed RUN. The END statement is used to stop execution of the program; if it is ommitted an error message will be given.

4.2 INPUT

Type NEW again, and then enter the following program:

```
10 INPUT N
20 N=N+1
30 PRINT N
40 END
```

This time we have printed the program as it would appear when listed, so when you type it in, you need not type in the space after the line number (although it would not matter if you did in this case). The INPUT statement enables you to supply numbers to a running program. When it is executed it will print a question mark and wait for a number to be typed in. The variable specified in the INPUT statement will then be set to the value typed in. To illustrate, type:

```
RUN
```

The program will add 1 to the number you type in; try running it again and try different numbers.

The INPUT statement may contain more than one variable; a question mark will be printed for each one, and the values typed in will be assigned to the variables in turn.

The INPUT statement may also contain strings; these will be printed out before each question mark. The following program illustrates this; it converts Fahrenheit to Celsius (Centigrade), giving the answer to the nearest degree:

```
10 INPUT "Fahrenheit" F
```

```
20 PRINT (10*F-315)/18 " Celsius" '
30 END
```

The value, in Fahrenheit, is stored in the variable F. The expression in the PRINT statement converts this to Celsius.

4.3 Comments - REM

The REM statement means 'remark'; everything on the line following the REM statement will be ignored when the program is being executed, so it can be used to insert comments into a program. For example:

```
5 REM Program for temperature conversion
```

4.4 Functions

Functions are operations that return a value. Functions are like statements in that they have names, consisting of a sequence of letters, but unlike statements they return a value and so can appear within expressions.

4.4.1 RND

The RND function returns a random number with a value anywhere between the most negative and most positive numbers that can be represented in BASIC. To obtain smaller random numbers the '%' remainder operator can be used; for example:

```
PRINT RND%4
```

will print a number between -3 and +3.

4.4.2 TOP

TOP returns the address of the first free memory location after the BASIC program.

```
PRINT &TOP
```

will print TOP in hexadecimal. This will be #3002 if you have just entered the interpreter.

```
PRINT TOP-#3000
```

is a useful way of finding out how many bytes are used up by a program.

4.4.3 ABS

The ABS function can be used to give the absolute or positive value of a number; the number is written in brackets after the function name. For example:

```
PRINT ABS(-57)
```

will print 57. One use of ABS is in generating positive random numbers. For example:

```
PRINT ABS(RND)%6
```

gives a random number between 0 and 5.

4.5 Escape - ESC

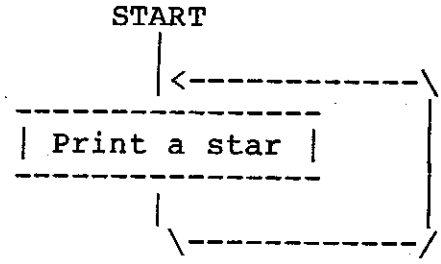
It is possible to create programs which will never stop; see the following example in section 4.6. The escape key 'ESC' at the top left of the keyboard will stop any BASIC program and return control to the '>' prompt.

4.6 GOTO

In the above programs the statements were simply executed in ascending order of their line numbers. However it is sometimes necessary to transfer control forwards or backwards to somewhere other than the next numbered statement. The GOTO (go to) statement is used for this purpose; the GOTO statement specifies the statement to be executed next. For example, type:

```
 1 REM Stars
10 PRINT "*"
20 GOTO 10
```

A flowchart for this program makes it clear that the program will never stop printing stars:



To stop the program you will have to type ESC (escape).

4.6.1 Labels - a to z

Acorn BASIC offers another option for the GOTO statement. Instead of giving the number of the statement to be executed next, a statement can be designated by a 'label', and the GOTO is followed by the required label. A label can be one of the lower-case letters a to z.

To illustrate the use of labels, rewrite the 'STARS' program as follows, using the label 's':

```
10 s PRINT "*"
20 GOTO s
```

Note that there must be no spaces between the line number 10 and the label s when you type it in (and therefore only one space when it is listed).

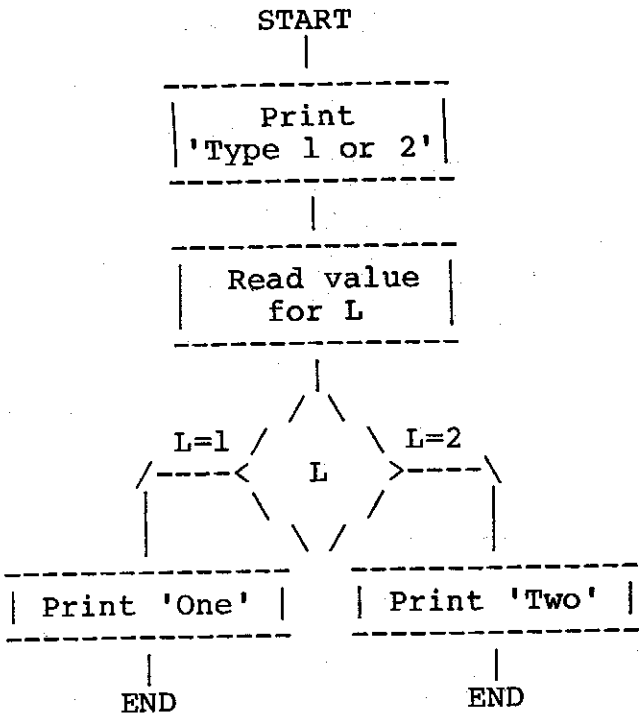
There are two advantages to using labels, rather than line numbers, in GOTO statements. First, programs are clearer, and do not depend on how the program lines are numbered. Secondly, the GOTO statement is faster using a label than using a line number because the Acorn BASIC interpreter remembers the positions of all 26 labels, whereas line numbers are found by searching through the text sequentially.

4.6.2 Switches

The GOTO statement may be followed by any expression which evaluates to a valid line number; for example:

```
10 REM Two-Way Switch
20 INPUT "Type 1 or 2" L
30 GOTO (40*L)
40 PRINT "One"
50 END
80 PRINT "Two"
90 END
```

If L is 1 the expression (40*L) will be equal to 40, and the program will print 'One'. If L is 2 the expression will be equal to 80 and the program will print 'Two'. The flowchart for this program is as follows:



4.6.3 Multi-Way Switches

Finally here is an example of a multi-way switch using GOTO. The program calculates a random number between 0 and 5 and then goes to a line number between 30 and 35. Each of these lines consists of a PRINT statement which prints the face of a dice. The single quote in the print statement gives a 'return' to the start of the next line

```
10 REM Dice Tossing
20 GOTO (30+ABS(RND)%6)
30 PRINT'" *"'; END
31 PRINT"  *"'"*"; END
32 PRINT"  *"'" *"'"*"; END
33 PRINT"* *"'"* *"; END
34 PRINT"* *"'" *"'"* *"; END
35 PRINT"* *"'"* *"'"* *"; END
```

Description of Program:

20	Choose random number between 30 and 35
30-35	Print corresponding face of a dice

Sample runs:

```
>RUN
  *
 *
*
>RUN
* *
 *
* *
```

```
>RUN
* *
* *
* *
```

4.7 Conditions - IF ... THEN

One of the most useful facilities in BASIC is the ability to execute a statement only under certain specified conditions. To do this the IF...THEN statement is used; for example:

```
IF A=0 THEN PRINT "Zero"
```

will execute the PRINT statement, and print "Zero", only if the condition A=0 is true; otherwise everything after THEN will be skipped and execution will continue with the next line.

4.7.1 Relational Operators

The part of the IF ... THEN statement after the IF is the 'condition' which can be any two expressions separated by a 'relational operator' which compares the two expressions. Six different relational operators can be used:

=	equal	<>	not equal
>	greater than	<=	less than or equal
<	less than	>=	greater than or equal

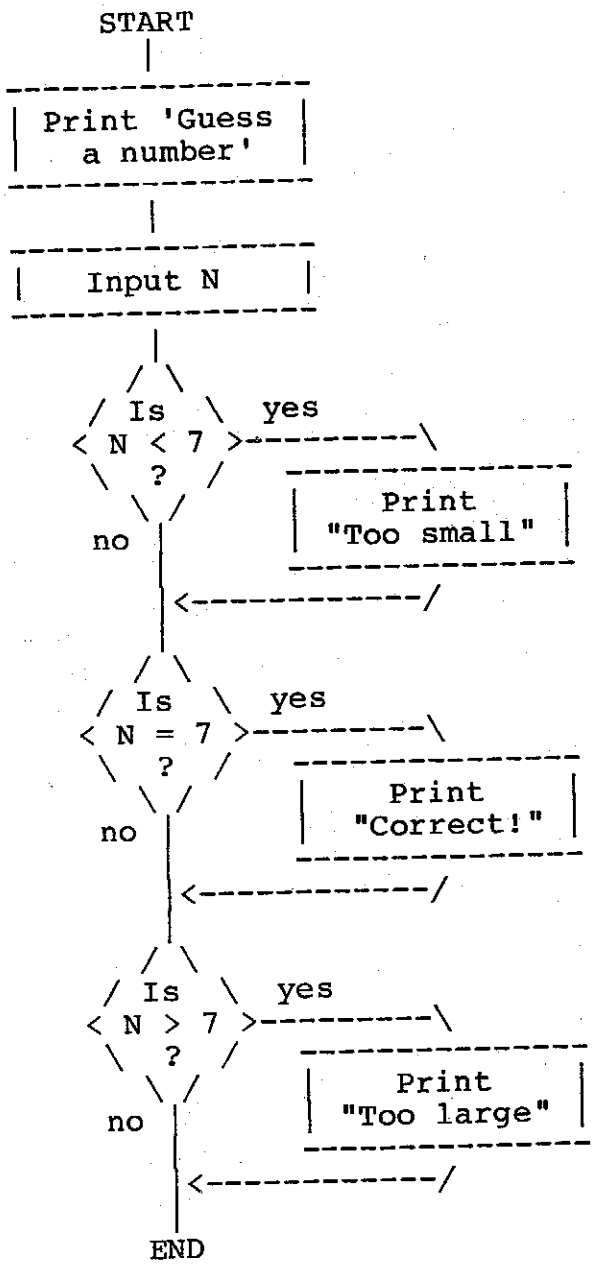
where each operator on the left is the opposite of the operator on the right.

The expressions on each side of the relational operators can be as complicated as required, and the order is unimportant. There is no need to put brackets around the expressions.

For example, the following program prints one of three messages depending on whether a number typed in is less than 7, equal to 7, or greater than 7:

```
10 REM Guess a number
20 INPUT"Guess a number " N
30 IF N<7 THEN PRINT "Too small"
40 IF N=7 THEN PRINT "Correct!"
50 IF N>7 THEN PRINT "Too large"
60 END
```

A flowchart for this program is as follows:



4.7.2 THEN Statement

The statement after THEN can be any statement, even an assignment statement as in:

```
IF A=7 THEN A=6
```

Note that the meaning of each '=' sign is different. The first 'A=7' is a condition which can be either true or false; the second 'A=6' is an assignment statement which instructs the computer to set the variable A to the value 6. To make this distinction clear the above statement should be read as: 'If A is equal to 7 then A becomes 6'. The word THEN can generally be left out, unless the following statement starts with a T, when THEN must be present to stop the interpreter objecting to a misspelt THEN.

4.7.3 Conjunctions - AND and OR

Conditions can be strung together using the conjunctions AND and OR, so, for example:

```
10 INPUT A,B
20 IF A=2 AND B=2 THEN PRINT "Both"
30 GOTO 10
```

will only print "Both" if both A and B are given the value 2. Alternatively:

```
10 INPUT A,B
20 IF A=2 OR B=2 THEN PRINT "Either"
30 GOTO 10
```

will only print "Either" if at least one of A and B is equal to 2.

4.8 Logical Variables

An alternative form for the condition in an IF ... THEN statement is to specify a variable whose value denotes either 'true' or 'false'. The values 'true' and 'false' are represented by the least significant byte being non-zero and zero respectively, so:

```
A=1; B=0
```

sets A to 'true' and B 'false'. Logical variables can be used in place of conditions in the IF statement; for example:

```
IF A THEN PRINT "True"
```

will print "True".

4.9 Iteration

One way of printing the powers of 2 would be to write:

```
10 REM Powers of Two
20 P=1; T=2; @=0
30 PRINT "2 ** ", P, " = ", T '
40 T=T*2; P=P+1
50 GOTO 30
```

which will print out:

```
2 ** 1 = 2
2 ** 2 = 4
2 ** 3 = 8
2 ** 4 = 16
2 ** 5 = 32
2 ** 6 = 64
```

and so on without stopping. This is a bit inelegant; suppose we wished to print out just the first 12 powers of 2. It is simply a matter of detecting when the 12th power has just been printed out, and stopping then. This can be done with the IF statement as follows:

```
10 REM First Twelve Powers of Two
20 P=1; T=2; @=0
30 PRINT "2 ** ", P, " = ", T '
40 T=T*2; P=P+1
50 IF P<=12 GOTO 30
60 END
```

The IF statement is followed by a GOTO statement; if P is less than 12 the condition will be true, and the program will go back to line 30. After the twelth power of 2 has been printed out P will have the value

13, which is not less than or equal to 12, and so the program will stop.

With the IF statement we have the ability to make the computer do vast amounts of work as a result of very little effort on our part. For example we can print 256 exclamation marks simply by running the following program:

```
10 I=0
20 PRINT"!"; I=I+1
30 IFI<256 GOTO 20
40 END
```

5.0 LOOPS

The previous section showed how the IF statement could be used to cause the same statements to be executed several times. Recall the program:

```
10 I=0
20 PRINT"!"; I=I+1
30 IF I<256 GOTO 20
40 END
```

which prints out 256 exclamation marks. This iterative loop is such a frequently-used operation in BASIC that all BASICs provide a special pair of statements for this purpose, and Acorn BASIC provides a second type of loop for greater flexibility.

5.1 FOR ... NEXT Loops

The FOR statement, together with the NEXT statement, causes a set of statements to be executed for a range of values of a specified variable. To illustrate, the above example can be rewritten using a FOR ... NEXT loop as follows:

```
10 FOR I=1 TO 256
20   PRINT "!"
30 NEXT I
40 END
```

The FOR statement specifies that the statements up to the matching NEXT statement should be executed for each value of I from 1 to 256 (inclusive). In this example there is one statement between the FOR and NEXT statements, namely:

```
PRINT "!"
```

This statement has been indented in the program to make the loop structure clearer; in fact the spaces are ignored by BASIC.

The NEXT statement specifies the variable that was specified in the corresponding FOR statement. This variable, I in the above example, is called the 'control variable' of the loop; it can be any of the variables A to Z.

The value of the control variable can be used inside the loop, if required. To illustrate, the following program prints out all multiples of 12 up to 12*12:

```
10 FOR M=1 TO 12
20   PRINT M*12
30 NEXT M
40 END
```

The range of values specified in the FOR statement can be anything you wish, even arbitrary expressions. Remember, though, that the loop is always executed at least once, so the program:

```
10 FOR N=1 TO 0
20   PRINT N
30 NEXT N
40 END
```

will print '1' before stopping.

5.1.1 STEP Size.

It is also possible to specify a STEP size in the FOR statement; the STEP size will be added to the control variable each time round the loop, until the control variable exceeds the value specified after TO. If the STEP size is omitted it is assumed to be 1. This provides us with an alternative way of printing the multiples of 12:

```
10 FOR M=12 TO 12*12 STEP 12
20   PRINT M
30 NEXT M
40 END
```

5.2 DO ... UNTIL Loops

Acorn BASIC provides an alternative pair of loop-control statements: DO and UNTIL. The UNTIL statement is followed by a condition, and everything between the DO statement and the UNTIL statement is repeatedly executed until the condition becomes true. So, to print 256 exclamation marks in yet another way write:

```
10 I=0
20 DO
30   I=I+1
40   PRINT "!"
50 UNTIL I=256
60 END
```

Again, the statements inside the DO ... UNTIL loop may be indented to make the structure clearer.

The DO ... UNTIL loop is most useful in cases where a program is to carry on until certain conditions are satisfied before it will stop. To illustrate, the following program prompts for a series of numbers, and adds them together. When a zero is entered the program terminates and prints out the sum:

```
10 S=0
20 DO INPUT J
30    S=S+J
40 UNTIL J=0
50 PRINT "SUM =", S '
60 END
```

Note that a statement may follow the DO statement, as in this example.

5.2.1 Greatest Common Divisor

The following simple program uses a DO ... UNTIL loop in the calculation of the greatest common divisor (GCD) of two numbers; i.e. the largest number that will divide exactly into both of them. For example, the GCD of 26 and 65 is 13. If the numbers are co-prime the GCD will be 1:

```
 1 REM Greatest Common Divisor
80 INPUT A,B
```

```
 90 DO A=A%B
100 IF ABS(B)>ABS(A) THEN T=B; B=A; A=T
120 UNTIL B=0
130 PRINT "GCD =" A '
140 END
```

Description of Program:

```
80        Input two numbers
90        Set A to remainder when it is divided by B
100       Make A the larger of the two numbers
120       Stop when B is zero
130       A is the greatest common divisor.
```

Variables:

```
A,B - Numbers
T - Temporary variable.
```

The method is known as Euclid's algorithm, and to see it working insert a line:

```
 95 PRINT A,B'
```

The ABS functions ensure that the program will work for negative as well as positive numbers.

5.2.2 Successive Approximation

The DO ... UNTIL loop construction is especially useful for problems involving successive approximation, where the value of a function is calculated by obtaining better and better approximations until some criterion of accuracy is met.

The following iterative program calculates the square root of any number up to about 2,000,000,000. Also shown is the output obtained when calculating the square root of 200,000,000:

```
 10 REM Square Root
 20 INPUT S
100 Q=S/2
110 DO Q=(Q+S/Q)/2
120 UNTIL (Q-1)*(Q-1)<S AND (Q+1)*(Q+1)>S
130 PRINT Q'
140 END
```

Description of Program:

```
20        Input number
100       Choose starting value
110       Calculate next approximation
120       Carry on until the square lies between the squares of the
          numbers either side of the root.
130       Print square root.
```

Variables:

```
Q - Square root
S - Number.
```

Sample run:

```
>RUN
?200000000
14142
>
```

5.3 Nested Loops

FOR ... NEXT and DO ... UNTIL loops may be nested; the following example will print the squares, cubes, and fourth powers of the numbers 1 to 15 in a neat table:

```
 1 REM Powers of Numbers
 3 @=8
 5 PRINT "        X      X^2"
 8 PRINT "      X^3      X^4"
10 FOR N=1 TO 15
20   J=N
30   FOR M=1 TO 4
40     PRINT J; J=J*N
50   NEXT M
55   PRINT'
60 NEXT N
70 END
```

The statements numbered 20 to 50 are executed 15 times, for every value of N from 1 to 15. For each value of N the statements on line 40 are executed four times, for values of M from 1 to 4. Thus 15*4 or 60 numbers are printed out.

5.3.1 Mis-Nested Loops

Note that loops must be nested correctly. The following attempt at printing out 100 pairs of numbers will not work:

```
10 FOR A=1 TO 10
20 FOR B=1 TO 10
30 PRINT A,B
40 NEXT A
50 NEXT B
```

The program will, if RUN, give an error (Error 230). The reason for the error will become clear if you try to indent the statements within each loop, as in the previous example.

6.0 SUBROUTINES

As soon as a program becomes longer than a few lines it is probably more convenient to think of it as a sequence of steps, each step being written as a separate 'routine', an independent piece of program which can be tested in isolation, and which can be incorporated into other programs when the same function is needed.

6.1 GOSUB

Sections of program can be isolated from the rest of the program using a BASIC construction called the 'subroutine'. In the main program a statement such as:

```
GOSUB 1000
```

causes control to be transferred to the statement at line 1000. The statements from line 1000 comprise the subroutine. The subroutine is terminated by a statement:

```
RETURN
```

which causes a jump back to the main 'calling' program to the statement immediately following the GOSUB 1000. It is just as if the statements from 1000 up to the RETURN statement had simply been inserted in place of the GOSUB 1000 statement in the main program.

As an example, consider the following program:

```
 10 A=10
 20 GOSUB 100
 30 A=20
 40 GOSUB 100
 50 END

100 PRINT A '
110 RETURN
```

Lines 100 and 110 form a subroutine, separate from the rest of the program, and they are terminated by RETURN. The subroutine is called twice from the main program, in lines 20 and 40. The program, when RUN, will print:

```
 10
 20
>
```

6.1.1 Chequebook-Balancing Program

As a more serious example, consider a program for balancing a chequebook. The program will have three distinct stages; reading in the credits, reading in the debits, and printing the final amount. We can immediately write the main program as:

```
 10 REM Chequebook-Balancing Program
 20 PRINT "Enter Credits"'
 30 GOSUB 1000
```

```
40 PRINT "Enter debits"'
50 GOSUB 2000
60 PRINT "Total is "
70 GOSUB 3000
80 END
```

Now all we have to do is write the subroutines at lines 1000, 2000, and 3000!

The subroutines might be written as follows:

```
1000 REM Sum Credits in C
1010 REM Changes C,J
1020 C=0
1030 DO INPUT J; C=C+J
1040 UNTIL J=0
1050 RETURN

2000 REM Sum Debits in D
2010 REM Changes D,J
2020 D=0
2030 DO INPUT J; D=D+J
2040 UNTIL J=0
2050 RETURN

3000 REM Print Total in T
3010 REM Changes T; Uses C,D
3020 T=C-D; @=5
3030 PRINT T/100," Pounds",T%100," Pence"
3040 RETURN
```

Values are entered in pence, and entering zero will terminate the list of credits or debits.

The two subroutines at 1000 and 2000 are strikingly similar, and this suggests that it might be possible to dispense with one of them. Indeed, the main part of the chequebook-balancing program can be written as follows, eliminating subroutine 1000:

```
10 REM Chequebook-Balancing Program
20 PRINT "Enter Credits"
30 GOSUB 2000
40 C=D
50 PRINT "Enter debits"
60 GOSUB 2000
70 PRINT "Total is "
80 GOSUB 3000
90 END
```

In conclusion, subroutines have two important uses:

1. To divide programs into modules that can be written and tested separately, thereby making it easier to understand the operation of the program

2. To make it possible to use the same piece of program for a number of similar, related, functions.

As a rough guide, if a program is too long to fit onto the screen of the VDU it should be broken down into subroutines. Each subroutine should state clearly, in REM statements at the start of the subroutine, the purpose of the subroutine, which variables are used by the subroutine, and which variables are altered by the subroutine. A few moments spent documenting the operation of the subroutine in this

way will save hours spent at a later date trying to debug a program which uses the subroutine.

6.2 GOSUB Label

The GOSUB statement is just like the GOTO statement that has already been described, in that it can be followed by a line number, an expression evaluating to a line number, or a label. Labels are of the form a to z, and the first line of the subroutine should contain the label immediately following the line number.

6.2.1 Linear Interpolation

The following program uses linear interpolation to find the roots of an equation using only integer arithmetic, although the program could be modified to use floating-point statements.

The equation is specified in a subroutine, y, giving Y in terms of X; the program finds solutions for Y=0.

As given, the program finds the root of the equation:

$$x^2 - x - 1 = 0$$

The larger root of this equation is phi, the golden ratio. A scaling factor of S=1000 is included in the equation so that calculations can be performed to three decimal places.

The program prompts for two values of X which lie either side of the root required

```
  1 REM Linear Interpolation
  5 S=1000; @=0; I=1
 10 INPUT "X1",A,"X2",B
 20 A=A*S; B=B*S
 30 X=A; GOSUB y; C=Y
 40 X=B; GOSUB y; D=Y
 50 IF C*D<0 GOTO 80
 60 PRINT "Root not bracketed"
 70 END
 80 DO I=I+1
 90 X=B-(B-A)*D/(D-C); GOSUB y
100 IF C*Y<0 THEN A=X; C=Y; GOTO 120
110 B=X; D=Y
120 UNTIL ABS(A-B)<2 OR ABS(Y)<2
130 PRINT"Root is X= "
140 IF X<0 PRINT "-"
145 PRINT ABS(X)/S,"."
150 DO X=ABS(X)%S; S=S/10
155 PRINT X/S; UNTIL S=1
160 PRINT'"Needed ",I," iterations."'
170 END
200 yY=X*X/S-X-1*S
210 RETURN
```

Description of Program:

5-70	Check that starting values bracket a root
80-120	Find root by successive approximation
130-145	Print integer part of root
150-155	Print decimal places
160	Print number of iterations needed
200-210	y: Subroutine giving Y in terms of X, with appropriate scaling.

Variables:

A - Lower starting value of X
B - Upper starting value of X
C - Value of Y for X=A
D - Value of Y for X=B
I - Iteration number
S - Scaling factor; all numbers are multiplied by S and held as integers
X - Root being approximated
Y - Value of equation for given X.

Sample run:

```
>RUN
X1?1
X2?3
Root is X= 1.618
Needed 7 iterations.
```

6.3 Subroutines Calling Subroutines

Often the task carried out by a subroutine may itself usefully be broken down into a number of smaller steps, and so it might be convenient to include calls to subroutines within other subroutines. This is perfectly legal, and subroutines may be nested up to a maximum depth of 15 calls.

6.4 Recursive Subroutine Calls

Sometimes a problem can be more simply expressed if it is allowed to include a reference to itself. When a subroutine includes a call to itself in this way it is known as a 'recursive' subroutine call, and it is possible to use recursive calls in Acorn BASIC provided that the depth of recursion is limited to 15 calls. The following half-hearted program uses a recursive call to print out ten stars without using a loop:

```
   10 REM Recursive Stars
   20 P=10; GOSUB p
   30 END

  100 pREM Print P stars
  110 IF P=0 RETURN
  120 P=P-1
  130 GOSUB p; REM Print P-1 stars
  140 PRINT "*"
  150 RETURN
```

This program could, of course, be written more effectively using a simple FOR ... NEXT loop. The following programs, however, use recursion to great benefit to solve mathematical problems that would be much harder to solve using iteration alone.

6.3.1 Tower of Hanoi Problem

In the Tower of Hanoi problem three pegs are fastened to a stand, and there are a number of wooden disks each with a hole at its centre. The disks are all of different diameters, and they all start on one peg, arranged in order of size with the largest disk at the bottom of the pile.

The problem is to shift the pile to another peg by transferring one disk at a time, with the restriction that no disk may be placed on top of a smaller disk. The number of moves required rises rapidly with the number of disks used; the problem was classically described with

64 disks, and moving one disk per second the solution of this problem would take more than 500,000 million years!

A recursive solution to the problem, stated in words, is:

To move F disks from peg A to peg B:
1. Move F-1 disks from peg A to peg C
2. Move bottom disk from peg A to peg B
3. Move F-1 disks from peg C to peg B.

Also, when F is zero there is no need to do anything. Steps 1 and 3 of the procedure contain a reference to the whole procedure, so the solution is recursive.

The following program will solve the problem for up to 11 disks, and displays the piles of disks at every stage in the solution, with coloured disks on the teletext screen:

```
    1 REM Tower of Hanoi
   10 PRINT$12
   20 A=TOP;D=A+4
   30 T=#400+40*21;REM somewhere on teletext screen
   40 V=-3;W=-1;L=40;N=14
   60 !D=#1020300;!A=0;S=#20202020;M=#7F
   70 INPUT"Number of disks "F;R=T-L*F-1
   80 A?1=F;?D=F;C=1;FORZ=L TO F*L STEP L
   85 C=C+1;IFC>7 C=1
   87 R?(Z+1)=C+16
   90 FORQ=R+2TOR+Z/L+1;Z?Q=M;N.
  100 NEXT
  110 GOSUBh;END
 1000 hIF?D=0 RETURN
 1010 D!4=!D-1;D?6=D?1;D?5=D?2;D=D+4;GOSUBh
 1020 X=T-D?W?A*L+D?W*N+26
 1030 Y=T-D?V?A*L+D?V*N-N
 1040 FORQ=0TOD?-4STEP4;Y!Q=X!Q;X!Q=S;N.
 1050 A?(D?W)=A?(D?W)+W;A?(D?V)=A?(D?V)-W
 1060 D?3=D?-2;D?2=D?W;D?1=D?V;GOSUBh
 1070 D=D-4;RETURN
```

Description of Program: (for ! and ? see chapter 7)

```
100       Draw starting pile of disks
110       Subroutine h is called recursively to move the number of
          disks specified in ?D
1000      h: Subroutine to move ?D disks
1010      Recursive call to move ?D-1 disks
1020      Calculate position of disk to be moved
1030      Calculate positon to where disk will be moved
1040      Move disk
1050      Set up array A
1060      Recursive call to put back ?D-1 disks.
```

Variables:

```
A?N - Number of disks on pile N
D - Stack pointer
?D - How many disks to transfer
D?1 - Destination Pile
D?2 - Intermediate pile
D?3 - Source pile
F - Total number of disks
L - Length of screen line
M - Mark character
N - One third of screen width
S - Four space characters
```

V - Constant
W - Constant.

6.3.2 Eight Queens Problem

A classical mathematical problem consists of placing eight queens on a chessboard so that no queen attacks any other. The following program find all possible solutions to the problem, and prints out the total number of solutions:

```
  1 REM Eight Queens
 30 C=0;D=TOP;E=D+3;A=D+27;!D=0
 60 @=0;GOS.t;P."There are "C" solutions"';END
100 tIF?D=#FF C=C+1;RETURN
110 ?A=(?D|D?1|D?2):#FF
120 1IF?A=0R.
130 A?1=?A&-?A
140 ?E=?D|A?1;E?1=(D?1|A?1)*2;E?2=(D?2|A?1)/2
150 D=D+3;E=E+3;A=A+2;GOS.t;D=D-3;E=E-3;A=A-2
160 ?A=?A&(A?1:#FF);GOTO1
```

Description of Program:

30	Initialise array space. D is vector of attacks, ?D is row attacks, D?1 is left diagonal attacks, D?2 is right diagonal attacks
60	Call recursive analyser and print answer
100	t: Recursive analyser: if all rows attacked have found a solution
110	Calculate possible places to put new queen
120	If no possible place, end this recursive attempt
130	Find least significant bit in possible places to use as new queen position
140	Calculate new attacked values
150	Recursive call of analyser
160	Remove this position from possible positions and see if done.

Variables:
?A - Possible positions; value of A changes
C - Solutions counter
?D - Row attacks; value of D changes
E - Holds D+3 to make program shorter.

7.0 VECTORS

So far we have met just 26 variables, called A to Z. Suppose you wanted to plot a graph showing the mean temperature for every month of the year. You could, at a pinch, use the twelve letters A to L to represent the mean temperatures, and read in the temperatures by saying:

```
INPUT A,B,C,D,E,F,G,H,I,J,K,L
```

However there is a much better way. A mathematician might call the list of temperatures by the names:

t_1, t_2, t_3, ... t_{12}.

where the 'subscript', the number written below the line, is the number of the month in the year. This representation of the twelve temperatures is much more meaningful than using twelve different letters to stand for them, and there is no doubt about which symbol represents the temperature of, for example, the third month.

A similar series of variables can be created in Acorn BASIC; these are called 'vectors'. Each vector consists of a vector 'identifier', or name, corresponding to the name 't' in the above example, and a 'subscript'. On most computers there is no facility for writing subscripts, so some other representation is used. Each member of the vector can act as a completely independent variable, capable of holding a value just like the variables A to Z. The members of a vector are called the vector 'elements'. The total number of possible elements depends on how the vector was set up; in the above example there were twelve elements, with subscripts from 1 to 12.

Acorn BASIC provides two types of vectors, called 'byte vectors' and 'word vectors'. Byte vectors are useful when only a small range of numbers are needed, and they use less storage space than word vectors. Word vectors use enough store to hold any numerical value.

7.1 Word Vectors

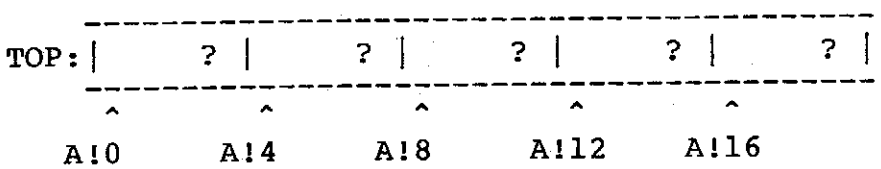
The word vector in Acorn BASIC uses an ordinary variable to hold the start address or base of the vector and the '!' (called 'pling') to represent a subscript; for example E!4. Each element in this type of vector can contain numbers as large as the simple variables A to Z, namely, between about -2000 million and 2000 million.

Before a vector can be used space must be reserved for it by utilising the value of TOP, the function which returns the first free byte after the end of the program. For example, to set up space for two vectors using A and B as bases with A having five elements and B having an unknown amount, the statement would be:

```
A=TOP; B=A+5*4
```

To leave space for N elements in B when allocating C, you should use C=B+N*4. Since the elements of a word vector require 4 bytes each the useful values for subscripts are all multiples of 4, and the allocation of space (done in bytes) must be done by multiplying by 4. We have now set things up so that A's first element, A!0 (which can be

written !A) would be at TOP, above the program text:



Note that the elements in a word vector are of the form A!(N*4). The question marks represent unspecified values, depending on what the memory contained when it was assigned. Space for the vector B is reserved immediately following on from A. Thus the fifth element of vector A is also the first element of vector B (A!20=!B).

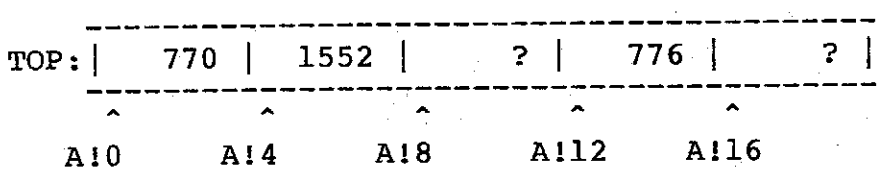
Vector elements can appear in expressions, and be assigned to, just like the simple variables A to Z. For example, to make the value of A!12 become 776 we would execute:

```
A!12=776
```

Then we could execute:

```
A!4=A!12*2
!A=A!12-6
```

and so on. The resulting vector would now be:



There are two places in BASIC programs where vector elements may not be used; these are:

1. As the control variable in a FOR ... NEXT loop
2. In an INPUT statement.

In these two cases the simple variables, A to Z, must be used.

7.1.1 Prime Numbers

The following program finds all the prime numbers up to 99999. It uses a word vector to store primes already found, and tests new candidates for divisibility by these numbers only. The amount of space given to the word vector is not determined in the program, but the program is only actively using the numbers in the vector up to the square root of the current candidate, the rest of the numbers do not need to be actually present:

```
 1 REM Prime Numbers
10 @=8;S=4;Z=0;J=TOP;G=J;!G=3;P=G+S
20 FORT=3TO99999STEP2
30 cIFT%!G=Z G=J;N.
40 IFT>!G*!G G=G+S;G.c
50 P.T;!P=T;G=J;P=P+S;N.
60 END
```

Description of Program:

```
10        Set up vector
20        Test all odd numbers
30        If divisible, try another
```

```
40          Have we tried enough divisors?
50          Must be prime - print it.
```

```
Variables:
!G - Divisor being tested
J - Equal to TOP
!P - Vector of divisors
S - Bytes per word
T - Candidate for prime
Z - Constant zero.
```

7.1.2 Sorting Program

The following program illustrates the use of vectors to sort a series of numbers into ascending order. It uses a fairly efficient sorting procedure known as the 'Shell' sort. The program, as written, reads in 20 numbers, calls a subroutine to sort the numbers into order, and prints the sorted numbers out:

```
  1 REM Sorting
  5 A=TOP
 10 FOR N=4 TO 20*4 STEP 4; INPUT J
 20 A!N=J; NEXT N
 30 GOSUB s
 40 FOR N=4 TO 20*4 STEP 4; PRINT A!N'
 50 NEXT N
 60 END
100 sQ=N/4
110 DO Q=(Q+2)/3;M=Q*4
120   FOR I=M+4 TO N STEP 4
130     FOR J=I TO M+4 STEP -M
140       IF A!J>=A!(J-M) GOTO b
150       T=A!J; A!J=A!(J-M); A!(J-M)=T
160     NEXT J
170 b NEXT I
180 UNTIL M=4; RETURN
```

```
Description of Program:
5-20        Read in vector of numbers
30          Call Shell sort
40-50       Print out sorted vector
100-180     s: Shell sort subroutine
140-150     Swap elements which are out of order.
```

```
Variables:
A!( 4 .. 80 ) - Vector to hold numbers
I,J - Loop counters
N - Elements in vector A
M - Subset step size
T - Temporary variable.
```

7.1.3 Subscript Checking

In Acorn BASIC it is left to the programmer to ensure that vector subscripts do not go out of range. Assigning to a vector whose subscript is out of range will change the values of other vectors, or strings, dimensioned after that vector.

If required, the programmer can easily add vector subscript checking; for example, if a vector A with 11 elements were assigned to:

```
R!A=35
```

the statement:

```
IF A>40 OR A<0 THEN ERROR
```

could be added before the assignment to cause an error if the vector subscript, A, went out of range.

7.2 Byte Vectors

The byte vector in Acorn BASIC again uses an ordinary variable to hold the start address of the vector and the '?' query to represent a subscript; for example E?3. Each element in this type of vector can contain numbers between 0 and 255, i.e. one byte long. Space is allocated by allowing one byte per element; for example, to allocate space for a byte vector A with six elements, execute:

```
A=TOP; B=A+6
```

The elements of this byte vector would then be:

```
A?0, A?1, A?2, A?3, A?4, A?5.
```

For many jobs, e.g. arrays of small numbers, or arrays of logical values, byte vectors are ideal since they are easier to use, and use less space than word vectors. The byte vector is also used for string manipulation, see section 8.4.1.

7.2.1 Arbitrary-Precision Arithmetic

The following program allows powers of two to be calculated to any precision, given enough memory. As it stands the program will calculate all the powers of 2 having less than 32 digits. The digits are stored in a vector A, one digit per vector element. Every power of 2 is obtained from the previous one by multiplying every element in the vector by 2, and propagating a carry when any element becomes more than one digit:

```
  5 REM Powers of Two
 10 A=TOP
 20 @=1; P=0
 30 A?0=1
 40 FOR J=1 TO 31
 50   A?J=0
 60 NEXT J
 70 DO J=31
 80   DO J=J-1; UNTIL A?J<>0
 85   PRINT'"2^" P "="
 90   FOR K=J TO 0 STEP -1
 94     PRINT A?K
 96   NEXT K
110   C=0
120   FOR J=0 TO 31
130     B=A?J*2+C
140     C=A/10
150     A?J=B%10
160   NEXT J
170   P=P+1
180 UNTIL A?31<>0
190 END
```

Description of Program:

```
40-60      Zero vector of digits
80         Ignore leading zeros
85-96      Print power
110-160    Multiply current number by 2
180        Stop when vector overflows.
```

Variables:

```
A - Vector of digits; one digit per element
C - Decimal carry from one digit to next
J - Digit counter
K - Digit counter
P - Power being evaluated.
```

7.3 Multiple Dimensions

The standard types of vector in Acorn BASIC are one-dimensional. In other words, they have just one subscript, and so can be visualised as lying in a straight line; hence the name 'vector'.

Sometimes it is convenient to make each element of a vector represent a cell in a square 'matrix'; each element would then have two subscripts corresponding to the column and row of that square. Such two-dimensional objects are called 'matrices'. Consider the following representation of a 3 by 6 matrix:

```
  0 1 2 3 4 5
 -------------
0| | | | | | |
 -------------
1| | | | | | |
 -------------
2| | | | |X| |
 -------------
```

The whole matrix has 3 x 6 = 18 elements, and the element shown with an X would have the subscripts (2,4).

Acorn BASIC does not have a direct representation for two-dimensional (or higher dimension) matrices, but they are easily represented using vectors as described in the following sections.

7.3.1 Calculation of Subscripts

To represent a two-dimensional matrix using a one-dimensional array imagine the matrix divided into rows as shown:

```
  0 1 2 3 4 5     0 1 2 3 4 5     0 1 2 3 4 5
 -------------   -------------   -------------
0| | | | | | | 1| | | | | | | 2| | | | | | |
 -------------   -------------   -------------
```

The first element of row 1, with subscripts (1,0), follows immediately after the last element of row 0, with coordinates (0,5). Consider the general case where the matrix has M rows numbered 0 to M-1, and N columns numbered 0 to N-1. The amount of space that the matrix requires (for a word vector) is:

```
(M*N-1)*4
```

Any matrix element, with subscripts A and B, can be referenced as:

```
X!((A*N+B)*4)
```

In the earlier example the matrix had dimensions 3 x 6 and so would require:

```
(3*6-1)*4=17*4=68
```

The array element with subscripts (2,4) would be given by:

```
X!64
```

Needless to say, you can actually write X!((A*N+B)*4) instead of working out the answer!

7.3.2 Solving Simultaneous Equations

The following program will solve a number of linear simultaneous equations, using a matrix to hold the coefficients of the equations, and a matrix inversion technique to find the solution. The program prints the solutions as integers, where possible, or as exact fractions.

This method has the advantage over the standard pivotal condensation technique that for integer coefficients the answers are exact integers or fractions.

The example run shown solves the pair of equations:

```
 a + 2b + 1 = 0
4a + 5b + 2 = 0
```

```
  10 REM Simultaneous Equations
  50 INPUT"Number of equations "N
  60 I=N*N;J=N*(N+1)
  65 A=TOP;C=A+4*I+4;I=C+J*4+4;REM I has N*4 bytes
  70 @=0;FOR J=1TON;FOR K=1TO N+1
  80 PRINT"C("J","K")=";INPUT B
  90 C!(((J-1)*(N+1)+K)*4)=B;NEXT;NEXT
 100 L=N*4+4;GOSUB c;E=D;M=1-2*(N%2)
 110 PRINT'"Solution:"'
 112 IF E<0 E=-E;M=-M
 115 IF E=0;PRINT"Degenerate!"';END
 120 FOR L=4TON*4STEP4;GOSUB c
 125 PRINT"X("L/4")=  "
 130 Z=M*D;B=E;DO Z=Z%B
 140 IF ABS(B)>ABS(Z) THEN T=B;B=Z;Z=T
 150 UNTIL B=0;Z=ABS(Z)
 151 P.M*D/Z;IF E/Z<>1 PRINT"/"E/Z
 155 M=-M;PRINT';NEXT L;END
 160 cFOR J=4TON*4STEP4;FOR K=4TON*4STEP4;Q=J*N-N*4+K
 170 IF K<L A!Q=C!(Q+J-4)
 180 IF K>=L A!Q=C!(Q+J)
 190 NEXT;NEXT
 200 dD=0;F=1;S=1
 210 FOR J=1TON;I!(J*4)=J;F=F*J;NEXT J
 215 GOSUB f
 220 FOR H=2TOF;GOSUB e;NEXT H;RETURN
 230 eJ=N*4-4;K=N*4
 240 gIF I!J>=I!(J+4) J=J-4;GOTO g
 250 hIF I!J>=I!K K=K-4;GOTO h
 260 GOSUB i;J=J+4;K=N*4;IF J=K GOTO f
 270 DO GOSUB i;J=J+4;K=K-4;UNTIL J>=K
 280 fP=1;FOR Y=4TON*4STEP4;P=A!(N*Y+(I!Y-N)*4)*P
 290 NEXT Y;D=D+S*P;RETURN
 300 iY=I!J;I!J=I!K;I!K=Y
 310 S=-S;RETURN
```

Description of Program:

```
50-60     Allocate space for matrix
```

```
70-90      Read in matrix of coefficients
120-155    Print solutions
130-150    Find GCD of solution, so it is printed in lowest terms
160-190    c: Permute terms to obtain next addition to determinant; i.e.
           for 5 equations, starting with (1,2,3,4,5) run through all
           permutations to (5,4,3,2,1)
280-290    f: Add in next product to determinant
300-310    i: Swap terms in permutation.
```

Variables:
A(1 ... N*N) - Matrix
C(1 ... N*N+N) - Matrix of coefficients
S - Signature of permutation.

Sample run:

```
>RUN
Number of equations ?2
C(1,1)=?1
C(1,2)=?2
C(1,3)=?1
C(2,1)=?4
C(2,2)=?5
C(2,3)=?2

Solution:
X(1)=  1/3
X(2)=  -2/3
```

7.4 Call by Reference

A major advantage of vectors over the arrays found in more normal BASIC interpreters is that their base addresses are available as values, and so can be passed to subroutines. As an example, consider this program:

```
   10 A=TOP; B=A+40
   .
   .
   90 P=A; GOSUB p; REM Output A
   94 P=B; GOSUB p; REM Output B
   98 END

  100 pREM Print 10 Elements of array P
  105 @=8; PRINT '
  110 FOR J=0 TO 39 STEP 4
  120   PRINT P!J
  130 NEXT J
  140 PRINT '
  150 RETURN
```

In this example subroutine p can be used to print any vector by passing its base address over in the variable P; this is known as a 'call by reference' because the subroutine is given a reference to the vector, rather than the actual values in the vector.

7.4.1 Arbitrary Precision Powers

The following program illustrates the use of word vectors to calculate the value of any number raised to any other number exactly, limited only by the amount of memory available. The program stores four decimal digits per word, so that the product of two words will not cause overflow, and the result is calculated as a word vector:

```
  1 REM Arbitrary Precision Powers
  5 T=TOP;DOT=T+1;?T=#55;UNTIL?T<>#55
 10 H=(T-TOP-3)/3;P=TOP;S=P+H;D=P+H
 15 H=10000
 20 @=0;PRINT'"     Power Program"
 30 PRINT'"  Computes Y^X, Where X>0 and Y>0"
 40 INPUT'"  Value of Y"Y,"  Value of X"X
 50 IFX<1ORY<1PRINT"  Value out of range";GOTO40
 60 M=Y;N=X;GOSUBp
 70 PRINT Y"^"X"="P!!P;IF!P<8 GOTO10
 90 FORL=!P-4TO4STEP-4
 95 IFL!P<1000P.0
100 IFL!P<100P.0
110 IFL!P<10P.0
120 P.L!P;N.;GOTO10
200 pJ=M;IFN%2=0J=1
210 R=P;GOS.e;J=M;R=S;GOS.e;IFN=1R.
250 B=S;DOA=B;GOS.m;B=E
255 N=N/2;A=P;IFN%2GOS.m;P=E
260 U.N<2;R.
300 m!D=!A+!B+4;F.J=4TO!D+4S.4
310 D!J=0;N.;W=D-4
320 F.J=4TO!B S.4;C=0;G=B!J
325 V=W+J;F.L=4TO!A S.4
330 Q=A!L*G+C+V!L;V!L=Q%H
340 C=Q/H;N.;V!L=C;N.
370 DO!D=!D-4;U.D!!D<>0;E=D;D=A;R.
400 e!R=0;DOd=!R+4;R!!R=J%H
410 J=J/H;U.J<1;R.
```

Description of Program:

5	Set T to top of available memory; if you know the address, replace by e.g. T=#4000
10	Divide available memory between P, S, and D
20-40	Read in values of Y and X
50	Disallow negative values
60	Calculate power
70	Print result if it fits in one word
90	Print rest of result, filling in leading zeros
140	Blank line to make listing clearer
200-260	p: Calculates power. Looks at binary representation of X and for each bit squares B, and if bit is a 1 multiplies P by current B
300-370	m: Multiply together the vectors pointed to by A and B and put the result into the vector pointed to by D. Pointers to vectors get changed; E points to result
400-410	e: Unpack J into vector pointed to by R; store number of words in !R.

Variables:

D!0 ... - Workspace vector
H - Radix for arithmetic
P!1 ... - Vector for unpacked result
!P - Number of elements used in P
S!0 ... - Workspace vector
T - Top of available memory.

Sample run:

```
>RUN
```

```
      Power Program
    Computes Y^X, Where X>0 and Y>0
    Value of Y?16
    Value of X?64
  16^64=1157920892373161954235709850086879078532699846656405640394575
84007913129639936
```

7.5 Vectors of Vectors

A second way of representing two-dimensional arrays is possible using word vectors; this avoids the need for a multiplication to calculate the subscript, but does require slightly more storage. The idea is to think of a two-dimensional matrix as a vector of vectors; first a vector is created containing the addresses of the rows of the matrix. For example, for a byte matrix called X with columns 0 to M, and rows 0 to N, the following statements will set up the vector of row addresses:

```
X=TOP;T=X+2*N
FOR J=0 TO N*2 STEP 2; X!J=T;T=T+M; NEXT J
```

A word array is used to hold the base addresses. T is used to allocate the space for each vector. Now that the vector of row base addresses has been set up, the element with subscripts A,B is:

```
X!(A*2)?B
```

8.0 STRINGS

A 'string' is a sequence of characters; the characters can be anything - letters, digits, or punctuation marks. They can even be control characters.

8.1 Quoted Strings

Strings are represented in a program by enclosing the characters between quotation marks; quoted strings have already been introduced in the context of the PRINT and INPUT statements. For example:

```
"This is a string"
```

To represent a quotation mark in a quoted string the quotation mark is typed twice. Valid strings always contain an even number of quotation marks. For example:

```
PRINT"He said: ""This is a valid string"""
```

will print:

He said: "This is a valid string"

8.2 String Variables

The variables A to Z have already been met, where they are used to represent numbers. These variables can also be used to represent strings, and strings can be manipulated, input with the INPUT statement, printed with the PRINT statement, and there are several functions for manipulating strings.

8.2.1 Allocating Space for Strings

BASIC allows strings of any size up to 255 characters. To use string variables space for the strings should first be allocated as if you were about to use a byte vector, with an additional byte for the string terminator. For example, for a string of up to 10 characters using the variable A the statement would be:

```
A=TOP;B=A+11;REM B is next free byte
```

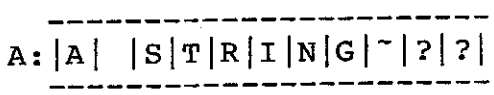
8.2.2 String Operator '$'

Having allocated space for the string it can then be assigned a value. For example:

```
$A="A STRING"
```

The '$' is the string-address operator. It specifies that the value following it is the address of the first character of a string.

The effect of the statement A=TOP is to set A to the address of free memory above the text of the BASIC program. In other words, A is a pointer to that area of memory. After the above assignment the contents of those locations are as follows:



The question-marks indicate that the last two locations could contain anything. The character '~' represents 'return' which is automatically stored in memory to indicate the end of the string. This is why you need to allocate one extra location to hold this terminator character, although you will not normally be aware of its presence when actually using the strings.

Note that it would be dangerous to allocate a string of more than 10 characters to A since it would exceed the space allocated to A.

8.2.3 Printing Strings

A string variable can be printed by writing:

```
PRINT $A
```

This would print:

```
A STRING>
```

and no extra spaces are inserted before or after the string.

8.2.4 String Assignment

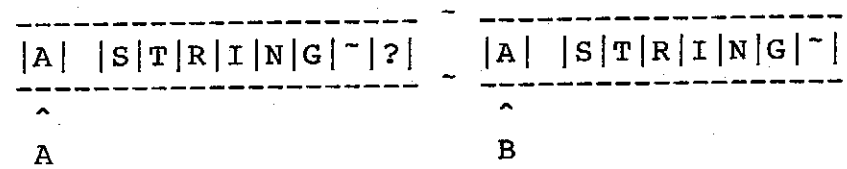
Suppose that two strings are allocated as follows:

```
A=TOP;B=A+256
```

The allocation allows for all eventualities, since the string A cannot exceed 255 (+1) characters. The string $A can be assigned to $B by the statement:

```
$B=$A
```

which should be read as 'string B becomes string A'. The result of this assignment in memory is as follows:



8.2.5 String Equality

It is possible to test whether two strings are equal with the IF statement. Example:

```
$A="CAT"; $B="CAT"
IF $A=$B PRINT "SAME"
```

would print SAME.

8.2.6 String Input

The INPUT statement may specify a string variable, in which case the string typed after the '?' prompt, and up to the 'return', will be assigned to the string variable. The maximum length of line that can be typed in to an INPUT statement is 64 characters so, for safety, the string variable in the INPUT statement should be allocated with a length of 64.

8.3 String Functions

Several functions are provided to help with the manipulation of strings.

8.3.1 Length of a String - LEN

The LEN function will return the number of characters in the string specified by its argument. For example:

```
$A="A STRING"
PRINT LEN(A)
```

will print the value 8. Note that:

```
$B=""""
PRINT LEN(B)
```

will print 1 since the string B contains only a single quote character.

8.3.2 CH

The CH function will return the ASCII value of the first character in the string specified by its argument. Thus:

```
CH"A"
```

will be equal to 65, the ASCII code for A. The string terminating character 'return' has a value of 13, so:

```
CH""
```

will be equal to 13.

8.4 String Manipulations

The following sections show how the characters within strings can be manipulated, and how strings can be concatenated into longer strings or broken down into substrings.

8.4.1 Character Extraction - '?'

Individual characters in a string can be accessed with the question-mark '?' operator. Consider again the representation of the string A. Number the characters, starting with zero:

```
-----------------------
|A| |S|T|R|I|N|G|~|?|?|
-----------------------
 0 1 2 3 4 5 6 7 8 9 10
 ^
 A
```

The value of the Nth character in the string is then simply A?N. For example, A?7 is "G", etc. In general A?B is the value of the character stored in the location whose address is A+B; therefore A?B is identical to B?A. In other words, a string is being thought of as a byte vector whose elements contain characters; see section 7.2.

The following program illustrates the use of the '?' operator to convert the case of all the characters in a string which is typed in:

```
 1 REM Convert String
 5 Q=TOP
10 INPUT $Q
20 FOR N=0 TO LEN(Q)-1
30 Q?N=Q?N : #20
```

```
   40 NEXT N
   50 PRINT $Q
   60 RUN
```

8.4.2 Encoding/Decoding Program

As a slightly more advanced example of string operations using the '?' operator, the following program will produce a very secure encoding of a message. The program is given a number, which is used to 'seed' BASIC's random number generator. To decode the text the negative of the same seed must be entered:

```
    1 REM Encoder/Decoder
   10 S=TOP; ?12=0
   20 INPUT'"Code number "T
   30 !8=ABS(T)
   40 INPUT'$S
   50 FOR P=S TO S+LEN(S)
   60 IF ?P<#41 GOTO 100
   70 R=ABS(RND)%26
   80 IF T<0 THEN R=26-R
   90 ?P=(?P-#41+R)%26+#41
  100 NEXT P
  110 PRINT $S
  120 GOTO 40
```

Description of Program:

20	Input code number
30	Use code number to seed random number generator
40	Read in line of text
50-100	For each character, if it is a letter add the next random number to it, modulo 26
110	Print out encoded string.

Variables:

P - Address of character in string
R - Next random number
S - Address of string; set to TOP
T - Code number.

Sample run:

```
>RUN

Code number ?123

?MEETING IN LONDON ON THURSDAY
BGYKPYI CM NHSHVO VU RGFGDHJI
? >
>RUN

Code number ?-123

?BGYKPYI CM NHSHVO VU RGFGDHJI
MEETING IN LONDON ON THURSDAY
```

To illustrate how secure this encoding algorithm is, try decoding the following quotation:

```
YUVHW ZY WKQN IAVUAG QM SHXTSDK
GSY IEJB RZTNOL UFQ FTONB JB BY
CXRK QCJF UN TJRB.
```

```
SWB FJA IYT WCC LQFWHA YHW OHRMNI OUJ
HTJ I TYCU GQYFT FT SGGHH HJ FRP ELPHQMD,
RW LN QOHD OQXSER CUAB.
DKLCLDBCV.
```

8.4.3 Concatenation

Concatenation is the operation of joining two strings together to make one string. To concatenate string B to the end of string A execute:

```
$A+LEN(A)=$B
```

For example:

```
10 A=TOP;B=A+20
20 $A="Acorn "
30 $B="BASIC"
40 $A+LEN(A)=$B
50 PRINT $A'
60 END
```

will print:

```
Acorn BASIC>
```

8.4.4 Right-String Extraction

The right-hand part of a string A, starting at character N, is simply:

```
$A+N
```

For example, executing:

```
10 A=TOP;B=A+20
20 $A="Acorn BASIC"
30 $B=$A+6
40 END
```

will give string B the value "BASIC".

8.4.5 Left-String Extraction

A string A can be shortened to the first N characters by executing:

```
$A+N=""
```

Since the 'return' character has the value 13, this is equivalent to:

```
A?N=13
```

8.4.6 Mid-String Extraction

The middle section of a string can be extracted by combining the techniques of the previous two sections. For example, the string consisting of characters M to N of string A is obtained by:

```
$A+N=""; $A=$A+M
```

For example, if the following is executed:

```
10 A=TOP
20 $A="Acorn BASIC"
30 $A+5=""; $A=$A+1
40 END
```

then string A will have the value "corn".

8.5 Vectors of Fixed-Length Strings

The word vectors may be used as string variables, thus providing the ability to have vectors of strings. To allocate space for a vector of strings the allocation statement can be incorporated into a FOR ... NEXT loop. For example, the following program allocates space for 21 strings, A!0 to A!80, each capable of holding 10 characters:

```
25 A=TOP;T=A+84
35 FOR N=0 TO 80 STEP 4
40   A!N=T
50   T=T+11
60 NEXT N
```

Note the use of variable T to allocate the space for each string. Individual elements of the string array can then be assigned to as follows:

```
$A!0="ZERO"
$A!4="ONE"
$A!40="TEN"
```

and so on.

8.5.1 Day of Week

The following program calculates the day of the week for any date in the 20th. century. It stores the names of the days of the week in a string vector. Note that this program uses a calculation to select each particular string:

```
  1 REM Day of Week
 10 A=TOP
 20 $A="Sunday";$A+10="Monday"
 30 $A+20="Tuesday";$A+30="Wednesday"
 40 $A+40="Thursday";$A+50="Friday"
 50 $A+60="Saturday"
 70 INPUT"Day of week"''"Year "Y,"Month "M,"Date in month "D
 80 Y=Y-1900
 90 IF Y<0 OR Y>99 PRINT"ONLY 20TH CENTURY !"';GOTO 70
100 IF M>2 THEN M=M-2; GOTO 120
110 Y=Y-1; M=M+10
120 E=(26*M-2)/10+D+Y+Y/4+19/4-2*19
130 PRINT"It is " $A+ABS(E%7)*10 ''
140 END
```

Description of Program:

10	Allocate space for string vector
20-50	Set vector elements
70	Input date
80-120	Calculate day
130	Print day of week.

Variables:

$A - String vector to hold names of days
D - Date in month
E - Expression which, modulo 7, gives day of week
M - Month
N - Counter
Y - Year in 20th century.

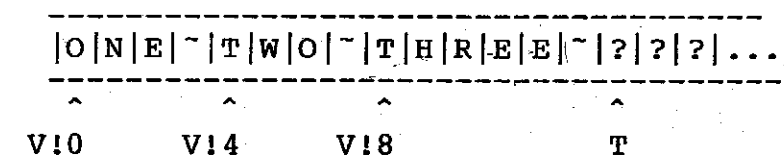
8.6 Arrays of Variable-Length Strings

The most economical way to use the memory available is to allocate only as much space as is needed for each string. For example the following program reads in 10 strings and saves them in strings called V!0 to V!36:

```
 10 V=TOP;T=V+40
 20 FOR N=0 TO 36 STEP 4
 30 INPUT $T
 40 V!N=T
 50 T=T+LEN(T)+1
 60 NEXT N
 70 INPUT "STRING NUMBER",N
 80 PRINT $V!N'
 90 GOTO 70
```

T is set to the address of the first free memory location. T is then incremented past each string to the next free memory location as each string is read in. Finally, when 10 strings have been read in the program prompts for a string number and types out the string of that number.

For example, if the first three strings entered were: "ONE", "TWO", and "THREE", the contents of memory would be:



8.7 Reading Text

Some BASICs have statements READ and DATA whereby strings listed in the DATA statements can be read into a string variable using the READ statement.

Although Acorn BASIC does not provide these actual statements, reading strings specified as text is a fairly simple matter. The following program reads the strings "ONE", "TWO" ... etc. into a string variable, $A, and prints them out. The strings for the numbers are specified as text after the program. They are identified by a label 't', and a call to subroutine 'f' sets Q to the address of the first string. Subroutine 'r' will then read the next string from the list:

```
 10 REM Read Text
 20 A=TOP; L=CH"t"
 25 GOSUB f
 30 FOR J=1 TO 20; GOSUB r
 40 PRINT $A '
 50 NEXT J
 60 END
500 fREM point Q to text
510 Q=?18*256
520 DO Q=Q+1
530 UNTIL ?Q=#D AND Q?3=L
540 Q=Q+4; RETURN
550 *
600 rREM read next entry into A
605 REM changes: A,Q,R
610 R=-1
```

```
  620 DO R=R+1; A?R=Q?R
  630 UNTIL A?R=CH"," OR A?R=#D
  640 IF A?R=#D Q=Q+3
  650 Q=Q+R+1; A?R=#D; RETURN
  660 *
  800 tONE,TWO,THREE,FOUR,FIVE
  810  SIX,SEVEN,EIGHT,NINE,TEN
  820  ELEVEN,TWELVE,THIRTEEN
  830  FOURTEEN,FIFTEEN,SIXTEEN
  840  SEVENTEEN,EIGHTEEN,NINETEEN
  850  TWENTY
```

Description of Program:

25	Find the text
30	Read in the next string
40	Print it out
500-550	f: Search for label t and point Q to first string
600-660	r: Read up to comma or return and put string into $A
800-850	t: List of 20 strings, note the space after the line number.

Variables:

$A - String
J - Counter
L - Label for text
Q - Pointer to strings
R - Temporary pointer.

The program can be modified to read from several different blocks of text with different labels by changing the value of L. Also note that the character delimiting the strings may be any character, specified in the CH function in line 630.

8.7.1 Reading Numeric Data

Numeric data can be specified as strings of characters as in in the Read Text program of the previous section, and converted to numbers using the VAL command in the extension ROM. For example, modify the Read Text program by changing line 40 to:

```
   40 FPRINT VAL A
```

and provide numeric data at the label 't', for example as follows:

```
  800 t1,2,3,4,1E30,27,66
  810  91,1.2,1.3,1.4,1.5
  820  13,14,15,16,17
  830  18,19,20
```

8.8 Printing Single Characters - '$'

A special use of the '$' operator in the PRINT statement is to print characters that can not conveniently be specified as a string in the program, such as control characters and graphics symbols. Normally '$' is followed by a variable used as the base address of the string. If, however, the value following the dollar is less than 255, the character corresponding to that code will be printed instead.

The most useful control codes are specified in the following sections; for a full list of control codes see section 10.1.

8.8.1 Cursor Movement

The cursor can be moved in any of the four directions on the screen using the following codes:

Hex	Decimal	Cursor Movement
#08	8	Left
#09	9	Right
#0A	10	Down
#0B	11	Up

The screen is scrolled when the cursor is moved off the bottom line of the screen; the cursor cannot be moved off the top of the screen.

8.8.2 Screen Control

The following control codes are useful for controlling the VDU screen:

Hex	Decimal	Control Character
#0C	12	Clear screen and home cursor
#1E	30	Home cursor to top left of screen

8.8.3 Random Walk

The following program prints characters on the screen following a random walk. One of the cursor control codes, chosen at random, is printed to move the cursor; a character, chosen at random, is then printed followed by a backspace to move the cursor back to the character position:

```
 1 REM Random Walk
10 DO
20 PRINT $ABS(RND)%4+8, $(#20+ABSRND%96), $8
30 UNTIL 0
```

9.0 READING AND WRITING DATA

The reader should now be familiar with the three types of data that can be manipulated using Acorn BASIC, namely:

1. Words i.e. numbers between -2000 million and 2000 million (approximately).
Storage required: 4 bytes
e.g. variables A to Z
word vectors A!4 ... etc.
indirection !A ... etc.

2. Bytes i.e. numbers between 0 and 255, or single characters, or logical values.
Storage required: 1 byte
e.g. byte vectors A?1 ... etc.
indirection ?A ... etc.

3. Strings i.e. sequences of between 0 and 255 characters, followed by a 'return'.
Storage required: Length+1 bytes
e.g. quoted string "A STRING"
string variable $A ... etc.

All these types of data can be written to files and read from files, making it very simple to make files of data generated by programs.

BASIC's functions and statements for file input and output are designed to be used with the disk operating system, but a restricted set of them can be used with the cassette operating system. When the disk operating system is used, several files can be used by one program, and the individual files are identified by a 'file handle', a number specifying which file is being referred to. Although this facility is not available when working with a cassette operating system, the file handle is still required for compatability.

9.1 Find Input and Find Output

The functions FIN (find input) and FOUT (find output) MUST be called before inputting from, or outputting to, files when used with the disk operating system; if used with the cassette operating system they will cause an error. The functions are called with a string as the argument which represents the name of the required file, and they return a value in the range 0 to 255, which will identify this file when it is used. This value is the 'file handle'.

The FOUT function is called as follows:

```
A=FOUT"EXAMPLE"
```

and it will open the file EXAMPLE on the current drive in the current qualifier for output. That is, if EXAMPLE exists its current size is used, but none of the data may be accessed, if it does not exist then a new file with the disk operating system's default file size will be created. Refer to your disk operating system manual for further details.

The FIN function is called as follows:

```
A=FIN"E.G.IN"
```

and it will open the file E.G.IN on the current drive in the current qualifier for input and updating. The file must exist, otherwise the value returned will be zero. Refer to your disk operating system manual for further details.

9.2 Output

To output a word to file the PUT statement is used. Its form is:

```
PUT A,W
```

where A and W are the file handle and word for output respectively.

To output a byte to file the BPUT statement is used; the form is:

```
BPUT A,B
```

where A is the file handle, and B is the byte for output.

To output a string the SPUT statement is used. The form is:

```
SPUT A,S
```

where A is the file handle, and S is the base address of the string.

9.3 Input

To read a word from file the GET function is used. Its form is:

```
GET A
```

where A is the file handle. The function returns the value of the word.

To read a byte the BGET function is used. Its form is:

```
BGET A
```

where A is the file handle. The BGET function returns the value of the byte, and can therefore be used in expressions; for example:

```
PRINT BGET A + BGET A
```

will read two bytes from files and print their sum.

To read strings the SGET statement is used. The form is:

```
SGET A,S
```

where A is the file handle, and S is the base address where the string will be stored. The string S should be large enough to accomodate the string being read.

Note the difference between SGET, which is a statement, and the functions BGET and GET; SGET cannot be used in expressions.

9.4 Data Control

With the disk operating system, BASIC can discover the length of files, and can control the position of the next byte to be read or written. The statements that control this will cause errors with the cassette operating system.

Once a file has been opened with FIN or FOUT, its extent (current length) may be read with the EXT function:

```
A=FIN"FRED"; PRINT"FRED is "EXT A" bytes long"'
```

When a file has been opened, it can be considered as a group of bytes numbered 0,1,... . The number of the next byte to be transferred (either read or written) is available as the value of the function PTR. The PTR of a file may also be updated:

```
A=FIN"FRED"
PRINT"Initial value "PTRA'; REM will be zero
PTRA=PTRA+20;REM skip first 20 bytes
```

Both input and output file pointers may be manipulated. After a byte is transferred a file's pointer is automatically incremented. Refer to your disk operating system manual for further details.

9.5 Data with Cassette

With the current cassette operating system a byte can be read from the moving tape, and bytes can be written to tape. However, when writing to tape an I/O port must first be initialised, the simplest way to do this is to use a single BPUT statement before asking for the drive to be started.

9.5.1 Data to Cassette

The following program prompts for a series of values, terminated by a zero, and saves them on a cassette tape. The first word saved on the tape is the number of words of data following:

```
  1 REM Data to Cassette
 10 V=TOP;BPUTA,-1;REM dummy bput to initialise
 20 N=0
 30 DO INPUT J
 40 V!N=J; N=N+4
 50 UNTIL J=0
 60 PRINT"Start tape recording"';LINK#FFE3;FORZ=0TO1000;N.
 70 PUT A,(N-4)
 80 FOR M=0 TO N-4 STEP 4
 90 PUT A,V!M
100 NEXT M
110 END
```

Description of Program:

```
30-50     Input numbers
60        Warn user to start tape; use #FFE3 to wait for a key
70        Output number of bytes
80-100    Save values on files.
```

Variables:

```
A - Dummy file handle
J - Temporary variable for values input
M - Counter
N - Counter for number of values
V!(0 ... 80) - Vector of numbers.
```

The next program reads the values back in and prints them out, together with the maximum and minimum values:

```
  1 REM Read from Cassette
 10 V=TOP
 20 PRINT"Play tape"';LINK#FFE3;N=GET A
 30 FOR M=0 TO N STEP 4
 40 V!M= GET A
 50 NEXT M
 60 REM X=Maximum, Y=Minimum
 70 X=!V; Y=X;PRINT"Data "X
 80 FOR M=4 TO N STEP 4
 90 PRINTV!M
100 IF X<V!M THEN X=V!M
110 IF Y>V!M THEN Y=V!M
120 NEXT M
130 PRINT'"Maximum "X'"Minimum "Y''
140 END
```

Description of Program:
20-50 Read values into array
70-130 Find maximum and minimum values in vector and print them

Variables:
A - Dummy file handle
M - Counter
N - Number of values in array
V(0 ...) - Vector of values
X - Maximum value
Y - Minimum value

9.5.2 Reading and Writing Speed

When writing data to the files it is important to remember that the program reading the data back will not be able to control the cassette; it will have to read the data before it has passed under the tape head. If the program to read the data will spend a substantial time between reading, it may miss bytes passing under the tape head unless a delay is inserted between bytes when writing to tape.

As a general guide, the program to read the data should take no longer to read each byte than the program to write the data takes to write it.

9.5.3 Animal Learning Program

The following program illustrates how a computer can be 'taught' information, so that a 'database' of replies to questions can be built up. The computer plays a game called 'Animal'; the human player thinks of an animal and the computer tries to guess it by asking questions to which the answer is either 'yes' or 'no'. Initially the computer only knows about a dog and a crow, but as the game is played the computer is taught about all the animals that it fails to guess.

The program uses the files input/output statements to load the database, or tree, from files at the start of the game, and to save the enlarged database at the end of the game.

First create a database by typing:

```
GOSUB 9000;
```

and record the database on a files. Then RUN the program and load the database you have just recorded. When the reply 'NO' is given to the question 'Are you thinking of an animal' the program will save the new, enlarged, database on files. Also given is a sample run which was obtained after several new animals had been introduced to the computer:

```
 1 REM Animals
10 REM Load Tree
20 PRINT"Play tape"';LINK#FFE3
25 FOR T=TOP TO TOP+GET F
30 ?T=BGET F; NEXT T
35 DO X=TOP
40 PRINT'"Are you thinking of an animal"
45 GOSUB q
48 IF Q=0 THEN GOSUB z; END
50 DO PRINT $X+1
60 GOSUB q
65 P=X+LENX+1+Q; X=!P+TOP
70 UNTIL ?X<>CH"*"
75 PRINT"IS IT " $X
80 GOSUB q
```

```
   85 IF Q=4 PRINT "HO-HO";UNTIL 0
   88 DO
   90 INPUT"What were you thinking of"$T
   95 UNTIL LEN T>2
   98 L=T; GOSUB s
  100 PRINT"   Tell me a question "
  110 PRINT"that will"'"distinguish "
  120 PRINT "between " $L" and " $X '
  130 $T="*"; R=T+1
  140 INPUT $R; !P=T-TOP; GOSUB s
  145 K=T; T=T+8; GOSUB j
  150 GOSUB q
  160 K!Q=X-TOP; K!(4-Q)=L-TOP
  170 UNTIL 0
 1000 qINPUT $T
 1010 IF ?T=CH"Y"OR?T=CH"y"THEN Q=4; RETURN
 1020 IF ?T=CH"Q"THEN END
 1030 Q=0; RETURN
 2000 j$T=$R
 2010 FOR A=2 TO LEN T-5
 2020 V=T?(A+4); $T+A+4=""
 2030 IF $T+A=" IT " GOTO 1
 2035 T?(A+4)=V
 2040 NEXT A
 2100 PRINT"What would the answer be"'
 2110 PRINT"For " $X
 2120 RETURN
 21501T?(A+4)=V; $T+A+1=""
 2160 PRINT $T,$X,$T+A+3
 2170 RETURN
 3000sT=T+LEN T+1; RETURN
 9000 REM Set-Up File
 9010 T=TOP; $T="*DOES IT HAVE FOUR LEGS"
 9015 GOSUB s; P=T; T=T+8; !P=T-TOP
 9020 $T="A CROW"; GOSUB s; P!4=T-TOP
 9025 $T="A DOG"; GOSUB s
 9100zREM Save Tree
 9110 BPUTF,F;PRINT"Record tape"';LINK#FFE0
 9115 PUT F,(T-TOP)
 9120 FOR N=TOP TO T
 9130 BPUT F, ?N
 9140 NEXT N
 9150 RETURN
```

Description of Program:

```
20-30      Load previous tree
35         Reset X to top of tree
50         Print next question
70         Carry on until not a question
75         Guess animal
90-95      Wait for a sensible reply
98         Find end of reply
1000-1030  q: Look for Y, N, or Q; set Q accordingly
2000-2120  j: Look for " IT " in question and print question with " IT "
           replaced by name of animal
3000       s: Move T to end of string $T
9000       Set up tree file
9100       z: Save tree file.
```

Variables:

F - Dummy file handle

```
K - Pointer to addresses of next two branches of tree
L - Pointer to animal typed in
P - Pointer to address of next question or animal
Q - Value of reply to question; no=0, yes=4
R - Pointer to question typed in
T - Pointer to next free location
X - Pointer to current position on tree.
```

Sample run:

```
>RUN
Are you thinking of an animal?Y
DOES IT HAVE FOUR LEGS?Y
CAN YOU RIDE IT?N
DOES IT HAVE STRIPES?N
IS IT A DOG?N
What were you thinking of?A MOUSE
   Tell me a question that will
distinguish between A MOUSE and A DOG
?DOES IT SQUEAK
DOES A DOG SQUEAK?NO

Are you thinking of an animal?Y
DOES IT HAVE FOUR LEGS?Y
CAN YOU RIDE IT?N
DOES IT HAVE STRIPES?N
DOES IT SQUEAK?Y
IS IT A MOUSE?Y
HO-HO
Are you thinking of an animal?N
Record tape
>
```

9.6 Data with Disk

All the programs in section 9.5 can be run with the disk by inserting appropriate FIN and FOUT statements, but the programs in this section cannot be run with the cassette operating system. With the disk, the invalid file handle of zero has the special function of reading (if GET) from the keyboard or writing (if PUT) to the output channel. Up to five files may be opened simultaneously.

9.6.1 Going Backwards

The following program illustrates random access to a disk file using PTR. It prompts for the name of a file and then prints the contents in reverse order, as hexadecimal numbers:

```
 10 INPUT"Name of file "$TOP
 20 A=FIN$TOP;IFA=0 GOTOn
 30 IFEXTA=0 PRINT"Nothing";END
 40 @=4;FORZ=EXTA TO 0 STEP-1
 50 PTRA=Z;PRINT&BGETA;NEXT
 60 PRINT';END
100 nPRINT"File not on disk""Here is catalogue"'
110 *CAT
120 RUN
```

9.6.2 File Editor

The following program allows you to create, edit and merge files, with a view to creating an EXEC file.

Up to three source files may be opened by typing their names in response to the "SOURCE" prompt. Typing return to this prompt at any stage ends the allocation of inputs. The program then prompts for an input string; the string input at this stage is normally allocated as the keyboard source; source 0.

Special strings input at this stage act as commands to the file editor:

n	Space n return	will take the next string off source n and transfer it to the output
Sn	"S" n return	will step to the next string in source n without affecting the output
Rn	"R" n return	will rewind source n without affecting the output
An	"A" n return	will transfer all of the remaining strings in source n to the output file
%DEL	"%DEL"	will delete the last string that was transferred from the output file
%END	"%END"	will terminate the creation of the output file.

After typing "%END" in response to the "INPUT STRING" prompt the program will prompt for the name of the destination file. The file that has been created will be saved with the name typed in at this point. If no destination file is required then an input of "%NONE" will create no output file.

If any errors occur or the escape key is pressed while the program is running the catalogue must be tidied up by typing:

```
SHUT0
*DELETE XXXX
```

The program may then be executed again with safety. Here is the program:

```
 10 P.$12
 20 P."SOURCE                          DESTINATION"
 30 A=TOP;B=A+4;C=B+40;D=C+80
 40 E=D+8;T=E+20
 50 P=0
 60 $E="."
 70 !D=#100B0600;D!4=0
100 GOSUBa;REM allocate source get strings
110 REM   Create destination
120  *SAVE XXXX 4000
130  I=FIN"XXXX"
140 DO X=22;Y=4;GOSUBc
150    P."Input string";GOSUBg
160    INPUT$T
170    IFT?1>#2F AND T?1<(#30+Z) AND T?2=13;GOSUBv;UNTIL0
180    IF$T="%DEL";P=P-1;GOSUB1;U.0
190    IF?T<>CH"%";GOSUBh;U.0
200 UNTIL$T="%END"
210 tS=PTRI;SHUT0
220 I=FIN"XXXX"
230 INPUT"Destination file name "$T
240  IF$T="%NONE";GOTO280
250 K=FIN$T;IFK=0;K=FOUT$T;GOTOi
260 SHUTK;$#100="DELETE    ";$#108=$T;LINK#FFF7;K=FOUT$T
270 iDO;SGETI,T;SPUTK,$T;U.PTRI=S
280 SHUT0
290 *DELETE"XXXX"
```

```
 300 *CAT
 310 END
 320 jDO;$T=$(C+J*20);GOSUBd;GOSUBe;U.$T="End of file";R.
 340 REM    Allocate sources
 360 aZ=1
 370 zJ=Z-1
 380  X=D?J;Y=D?(J+4);GOSUBc
 390  @=2;P."Source "Z
 400  GOSUBg
 410  INPUT" "$T
 420  IF$T="";P.$11;GOSUBg;GOTOx
 430   G=FIN$T;IFG=0;GOTO380
 440  A?J=G;$(B+J*10)=$T
 450  Z=Z+1;IFZ<4;GOTOz
 460 xIFZ=1;RETURN
 470  FORJ=0TOZ-2;GOSUBe;N.;R.
 490  REM New string to output file
 510 dSPUTI,$T
 520  P=P+1
 530 1 PTRI=0;Q=P
 540  IF P>15 DO;SGETI,T;Q=Q-1;UNTILQ<16
 550  X=17-Q;Y=20
 560  FOR R=1 TO Q;GOSUB c;GOSUB g
 570  SGETI,T;PRINT $T
 580  X=X+1;NEXT
 590 RETURN
 600 vJ=T?1-#30
 610  IF?T=CH"S"ANDJ<>0;J=J-1;GOSUBe;R.
 620  IF?T=CH"R"ANDJ<>0;J=J-1;PTR(A?J)=0;GOSUBe;R.
 630  IF?T=CH"A"ANDJ<>0;J=J-1;GOSUBj;R.
 640  IF?T<>32;GOSUBh;R.
 650  IFJ=0;$T=$E;GOSUBd;R.
 660  J=J-1
 670  $T=$(C+J*20);GOSUBd;GOSUBe;R.
 690  REM Move cursor to X,Y
 710 cP.$30;FORO=0TOX;P.$10;N.;FORO=0TOY;P.$9;N.;R.
 730  REM   New string off input
 750 e
 760  IFPTR(A?J)>=EXT(A?J);$T="End of file";GOTOy
 770  SGET(A?J),T
 780  IF LENT>19;$T="No string"
 790 y$(C+J*20)=$T
 800  X=D?J+1;Y=D?(J+4);GOSUBc;GOSUBg;P.$T
 810 RETURN
 830  REM   Update keyboard channel
 850 hX=18;Y=1;GOSUBc;P."SOURCE 0",$13,$10," KEYBOARD = "
 860  GOSUBg;P.$T
 870  $E=$T
 880 RETURN
 900  REM   Clear 20 spaces on screen
 920 gP."                    ";FORO=0TO18;P.$8;N.;R.
```

Description of program

```
10-20      Clear screen print titles
30-70      Initialise variables and pointers
100-130    Create inputs and outputs
140-200    Input string and check for commands
210-220    Shut all files and open "XXXX" again
230-240    Input destination name and check for "%NONE"
250-270    Transfer created file to destination
```

```
280-310    Shut all files and tidy up
320        j:Transfer all of source to destination
340-470    a:Allocate source files (up to three)
490-590    d:Put string $T onto destinaing $T onto destination
530-590    l:Print out contents of destination file
600-650    v:Interpret input command
660-670    Transfer one string from source to destination
690-710    c:Move cursor to X,Y
730-810    e:Get next string off source into $T
830-880    h:Put new string to keyboard source
900-920    g:Clear half a line on the screen.

Variables:
A - Pointer to source handles
B - Pointer to file names
C - Pointer to current top strings off files
D - Pointer to vector of heading positions
E - Pointer to keyboard string
G - Temporary store of file handle
I - Destination file handle
J - Number of current source file
O - Counter various uses
P - Number of strings on destination file
Q - Number of output strings to print
R - Number of string printed
S - Destination file length
T - Pointer to string transfer buffer
X - x coordinate for cursor move
Y - y coordinate for cursor move
Z - Number of source files.
```

10.2 Changing Text Spaces

The 'text space' is the region of memory used by the BASIC interpreter for storing the text of programs. On first entering the interpreter at #C2B2, the text space is initialised to be from #3000 upwards. However, it is possible to change the value of the text-space pointer so that text can be entered and stored in different areas of memory. It is even possible to have several different programs resident concurrently in memory, in different text spaces.

The memory location 18 (decimal) contains a pointer to the first page of the BASIC text. This value is refered to by the system in the following cases:

1. During line editing in direct mode
2. During a SAVE statement; the save parameters are ?18*256 and TOP
3. During a LOAD command; a new program is loaded to ?18*256
4. During the execution of a GOTO or GOSUB statement or a RUN statement, labels with known values being the exception.

Changing ?18 in programs permits a BASIC program in one text area to call subroutines in a BASIC program in another text area. The value of TOP will not change with this kind of use, so its use as a memory space allocator and pointer to the end of text in the line editor must be watched carefully.

10.2.1 Calling Subroutines in Different Text Spaces

The following example shows the entering of a subprogram and main program in different text spaces. First enter a subroutine in the first text space:

```
?18=#32
NEW
10 PRINT"TEXT AREA ONE"'
20 RETURN
```

Now change the value of the text-space pointer and enter a program to call this subroutine into the second text space:

```
?18=#33
NEW
10 REM CALL SUBROUTINE IN #32
20 ?18=#32
30 GOSUB 10
40 REM PROVE YOU'RE BACK
50 PRINT"TEXT AREA TWO"'
60 GOSUB 10
70 ?18=#33;REM BACK FOREVER
80 END
```

Now run the program:

```
RUN
TEXT AREA ONE
TEXT AREA TWO
TEXT AREA ONE
```

Note that switching back to the first text space by typing:

```
?18=#32
```

will not change the value of TOP. To reset TOP, type:

```
END
```

10.3 Renumbering Programs

The following routine can be used to renumber the line-numbers of a program or piece of text. The program and renumber routine must both be in memory at the same time, in different text spaces. Note that the renumber program only renumbers the line numbers; it does not renumber numbers in GOTO or GOSUB statements.

10.3.1 Renumbering

You must first enter the program to be renumbered into memory, e.g. load it from file. Then choose an area of memory for the renumber program (e.g. the next page after TOP), and set the text pointer there. Then load in the renumber program:

```
 1 REM Renumber
10 INPUT"TEXT SPACE TO RENUMBER"Z
15 Z=Z*256
20 INPUT"START AT"A,"STEP"B
30 ?18=Z/256
40 IFZ?1=255 END
50 DOZ?1=A/256;Z?2=A;A=A+B
55 Z=Z+3+LEN(Z+3)
60 UNTILZ?1=255;END
```

Then RUN the program, and reply to the prompts as follows:

```
TEXT SPACE TO RENUMBER ?#30
START AT?10
STEP?10
```

The program will switch back to the usual text space, and the renumbered program can be listed.

10.4 Appending Files

By using the operating systems load relocate, a BASIC program may be added to the end of a BASIC program. The procedure is :

```
LOAD"FRED"
PRINT&TOP'
 3284
>*LOAD"FRED+" 3282
>END
>
```

That is, load the new file to TOP-2.

10.5 Trapping Errors

The memory locations 16 and 17 contain a pointer, low byte in 16, high byte in 17, to the start of a BASIC program which is entered whenever an error occurs. In direct mode they are set to point at a program in the interpreter which reads:

```
  @=5;P.'"Error"?0;IF!1&#FFFF<>0P." at line "!1&#FFFF
0 P.';E.
```

Location 0 contains the error number and locations 1 and 2 contain the line number where the interpreter thinks it occurred. Programs intended to handle errors should store the value of !1 since it is changed whenever a return is executed. The first character in a text space that can be pointed to by ?16 and ?17 is at the start of the text space plus three, and this is the first character of the listed program. All interpreter stacks are cleared after an error but the

values of labels are not forgotten.

10.5.1 On Error GOTO

To provide a GOTO on an error it is necessary to provide a string containing the GOTO statement, and write the address of this string in locations 16 and 17. For example, to provide a jump to line 170 on an error:

```
10 A=TOP
20 $A="GOTO 170"
30 ?16=A; ?17=A/256
```

10.5.2 Calculator Program

The following program simulates a desk-top calculator; it will evaluate any expression which is typed in, and any error will cause the message "BAD SYNTAX" to be printed out:

```
10 E=TOP; $E="P.""BAD SYNTAX""';G.30"
20 ?16=E; ?17=E/256
30 @=0; DO IN.A; P.$320"="A; U.0
```

10.6 Program Chaining

Using programs in seperate text spaces, chaining of programs can be accomplished:

```
10 REM chain in TOTAL
20 *LOAD TOTAL 3200
30 *DIR
40 REM the DIR waits until DOS completion
50 ?18=#32
60 RUN
```

However, this does not have the entire effect, since the END of the program loaded has not affected TOP. With the DOS, chaining can easily be accomplished using an EXEC file:

```
10 REM chain in user file
20 INPUT"File name "$TOP
30 A=FIN$TOP;IFA=0 PRINT"NO such file";RUN
40 SHUTA;A=FOUT"XXXXXXX"
50 SPUTA,"LOAD$TOP"
60 SPUTA,"RUN"
70 SHUTA
80 *EXEC XXXXXXX
90 END
```

11.0 THE MACHINE

The Acorn BASIC will allow machine code subroutines to be executed during a program and has many abilities for actually manipulating the real computer. Thus the critical sections of programs, where speed is important, can be written in assembler, with the control and 'intelligence' written in BASIC.

When programming in BASIC it is not usually necessary to understand how the parts of the computer are working together, or where anything in particular lives in the computer. However in this chapter on the machine itself some understanding of these parts is needed.

11.1 Memory

The computer's memory can be thought of as a number of 'locations', each capable of holding a value. An Acorn computer capable of running the BASIC interpreter has at least 1024 locations used to hold the values necessary for the interpreter and operating system to function.

Somehow it must be possible to distinguish between one location and another. Houses in a town are distinguished by each having a unique address; even when the occupants of a house change, the address of the house remains the same. Similarly, each location in a computer has a unique 'address', consisting of a number. Thus the first few locations in memory have the addresses 0, 1, 2, 3 ... etc. Thus we can speak of the 'contents' of location 100 as the number stored in the location of that address.

11.2 Hexadecimal Notation

Having been brought up to count in tens it, seems natural for us to use a base of ten for our numbers, and any other system seems clumsy. We have just ten symbols, 0, 1, 2, ... 8, 9, and we can use these symbols to represent numbers as large as we please by making the value of the digit depend on its position in the number. Thus, in the number 171 the first '1' means 100, and the second '1' means 1. Moving a digit one place to the left increases its value by 10; this is why our system is called 'base ten' or 'decimal'.

It happens that base 10 is singularly unsuitable for working with computers; we choose instead base 16, or 'hexadecimal', and it will pay to spend a little time becoming familiar with this number system.

First of all, in base 16 we need 16 different symbols to represent the 16 different digits. For convenience we retain 0 to 9, and use the letters A to F to represent values of ten to fifteen:

Hexadecimal digit:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal value:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The second difference between base 16 and base 10 is the value accorded to the digit by virtue of its position. In base 16 moving a digit one place to the left multiplies its value by 16 (not 10).

Because it is not always clear whether a number is hexadecimal or decimal, hexadecimal numbers will be prefixed with a hash '#' symbol. Now look at the following examples of hexadecimal numbers:

```
#B1
```

The 'B' has the value 11*16 because it is one position to the left of the units column, and there is 1 unit; the number therefore has the decimal value 176+1 or 177.

```
#123
```

The '1' is two places to the left, so it has value 16*16*1. The '2' has the value 16*2. The '3' has the value 3. Adding these together we obtain: 256+32+3 = 291.

There is really no need to learn how to convert between hexadecimal and decimal because the BASIC interpreter can do it for you.

11.2.1 Converting Hexadecimal to Decimal

To print out the decimal value of a hexadecimal number, such as #123, type:

```
PRINT #123
```

The answer, 291, is printed out.

11.2.2 Converting Decimal to Hexadecimal

To print in hexadecimal the value of a decimal number, type:

```
PRINT &123
```

The answer, #7B, is printed out. The '&' symbol means 'print in hexadecimal'. Thus writing:

```
PRINT &#123
```

will print 123.

11.3 Binary Notation

The computer memory consists of electronic circuits that can be put into one of two different states. Such circuits are called bistables because they have two stable states, or flip/flops, for similar reasons. The two states are normally represented as 0 and 1, but they are often referred to by different terms as listed below:

State	
0	1
zero	one
low	high
clear	set
off	on

When the digits 0 and 1 are used to refer to the states of a bistable they are referred to as 'binary digits', or 'bits' for brevity.

With two bits you can represent four different states which can be listed as follows, if the bits are called A and B:

A	B
0	0
0	1
1	0
1	1

With four bits you can represent one of 16 different values, since 2x2x2x2=16, and so each hexadecimal digit can be represented by a four-bit binary number. The hexadecimal digits and their binary

equivalents are shown in the following table:

Decimal	Hexadecimal	Binary
0	0	0 0 0 0
1	1	0 0 0 1
2	2	0 0 1 0
3	3	0 0 1 1
4	4	0 1 0 0
5	5	0 1 0 1
6	6	0 1 1 0
7	7	0 1 1 1
8	8	1 0 0 0
9	9	1 0 0 1
10	A	1 0 1 0
11	B	1 0 1 1
12	C	1 1 0 0
13	D	1 1 0 1
14	E	1 1 1 0
15	F	1 1 1 1

Any decimal number can be converted into its binary representation by the simple procedure of converting each hexadecimal digit into the corresponding four bits. For example:

```
Decimal: 25
Hexadecimal: 19
                /   \
               /     \
Binary:  0001   1001
```

Thus the binary equivalent of #19 is 00011001 (or, leaving out the leading zeros, 11001).

11.4 Bytes

The size of each memory location is called a 'byte'. A byte can represent any one of 256 different values. A byte can hold a number between 0 and 255 in decimal, or from #00 to #FF in hexadecimal. Note that exactly two digits of a hex number can be held in one byte. Alternatively a byte can be interpreted as one of 256 different characters. Yet another option is for the byte to be interpreted as one of 256 different instructions for the processor to execute. We have already seen that we need exactly two hexadecimal digits to represent all the different possible values in a byte of information. It should now be clear that a byte corresponds to eight bits of information, since each hex digit requires four bits to specify it. The bits in a byte are usually numbered, for convenience, as follows:

```
7 6 5 4 3 2 1 0
0 0 0 1 1 0 0 1
```

Bit 0 is often referred to as the 'low-order bit' or 'least-significant bit', and bit 7 as the 'high-order bit' or 'most-significant bit'. Note that bit 0 corresponds to the units column, and moving a bit one place to the left in a number multiplies its value by 2.

11.4.1 Examining Memory Locations - '?'

We can now look at the contents of some memory locations in the Acorn's memory. To do this we use the '?' query operator, which means

'look in the following memory location'. The query is followed by the address of the memory location we want to examine. Thus:

```
PRINT ?#C000
```

will look at the location whose address is #C000, and print out its value, which will be 60 (the start of the interpreter). Try looking at the contents of other memory locations; they will all contain numbers between 0 and 255.

It is often convenient to look at several memory locations in a row. For example, to list the contents of the 32 memory locations from #80 upwards, type:

```
FOR N=0 TO 31; PRINT N?#80; NEXT N
```

The value of N is added to #80 to give the address of the location whose contents are printed out; this is repeated for each value of N from 0 to 31. Note that N?#80 is identical to ?(N+#80).

11.4.2 Changing Memory Locations

A word of caution: although it is quite safe to look at any memory location in the computer, care should be exercised when changing memory locations. The examples given here specify locations that are not used by the BASIC interpreter and operating system; if you change other locations, be sure you know what you are doing or you may lose the stored text, or have to reset the computer.

First print the contents of #80. The value there will be whatever was in the memory when you entered BASIC, because BASIC does not use this location. To change the contents of this location to 7, type:

```
?#80=7
```

To verify the change, type:

```
PRINT ?#80
```

Try setting the contents to other numbers. What happens if you try to set the contents of the location to a number greater than 255?

11.5 Numbers Representing Characters

If locations can only hold numbers between 0 and 255, how is text stored in the computer's memory? The answer is that each number is used to represent a different character, and so text is simply a sequence of numbers in successive memory locations. There is no danger in representing both numbers and characters in the same way because the context will always make it clear how they should be interpreted.

To find the number corresponding to a character the CH function can be used. Type:

```
PRINT CH"A"
```

and the number 65 will be printed out. The character "A" is represented internally by the number 65. Try repeating this for B, C, D, E... etc. You will notice that there is a certain regularity. Try:

```
PRINT CH"0"
```

and repeat for 1, 2, 3, 4 ... etc.

11.6 Printing a Character

The operating system contains routines for the basic operations of printing a character to the VDU, and reading a character from the keyboard, and these routines can be called from assembler programs or directly form the BASIC interpreter using the LINK statement. The addresses of these routines are standardised throughout the Acorn

range of software, and are as follows:

Name	Address	Function
OSWRCH	#FFF4	Puts character in accumulator to output (VDU)
OSRDCH	#FFE3	Read from input (keyboard) into accumulator

In each case all the other registers are preserved. The names of these routines are acronyms for 'Operating System WRite CHaracter' and 'Operating System ReaD CHaracter' respectively. All output generated by BASIC's PRINT statement goes to OSWRCH, and all input to BASIC's INPUT statement comes from OSRDCH.

11.7 LINK Statement

The LINK statement allows the interpreter to execute a machine code subroutine. The processor's A, X and Y registers are initialised to the least significant bytes of the variables A, X and Y, but no result is returned from the machine code subroutine. The subroutine can end with an RTS instruction or can execute a BRK instruction to cause an error and stop the interpreter. For example, to print character #21 (a '!')

```
A=#21; LINK#FFF4
```

11.8 Controlling the Outside World

A Versatile Interface Adapter, or VIA, can be added to the computer on an Acorn V.I.B. to provide two eight-bit parallel I/O ports, together with four control lines, a pair of interval timers for providing real time interrupts, and a serial to parallel or parallel to serial shift register. Both eight-bit ports and the control lines are connected to side B of the connector. The operating system can use the VIA as a Centronics printer interface.

Each of the 16 lines can be individually programmed to act as either an input or an output. The two additional control lines per port can be used to control handshaking of data via the port, and to provide interrupts. Several of the lines can be controlled directly from the interval timers for generating programmable frequency square waves or for counting externally generated pulses. Only the most basic use of the VIA will be explained here; for more of its functions consult the VIA data sheet (available from Acorn Computers) and the V.I.B. technical manual. The VIA registers occur in the following memory addresses:

Register	Address	Name
Data Register B	#C00	DB
Data Register A	#C01	DA
Data Direction Register B	#C02	DDRB
Data Direction Register A	#C03	DDRA
Timer 1 low counter, latch	#C04	T1CL
Timer 1 high counter	#C05	T1CH
Timer 1 low latch	#C06	T1LL
Timer 1 high latch	#C07	T1LH
Timer 2 low counter, latch	#C08	T2CL
Timer 2 high counter	#C09	T2CH
Shift Register	#C0A	SR
Auxiliary Control Register	#C0B	ACR
Peripheral Control Register	#C0C	PCR
Interrupt Flag Register	#C0D	IFR
Interrupt Enable Register	#C0E	IER
Data Register A	#C0F	DA

On reset all registers of the VIA are reset to 0 (except T1, T2 and

SR). This places all peripheral lines in the input state, disables the timers, shift register, etc. and disables interrupts.

11.8.1 Printer Interface

Port A has a high current output buffer leading to a 26-way printer connector to produce a Centronics-type parallel interface, capable of driving most parallel-interface printers with the software already in the operating system. Printer output is enabled by printing a CTRL-B character, and disabled by printing a CTRL-C character; see Section 10.1.

11.8.2 Parallel Input/Output

To use the ports in a simple I/O mode with no handshake, the Data Direction Register associated with each I/O register must be programmed. A byte is written to each of the DDR's to specify which lines are to be inputs and outputs. A zero in a DDR bit causes the corresponding bit in the I/O register to act as an input, while a one causes the line to act as an output. Writing to the data register (DA or DB) will affect only the bits which have been programmed as outputs, while reading from the data register will produce a byte composed of the current status of both input and output lines.

In order to use the printer port for handshaked I/O, the printer software driver must be removed from the output stream by setting the vector WRCVEC (address #208) to WRCVEC+3; e.g.:

```
!#208=!#208+3
```

It is fatal to execute this statement more than once !!!. For ordinary, non-handshake, I/O the vector need not be changed, but you should avoid turning the printer on.

11.8.3 Writing to a Port

The following program illustrates how to write to one of the VIA's output ports from a BASIC program:

```
20 ?#C0C=0
30 ?#C02=#FF
40 INPUT J
50 ?#C00=J
60 GOTO 40
```

Description of Program:

```
20        Remove all handshaking
30        Program all lines as outputs
50        Output bytes.
```

11.8.4 Timing to 1 Microsecond

The following program demonstrates how the VIA's timer 2 can be used to measure the execution-time of different BASIC statements to the nearest microsecond. The same method could be used to time events signalled by an input to one of the ports:

```
10 REM Microsecond Timer
20 B=#C08
30 !B=65535
40 X=Y
50 B?3=32;Q=!B&#FFFF
60 PRINT 65535-Q-1740 " microseconds"'
70 END
```

Description of Program:

```
20        Point to timer 2 in VIA
30        Set timer to maximum count
40        Line to be timed; if absent, time should be 0
50        Turn off timer; read current count
60        Print time, allowing for time taken to read count.
```

11.8.5 Music Program

The following program demonstrates how the VIA's timer 1 can be used to generate programmable frequency tones. PB7 (pin 17) of the VIA generates a programmable frequency square wave, which can be connected directly to a 16 ohm loudspeaker. The program uses a BGET 0 statement to read a key from the keyboard; if you have a cassette operating system, this should be replaced by:

```
LINK#FFE3; N=?#E21
```

Also, the cassette operating system cannot cope with the way that the program uses files:

```
  10 B=#C00
  20 B?2=255
  30 B?11=#C0
  40 B!4=0
  50 T=TOP;A=4545;REM COUNT FOR 220 HZ
  55 S=10594;L=8*12*2-2
  60 N=A*2;!T=N;REM STARTING NOTE
  70 FORF=2TOL STEP2;N=N*10000/S;T!F=N;N.
  75 E=T+L+6;!E=#04030201;E!4=#100C0806
  80 Q=E+8;$Q="ZSXCFVGBNJMK ";P=Q+LENQ+1
  85 $P="A A#B C C#D D#E F F#G G#   "
  90 S=P+LENP+1;REM STAVE
 100 !S=0;REM TUNE END
 102 A=0;REM PLAY IT OFF
 105 M=0;REM INSERTION
 110 O=CH"0";REM LOWEST OCTAVE
 115 D=CH"0";REM NOTE TIME
 118 GOS.f
 120 DO?#407=O;?#40E=D
 140 P.$21;N=BGET0;P.$6;IFN=13 GOS.p;U.0
 141 IFN=CH"Q" D=48;U.0
 142 IFN=CH"W" D=49;U.0
 143 IFN=CH"E" D=50;U.0
 144 IFN=CH"R" D=51;U.0
 145 IFN=CH"T" D=52;U.0
 146 IFN=CH"Y" D=53;U.0
 147 IFN=CH"U" D=54;U.0
 148 IFN=CH"I" D=55;U.0
 150 IFN>=CH"0"ANDN<=CH"7" O=N;U.0
 153 IFN=CH"["A=1
 155 IFN=CH"]"A=0
 160 IFN=CH"-"M=M-3;P.$8$8$8$8$8;U.0
 161 IFN=62GOS.d;GOS.f;C=FO.$C;F.F=0TOM+4;BP.C,S?F;N.;SH.C;U.0
 163 IFN=60GOS.d;C=FIN$C;F.F=0TOEXTC;S?F=BGETC;N.;SH.C;GOS.f;U.0
 165 IFN=10M=0;GOS.f;U.0
 170 IFN=#7FM=M-3;S!M=0;P.$127$127$127$127$127;U.0
 180 X=-1;DOX=X+1;IFQ?X=13 UNTIL1;UNTIL0
 190 UNTILQ?X=N;U=S!M;S?M=O;S?(M+1)=X;S?(M+2)=D
 200 IFA GOS.n
 205 IFA=0 GOS.m
```

```
 210 M=M+3;IFU=0 S!M=0
 220 UNTILO
1000 pM=0;P.$30'';IF!S=0 RETURN
1010 DOGOS.n;M=M+3;UNTILM!S=0;RETURN
2000 nW=0;IFS?(M+1)<>12W=T!((S?M&15*12+S?(M+1))*2)&#FFFF
2010 B!6=W;FORX=0TOE?(S?(M+2)&15)*300;N.
2020 B!6=0;FORX=X TOOSTEP-15;N.
2030 mP.$S?M$S?(M+2)" "$P?(S?(M+1)*2)$P?(S?(M+1)*2+1)
2040 RETURN
4000 dC=S+M+4;PRINT''$N;INPUT" NAME "$C;RETURN
4010 fP.$12"OCTAVE   NOTE "''
4020 eM=0;IFS!M=0 RETURN
4030 DOGOS.m;M=M+3;UNTILS!M=0;RETURN
```

Description of Program:

10-70	Create the frequency values for the 8 octaves
75-118	Set up variables and screen before entering main loop
120	Poke octave and duration to screen
140	Get key from keyboard
120-220	Main loop of program
1000-1010	p: Play out tune
2000-2020	n: Play current note
2030	m: Print current note
4000	d: Get file name
4010	f: Print out tune.

Variables:

A - Base note frequency/ play mode truth value
B - Pointer to VIA
C - Used for free space string
D - Current note length
E - Vector of note lengths
F - Local variable
L - Number of bytes required for 8 octaves
M - Position of end of tune vector
N - Input key
O - Current Octave
P - String of note characters for display
Q - String of input keys for notes
S - Stave vector
T - Vector of values for each note
W - Note value to be played
X - Local search variable.

The keys 0 to 7 program the octave that the notes are in. Q, W, E, R, T, Y, U, I program the note length, in order of increasing duration. Keys Z, S, X, C, F, V, G, B, N, J, M, K represent the notes A, A#, B, C, C#, D, D#, E, F, F#, G, G# respectively in the current octave. The space bar introduces a rest of the current note duration. The '-' key moves the cursor back over the tune so that notes can be edited. The DELETE key deletes the note. The RETURN key plays out the current tune, and the LINE FEED key writes out the current tune. The '>' saves and the '<' loads a tune to or from a specified file on disk. '[' turns echo mode on, ']' turns echo mode off. Notes are displayed as OCTAVE LENGTH NOTE.

Since the program disables the screen, you may need to type an ACK to enable output. If you press ESC during a tune, the noise can be stopped by either RUNning the program again, or typing B!6=0.

11.9 Controlling the Screen

The Acorn teletext VDU has a 6845 screen controller on board. BASIC can be used to reprogram the screen controller to provide different screen formats, cursor shape control, light pen control, and screen on/off. A brief description of the 6845 is given here: a comprehensive description is given in Acorn Computer's 6845 data sheet.

The 6845 is programmed through two registers: one is the address register, at location #800, and the other is the data register at location #801. The data in the address register determines which of the 18 internal registers the data register represents. The registers are:

Register	Function	Initial Value for Teletext Card
#00	Horizontal Total	#3F
#01	Horizontal Displayed	#28
#02	H Sync position	#33
#03	Sync Width	#44
#04	Vertical Total	#1E
#05	Vertical Total Adjust	#02
#06	Vertical Displayed	#19
#07	V Sync position	#1B
#08	Interlace and Skew	#03
#09	No. of scan lines	#12
#0A	Cursor start and flash	#72
#0B	Cursor end	#13
#0C	Screen start address H	
#0D	Screen start address L	
#0E	Cursor address H	
#0F	Cursor address L	
#10	Light pen address H	
#11	Light pen address L	

To program register YY with contents XX from BASIC, execute

```
!#800=#10XXYY
```

To reset all 6845 registers output a form feed (character 12).

11.8.1 Reprogramming the Screen Format

Note that the operating system software VDU driver will still assume the original number of rows and columns.

Example, a screen with only 16 rows of 32 columns

```
!#800=#102001;!#800=#101006
```

11.9.2 Reprogramming the Cursor

The number of video lines and the flashing rate of the cursor may be changed. Examples:

```
Cursor off:           !#800=#101F0A
Flashing underline:   !#800=#10720A
Static underline:     !#800=#10120A
Static block:         !#800=#10000A
Flashing block:       !#800=#10600A
Fast flashing block:  !#800=#10400A
```

11.9.3 Reading the Light Pen

The last known position of the light pen on the screen, in terms of the row and column, may be found:

```
10 A=TOP;B=A+4;!A=0;B=0;C=#800
20 ?C=#10;A?1=C?1
30 ?C=#11;?A=C?1
40 ?C=#C;B?1=C?1
50 ?C=#D;?B=C?1
60 A=A-B
70 B=A/40;REM row
80 A=A%40;REM column
```

11.9.4 Screen On/Off

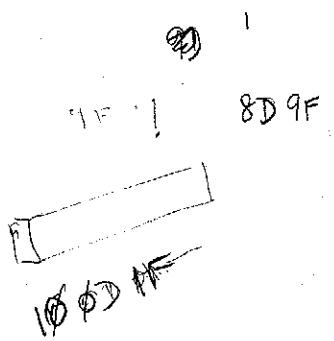
The screen display can be turned off, characters written to the VDU, and the screen display turned back on again:

```
!#800=#100006;REM screen off
!#800=#101906;REM screen on
```

11.10 Using Teletext Characters

The teletext standard VDU in the standard Systems Two and Three produces coloured alphanumerics and graphics characters as well as some other features. The options are as follows:-

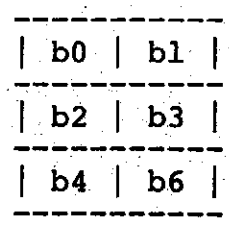
Decimal	Hex	Function
129	#81	Red alphanumerics
130	#82	Green alphanumerics
131	#83	Yellow alphanumerics
132	#84	Blue alphanumerics
133	#85	Magenta alphanumerics
134	#86	Cyan alphanumerics
135	#87	White alphanumerics
136	#88	Flash
137	#89	Steady
138	#8A	End box
139	#8B	Start box
140	#8C	Normal height
141	#8D	Double height
145	#91	Red graphics
146	#92	Green graphics
147	#93	Yellow graphics
148	#94	Blue graphics
149	#95	Magenta graphics
150	#96	Cyan graphics
151	#97	White graphics
152	#98	Conceal display
153	#99	Contiguous graphics
154	#9A	Separated graphics
156	#9C	Black background
157	#9D	New background
158	#9E	Hold graphics
159	#9F	Release graphics



These characters only affect the characters on the right of them on a line, and provide the following options:-

Alpha (colour) causes following characters on the line to be alphanumeric characters in the colour specified

Graphics (colour) causes following characters on the line to be graphics characters in the colour specified. In graphics mode, each character space is divided into 6 cells, and each cell is 'on' if the corresponding bit in the character code is set. The assignments are:



Bit 5 in the code must always be set for a graphics character. If bit 5 is clear, then the upper case alphabetic characters are available.

Flash causes the following characters on the line to flash

Steady causes the following characters on the line not to flash

Start & End box options for using the VDU board to superimpose text onto a normal TV picture

Double height duplicating a line with a double height character on it will cause a line twice as tall as usual

Normal height allows single height characters to appear on the upper line of a double height pair

Conceal display following characters are not displayed unless the VDU board recieves a REVEAL command along its serial interface

Separated graphics causes the cells in graphics mode to be separated

Contiguous graphics causes the cells in graphics mode to be in their normal (connected) mode

New background sets the background to the colour of the last colour specifying character

Black background resets the background to black

Hold graphics normally, control characters appear as spaces on the screen. This option causes control characters to appear as the last entered graphics character

Release graphics causes control characters to be displayed as a space.

The VDU assumes a setting of alpha white, steady, end box, normal height, contiguous graphics, black background, and release graphics, at the start of each line

For example, the following program writes out 'Acorn Computers' in yellow and green flashing double height letters:

```
 10 PRINT$136$141
 20 GOSUBa
 30 PRINT$136$141
 40 GOSUBa
 50 END
100 aPRINT$130"Acorn"$131"Computers"'
110 RETURN
```

11.10.1 Plotting with Teletext Characters

When the teletext screen is operating in its non-scrolled position (just after a form feed), we can use BASIC to draw lines in the graphics characters. The VDU should be set up with the following routine:

```
 10 PRINT $12
 20 FORZ=0TO24*40STEP40
 30 Z?#400=135;NEXT
 40 S=TOP;!S=#8040201;S!4=#4010
```

Then this subroutine calculates values for the particular bit specified by the (x,y) coordinates, the origin being at the bottom left:

```
100 pP=X/2+(74-Y)/3*40+#401;C=S?(X&1+(74-Y)%3*2);RETURN
```

We may change the bit in one of three ways:

```
?P=?P|C          switch bit on
?P=?P:C          change state of bit
?P=?P&(C:-1)     switch bit off
```

As an example, the following program draws a random lissajou pattern in a random colour:

```
 10 S=TOP;!S=#8040201;S!4=#4010
 20 C=ABSRND%7+#91;V=ABSRND%11+5
 30 W=ABSRND%11+5
 40 PRINT$12;FORZ=0TO24*40STEP40
 50 Z?#400=C;NEXT
 60 R=34000;T=10000
 70 Q=R;U=0;D=1000;E=D
 80 DOR=R+T/W;T=T-R/W;Q=Q+U/V;U=U-Q/V
 90 X=R/D+36;Y=Q/E+39;GOSUBp;?P=?P|C
100 UNTIL0
200 pP=X/2+(74-Y)/3*40+#401;C=S?(X&1+Y%3*2);RETURN
```

Description of program:

10	Initialise cell vector
20-30	Choose random colour and random frequencies for the two waves generating the lissajou pattern
40-50	Set screen to graphics colour
60-70	Initial values for successive approximation sine and cosine generators
80	One step in each of the sine and cosine generators
90	Plot a point in the pattern
200	Point subroutine.

11.11 Reading the keyboard

The keyboard on a System 2 or 3 is available at address #E21 as ASCII data. If the byte is negative, no key is currently pressed. Examples of this:

```
10 DOPRINT"Acorn ";UNTIL?#E21=CH" "

20 DOPRINT"Computer ";UNTIL?#E21<128
```

12.0 MORE SPACE AND MORE SPEED

This chapter shows how to abbreviate programs so that they will fit into a smaller amount of memory, and how to write programs so that they will run as fast as possible.

12.1 Abbreviating BASIC Programs

Most versions of BASIC demand a large amount of redundancy. For example, the command PRINT must usually be specified in full, even though there are no other statements beginning with PR. In Acorn BASIC it is possible to shorten many of the statement and function names and omit many unnecessary parts of the syntax, in order to save memory and increase execution speed. The examples in this manual have generally avoided such abbreviations because they make the resulting program harder to read and understand, but a saving of up to 30% in memory space can be obtained by abbreviating programs as described in the following sections.

12.1.1 Statements and Functions

All statement and function names can be abbreviated to the shortest sequence of characters needed to distinguish the name, followed by a full stop. This does not speed the interpretation up noticeably, but will save space. The following abbreviations are possible:

Name	Abbreviation
ABS	A.
AND	A.
BGET	B.
BPUT	B.
CH	
COUNT	C.
DO	
END	E.
EXT	E.
FIN	F.
FOR	F.
FOUT	FO.
GET	G.
GOSUB	GOS.
GOTO	G.
IF	
INPUT	IN.
LEN	L.
LET	L.
LINK	LI.
LIST	L.
LOAD	LO.
NEW	N.
NEXT	N.
OR	
PRINT	P.
PTR	

```
PUT
REM
RETURN      R.
RND         R.
RUN
SAVE        SA.
SGET        S.
SHUT        SH.
SPUT        SP.
STEP        S.
THEN        T.
TO
TOP         T.
UNTIL       U.
```

12.1.2 Spaces

Spaces are largely irrelevant to the operation of the BASIC interpreter, and they are ignored when encountered in a program. Their only effect is to cause a 13 microsecond delay per space in execution. There is one place where a space is necessary to avoid an ambiguity as in the following example:

```
FOR A=B TO C
```

where the space after B is compulsory to make it clear that B is not the first letter of a function name.

12.1.3 LET

Some BASICs demand that every assignment statement begin with the word LET; e.g.:

```
LET A=B
```

In Acorn BASIC the LET statement may be omitted, with a decrease in execution time.

12.1.4 THEN

The word THEN in the second part of an IF statement may be omitted. Example:

```
IF A=B C=D
```

is perfectly legal. However, note that if the second statement begins with a T, or a '?' or '!' unary operator, some delimiter is necessary:

```
IF A=B THEN T=Q
```

Alternatively a statement delimiter ';' can be used as the delimiter:

```
IF A=B; T=Q
```

Using a ';' is slower than a THEN or T..

12.1.5 Brackets

Brackets enclosing a function argument, or an array identifier, are unnecessary and may be omitted when the argument, or array subscript, is a single variable or constant.

For example ABS(RND) may be written ABSRND, but ABS(B+2) cannot be abbreviated.

12.1.6 Commas

The commas separating elements in a PRINT statement can be omitted when there is no ambiguity. Example:

```
PRINT A,B,C,"RESULT",J
```

may be shortened to:

```
PRINTA B C"RESULT"J
```

Note that the comma in:

```
PRINT &A,&B
```

is, however, necessary to distinguish the numbers from the single number (A&B) printed in hex.

12.1.7 Multi-Statement Lines

Each text line uses one byte per character on the line, plus two bytes for the line number and a one-byte terminator character; thus writing several statements on one line saves two bytes per statement. Note that there are two occasions where this cannot be done:

1. After an IF statement, because the statements on the line following the IF statement would be skipped if the condition turned out false

2. Where the line number is referred to in a GOTO or GOSUB statement.

12.1.8 Control Variable in NEXT

The FOR...NEXT control variable may be omitted from the NEXT statement; the control variable will be assumed to be the one specified in the most recently activated FOR statement.

12.2 Maximising Execution Speed

Acorn BASIC is one of the fastest BASIC interpreters available, and all of its facilities have been carefully optimised for speed so that calculations will be performed as quickly as possible.

To obtain the best possible speed from a program the following hints should be borne in mind; but note that many of these suggestions reduce the legibility of the program, and so should only be used where speed is critical:

1. Use the FOR ... NEXT loop or DO ... UNTIL loop in preference to an IF statement and a GOTO

2. Use labels, rather than line numbers, in GOTO and GOSUB statements

3. Avoid the use of constants specified in the body of programs; instead use variables which have been set to the correct value at the start of the program. For example, replace:

```
A=A*1000
```

by:

```
T=1000
.
.
A=A*T
```

4. Write statements in-line, rather than in subroutines, when the subroutines are only called once, or when the subroutine is only two or three lines

5. If a calculation is performed every time around a loop, make sure that the constant part of the calculation is performed only once outside the loop. Example:

```
FOR J=1 TO 10
```

```
FOR K=1 TO 10
PRINT J*J+K
NEXT K
NEXT J
```

could be written as:

```
FOR J=1 TO 10
Q=J*J
FOR K=1 TO 10
PRINT Q+K
NEXT K
NEXT J
```

6. Where several nested FOR ... NEXT loops are being executed, and the order in which they are performed is not important, arrange them so that the one executed the greatest number of times is at the centre. Example:

```
FOR J=1 TO 2
FOR K=1 TO 1000
.
.
NEXT K
NEXT J
```

is faster than:

```
FOR K=1 TO 1000
FOR J=1 TO 2
.
.
NEXT J
NEXT K
```

because in the second case the overhead for setting up the inner loop is performed 1000 times, whereas in the first example it is only performed twice

7. Choose the FOR ... NEXT loop parameters so as to minimise calculations inside the loop. Example:

```
FOR N=0 TO 9
PRINT A!(N*4)
NEXT N
```

could be rewritten as the faster:

```
FOR N=0 TO 36 STEP 4
PRINT A!N
NEXT N
```

8. Use word operations rather than byte operations where possible. For example, to clear a section of memory to 0 it is faster to execute:

```
FOR N=#8000 TO #9000 STEP 4; !N=0; NEXT N
```

than the following:

```
FOR N=#8000 TO #9000; ?N=0; NEXT N
```

9. The IF statement containing several conditions linked by the AND connective, as, for example:

```
IF A=2 AND B=2 AND C=2 THEN
```

will evaluate all the conditions even when the earlier ones are false. Rewriting the statement as:

```
IF A=2 IF B=2 IF C=2 THEN
```

avoids this, and so gives faster execution.

13.0 WHAT TO DO IF BAFFLED

This section is the section to read if all else fails; you have studied your program, and the rest of the manual, and you still cannot see anything wrong, but the program refuses to work.

There are two types of programming errors: errors of syntax and errors of logic.

13.1 Syntax Errors

Syntax errors are caused by writing something in the program that is not legal, and that is therefore not understood by the BASIC interpreter. Usually this will give rise to an error, and reading the description of that error code in Chapter 18 should make the mistake obvious.

Typical causes of syntax errors are:

1. Mistyping a digit '0' for a letter 'O', and vice-versa. E.g.
```
F0R N=1 T0 3
```

2. Mistyping a digit '1' for a letter 'I', and vice-versa. E.g.
```
1F J=2 PR1NT "TWO"
```

3. Forgetting to enclose an expression in brackets when it is used as a parameter in a statement. E.g.
```
ABS X+32
```

In some cases a syntax error is interpreted as legal by BASIC, but with a different meaning from that intended by the programmer, and no error message will be given. E.g.
```
IFA=O THEN PRINT "ZERO"
```
was intended to test A to see if it was zero, but in fact tests for equality with the variable O.

13.2 Logical Errors

Errors of logic arise when a program is perfectly legal, but does not do what the programmer intended, probably because the programmer misinterpreted something in this manual, or because a situation arose that was not foreseen by the programmer. Common logical errors are:

1. Unitialised variables. Remember that the variables A-Z initially contain unpredictable values, and so all the variables used in a program should appear on the left-hand side of an assignment statement, in an INPUT statement, or as the control variable in a FOR ... NEXT loop, at least once in the program. These are the only places where the values of variables are changed

2. The same variable is used for two purposes. It is very easy to forget that a variable has been used for one purpose at one point in the program, and to use it for another purpose when it was intended to save the variable's original value. It is good practice to keep a list of the variables used in a program, similar to the list given after the application programs in this manual, to avoid this error

3. Assigning to a string variable and exceeding the allocated space. Care should be taken that enough space has been allocated to string

variables to receive the strings allocated to them

4. Assigning outside the bounds of a vector. Assigning to vector elements above the range allocated will overwrite other vectors, or strings

5. Mistaking the priority of the '!' and '?' operators. A!4*5 is not equivalent to 5*A!4; the first is (A!4)*5 and the second is (5*A)!4.

13.3 Suspected Hardware Faults

This section deals with faults on an Acorn Computer which is substantially working, but which exhibits faults which are thought to be due to hardware faults rather than programming faults. Hardware fault-finding details are provided in the various technical manuals; this section describes only those hardware problems that can be tested by running software diagnostics.

13.3.1 RAM Memory Faults

The following BASIC program can be used to verify that the computer's memory is working correctly:

```
 1 REM MEMORY TEST
10 INPUT"FROM"A,"  TO"B
20 DO ?11=0; R=!8
30 FOR N=A TO B STEP4;!N=RND; NEXT N
35 ?11=0; !8=R
40 FOR N=A TO B STEP4
50 IF !N<>RND PRINT'"FAIL AT "&N'
60 NEXT N
70 P." OK"; UNTIL 0
```

The first address entered should be the lowest address to be tested, and the second address entered should be four less than the highest address to be tested. For example, to test 4K memory from #2000 to #2FFF enter:

```
>RUN
FROM?#2000
  TO?#2FFF
```

The program stores random numbers in the memory locations, and then re-seeds the random-number generator and checks each location is correct.

13.3.2 ROM Memory Faults

The BASIC interpreter in 4K ROM could be at fault although as all ROMs are tested before despatch it is very unlikely that a fault can be present. However, if a user suspects a ROM fault the following program should be entered and run; the program obtains a 'signature' for the whole ROM, this signature consisting of a four-digit hexadecimal number:

```
 1 REM CRC Signature
10 INPUT "PROM ADDRESS", P
20 C=0;Z=#FFFF;Y=#2D
30 FOR Q=0 TO #FFF
35 A=P?Q
40 FOR B=1 TO 8
60 C=C*2+A&1;A=A/2;IFC>Z C=C:Y;C=C&Z
80 NEXT B; NEXT Q
110 PRINT "SIGNATURE IS" &C'
```

```
  120 END
```

Sample run:

```
>RUN
PROM ADDRESS?#C000
SIGNATURE IS 193F
>
```

If this signature is obtained the ROM is correct, (to say nothing of it having run the program).

14.0 EXTENDING THE BASIC

At the time of writing, Acorn have two extension floating point programs, and the ONLI extension. The addition of one of these extensions does not alter any of the functions of the original BASIC interpreter.

14.1 Floating-Point Extension to BASIC

The floating-point extension adds 27 new variables, %@ and %A to %Z, and the following special statements and functions to the existing integer BASIC. Due to Acorn's flexibility there are several versions of the floating-point package, and only the minimum set of operations is given below:

Floating-Point Statements

FIF, FINPUT, FPRINT, FPUT, FUNTIL, STR

Floating-Point Functions

ABS, ACS, ASN, ATN, COS, DEG, EXP, FGET, FLT, HTN, LOG, PI, RAD, SGN, SIN, SQR, TAN, VAL

Floating-Point Operators

!, %, ^.

The two versions currently available differ in providing either graphics PLOT, DRAW and MOVE for the teletext VDU, or floating point arrays and degree to radian conversions. The version with arrays requires two locations to be set up at the start of the program:

```
10 ?#23=TOP;?#24=TOP/256
```

in order to allocate space for the arrays after the program.

All the extension statements and functions, except FLT, and all the extension operators expect floating-point expressions as their arguments.

Whenever the context demands a floating-point expression, or factor, all calculations are performed in floating-point arithmetic and all integer functions and variables are automatically floated. An integer expression may be explicitly floated with the FLT function, which takes an integer argument. For example:

```
FPRINT FLT(2/3)
```

will print 0.0 because the division is performed in integer arithmetic and then floated. Therefore:

```
FPRINT FLT(PI)
```

will convert PI to an integer, and then float it, printing 3.00000000.

When the context demands an integer expression, or factor, all calculations are performed in integer arithmetic, and floating-point functions will be automatically converted to integers. For example:

```
PRINT SQR(10)
```

will print 3. Floating-point expressions used in an integer context must be fixed by the '%' operator. For example:

```
PRINT %(3/2+1/2)
```

will print 2, since the expression is evaluated using floating-point arithmetic and then fixed, whereas:

```
PRINT 3/2+1/2
```

will print 1, since in each case integer division is used.

Since there are both integer and floating-point versions of the ABS function, the context will determine how its argument is evaluated. For example:

```
PRINT ABS(2/3+1/3)
```

will print 0, whereas:

```
FPRINT ABS(2/3+1/3)
```

will print 1.00000000. The floating-point function may be obtained in an integer context by prefixing it with the '%' operator. Thus:

```
PRINT %ABS(2/3+1/3)
```

will print 1.

14.1.2 Floating-Point Statements

FIF Floating-point IF

Same syntax as IF, but connectives as AND and OR are not allowed. Example:

```
FIF %A < %B FPRINT %A " IS LOWER THAN " %B
```

FINPUT Floating-point Input **FIN.**

Exactly as INPUT, but takes a floating-point variable or array element, and does not allow strings to be input. Example:

```
FINPUT"Your weight "%A
```

FPRINT Floating-point Print **FP.**

Exactly as PRINT except that no $ expressions are allowed, and all expressions are treated as floating-point expressions. Floating-point numbers are printed out right justified in a field size determined by the value of @. Example:

```
FPRINT"You are "%H" metres tall"''
```

FPUT Floating-point Put

FPUT writes the 5 bytes representing a floating-point number to the sequential file whose handle is specified by its argument. Example:

```
FPUTA,2^32+1
```

FUNTIL Floating-point Until **FU.**

As UNTIL, but the connectives AND and OR are not allowed. Matches with DO statement. Example:

```
DO%A=%A+.1;FUNTIL%A>2
```

STR Convert to String

STR converts a floating-point expression into a string of characters.

It takes two arguments, the floating point expression, and an integer expression which is evaluated to give the address where the string is to be stored. Example:

```
      STR PI, TOP
      PRINT $TOP'
3.14159265
```

14.1.3 Floating-Point Functions

ABS Absolute value

Returns the absolute value of a floating-point argument. Example:

```
      FPRINT ABS -2.2
2.20000000
```

ACS Arc cosine

Returns arc cosine of argument, in radians. Example:

```
      FPRINT ACS 1
     0.0
```

ASN Arc sine

Returns arc sine of argument, in radians. Example:

```
      FPRINT ASN 1
1.57079633
```

ATN Arc tangent

Returns arc tangent of argument, in radians. Example:

```
      FPRINT ATN 1
7.85398163E-1
```

COS Cosine **C.**

Returns cosine of angle in radians. Example:

```
      FPRINT COS 1
5.40302306E-1
```

EXP Exponent **E.**

Returns exponent (i.e. e^<factor>). Example:

```
      FPRINT EXP 1
2.71828183
```

FGET Floating-point GET

Same as GET, but reads five bytes from a serial file and returns a floating-point number.

FLT Float **F.**

Takes an integer argument and converts it to a floating-point number. Example:

```
      FPRINT FLT(4/3)
1.00000000
```

HTN Hyperbolic tangent **H.**

Returns the hyperbolic tangent of its argument. Example:

```
      FPRINT HTN 1
7.61594156E-1
```

LOG Natural logarithm **L.**

Returns the natural logarithm of its argument. Example:

```
      FPRINT LOG 1
0.0
```

PI

Returns the constant pi. Example:

```
      FPRINT PI
3.14159265
```

SIN Sine

Returns sine of an angle in radians. Example:

```
      FPRINT SIN PI
0.0
```

SQR Square root

Returns square root of argument. Example:

```
      FPRINT SQR 2
1.41421356
```

TAN Tangent **T.**

Returns tangent of angle in radians. Example:

```
      FPRINT TAN PI
     0.0
```

VAL Value of String **V.**

Returns a number representing the string converted to a number. If no number is present, zero will be returned. VAL will read up to the first illegal character, and cannot cause an error. Example:

```
      FPRINT VAL "2.2#"
2.20000000
```

14.1.4 Floating-Point Operators

! Floating-point indirection **{pling}**

The floating-point indirection operation makes it possible to set up vectors of floating-point numbers. The operator returns the five bytes at the address specified by its operand. For example, to set up a floating-point vector of three elements:

```
      A=TOP; %!A=PI; %!(A+5)=3; %!(A+10)=4
```

% Convert to integer **{percent}**

The unary % operator converts its floating-point argument to an integer. Example:

```
      PRINT %(3/2+1/2)
       2
```

^ Raise to power **{up arrow}**

Binary operator which raises its left-hand argument to the power of

its right-hand argument; both arguments must be floating-point factors. Example:

```
      FPRINT 2^32
4.29496728E9>
```

14.1.5 Floating-Point Variables

The floating-point variables %@ and %A to %Z are stored from #2800 onwards, five bytes per variable, thus taking a total of 135 bytes. Thus, for example, a floating-point vector:

```
      %!#2800
```

may be set up whose elements:

```
      %!(#2800+0), %!(#2800+5), %!(#2800+10) ...
```

will correspond to the variables:

%@, %A, %B ... etc.

For example, the floating-point variables may be initialised to zero by executing:

```
      FOR J=0 TO 26*5 STEP 5
      %!(#2800+J)=0
      NEXT J
```

14.2 ONLI extension to BASIC

ONLI is a 2K extension to BASIC with extra operations added based on the ONLI system, developed by Dr. Stephen Lea of the Cambridge University Department of Experimental Psychology. The extra commands are designed for control of experiments, and use the Acorn Laboratory Interface to provide a real-time clock and isolated connections to the experimental equipment. Its principal feature is that it allows the programmer to specify a large number of states, such as a timed interval, or waiting for an event, which, once set up, will behave independently. For example separate outcomes can be programmed for different inputs, and the arrival of one event does not stop the system from looking for other events, unless specifically told to.

The system allows the specification of up to eight independently running clocks, accurate to 10mS, and the specification of outcomes for the switching of up to sixteen external events. Up to 32 lines can be controlled each as inputs or outputs. A special function allows the user to set a pattern of outputs, for a time measured in units of 200 microseconds, and wait for one of a set of specified inputs, returning an accurate time in units of 200 microseconds. The standard PLOT, MOVE and DRAW commands for the teletext VDU are also present.

15.0 BASIC STATEMENTS, FUNCTIONS, AND COMMANDS

All the BASIC statements, functions, and commands are listed in the following pages in alphabetical order. Following each name is, where applicable, an explanation of the name and the shortest abbreviation of that name. The following symbols will be used; these are defined more fully in Chapter 17:

<variable> - one of the variables A to Z, or @.

<factor> - a variable, a constant, a function, an array, an indirection, or an expression in brackets, any of which may optionally be preceded by a + or - sign; e.g.:

A, -1234, ABS(12), !A, (2*A+B).

<expression> - any arithmetic expression; e.g.:

A+B/2*(27-R)&H.

<relational expression> - an expression, or a pair of expressons linked by a relational operator; e.g.:

A, A>=B, $A="CAT".

<testable expression> - any number of <relational expressions> connected by AND or OR; e.g.:

A>B AND C>D.

<string right> - a quoted string, or an expression optionally preceded by a dollar; e.g.:

"STRING", $A.

ABS Absolute value **A.**

This function returns the absolute value of its argument, which is a <factor>. ABS will fail to take the absolute value of the maximum negative integer, -2147483648, since this has no corresponding positive value. The most common use of ABS is in conjunction with RND to produce random numbers in a specified range, see RND. Example:

```
      PRINT ABS-1,ABS(-1),ABS1,ABS(1)'
    1     1     1     1
```

AND Relational AND **A.**

This symbol provides the logical AND operation between two <relational expression>s. Its form is <relational expression a> AND <relational expression b> and the result will be true only if both <relational expression>s are true. AND has the same priority as OR. Example:

```
      IF A=B AND C=D PRINT"EQUAL PAIRS"'
```

BGET Byte get **B.**

This function returns a single byte from a random file. The form of the instruction is:

```
BGET <factor>
```

where <factor> is the file handle returned by the FIN function. The next byte from the random file is returned as the least significant byte of the value, the other three bytes being zero. In the DOS the sequential pointer will be moved on by one and the operating system will cause an error if the pointer passes the end of the file. Example:

```
A=FIN"FRED"
PRINT "THE FIRST BYTE FROM FRED IS "BGET A'
```

BPUT Byte put **B.**

This statement sends a single byte to a random file. The form of the statement is:

```
BPUT <factor>, <expression>
```

where <factor> is the file handle returned by the FOUT function; the <expression> is evaluated and its least significant byte is sent to the random file. If you are using the DOS, the random file's sequential pointer will be moved on by one and the operating system will cause an error if the length of the file exceeds the space allowed. Example:

```
A=FOUT"FRED"
BPUT A, 23
```

CH Change character to number **CH**

This function returns the number representing the first ASCII character of the string supplied as its argument. It differs from straight use of the '?' operator in that it can take an immediate string argument or an <expression>. Examples:

```
   PRINT CH""'
13       (value of string terminating character)

   PRINT CH"BETA"'
66

   S=TOP;$S="BETA"
   PRINT ?S,CH$S,CHS'
66   66   66

   PRINT S?LENS,CH$S+LENS'
65   65
```

COUNT Count of characters printed **C.**

This function returns the number of characters printed since the last return, and is thus (usually) the column position on a line at which the next character will be printed. COUNT is useful for positioning table elements etc. Example:

```
   DO PRINT"=";UNTIL COUNT=14
==============>
```

DO **Start of DO ... UNTIL loop** **DO**

This statement is part of the DO...UNTIL control expression. As the BASIC interpreter passes DO it saves that position and will return to it if the UNTIL statement's condition is false. No more than 11 active DO statements are allowed. See UNTIL for examples.

END **End of program** **E.**

This statement has two functions:

1. Termination of an executing program

2. Resetting the value of TOP to point to the first free byte after the program text.

END can be used in direct mode to set TOP. Programs can have as many END statements as required and they do not need to have an END statement as a last line, although an error will be caused on execution past the end of the program. See also TOP. Example:

```
IF $Z="FINISH" END; REM conditional end
```

EXT **Extent of random file** **E.**

In the DOS this function returns the EXTent (length) of a random file in bytes. The file can be either an input or an output file, and the form of the instruction is

```
EXT<factor>
```

where factor is the file handle found using either FIN or FOUT.
In the COS, execution of this function results in an error. Example:

```
A=FIN"FRED"
PRINT "FRED IS "EXT(A)" BYTES LONG"'
```

FIN **Find input** **F.**

In the DOS this function initialises a random file for input (with GET, BGET and SGET) and updating (with PUT, BPUT and SPUT), and returns a number which uniquely represents the file. This 'file handle' is used in all future references to the file. Zero is returned if the file does not exist. The file handle is only one byte long (1 - 255) and can be stored in variables or using ! or ?. Usage of a file handle not given by the operating system will result in an error.
In the COS FIN causes an error.

FOR **Start of FOR ... NEXT loop** **F.**

This statement is the first part of the FOR ... NEXT loop, which allows a section of BASIC text to be executed several times. The form of the FOR statement is:

```
FOR (a) = (b) TO (c) STEP (d)
```

where (a) is the CONTROL VARIABLE which is used to test for loop completion
(b) is the initial value of the control variable
(c) is the limit to the value of the control variable
(d) is the step size in value of the control variable for each pass of the loop; if omitted, it is assumed to be 1.

Items (b) (c) (d) are <expression>s; they are evaluated only once, when the FOR statement is encountered, and the values are stored for later reference by the NEXT statement. No more than 11 nested FOR statements are allowed by the interpreter. Examples:

```
FOR Z=0 TO 11
FOR @=X TO Y
FOR U=-7 TO 0
FOR G=(X+1)*2 TO Y-100
FOR J=0 TO 9 STEP 3
FOR K=X+1 TO Y+2 STEP I
FOR Q=-10*ABSX TO -14*ABSY STEP -ABSQ
```

FOUT · Find output **FO.**

In the DOS this function initialises a random file for output (with PUT, BPUT and SPUT), and returns a number which uniquely specifies the output file. This 'file handle' is used in all future references to the file. An error will occur if there is a problem associated with using the file as an output file; e.g.

(a) write protected file
(b) write protected disc
(c) insufficient space in directory
(d) insufficient memory space.

The number returned is only one byte long (1-255) and can be stored in variables or using ! or ?. Usage of a number not given by the operating system will result in an error.

In the COS FOUT causes an error.

Example:

```
A=FOUT"FRED"
IF A=0 PRINT "WE HAVE A PROBLEM WITH FRED"'
```

GET Get word from file **G.**

This function reads a 32 bit word from a random file and returns its value. The form of the instruction is:

```
GET<factor>
```

where <factor> is the file handle found with the FIN function. The first byte fetched from the file becomes the least significant byte of the value.

In the the DOS the random file sequential pointer will be moved on by 4 and the operating system will cause an error if the pointer passes the end of the file.

Example:

```
A=FIN"FRED"
PRINT "THE FIRST WORD FROM FRED IS "GET A'
```

GOSUB Go to subroutine 3'114 mS. 3;114 µS. **GOS.**

This statement gives the ability for programs to call subprograms. The GOSUB statement stores its position so that it can come back later on execution of a RETURN statement. Like GOTO it can be followed by a <factor> whose value is a line number, or by a label. No more than 15 GOSUB statements without RETURNs are allowed. Example:

```
 10 GOSUB a
 20 GOSUB a
 30 END
100 a PRINT"THIS IS A SUB PROGRAM"'
140 RETURN
```

When RUN this will print:

```
THIS IS A SUB PROGRAM
```

```
THIS IS A SUB PROGRAM
>
```

GOTO Go to line **G.**

This statement overrides the sequential order of program statement execution. It can be used after an IF statement to give a conditional change in the program execution. The form of the statement is either:

```
      GOTO <factor>
or    GOTO <label>
```

The GOTO statement can transfer to either an unlabelled line, by specifying the line's number, or to a labelled line, by specifying the line's label. Examples:

```
   10 IF A=0 PRINT"ATTACK BY KLINGON "Z;GOTO x
   20 PRINT"YOU ARE IN QUADRANT "X Y
   30x PRINT'"STARDATE "T'

  100m INPUT"CHOICE "A
  110 IF A<1 OR A>9 PRINT"!!!!!"; GOTO m
  120 GOTO(A*200); REM GO EVERYWHERE !
```

IF If statement **IF**

This statement is the main control mechanism of BASIC. It is followed by a <testable expression>, which is a single byte. If TRUE (non-zero) the remainder of the line will be interpreted; if FALSE (zero) execution will continue on the next line. After the <testable expression>, IF can be followed by one of two different options:

1. The symbol THEN, followed by any statement

2. Any statement, provided that the statement does not begin with T or a unary operator '!' or '?'.

Examples:

```
IF A<3 AND B>4 THEN C=26
IF A<3 IF B>4 C=26; REM equivalent condition to above
IF A>3 OR B<4 THEN C=22; REM complementary condition to above
IF A>3 AND $S="FRED" OR C=22; REM AND and OR have equal priority
```

INPUT Input statement **IN.**

This statement receives data from the keyboard. The INPUT statement consists of a list of items which can be:

(a) a string delimited by " quotes
(b) any ' new-line symbols
(c) a <variable> or a $<expression> separated from succeeding items by a comma.

Items (a) and (b) are printed out, and for each item (c) a '?' is printed and the the program will wait for a response. If the item is a <variable>, the response can be any valid <expression>; if the item was a $<expression>, the response is treated as a string and will be located in memory starting at the address given by evaluating the <expression>. If an invalid response is typed, no change to the original is made. Example:

```
INPUT"WHAT IS YOUR NAME "$TOP,"AND HOW OLD ARE YOU "A
```

When RUN this will produce:

```
WHAT IS YOUR NAME ?FRED
```

```
AND HOW OLD ARE YOU ?100
```

LEN Length of string **L.**

This function returns the number of characters in a string. The argument for LEN is a <factor> which points to the first character in the string. Valid strings have between 0 and 255 characters before a terminating return; invalid strings for which the terminating return is not found after 255 characters will return length zero. Example:

```
$TOP="FRED";PRINT"LENGTH OF "$TOP" IS "LEN TOP'
```

LET Assignment statement **omit**

This statement is the assignment statement and the word LET is optional. There are two types of assignment statement:

1. Arithmetic

```
LET<variable>=<expression>
   <variable>!<factor>=<expression>
   <variable>?<factor>=<expression>
   !<factor>=<expression>
   ?<factor>=<expression>
```

2. String movement

```
LET$<expression>=<string right>
```

In each case the value of the right-hand side is evaluated, and then stored as designated by the left-hand side. The word LET is not legal in an array assignment.

LINK Link to machine code subroutine 1.650 mS. **LI.**

This statement causes execution of a machine code subroutine at a specified address. Its form is:

```
LINK <factor>
```

where <factor> specifies the address of the subroutine. The processor's A, X and Y registers will be initialised to the least significant bytes of the BASIC variables A, X and Y, and the decimal mode flag will be cleared. The return to the interpreter from the machine code program is via an RTS instruction. Examples:

```
Q=TOP; !Q=#6058; LINK Q; REM clear interrupt flag
Q=TOP; !Q=#6078; LINK Q; REM set interrupt flag
LINK #FFE3;REM wait for key to be pressed
```

LIST List BASIC text **L.**

This command will list program lines in the current text area. It can be interrupted by pressing ESC and can take any of these forms:

```
LIST          list all lines
LIST 10       list line 10
LIST , 40     list all lines up to 40
LIST 100 ,    list all lines from 100
LIST 10,40    list all lines between 10 and 40
```

LOAD Load BASIC program **LO.**

This command will load a BASIC program into the current text area. Its form is:

```
LOAD <string right>
```

and it will pass the string to the operating system and request the operating system to complete the transfer before returning (in case

the transfer is by interrupt or direct memory access). Then the text area is scanned through to set the value of TOP; if the file was machine code or data and not a valid BASIC program the prompt may not reappear. Example:

```
LOAD"FRED"
```

NEW Initialise text area **N.**

This command inserts an 'end of text' marker at the start of the text area, and changes the value of TOP accordingly. The OLD command provides an immediate recovery.

NEXT Terminator of FOR ... NEXT loop
N.

This statement is half of the FOR ... NEXT control loop. When the word NEXT is encountered, the interpreter increases the value of the control variable by the step size, and, if the control variable has not exceeded the loop termination value, control is transfered back to the statement after the FOR statement; otherwise execution proceeds to the statement after the NEXT statement. The NEXT statement optionally takes a <variable> which will cause a return to the same level of nesting as the FOR statement with the same control variable, or an error if no such FOR statement is active. Examples:

```
      @=2
      FOR Z=0 TO 9; PRINT Z; NEXT; PRINT'
 0 1 2 3 4 5 6 7 8 9

      FOR Z=0 TO 9 STEP 2; PRINT Z; NEXT Z;PRINT'
 0 2 4 6 8

      FOR Z=0 TO 9; PRINT Z; NEXT Y
 0
Error   230
>
```

OR Relational OR **OR**

This symbol provides the logical OR operation between two <relational expression>s. Its form is <relational expression a> OR <relational expression b> and the result will be true (non-zero) if either <relational expression> is true. OR has the same priority as AND. Example:

```
IF A=B OR C=D PRINT"At least one pair equal"'
```

PRINT Print statement **P.**

This statement outputs results and strings to the screen. A PRINT statement consists of a list of the following items:

(a) a string delimited by " quotes, which will be printed.
(b) any ' symbols which will cause a 'new line'.
(c) the character '&' which forces hexadecimal numerical print out until the next comma.
(d) an <expression> whose value is printed out in either decimal or hexadecimal, right-hand justified in a field width defined by '@'
(e) a $<expression>; if the value of the <expression> is between 0 and 255, the ASCII character corresponding to that value will be printed out; otherwise the string pointed to by that value will be printed out.

Examples:

```
      PRINT'

      PRINT"Hello"'
Hello

      PRINT 1'
    1

      PRINT 1'2'3'
    1
    2
    3

      PRINT"40*25="40*25'
40*25= 1000

      PRINT$CH"e"'
e

      PRINT$12

      DO INPUT"Who are you "$TOP;PRINT"Hi "$TOP'; UNTIL $TOP=""
Who are you ?fred
Hi fred
Who are you ?

      PRINT&0 10 20 30'
    0    A   14   1E
```

PTR Pointer of random file **PTR**

In the DOS this function and statement allows the manipulation of the pointers in sequential files. Its form is:

```
      PTR<factor>
```

where <factor> is the file handle found using FIN or FOUT, and it may appear on the left-hand side of an equal sign or in an expression.
In the COS PTR will cause an error.
Examples:

```
      A=FIN"FRED"
      PRINT PTR A'
    0

      PTRA=PTRA+23
```

PUT Put word to random file **PUT**

This statement sends a four byte word to a sequential output file. The form of the instruction is:

```
      PUT <factor> , <expression>
```

where <factor> is the file handle returned by the FOUT function. The <expression> is evaluated and sent, least significant byte first, to the sequential output file. The sequential output file pointer will be moved on by four and the operating system will cause an error if the length of the file exceeds the space allowed. Example:

```
      A=FOUT"FRED"
      PUT A , 123456
```

REM Remark **REM**

This statement causes the interpreter to ignore the rest of the line, enabling comments to be written into the program. Alternatively comments can be written on lines branched around by a GOTO statement.

RETURN Return from subroutine **R.**

This statement causes a return to the last encountered GOSUB statement. See GOSUB for examples.

RND Random number **R.**

This function returns a random number between -2147483648 and 2147483647, generated from a 33 bit pseudo-random binary sequence generator which will only repeat after over eight thousand million calls. The sequence is not initialised on entering the interpreter, but locations 8 to 12 contain the seed, and can be set using '!' to a chosen starting point. To produce random numbers in some range A to B use:

```
ABSRND%(B-A)+A
```

RUN Execute BASIC text from beginning **RUN**

This statement will cause the interpreter to commence execution at the lowest numbered line of the current text area. Since it is a statement, it may be used in both direct mode and programs.

SAVE Save BASIC text space **SA.**

This statement will cause the current contents of the memory between the start of the text area, given by ?18*256, and the value of TOP, to be saved by the operating system with a specified name. The operating system is not requested to wait until the transfer is finished before returning to the interpreter. Example:

```
SAVE"FRED"
```

SGET String get **S.**

This statement reads a string from a random file. The form of the statement is:

```
SGET <factor>, <expression>
```

where <factor> is the file handle returned by the FIN function. The <expression> is evaluated to form an address, and bytes are taken from the sequential input file and put in memory at consecutive locations starting at that address, until a 'return' is read. The sequential input file pointer will be moved on by the length of the string plus one and the operating system will cause an error if the pointer passes the end of the input file.

SHUT Finish with random file **SH.**

In the DOS this statement closes random input or output files. The form of the statement is:

```
SHUT <factor>
```

where <factor> is the file handle found with either FIN or FOUT. If it is an output file any information remaining in buffer areas in memory is written to the file. If the <factor> has value zero, all current sequential files will be closed. In the COS this statement causes an error.

SPUT String put **SP.**

This statement writes a string to a random file. The form of the instruction is:

```
SPUT <factor>, <string right>
```

where <factor> is the file handle returned by the FOUT function. Every byte of the string, including the terminating 'return' character, is sent to the file. In the DOS the random file sequential pointer will be moved on by the length of the string plus one, and the operating system will cause an error if the length of the file exceeds the space allowed. Example:

```
A=FOUT"FRED"
SPUT A , "THIS IS FILE FRED"
```

STEP Step specifier in FOR statement **S.**

This symbol is an optional parameter in the FOR statement, used to specify step sizes other than the default of +1. It is followed by an <expression> which is evaluated and its value stored along with the other FOR parameters. See FOR for examples.

THEN Connective in IF statement **omit**

This symbol is an option in the IF statement; it can be followed by any statement.

TO Limit specifier in FOR statement **TO**

This symbol is required in a FOR statement to specify the limit which is to be reached before the FOR ... NEXT loop can be terminated. See FOR for examples.

TOP First free byte **T.**

This function returns the address of the first free byte after the end of a stored BASIC program. Its value is adjusted during line editing and by the END statement and LOAD command. It is vital for TOP to have the correct value (set by END) before using the line editor. See also ?18 and END.

UNTIL Terminator of DO ... UNTIL loop **U.**

This statement is part of the DO ... UNTIL repetitive loop. UNTIL takes a <testable expression> and will return control to the character after DO if this is zero (false), otherwise execution will continue with the next statement. Examples:

```
DO PRINT"#";UNTIL 0; REM do forever

DO PRINT"#"; UNTIL COUNT=20; PRINT'
####################

DO INPUT"Calculation "A; PRINT"Answer is "A'; UNTIL A=12345678
Calulation ?2*3
Answer is     6
Calculation ?A
Answer is     6
Calculation ?12345678
Answer is 12345678
```

16.0 BASIC CHARACTERS AND OPERATORS

This section lists all BASIC's special characters and operators. They are followed by a description of the character or operator, and its name enclosed in {} brackets. Lower case characters in <> brackets refer to the syntax definition in Chapter 17.

16.1 Special Characters

Line terminater **{RETURN}**

This character is used to terminate a statement or command, or a line input to the INPUT statement, and as the terminator for strings.

Cancel input **{CAN (CTRL-X)}**

This character will, when typed from the keyboard, delete the current input buffer and give a new line.

Escape **{ESC}**

This character, typed on the keyboard, will stop any BASIC program and return to direct mode. BASIC checks for escape at every statement terminator.

Separator **{space}**

This character is stored intact to allow formating of programs. Space may be used anywhere except:

1. In control words
2. After the # {hash} symbol
3. Between line number and label.

It may be necessary to insert spaces to avoid ambiguity as, for example, in:

```
FORZ=V TOW STEPX
```

Here a separator character is needed between V and T, and similarly between W and S, to eliminate the possibility of a function called VTOWSTEP.

" **String delimiter** **{double quote}**

This character is used as the delimiting character whenever a string is to be part of a BASIC statement (i.e. everywhere except when inputting strings with an INPUT statement). If you wish to include " in a string it should be written "". The simple rule for valid strings is that they have an even number of " characters in them.

' **New line** **{single quote}**

This character may be used in PRINT and INPUT statements to generate a new line by generating both CR and LF codes. The value of COUNT will be set to zero.

( ) **{round brackets}**

These characters provide a means of overriding the normal arithmetic priority of the operators in an <expression>. The contents of brackets are worked out first, starting with the innermost brackets.

, **Separator** **{comma}**

This character is used to separate items in PRINT and INPUT statements.

. **{stop}**

This character is used to allow a shorter representation for some of the key-words, thus using less memory space to store the program.

; **Statement terminator** **{semi-colon}**

This character is the statement terminator used in multi-statement lines.

@ **Numeric field width** **{at}**

This character is a variable which controls the PRINT statement. It specifies the number of spaces in which a number will be printed, right-justified. If the field size is too small to print the number, the number is printed in full without any extra spaces; thus field sizes of 0 and 1 give the same result of minimum-width printing. The - sign is printed in front of a negative number and counts towards the number of characters in the number. On initial entry into BASIC, any error, or following use of the LIST statement or assembler, @ is set to 5. Example:

```
      @=5;PRINT1,12,123,1234,12345,123456'
    1   12  123 123412345123456
```

a - z **Labels**

These characters provide a very fast means of transferring control with the GOTO and GOSUB statements. A line may be labelled by putting one of a-z immediately after the line number (no blanks are allowed before the label). Transfer to a labelled line is achieved by a GOTO or GOSUB statement followed by the required label. Example:

```
   10a PRINT"looping"'
   20 GOTO a
>RUN
looping
looping
looping
```

16.2 Operators

! **Word indirection** **{pling}**

This character provides word indirection. It can be both a binary and a unary operator and appear on the left-hand side of an equal sign as well as in <expression>s.

As a unary operator on the LEFT of an equals sign it takes a <factor> as an argument and will treat this as an address. The <expression> on the right of the equals sign is evaluated and then stored, starting with the least significant byte, in the four locations starting at this address. Example:

```
!A=#12345678
```

will store values in memory as follows:

```
---------------------
| 78 | 56 | 34 | 12 |
---------------------
  A   A+1  A+2  A+3
```

As a binary operator on the LEFT of an equals sign it takes two arguments; a <variable> on the left and a <factor> on the right. These two values are added together to create the address, and the value is stored at this address as above. Example:

```
A!B=#12345678
```

As a unary operator in an <expression> it takes a <factor> as an argument and will treat this as an address. The value is that contained in the four bytes at this address. For example, if the contents of memory are as follows:

```
---------------------
| 18 | 00 | 00 | 00 |
---------------------
  A   A+1  A+2  A+3
```

Then the value printed by

```
PRINT !A
```

will be 24 (decimal).

As a binary operator in an <expression> it takes two arguments, a <factor> on either side. The sum of these two values is used as the address, as above. Example:

```
PRINT A!B
```

 # Hexadecimal constant {hash or pound}

This character denotes the start of a hexadecimal value in <factor>. It cannot be followed by a space and there is no check made for overflow of the value. The valid hexadecimal characters are 0 to 9 and A to F.

$ String pointer {dollar}

This character introduces a pointer to a string; whenever it appears it can be followed by an <expression>. In a PRINT statement, if the pointer is less than 256, the ASCII character corresponding to the value of the pointer will be printed. Dollar can be used on the left of an equals sign as well as anywhere a string can be used. If the only choice allowed is either a dollar or a string in double quotes, then it is possible to omit the dollar. Strings may contain up to 255 characters. Examples:

```
IF$A=$B.....    string equality test
IF$A="FRED"....  string equality test
$A="JIM".....   move string JIM to where A is pointing
$A=$B.....      copy B's string to where A points
PRINT$A.....    print the string A is pointing at
PRINT$A+1.....  print the string (A+1) is pointing at
PRINT$64.....   print ASCII character 64 i.e. @
```

% **Remainder** **{percent}**

This character is the operation of signed remainder between two values. Its form is <factor a>%<factor b>. The sign of the result is the same as the sign of the first operand.

& **Hexadecimal/AND** **{ampersand}**

This character has two distinct uses:

1. To print hexadecimal values in the PRINT statement. Its form here is as a prefix in front of the particular print item which is to be printed in hexadecimal

2. As the operation of bitwise logical AND between two values. Its form here is <factor a>&<factor b> and the result is a 32 bit word, each bit of which is a logical AND between corresponding bits of the operands.

***** **Multiply** **{star}**

This character is the operation of signed multiplication between two 32 bit values. Its form is <factor a>*<factor b>.

+ **Add** **{plus}**

This character has two similar uses:

1. As the unary operation "do not change sign". Its form here is +<factor>

2. As the operation of addition between two 32 bit values. Its form here is <term a>+<term b>.

- **Subtract** **{minus}**

This character has two similar uses:

1. As the unary operation of negate. Its form here is -<factor>, and the result is 0 - <factor>

2. As the operation of subtraction between two 32 bit values. Its form here is <term a>-<term b> and the result is found by subtracting <term b> from <term a>.

/ **Divide** **{slash}**

This character is the operation of signed division between two 32 bit values. Its form is <factor a>/<factor b> and the result is found by dividing <factor a> by <factor b>.

: **Exclusive-OR** **{colon}**

This character is the operation of bitwise logical exclusive-OR between two 32 bit <term>s. Its form is <term a>:<term b> and the result is a 32 bit word each bit of which is the exclusive-OR of corresponding bits in <term a> and <term b>.

< **Less-than** **{left triangular bracket}**

This character is the relational operator "less than" between two <expression>s. Its form is <expression a> < <expression b> and it returns a logical value, of 'true' if <expression a> is less than <expression b> and 'false' otherwise, which can be tested by IF and UNTIL statements.

= Equals **{equal}**

This character has two uses:

1. As the relational operator "equal to" between two <expression>s. Its form is <expression a> = <expression b> and it returns a logical value, of 'true' if <expression a> is equal to <expression b> and 'false' otherwise, which can be tested by IF and UNTIL statements

2. As the assignment operation "becomes". The object on the left-hand side is assigned the value of the right-hand side. There are three similar uses of this:

	Example:
1. Arithmetic	
<variable>=<expression>	A=2
<variable>!<factor>=<expression>	A!J=3
<variable>?<factor>=<expression>	A?J=4
!<factor>=<expression>	!J=5
?<factor>=<expression>	?J=6
2. String movement	
$<expression>=<string right>	$A="FRED"
3. FOR statement	
FOR<variable>=<expression> ...	FOR A=0 TO ...

> Greater-than **{right triangular bracket}**

This character is the relational operator "greater than" between two <expression>s. Its form is <expression a> > <expression b> and it returns a logical value, of 'true' if <expression a> is greater than <expression b> and 'false' otherwise, which can be tested by IF and UNTIL statements.

? Byte indirection **{query}**

This character provides byte indirection. It can be either a binary or a unary operator and appear on the left-hand side of an equals sign as well as in <expression>s.

As a unary operator on the LEFT of an equals sign it takes a <factor> as an argument and will treat this as an address; the <expression> on the right of the equals sign is evaluated and its least significant byte is stored at that address. Example:

```
?A=#12345678
```

will store into memory as follows:

```
------
| 78 |
------
  A
```

As a binary operator on the LEFT of an equals sign it takes two arguments, a <variable> on the left and a <factor> on the right. These two values are added together to create the address where the value will be stored as above. Example:

```
A?B=#12345678
```

As a unary operator in an <expression> it takes a <factor> as an argument and will treat this as an address; the value is a word whose most significant three bytes are zero and whose least significant byte is the contents of that address. Example:

```
PRINT ?A
```

As a binary operator in an <expression> it takes two arguments, a <factor> on either side. The sum of these two values is the address used as above. Example :

```
PRINT A?B
```

| OR {inverted backslash}

This character is the binary operation of bitwise logical OR between two 32 bit <term>s. Its form is <term a>|<term b> and the result is a 32 bit word each bit of which is an OR operation between corresponding bits of <term a> and <term b>.

<> Not equal {left & right triangular brackets}

This symbol is the relational operator "not equal to" between two <expression>s. Its form is <expression a> <> <expression b> and it returns a logical value, of 'true' if <expression a> is not equal to <expression b> and 'false' otherwise, which can be tested by IF and UNTIL statements.

<= Less or equal {left triangular bracket, equal}

This symbol is the relational operator "less than or equal" between two <expression>s. Its form is <expression a> <= <expression b> and it returns a logical value, of 'true' if <expression a> is less than or equal to <expression b> and 'false' otherwise, which can be tested by IF and UNTIL statements.

>= Greater or equal {right triangular bracket, equal}

This symbol is the relational operation "greater than or equal to" between two <expression>s. Its form is <expression a> >= <expression b> and it returns a logical value, of 'true' if <expression a> is greater than or equal to <expression b> and 'false' otherwise, which can be tested by IF and UNTIL statements.

17.0 FORMAL SYNTAX DEFINITION

This syntax definition is written in B.N.F., or Backus-Naur Form, with some additions. In the places where a proper definition in B.N.F. would be far too long, a description has been used. The rules are:

Things in triangular <> brackets are defined things, "syntactic entities", everything else is itself

The ::= symbol is read as "is defined"

The | sign is read as OR: one of the alternatives must be true

Concatenation of things is read as "followed by"

The ^ sign is read as "any number of"

The {} brackets allow concatenations to be grouped together.

17.1.1 Basic Symbols

! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z [| <> <= >= ABS AND END EXT FIN FOR GET LEN LET NEW PTR PUT REM RND RUN TOP BGET BPUT FOUT GOTO LINK LIST LOAD NEXT SAVE SGET SHUT SPUT STEP THEN COUNT GOSUB INPUT PRINT UNTIL RETURN

No multi-character basic symbols may include blanks; otherwise blanks may be used freely to improve the readability of the program. The character '.' can be used to provide a shorter representation of all multi-character basic symbols.

<asciic>::={ascii characters excluding carriage return}

<digit>::=0|1|2|3|4|5|6|7|8|9

<hex digit>::=<digit>|A|B|C|D|E|F

<positive number>::=<digit><digit>^
such that <positive number> is less than 2147483648

<hex number>::=<hex digit><hex digit>^

<integer field size>::=@

<p-variable>::=<integer field size>|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<variable>::=<p-variable>{character which is not <p-variable> or .}

<label>::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<conjunction>::=AND|OR

<relation operation>::=<|>|<=|>=|=|<>

```
<expression operator>::=+|-|||:

<term operator>::=*|/|%|&|!|?

<factor>::=+<unary plus>|-<unary plus>|<unary plus>

<unary plus>::=<variable>|<positive number>|#<hex number>|
        (<testable expression>)|!<factor>|?<factor>|TOP|COUNT
        |RND|ABS<factor>|LEN<factor>|CH<string right>
        |PTR<factor>|EXT<factor>|GET<factor>|BGET<factor>|
        FIN<string right>|FOUT<string right>

<term>::=<factor>{<term operation><factor>}^

<expression>::=<term>{<expression operator><term>}^

<relnl expression>::=<expression>|<expression><relation operation>
        <exression>|$<expression>=<string right>

<testable expression>::=<relnl expression>{<conjunction>
        <relnl expression>}^

<delimit quote>::="{any ascii character not a "}

<string right>::=<expression>|$<expression>|"<asciic>^<delimit quote>

<sd>::=<statement delimiter>::={carriage return}|;

<working let>::={{{<variable>|!<factor>|?<factor>|<variable>!<factor>|
        <variable>?<factor>}=<expression>}|$<expression>=
        <string right>}<sd>

<let statement>::=LET<working let><sd>|<working let><sd>

<printable string>::={'|"<asciic>^<delimit quote>}^

<input section>::=<printable string>{<variable>|$<expression>|{null}}

<input statement>::=INPUT<input section>{,<input section>}^<sd>

<return statement>::=RETURN<sd>

<new command>::=NEW<sd>

<link statement>::=LINK<factor><sd>

<OS statement>::=*<asciic>^

<go entity>::=<label>|<factor>

<goto statement>::=GOTO<go entity><sd>

<gosub statement>::=GOSUB<go entity><sd>

<end statement>::=END<sd>

<do statement>::=DO

<until statement>::=UNTIL<testable expression><sd>
```

```
<next statement>::=NEXT<sd>|NEXT<variable><sd>

<half for>::=FOR<variable>=<expression>TO<expression>

<for statement>::=<half for><sd>|<half for><STEP<expression><sd>

<save statement>::=SAVE<string right><sd>

<load command>::=LOAD<string right><sd>

<run statement>::=RUN<sd>

<list command>::=LIST<sd>|LIST<positive number><sd>|
          LIST,<positive number><sd>|LIST<positive number>,<sd>|
          LIST<positive number>,<positive number><sd>

<if statement>::=IF<testable expression>{THEN<statement>|<statement>}

<print comma>::={nothing, if possible}|,

<print statement>::=PRINT{<printable string>{<expression>|
          $<expression>|{nothing}}<print comma>}^<sd>

<enter line command>::=<positive number><asciic>^{carriage return}

<put statement>::=PUT<factor>,<expression><sd>

<bput statement>::=BPUT<factor>,<expression><sd>

<sput statement>::=SPUT<factor>,<string right><sd>

<sget statement>::=SGET<factor>,<expression><sd>

<ptr statement>::=PTR<factor>=<expression><sd>

<null statement>::=<sd>
```

18.0 ERROR CODES

The following is a list of all 4K BASIC errors. Note that it is possible to obtain errors not on this list by executing a BRK in a machine-code program, or using a BASIC extension.

18 Too many DO statements, UNTIL with no DO

The largest permitted number of nested DO ... UNTIL loops is 15. This limit has been exceeded.

An UNTIL statement was encountered without a DO being active. Example:

```
20 IF A=1 DO A=A+1
30 UNTIL A=3    (if A<>1 the DO is not executed)
```

29 Unknown or missing function

The statement contains a sequence of characters which are not the name of a function. Example:

```
10 J=RAN+10     (where RND was intended).
20 FPRINT $A    (string variables not permitted in FPRINT)
```

31 RETURN without GOSUB, Too many GOSUBs, Line not found

A RETURN was found in the main program. RETURN is only meaningful in a subroutine.

The largest permitted depth of subroutine nesting is 15. This error means that more than 11 GOSUB statements have been executed without matching RETURN statements. Example:

```
10 GOSUB 10
20 END
```

The line number specified in a GOTO or GOSUB was not found. Example:

```
10 GOTO 6
15 N=6; GOTO N    (where there is no line 6)
```

91 No hexadecimal number after '#'

The characters immediately following the '#' symbol must be legal hexadecimal characters 0-9 or A-F. Spaces are not permitted. Example:

```
10 PRINT #J
```

94 Unknown command; invalid statement terminator; missing END GOSUB without RETURN; FOR without NEXT

The statement has not been recognised as a legal BASIC statement. The error may also be caused by an illegal character after a valid statement, or by an attempt to execute past the end of the program. Example:

```
10 LIST    (LIST is not allowed in a program)
```

```
20  s A=B     (no space permitted between label and line number)
```

The GOSUB statement, when used in direct mode, must be followed by a semicolon. Example:

```
GOSUB 10
```

The FOR statement was used in direct mode without a NEXT statement.

109 Number too large, Attempt to use variable in LIST

Attempt to enter a number which is too large to be represented in BASIC. Example:

```
20 J=9999999999
```

Error also occurs if the largest negative number is entered:

```
30 J=-2147483648
```

even though this number can be represented internally. To input this number, use the hexadecimal form #80000000.

The LIST command may only be used with constants as its arguments. Example:

```
LIST A,B
```

111 Missing variable in FOR; too many FOR statements

The control variable in a FOR ... NEXT loop must be one of the simple variables A to Z. Example:

```
35 FOR !C=1 TO 10
```

The maximum permitted number of nested FOR ... NEXT loops is 15; this number has been exceeded.

127 Label not found

A label, a-z, was specified in a GOTO or GOSUB, but no statement starting with that label was found. Example:

```
40 GOTO s
```

129 Division by zero

A number was divided by zero. Example:

```
10 J=J/(A-B)     (where A and B were equal)
```

159 Unmatched quotes in PRINT or INPUT

Strings in PRINT statements, or entered in INPUT statements, should have an even number of '"' quotation marks. Example:

```
PRINT "THIS IS A QUOTE:""
```

174 Significant item missing or malformed

An unexpected character was encountered during the interpretation of a statement. Example:

```
10 G0TO 20     (O mistyped as zero; should be GOTO)
20 FOR J TO 4     (expected '=' after J)
30 FOR J=1 STEP 1 TO 4     (order should be TO ... STEP)
```

200 Unmatched quotes in string

Strings appearing in a program should have an even number of '"' quotation marks.

230 NEXT without matching FOR

If a control variable is specified in a NEXT statement then the variable must match the control variable in the corresponding FOR statement. Example:

```
10 FOR N=1 TO 10
20 FOR J=1 TO 10
30 PRINT "*"
40 NEXT N
50 NEXT J
```

A NEXT statement was encountered without any FOR statement being active.