ATOMIC THEORY AND PRACTICE

A beginner's course in BASIC and machine code programming

David Johnson-Davies

ACORN ATOM

ATOMIC THEORY AND PRACTICE

Contents

Introduction

This manual explains how to connect up the ATOM, and how to program it in BASIC or Assembler. The manual is arranged in three sections printed on different coloured paper. If you have never programmed before you should read the BASIC section, on plain paper, starting from Chapter 1; but be warned that you are setting off on an adventure which will 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 the ATOM's BASIC.

If you have already programmed in BASIC you may prefer to turn to chapters 20 and 21 in the reference section; these contain a complete summary of all the BASIC statements, functions, commands, and operators. You will be pleased to discover a number of extensions in ATOM BASIC that are not found in other BASICS.

If you want to learn to program in Assembler you should turn to the second section of the manual which is printed on coloured paper, and read from chapter 13 onwards. Readers experienced in Assembler programming can jump to chapter 23 in the reference section, which gives a concise description of the ATOM assembler.

The third section of the manual, printed on plain paper, is the reference section. It contains a summary of all the ATOM's facilities, a listing of the special addresses in the ATOM, and the error codes.

If you have a minimal ATOM you will be able to run all programs whose sizes are given as less than 512 bytes, or which are so short that no size is given. Longer programs will require additional memory, but many programs can be reduced in size by using the abbreviations explained in chapter 10.

Acknowledgements

CONNECTIONS TO ATOM

ACORN BUS EXTENSION

VIA CONNECTOR PRINTER OR USER I/O

HEATSINK ACCESS

B/W AND COLOUR SIGNALS

D.C. POWER SUPPLY JACK

TAPE AND USER INPUT/OUTPUT

UHF MODULATED OUTPUT

ATOM KEYBOARD

STANDARD TYPEWRITER KEYS

COMPUTER FUNCTION KEYS

EDITING KEYS

CHANGE MODE KEYS

2

1 Start Here

If you bought the ATOM ready built, together with a power supply and a cable to connect it to a TV set, then carry on reading. Otherwise you should refer to the Technical Manual for details of how to assemble an ATOM kit, and for details of the required accessories.

The ATOM connects to the aerial socket of an ordinary black-and-white or colour TV set. The ATOM will not affect the normal operation of the TV in any way. Connect the UHF output from the ATOM to the aerial socket of the TV set; see Fig. 1. Connect the ATOM's power supply to a mains socket, and plug the power connector into the back of the ATOM; again, see Fig. 1. Press the key marked BREAK on the top right of the ATOM's keyboard. Switch on the TV set, and turn the set's volume control down. The ATOM makes use of a TV channel that is not occupied by any TV stations, and it is necessary to tune to this channel in order to get the correct display from the ATOM. If the TV set you are using has push buttons to select stations, choose an unused button and tune the TV by rotating the button. If the TV has a single numbered tuning dial, turn the dial to somewhere near channel 36. Tune in the TV set until the screen is black, with the following display in the top left-hand corner of the screen:

ACORN ATOM

>█

Adjust the contrast and brightness controls so that the letters are clearly legible, and tune the TV set carefully until the letters are sharp and clear.

The '>' sign is called the ATOM's 'prompt'. It indicates that the computer is waiting for something to be typed in; a command, perhaps, or a program. The white rectangle, '█', is called the 'cursor'; it indicates where on the screen the next character you type in will appear.

1.1 What the ATOM Can Do

The ATOM 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, CLEAR, DIM, DO, DRAW, END, FOR, GOSUB, GOTO, IF, INPUT, LET, LINK, MOVE, NEXT, OLD, PLOT, PRINT, PUT, REM, RETURN, RUN, SAVE, SGET, SHUT, SPUT, UNTIL, WAIT.

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 the ATOM 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 the ATOM's '>' prompt. The ATOM will reply with a 'bleep' and will print:

    ERROR 94

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

1.2 A Demonstration

Now that you are in control of your ATOM you may like a quick demonstration of some more complicated things that it can do. No attempt is made here to explain how these examples work; for that you will have to read the rest of the first section of this manual.

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

    DO PRINT "ATOM-"; 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,$128; UNTIL 0

You will need to use the SHIFT key to get some of the special symbols. This program is longer than one line of the screen, but just keep typing and it will appear on a second line. Then press RETURN to run the program. Again, you will have to type ESC to stop this program.

    To demonstrate the graphics commands type:

    CLEAR0; MOVE 10,0; DRAW 60,50

and the ATOM will draw a line on the screen. If you feel like trying a more complicated graphics program, type in the following:

4

```
CLEAR0;MOVE32,24;Y=1;DOPLOT1,0,Y;PLOT1,Y,0;Y=-Y-2*Y/A.Y;U.0
```

Press ESC to get back the ATOM's prompt.

To demonstrate the ATOM's assembler enter the following line after the prompt:

```
P=320;[INX; LDA 0,X; STA #B002; JMP 320;]
```

An assembler listing will be printed out, and the machine code will be put into memory at 320. To execute the program, type:

```
LINK 320
```

and the ATOM will make a buzzing noise. It is playing the random contents of its memory through its internal loudspeaker. To stop the program you will have to type BREAK, because it is a machine-code program.

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

1.3 The Keyboard

The ATOM keyboard is designed to the standard layout generally accepted in the computer industry; see Fig. 2. In most respects it is just like the keyboard of an ordinary typewriter, but there are some important differences. For a start there are several keys not found on typewriters, such as DELETE, REPT, CTRL, and BREAK. The purpose of each of these keys will be explained in the following sections.

Another difference is that the letters A-Z will appear in capitals, rather than lower-case, when they are typed by themselves. Try typing in the letters 'ABC' and observe that they appear, as you type them, on the screen:

>ABC█

From now on, in the examples, the cursor will not be drawn in for simplicity.

1.3.1 SHIFT

Some keys carry two legends. For example, each digit key (except 0) also has a special symbol or punctuation mark above it. The lower symbol on each of these keys is obtained by simply typing that key; the upper symbol is obtained by holding one of the SHIFT keys down, and typing that key. This aspect of the keyboard is just like a typewriter.

If the SHIFT key is held down in conjunction with one of the keys bearing a single legend, such as A-Z and @, [, etc, then the character will appear inverted; i.e. as a black character on a white square. Inverted A-Z correspond to lower case letters, and will be represented by lower case letters a-z in this manual. Inverted @, [, \ etc. will be represented by @, [, \ etc.

1.3.2 LOCK

The LOCK key, when pressed on its own, changes the way the SHIFT key operates with the letters A-Z. Initially the keyboard will give inverted A-Z in conjunction with the SHIFT key, and plain A-Z otherwise. If the LOCK key is now pressed once the keyboard will normally give inverted A-Z, and will give plain A-Z only when the SHIFT key is held down. Pressing LOCK again will revert to the previous state.

5

1.3.3 DELETE

The advantage of a TV screen over a piece of paper is that mistakes can be corrected without trace of the error. The DELETE key will erase the last character on the line, and the cursor will back up one space. Mistakes can thus be deleted and retyped with little effort.

1.3.4 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.

1.3.5 Repeat - REPT

If the 'repeat' key, marked REPT, is held down with another key, that key is typed repeatedly. REPT is useful in conjunction with DELETE to erase several characters very rapidly. Note that pressing REPT on its own will have no effect.

1.3.6 Control - CTRL

There are several special functions available from the keyboard which are obtained by typing certain keys with the 'control' key - marked CTRL - held down. Only the following two control functions will be mentioned here:

CTRL-G gives a bleep in the ATOM's loudspeaker.

CTRL-L clears the screen.

1.3.7 BREAK

The BREAK key will reset the computer, and return it to the state it was in just after switching on. It should not normally be necessary to type BREAK, but some assembler programs can cause loops which cannot be stopped in any other way. Note that the contents of memory are preserved when BREAK is typed, and any stored program can be recovered.

1.4 Scrolling

When the cursor reaches the bottom of the screen further lines typed in will cause the screen to 'scroll'; every line is shifted up so that you always see the last 16 lines of what has been typed, and the top line of text on the screen will be lost.

1.5 Storing Text

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

```
10 IN XANADU DID KUBLA KHAN
20 A STATELY PLEASURE-DOME DECREE:
30 WHERE ALPH, THE SACRED RIVER, RAN
40 DOWN TO A SUNLESS SEA.
```

Remember to type RETURN at the end of each line. Each line number can be followed by up to 64 characters; if you try to type more than 64 characters the ATOM 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:

```
36 THROUGH CAVERNS MEASURELESS TO MAN
```

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

1.6 Commands

Commands typed in after the '>' prompt, without a preceding line number, and followed by RETURN, are executed immediately by ATOM rather than being stored in its memory. For example, now type the 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.
```

There are several options with the LIST command. For example:

```
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).

1.7 Editing

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

```
20 NEW LINE TWO
```

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

1.8 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.

OLD can be typed after typing BREAK to retrieve the text previously in

7

memory. Note that you should only type OLD if there is already text in memory.

1.9 Errors

By now you the ATOM will probably have made a 'bleep' followed 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 the ATOM was not expecting or could not interpret.

2. The ATOM 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 the ATOM, 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 27.

1.10 Saving Text or Programs on Tape

Having entered some stored text into the ATOM's memory, this section will show how to save this text, and load it back at a later time.
Text and programs can be saved on standard cassette (or reel-to-reel) tapes using the ATOM's cassette interface. Connect the cassette output from the ATOM to the input of a cassette recorder, and the output from the recorder to the input of the ATOM. The tape load routine uses software averaging techniques to minimise the likelihood of errors on loading, and no trouble should be experienced in transferring tapes from one machine to another.

1.10.1 Setting Up

Before loading and saving files using the cassette interface it is worth entering the following simple routines to check that the cassette system is working correctly, and to find out the best setting of the recorder's volume control.
Enter the following line after the ATOM's prompt:

    DO BPUT A,88; WAIT; WAIT; WAIT; WAIT; UNTIL 0

Type RETURN and record on the recorder for a few minutes. To stop the program type ESC (escape). This program has recorded a sequence of Xs, in coded form, on the tape. If you play it back it should sound like a series of short buzzes.
Now enter the following line, which is a program to read characters from the tape and print them on the screen:

    DO PRINT $BGET A; UNTIL 0

The dollar symbol is obtained by holding the SHIFT key down and typing '4'. Press RETURN, rewind the tape, and play back the 'X's that you recorded. If all is well a stream of 'X's should be printed out, and adjust the volume setting on the recorder so that no other characters appear, indicating errors. When you are satisfied that all is well,

proceed to the next section.

1.10.2 Text Files

The information is stored as a stream of audible tones on tape; each section of information is referred to as a 'file'. Several different files can be saved on one tape, and they are identified by having unique 'filenames'. Filenames can be anything containing up to 16 letters, digits, or spaces: suitable names are "DATA FILE", "22/4/80", etc.

1.10.3 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 filename chosen for illustration. Type RETURN, and the message:

RECORD TAPE

will be printed on the screen. Put the tape recorder to record, and allow the tape to run well past the leader. Now type RETURN (or any other key) and the cursor will move to the start of the next line, indicating that the text is being recorded. After a short delay the '>' prompt will reappear, and you can turn the tape-recorder off.

1.10.4 *CAT

The *CAT command will give a complete catalogue of all the files on a cassette. The '*' asterisk is used to distinguish the cassette operating-system commands from the BASIC commands. Rewind the tape and type:

        *CAT

The ATOM will reply with:

PLAY TAPE

and you should then play the tape, and press any key to start the catalogue. As a file is encountered on the tape the filename will be printed out, together with additional information about the file:

EXAMPLE   XXXX XXXX XXXX XX

where the 'X's represent four numbers which you can ignore for the moment (see Section 19.3 for details).
     When you have finished you can get back to the '>' prompt by typing CTRL (control).

1.10.5 LOAD

Switch off the ATOM, in order to cause the saved text to be lost, and then switch on again and type:

        LOAD "EXAMPLE"

The ATOM will reply with:

PLAY TAPE

and the tape should be rewound and played, and RETURN pressed. The computer will search through the tape for a file of the specified filename, EXAMPLE in this case, and then load it into its memory. If all is well the prompt should reappear, and then typing:

        LIST

9

will give a listing of the text that was previously saved.

1.10.6 File Blocks

If you save a long file on tape, and play it back, you will discover that it is broken up into a number of short blocks, with gaps in between, and that when the file is catalogued its name appears several times, once for each block. This is done for greater reliability, and if the tape is damaged in the middle of one block it will still be possible to load back the other blocks of the file.

One further message that may be given when loading tapes is:

REWIND TAPE

This implies that you have started playing the tape in the middle of the file you wanted to load. Rewind the tape, press RETURN, and the message:

PLAY TAPE

will be given again.

1.10.7 Errors when Using Tape

If an error is found when loading back a tape file, the message:

SUM
ERROR 6

is given. This might be caused by bad adjustment of the tape-recorder playback volume, a damaged or dirty tape, or recording a file over part of a previous file.

If you choose an invalid name for a file, the message:

NAME
ERROR 118

will be given.

2 Calculating in BASIC

The ATOM computer understands a language called BASIC which, because of the ease of writing programs in it, has become the most popular language for use on small computer systems. 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

The ATOM 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. ATOM 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 ATOM BASIC can evaluate, provided they will fit on two lines of the VDU. You will notice that ATOM BASIC calculates extremely rapidly. Try typing:

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

ATOM 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 eight 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 ATOM 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"
```

12

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 8 character spaces. If the number needs more than 8 spaces the field

13

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:

```
@=32
```

will print one number per line, because there are 32 character positions on each line.

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" ' "T" ' "O" ' "M" '
```

will print out:

```
A
T
O
M
>
```

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

2.6 Multiple-Statement Lines - ';'

ATOM 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 13.1.1; 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. #8000 is the address of the display area, and is a more convenient way of specifying this address than its decimal equivalent.

A number can be printed in hexadecimal by prefixing it with an '&' ampersand in the PRINT statement. Thus:

```
PRINT &32768
```

14

will print:

    8000>

2.8 Logical Operations

In addition to the arithmetic operations already described, ATOM 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. Note that the '▢' symbol is obtained on the keyboard by typing 'shift \', and it will appear on the display as an inverted '\'.

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

Operands A 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 ATOM 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 ATOM ROM).

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

    ?A=13

For example, to change the contents of the memory location corresponding to the top left-hand corner of the screen type:

?#8000=127

and a white block will appear in the top left of the screen (see section 18.5 for an explanation).
        As another useful example try:

        ?#E1=0

which will turn the cursor off. To turn the cursor back on again type:

        ?#E1=#80

3 Planning a Program

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

The Atom 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 'stir', 'bake', with certain objects such as 'flour', 'eggs':
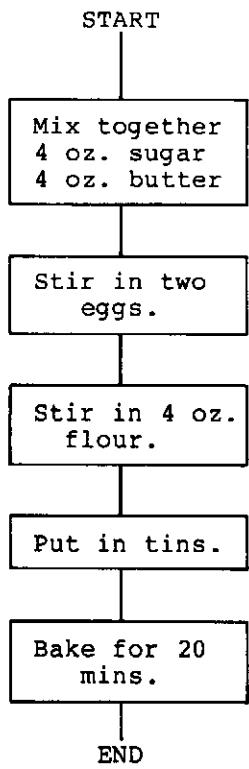
> Recipe 1. Sponge Cake
> 1. Mix together 4 oz. sugar and 4 oz. butter.
> 2. Stir in 2 eggs.
> 3. Stir in 4 oz. flour.
> 4. Put into tins.
> 5. Bake for 20 mins. at Mark 4.
> 6. Remove from oven and eat.
> 7. END

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 in BASIC or Assembler 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:

```
                        START
                          |
            +-------------------------+
            |  Mix together           |
            |  4 oz. sugar            |
            |  4 oz. butter           |
            +-------------------------+
                          |
            +-------------------------+
            |  Stir in two            |
            |     eggs.               |
            +-------------------------+
                          |
            +-------------------------+
            |  Stir in 4 oz.          |
            |     flour.              |
            +-------------------------+
                          |
            +-------------------------+
            |  Put in tins.           |
            +-------------------------+
                          |
            +-------------------------+
            |  Bake for 20            |
            |     mins.               |
            +-------------------------+
                          |
                        END
```
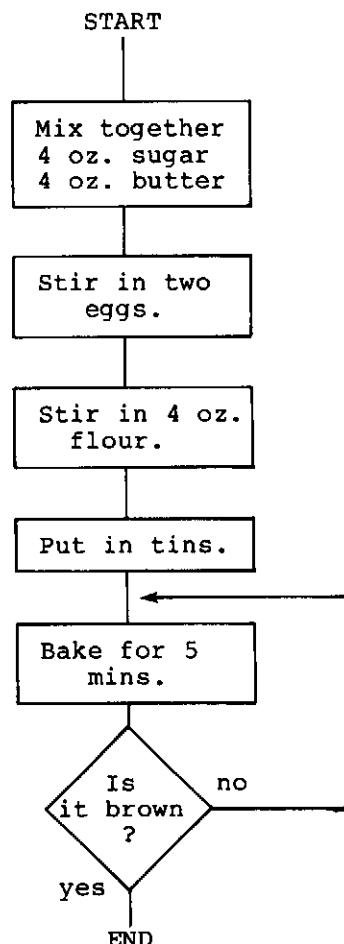
3.2 Decisions

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:

START

```
        |
 ┌──────────────┐
 │ Mix together │
 │ 4 oz. sugar  │
 │ 4 oz. butter │
 └──────────────┘
        |
 ┌──────────────┐
 │ Stir in two  │
 │    eggs.     │
 └──────────────┘
        |
 ┌──────────────┐
 │ Stir in 4 oz.│
 │    flour.    │
 └──────────────┘
        |
 ┌──────────────┐
 │ Put in tins. │
 └──────────────┘
        |
 ┌──────────────┐
 │  Bake for 5  │ ◄──────┐
 │    mins.     │        │
 └──────────────┘        │
        |                │
      ╱   ╲              │
     ╱ Is  ╲    no       │
    ╱it brown╲───────────┘
    ╲   ?   ╱
     ╲     ╱
      ╲   ╱
   yes  |
       END
```

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.
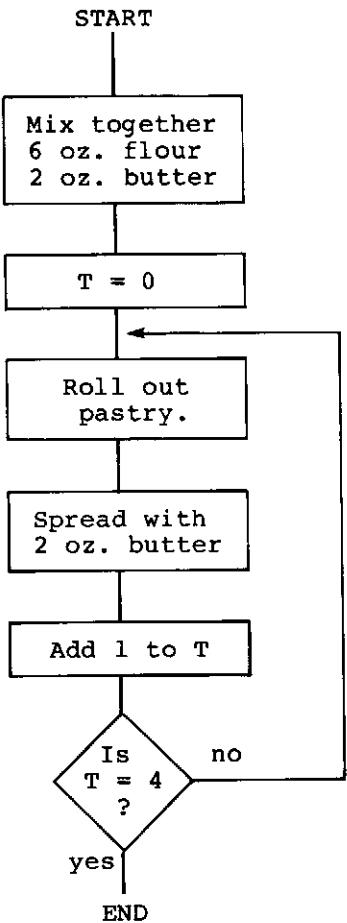
3.3 Counting

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:

START

```
┌─────────────────┐
│ Mix together    │
│ 6 oz. flour     │
│ 2 oz. butter    │
└─────────────────┘

┌─────────────────┐
│    T = 0        │
└─────────────────┘

┌─────────────────┐
│  Roll out       │
│   pastry.       │
└─────────────────┘

┌─────────────────┐
│ Spread with     │
│ 2 oz. butter    │
└─────────────────┘

┌─────────────────┐
│  Add 1 to T     │
└─────────────────┘

      ╱╲
     ╱  ╲   no
    ╱ Is ╲
    ╲T = 4╱
     ╲ ? ╱
      ╲╱
   yes │

     END
```

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

To perform an operation several times in a BASIC or Assembler program an identical method can be used; a counter, such as T, is used to count the number of operations and the counter is tested each time to determine whether enough operations have been completed.

3.4 Subroutines

A recipe may include a reference to another recipe. For example, a 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.
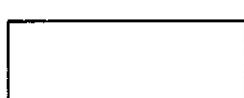
Note that, in order not to lose his place, the cook needed a marker to insert in the recipe book so that he should know where to return to at the end of the subroutine. In BASIC or assembler programs the computer keeps a record of where you were when you call a subroutine, and returns you there automatically 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
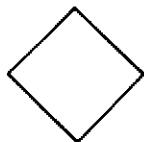
Before writing a program in BASIC or Assembler it is a good idea to express the problem in one of the forms used in this chapter:

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 in BASIC or Assembler will be relatively easy.

4 Writing a BASIC Program

Commands and statements typed after the ATOM's prompt are executed immediately, as we have seen in Chapter 3. However if you start the line with a number, the line is not executed but stored as text in the ATOM's memory.

4.1 RUN

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

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

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
```

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 Farenheit to Celsius (Centigrade), giving the answer to the nearest degree:

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

The value, in Farenheit, 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 #8202 if you have not entered a program (or have just typed NEW) on the unexpanded ATOM, and #2902 on an expanded ATOM.

    PRINT TOP-#8200

is a useful way of finding out how many bytes are used up by a program; on an unexpanded ATOM there is a total of 512 bytes for programs.

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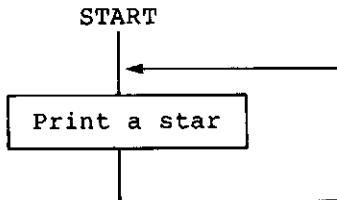
It is possible to create programs which will never stop; see the following example in section 4.6. The escape key 'ESC' at the top left of the keyboard will stop any BASIC program and return control to the '>' prompt.

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

ATOM 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, which are obtained on the ATOM by typing the letter with the shift key held down. Labels appear on the VDU as upper-case inverted letters, so they are very easily identified in programs. For typographical convenience labels will be represented as lower-case letters in this manual.

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

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

Note that there must be no spaces between the line number and the label.

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. To demonstrate this, enter the following program which generates a tone of 187 Hz in the loudspeaker:

```
10 P=#B002
20a ?P=?P:4; GOTO a
```

This program works as follows: P is the location corresponding to the input/output port, and exclusive-ORing this location with 4 will change the output line connected to the loudspeaker. The frequency generated implies that the statements on line 20 are executed in about 2.5 milliseconds (twice per cycle).

25

Try removing the label and rewrite the program as follows:

```
10 P=#B002
20 ?P=?P:4; GOTO 20
```

The GOTO statement is now slightly slower, and the tone generated will have the lower frequency of 144 Hz. The highest frequency that can be generated by a BASIC program is 322 Hz, as follows:

```
10 REM 322 Hz
20 P=#B002
30 FOR Z=0 TO 10000000 STEP 4;?P=Z;N.
```
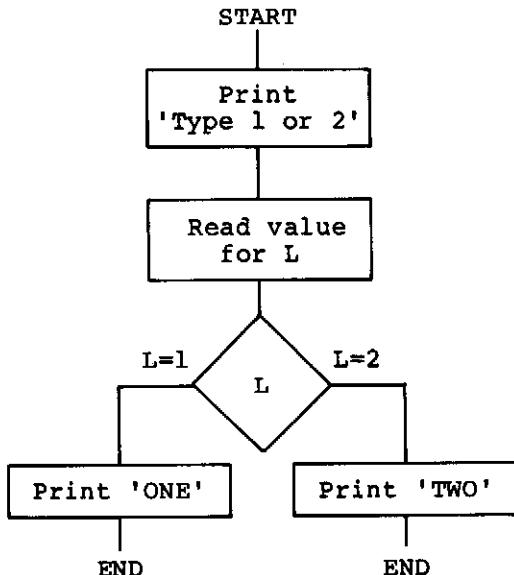
To play tunes you will need to use an assembler program; see Section 15.4.

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:

```
                      START
                        |
              +-------------------+
              |      Print        |
              |  'Type 1 or 2'    |
              +-------------------+
                        |
              +-------------------+
              |    Read value     |
              |      for L        |
              +-------------------+
                        |
                       / \
                      /   \
            L=1      /  L  \      L=2
             +------<       >------+
             |       \     /       |
             |        \   /        |
             |         \ /         |
   +------------------+   +------------------+
   |  Print 'ONE'     |   |  Print 'TWO'     |
   +------------------+   +------------------+
             |                     |
            END                   END
```

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

26

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
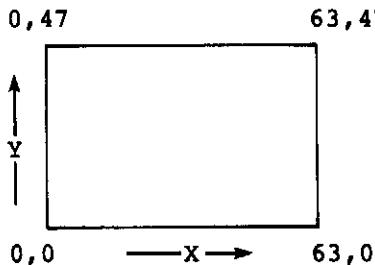
Sample runs:

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

4.7 Graphics

The ATOM has no less than 9 different graphics modes available from
BASIC. This section provides a brief introduction to graphics mode 0,
the lowest resolution mode, which is available on the unexpanded ATOM.
With more memory added to the ATOM the other graphics modes are
available, and these are explained in Chapter 11. A special feature of
mode 0 is that it is possible to mix graphics with any of the ATOM's
characters.

Graphics treats the screen as a piece of graph paper on which it
is possible to draw lines and plot points. Points on the screen are
called 'picture elements' or 'pixels' for short, because they are
actually small squares. Each pixel on the screen is specified by its
coordinates in the two directions, horizontal and vertical, and these
coordinates will be referred to as X and Y respectively. The graphics
screen is labelled as follows in mode 0:

27

4.7.1 CLEAR

To prepare the screen for graphics the statement CLEAR is used. It is followed by the graphics mode number. On the unexpanded ATOM the only legal option is:

        CLEAR 0

4.7.2 MOVE

Any point on the screen can be specified by moving the 'graphics cursor' to that point with the MOVE statement. The graphics cursor does not show on the screen, and it is different from the ordinary cursor which is visible in character mode. The format of the statement is:

        MOVE X,Y

where X and Y can be numbers, or arbitrary expressions provided they are enclosed in brackets. For example, to move the graphics cursor to the origin, X=0 Y=0, type:

        MOVE 0,0

The MOVE statement will normally be the first graphics statement of any program.

4.7.3 DRAW

The DRAW statement will plot a line anywhere on the screen. The line starts from the position of the graphics cursor, and ends at the point specified in the statement, and tie graphics cursor will be moved to that point. For example:

        DRAW 63,47

will draw a line to the top right-hand corner of the screen, and leave the graphics cursor at that point. It is quite legal, and safe, to draw off the screen; the line will just not appear.

4.7.4 Example

The following simple program will draw a rectangle, rotated by the amount entered for R. Try typing in numbers between 0 and 47 for R:

```
10 REM Rotating Rectangle
20 X=63; Y=47
30 INPUT R
40 CLEAR 0
50 MOVE R,0
60 DRAW X,R; DRAW (X-R),Y
70 DRAW 0,(Y-R); DRAW R,0
80 GOTO 30
```

4.7.5 Plotting Points

One way of plotting a single point at X,Y on the screen is to write:

        MOVE X,Y; DRAW X,Y

A more elegant way is given in Section 11.3.

4.8 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.

4.8.1 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
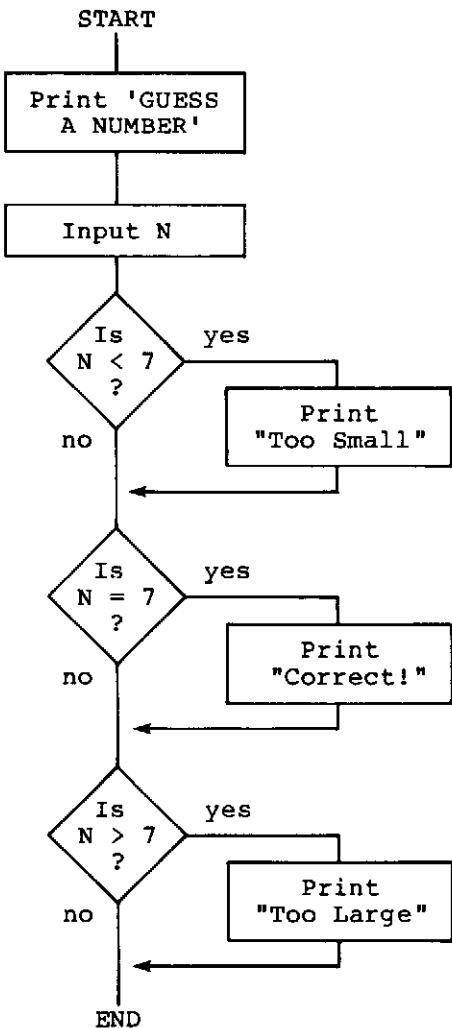
where each operator on the left is the opposite of the operator on the
right.
    The expressions on each side of the relational operators can be as
complicated as required, and the order is unimportant. There is no
need to put brackets around the expressions.
    For example, the following program prints 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:

29

```
                          START
                            |
              ┌─────────────────────────┐
              │   Print 'GUESS           │
              │    A NUMBER'             │
              └─────────────────────────┘
                            |
              ┌─────────────────────────┐
              │   Input N                │
              └─────────────────────────┘
                            |
                         ╱  Is  ╲    yes
                        ╱  N < 7 ╲──────────┐
                        ╲    ?   ╱          │
                         ╲      ╱    ┌──────────────┐
                         no │         │   Print      │
                            │         │ "Too Small"  │
                            │◄────────└──────────────┘
                            |
                         ╱  Is  ╲    yes
                        ╱  N = 7 ╲──────────┐
                        ╲    ?   ╱          │
                         ╲      ╱    ┌──────────────┐
                         no │         │   Print      │
                            │         │ "Correct!"   │
                            │◄────────└──────────────┘
                            |
                         ╱  Is  ╲    yes
                        ╱  N > 7 ╲──────────┐
                        ╲    ?   ╱          │
                         ╲      ╱    ┌──────────────┐
                         no │         │   Print      │
                            │         │ "Too Large"  │
                            │◄────────└──────────────┘
                            |
                          END
```

4.8.2 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'.

4.8.3 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

30

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 40
```

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

4.9 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 1 and 0 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".
  A logical variable can also be set to the value of a condition:

```
A=(L=100)
```

This statement will set A to 'true' if L is 100, and to 'false'
otherwise. The condition must be placed in brackets as shown.

4.10 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.

31

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 IF I<256 GOTO 20
40 END
```

4.10.1 Cubic Curve

Perhaps a more useful example is the following program, which calculates the value of:

$$x^3 - 600x$$

for 64 values of x and plots a graph of the resulting curve:

```
 1 REM Cubic Curve
10 CLEAR 0
20 MOVE 0,24; DRAW 63,24
30 MOVE 32,0; DRAW 32,47
40 MOVE -1,-1
50 X=-33
55 Y=(X*X*X-600*X)/400
60 DRAW (32+X),(24+Y)
70 X=X+1
80 IF X<33 THEN GOTO 55
90 END
```

Description of Program:
10	Use graphics mode 0
20-30	Draw axes
40	Move graphics cursor off screen
50-80	Plot curve for values of X from -32 to 32
55	Equation to be evaluated divided by 400 to bring the interesting part of the cubic curve into range
60	Draw to next point, with origin at (32,24).

Program size: 190 bytes

5 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 (half a screen full). This iterative loop is such a frequently-used operation in BASIC that all BASICs provide a special pair of statements for this purpose, and ATOM 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:

33

```
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 ommitted 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
```

5.1.2 Graph Plotting Using FOR...NEXT

The FOR...NEXT loop is extremely useful when plotting graphs using the
ATOM's graphics facilities. Try rewriting the Cubic Curve program of
Section [7.8] using a FOR...NEXT loop. use 4.10.1.
    The following curve-stitching program is quite fun, especially in
the higher graphics modes. It simulates the curves produced by
stitching with threads stretched between two lines of holes in a
square of cardboard. The curve produced as the envelope of all the
threads is a parabola:

```
 1 REM Curve Stitching in a Square
10 V=46
20 INPUT Q
30 CLEAR 0
40 FOR Z=0 TO V STEP Q; Y=V-Z
50    MOVE 0,Z; DRAW Y,0
60    MOVE Y,V; DRAW V,Z
70 NEXT Z
80 END
```

The value of Q typed in should be between 2 and 9 for best results; V
determines the size of the square that is drawn. The program works
best when V is a multiple of Q.

5.2 DO...UNTIL Loops

ATOM 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.

34

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 coprime the GCD will be 1.

```
  1 REM Greatest Common Divisor
 80 INPUT A,B
 90 DO A=A%B
100 IFABS(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

Program size: 137 bytes

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                115 I7 Q>65535 THEN Q= 65535
```

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

Program size: 118 bytes

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
 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
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.

36

5.4 WAIT Statement

ATOM BASIC includes an accurate timing facility, derived from the main CPU clock. To understand the operation of the WAIT statement, imagine that the ATOM contains a clock which 'ticks' sixty times a second. The WAIT statement causes execution to stop until the next clock tick. Thus it automatically synchronises the program to an accurate time.

The WAIT statement makes it a simple matter to write programs to give any required delay. For example, the following program gives a delay of 10 seconds:

```
FOR N=1 TO 10*60; WAIT; NEXT N
```

You are perhaps wondering why WAIT does not just give a delay of 1/60 second, rather than waiting for the next clock tick. The reason is that if only a delay function were provided, you would have to know exactly how long the other statements in the loop took to execute if you wanted accurate timing. In fact, with the WAIT function, all you need to do is to ensure that the statements in the loop take less than 1/60th. of a second, so as not to miss the next tick.

5.4.1 Digital Clock

The following digital clock displays the time as six digits in the top left-hand corner of the screen.

```
10 REM Digital Clock
20 INPUT "TIME" H,M,S
30 PRINT $12; ?#E1=0
40 T=((H*100)+M)*100+S
50 DO FOR S=1 TO 55; WAIT; NEXT S
60 PRINT $30,T; T=T+1
70 IF T%100=60 THEN T=T+40
80 IF T%10000=6000 THEN T=T+4000
90 UNTIL 0
```

Description of Program:
20	Input the starting time
30	Clear screen; turn off cursor
40	Set up time as 6-digit number
50	Use up rest of a second
60	Print T in top left-hand corner of screen
70-80	Update minutes and hours

Variables:
H - Hours
M - Minutes
S - Seconds
T - Six-digit number representing time

Program size: 216 bytes

To turn the cursor back on after running this program type a form-feed; i.e. CTRL-L.

5.4.2 Reaction Timer

The following reaction-timer program uses WAIT to calculate your reaction time, and prints out the time in centiseconds (i.e. hundredths of a second) to the nearest 2 centiseconds. It blanks the screen, and then, after a random delay, displays a dot at a random place on the screen. When you see the dot you should press the SHIFT key as quickly as possible; the program will then display your reaction time.

```
  1 REM Reaction Timer
 10 CLEAR 0
 20 X=ABS(RND)%64; Y=ABS(RND)%48
 30 FOR N=1 TO ABS(RND)%600+300
 40 WAIT; NEXT N              35 If ?#B001<>#FF Print
 50 MOVE X,Y; DRAW X,Y                "CHEAT!"; G.120
 60 T=0
 70 DO T=T+1; WAIT
 80 UNTIL ?#B001<>#FF
 90 PRINT "REACTION TIME ="
100 PRINT T*10/6, " CSEC." '
110 IF T>18 PRINT "WAKE UP!" '
120 END
```

Description of Program:

20	Choose random X,Y coordinates for point on screen.
30-40	Wait for random time between 6 and 9 seconds.
50	Plot point at X,Y
60-70	Count sixtieths of a second
80	#B001 is the address of the input port to which the SHIFT key is connected; the contents of this location are #FF until the SHIFT key is pressed.
90-100	Print reaction time converted to centiseconds.
110	If appalling reactions, print message.

Variables:
N - Counter for random delay
T - counter in sixtieths of a second for reaction time
X,Y - random coordinates for point on screen.

Program size: 273 bytes

6 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

39

```
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 expession 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
200yY=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

Program size - 466 bytes

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 ATOM 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

100pREM 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.

6.4.1 Tower of Hanoi Problem

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

42

one disc at a time, with the restriction that no disc may be placed on top of a smaller disc. The number of moves required rises rapidly with the number of discs used; the problem was classically described with 64 discs, and moving one disc 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 discs from peg A to peg B:
1. Move F-1 discs to peg C.
2. Move bottom disc to peg B.
3. Move F-1 discs 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 15 discs, and displays the piles of discs at every stage in the solution:

```
   1 REM Tower of Hanoi
  10 PRINT$12
  20 A=TOP;D=A+4
  40 V=-3;W=-1
  60 !D=#1020300;!A=0
  70 INPUT"NUMBER OF DISCS "F
  80 A?1=F;?D=F
  85 N=64/3
  90 CLEAR0
 100 FORQ=1TOF;MOVE(F-Q),(2*(F-Q));PLOT1,(2*Q-1),0;NEXT
 110 GOSUBh;END
1000hIF?D=0 RETURN
1010 D!4=!D-1;D?6=D?1;D?5=D?2;D=D+4;GOSUBh
1020 MOVE(F-D?-4+D?V*N-N),(D?V?A*2);PLOT1,(D?-4*2-1),0
1030 MOVE(D?W*N-N),(D?W?A*2-2);PLOT3,(F+D?-4),0
1040 A?(D?W)=A?(D?W)+W;A?(D?V)=A?(D?V)-W
1050 D?3=D?-2;D?2=D?W;D?1=D?V;GOSUBh
1060 D=D-4;RETURN
```

Description of Program:
100	Draw starting pile of discs
110	Subroutine h is called recursively to move the number of discs specified in ?D.
1000	h: Subroutine to move ?D discs
1010	Recursive call to move ?D-1 discs
1020	Draw new disc on screen
1030	Remove old disc from screen
1040	Set up array A
1050	Recursive call to put back ?D-1 discs

Variables:
A?N - Number of discs on pile N
D - Stack pointer
?D - How many discs to transfer
D?1 - Destination Pile
D?2 - Intermediate pile
D?3 - Source pile
F - Total number of discs
N - One third of screen width
V - Constant
W - Constant

Program size: 461 bytes
Stack usage: (4 * number of discs) bytes

43

6.4.2 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 draws a diagram of the board to show each solution as it is found. The program uses many abbreviations to keep it small enough to fit on an unexpanded ATOM (for a complete explanation of these abbreviations, see section 10.1):

```
   1 REM Eight Queens
  30 C=0;D=TOP;E=D+3;A=D+27;!D=0
  60 @=0;GOS.t;P.$13"THERE ARE "C" SOLUTIONS"';END
 100tIF?D=#FF C=C+1;GOTOd
 110 ?A=(?D\D?1\D?2):#FF
 120lIF?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
 200dCLEAR0;FORZ=0TO32S.4;MOVE0,Z;DRAW31,Z;MOVEZ,0;DRAWZ,32;N.
 210 Q=0;FORZ=3TO34STEP3;P=TOP?Z-Q;S=-2;DOS=S+4;P=P/2;UNTILP=0   use 24
 220 Q=TOP?Z;PLOT13,(Z/3*4-2),S;N.;P.$30 C;R.
```

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 position and see if done.
200	d: Have solution, display board matrix.
210	Plot pixels at positions of queens.
220	Print the solution number at screen top and end recursion.

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

Program size: 440 bytes
Vectors: 30 bytes
Total storage: 470 bytes.

7 Arrays and 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:

$$t_1, t_2, t_3, \ldots\ldots t_{12}.$$

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 ATOM BASIC, and these are called 'arrays'. Each array consists of an array '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 array can act as a completely independent variable, capable of holding a value just like the variables A to Z. The members of an array are called the array 'elements'. The total number of possible elements depends on how the array was set up; in the above example there were twelve elements, with subscripts from 1 to 12.

In addition to the standard type of array, ATOM BASIC provides two other types of array 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 arrays. Word vectors use the same amount of storage as arrays, but can be manipulated in a more flexible manner.

7.1 Arrays – AA to ZZ

The array in ATOM BASIC consists of a pair of identical letters followed by the subscript in brackets: for example, EE(3). Each element in this type of array can contain numbers as large as the simple variables A to Z, namely, between about −2000 million and 2000 million.

Before an array can be used space must be reserved for it by a DIM, or 'dimension', statement which tells BASIC how large the array is to be. For example, to reserve space for an array called AA with the five elements AA(0), AA(1), AA(2), AA(3), and AA(4), the statement would be:

        DIM AA(4)

The DIM statement allocates space for arrays starting at the first free memory location after the program text. If this were the first DIM statement encountered in the program the element AA(0) would be at

TOP, above the program text:

?	?	?	?	?

TOP:
```
     ^      ^      ^      ^      ^
   AA(0)  AA(1)  AA(2)  AA(3)  AA(4)
```

The question marks represent unspecified values, depending on what the array contained when it was dimensioned. If now another array were dimensioned with the statement:

    DIM BB(3)

space for the array BB would be reserved immediately following on from AA.

Array elements can appear in expressions, and be assigned to, just like the simple variables A to Z. For example, to make the value of AA(3) become 776 we would execute:

    AA(3)=776

Then we could execute:

    AA(1)=AA(3)*2
    AA(0)=AA(3)-6

and so on. The resulting array would now be:

TOP:

770	1552	?	776	?

```
     ^      ^      ^      ^      ^
   AA(0)   AA(1)  AA(2)  AA(3)  AA(4)
```

There are two places in BASIC programs where array 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 Histogram

The following program illustrates the use of arrays to plot a histogram of the temperature over the twelve months of the year. The temperatures, assumed to be in the range 0 to 100, are first entered in and are stored in the array TT(1..12).

```
  1 REM Histogram
 10 DIM TT(12)
 20 FOR J=1 TO 12;INPUT K
 30 TT(J)=K; NEXT J
 40 PRINT $12; CLEAR 0; @=5
 50 MOVE 60,12; DRAW 12,12
 60 DRAW 12,42
 70 FOR N=11 TO 0 STEP -1
 80 IF N=7 PRINT "TEMP."
 90 IF N%2=0 PRINT N*10
100 PRINT';NEXT N
110 PRINT "       JAN MAR MAY JUL SEP NOV"'
120 PRINT "         FEB APR JUN AUG OCT DEC"'
130 PRINT "                  MONTH"'
```

46

```
      140 FOR N=1 TO 12; J=11+4*N
      150 MOVE J,12; DRAW J,(TT(N)*3/10+12)
      160 NEXT N; END
```

Description of Program:
```
20-30       Input 12 values
40          Clear screen
50-60       Draw axes
70-100      Label vertical axis
110-130     Label horizontal axis
140-160     Plot histogram bars
```

Program size: 415 bytes
Array storage: 52 bytes

7.1.2 Sorting Program

The following program illustrates the use of arrays 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 DIM AA(20)
   10 FOR N=1 TO 20; INPUT J
   20 AA(N)=J; NEXT N
   30 GOSUB s       N=20; GOSUB s
   40 FOR N=1 TO 20; PRINT AA(N)'
   50 NEXT N
   60 END
  100sM=N
  110 DO M=(M+2)/3
  120    FOR I=M+1 TO N
  130       FOR J=I TO M+1 STEP -M
  140          IF AA(J)>=AA(J-M) GOTO b
  150          T=AA(J); AA(J)=AA(J-M); AA(J-M)=T
  160       NEXT J
  170b   NEXT I
  180 UNTIL M=1; RETURN
```

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

Variables:
AA(1..20) - Array to hold numbers
I,J - Loop counters
N - Number of elements in array AA
M - Subset step size
T - Temporary variable

Program size: 332 bytes
Array storage: 84 bytes

7.1.3 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 an array AA, one digit per array element. Every power of 2 is obtained from the previous one by multiplying every element in the array by 2, and propagating a carry when any element becomes more than one digit.

```
  5 REM Powers of Two
 10 DIM AA(31)
 20 @=1; P=0
 30 AA(0)=1
 40 FOR J=1 TO [39] 31
 50    AA(J)=0
 60 NEXT J
 70 DO J=31
 80    DO J=J-1; UNTIL AA(J)<>0
 85    PRINT'"2^" P "="
 90    FOR K=J TO 0 STEP -1
 94      PRINT AA(K)
 96    NEXT K
110    C=0
120    FOR J=0 TO 31
130      A=AA(J)*2+C
140      C=A/10
150      AA(J)=A%10
160    NEXT J
170    P=P+1
180 UNTIL AA(31)<>0
190 END
```

Description of Program:

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

Variables:
AA - Array of digits; one digit per element
C - Decimal carry from one digit to next
J - Digit counter
K - Digit counter
P - Power being evaluated

Program size: 356 bytes
Array usage: .124 bytes
Total memory: 480 bytes.

7.1.4 Digital Waveform Processing

The following program uses a 256-element array to store a waveform which can be low-pass filtered, converted to a square wave, or printed out.

```
  1 REM Digital Waveform Processing
  5 DIM AA(255)
 10 H=2000
 15 CLEAR4
 23 GOS.s; GOS.q
 25 Z=160; GOS.p
 28 GOS.1
 30 Z=96; GOS.p
 32 GOS.s
 34 Z=32; GOS.p
```

```
      90 END
   1000pREM Plot Waveform
   1005 MOVE 0,96
   1010 FOR N=0 TO 255
   1020 PLOT13,N,(Z+AA(N)/H)
   1030 NEXT N
   1040 RETURN
   2000sREM Make Sine Wave
   2010 S=0;C=40000
   2020 FOR N=0 TO 255
   2030 AA(N)=-S
   2040 C=C-S/10
   2050 S=S+C/10
   2060 NEXT N
   2070 RETURN
   3000qREM Make Square Wave
   3010 FOR N=0 TO 255
   3020 IF AA(N)>=0 AA(N)=40000
   3030 IF AA(N)<0 AA(N)=-40000
   3035 NEXT N
   3040 RETURN
   4000lREM Low Pass Filter
   4010 B=0
   4020 FOR N=0 TO 255
   4030 B=AA(N)*360/1000+B*697/1000
   4040 AA(N)=B; NEXT N
   4050 RETURN
```

Description of Program:
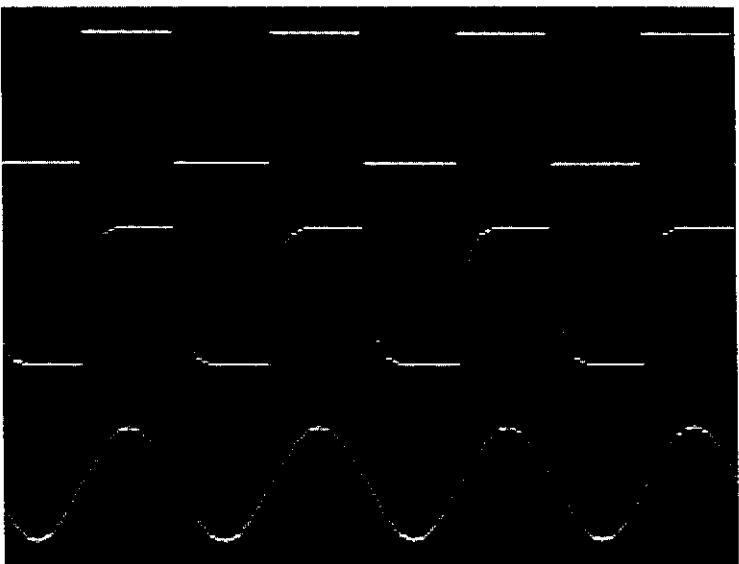
23	Calculate a square wave
25	Plot it at top of screen
28	Low-pass filter the square wave
30	Plot it in centre of screen
32	Calculate a sine wave
34	Plot it at bottom of screen
1000-1040	p: Plots waveform
2000-2070	s: Calculates a sine wave.
3000-3040	q: Squares-up the waveform
4000-4050	l: Low-pass filters the waveform

Variables:
AA(0...255) - Array of points, values between -40000 and 40000.
B - Previous value for low-pass filter
C - Cosine of waveform
H - Scaling factor for plotting waveforms
N - Counter
S - Sine of waveform
Z - Vertical coordinate for centre of waveform.

Program size: 564 bytes.
Array storage: 1024 bytes
Total memory: 1588 bytes

Sample plot:

7.1.5 Subscript Checking

Many BASIC interpreters perform extensive checking whenever an array element is used in a program. For example, if an array were dimensioned:

    DIM RR(10)

then every time the array were used the subscript would be checked to make sure that it was both 0 or greater, and 10 or less. Obviously these two checks slow down the execution of a program, and so in ATOM BASIC only the first check is performed, so that only positive subscripts are allowed. It is left to the programmer to ensure that subscripts do not go out of range. Assigning to an array whose subscript is out of range will change the values of other arrays, or strings, dimensioned after that array.

If required, the programmer can easily add array subscript checking; for example, if the array assignment were:

    RR(A)=35

the statement:

    IF A>10 THEN ERROR

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

7.1.6 Multi-Dimensional Arrays

The standard types of array in ATOM 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 'array'.

Sometimes it is convenient to make each element of an array 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 arrays 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).

ATOM BASIC does not have a direct representation for two-dimensional (or higher dimension) arrays, but they are easily represented using the single-dimension arrays AA to ZZ as described in the following sections.

7.1.7 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 matrix can be dimensioned, using a one-dimensional array, with the DIM statement:

        DIM XX(M*N-1)

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

        XX(A*N+B)

In the earlier example the array had dimensions 3 x 6 and so would be dimensioned:

        DIM XX(17)

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

        XX(16)

7.1.8 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)
```

51

```
 65 DIM AA(I),CC(J),II(N)
 70 @=0;FOR I=1TON;FOR J=1TO N+1
 80 PRINT"C("I","J")=";INPUT C
 90 CC((I-1)*(N+1)+J)=C;NEXT J;NEXT I
100 L=N+1;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=1TON;GOSUB c
125 PRINT"X("L")=   "
130 A=M*D;B=E;DO A=A%B
140 IF ABS(B)>ABS(A) THEN T=B;B=A;A=T
150 UNTIL B=0;A=ABS(A)
151 P.(M*D)/A;IF E/A<>1 PRINT"/"E/A
155 M=-M;PRINT';NEXT L;END
160cFOR I=1TON;FOR J=1TON;K=I*N-N+J
170 IF J<L AA(K)=CC(K+I-1)
180 IF J>=L AA(K)=CC(K+I)
190 NEXT J;NEXT I
200dD=0;F=1;S=1
210 FOR J=1TON;II(J)=J;F=F*J;NEXT J
215 GOSUB f
220 FOR H=2TOF;GOSUB e;NEXT H;RETURN
230eI=N-1;J=N
240gIF II(I)>=II(I+1) I=I-1;GOTO g
250hIF II(I)>=II(J) J=J-1;GOTO h
260 GOSUB i;I=I+1;J=N;IF I=J GOTO f
270 DO GOSUB i;I=I+1;J=J-1;UNTIL I>=J
280fP=1;FOR K=1TON;P=P*AA(N*K-N+II(K))
290 NEXT K;D=D+S*P;RETURN
300iK=II(I);II(I)=II(J);II(J)=K
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:
```
AA(1...N*N) - Matrix
CC(1...N*N+N) - Matrix of coefficients
S - Signature of permutation.
```

Program Size: 932 bytes.
Variable Space: (2*(N*N+N)+3)*4 bytes

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
```

52

SOLUTION:
X(1)= 1/3
X(2)= -2/3

7.2 Byte Vectors Using '?'

It is sometimes wasteful of memory to allocate space for numbers over
the range provided by word arrays so a second type of array
representation is provided which only allocates one byte, rather than
four bytes, for each array element. These are referred to as 'byte
vectors', and they are in effect one-dimensional arrays. Byte vectors
differ from word arrays in that they use one of the simple variables A
to Z to hold the 'base' address of the array; i.e. the address in
memory where the zeroth element of the array will reside. The array
subscripts are simply 'offsets' from this base address; i.e. the
subscript is added to the base address to give the address of the
array element. The vector elements are written as:

    A?0, A?1, A?2, ... etc

where A is the simple variable used to hold the base address of the
vector, and the number following the question mark is the subscript.
    Note that the zeroth element of a byte vector, A?0, is equivalent
to ?A, the contents of the location with address A. Similarly A?1 is
equivalent to ?(A+1), and so on.
    Byte vectors can be dimensioned by the DIM statement; for example,
to dimension a byte vector with elements from A?0 to A?11 the
statement would be:

    DIM A(11)

Because the DIM statement dimensions arrays and vectors from the end
of the program onwards, the above DIM statement is equivalent to:

    T=TOP; A=T; T=T+12

where T is a variable used to keep track of the next free memory
location. Note that space for vectors can be reserved anywhere in
memory, as distinct from arrays which can only be assigned from TOP
onwards using the DIM statement. For example, to assign space for a
vector S corresponding to the screen memory, simply execute:

    S=#8000

Elements of this vector would then correspond to locations on the
screen; e.g. S?31 is the location corresponding to the top right-hand
corner of the screen.
    Each element of a byte array can hold a positive number between 0
and 255, or a single character. Strings are simply byte vectors
containing characters. Note that the subscript of a byte array can be
an arbitrary expression provided that it is enclosed in brackets.

7.3 Word Vectors Using '!'

A second representation for word arrays is provided in ATOM BASIC
using the word indirection operator '!', and is mentioned here for
completeness, although for simple problems involving arrays the word
arrays AA to ZZ are probably more convenient. Word vectors are similar
to the byte vectors already described, but each element of the vector
consists of a word rather than a byte. Each element consists of the
base address variable separated from the subscript, or offset, by a
'pling' '!'. Note that the subscript should be incremented by 4 for
each element, since each element is offset 4 bytes from the previous
one. For example, a word vector W might have the six elements:

53

W!0, W!4, W!8, W!12, W!16, W!20.

Space can be dimensioned for word vectors by using the DIM statement, and allowing 4 bytes per element; for example, to provide storage for the above 6 elements, execute:

    DIM W(23)

Note that the zeroth element of the vector, W!0, is equivalent to !W.

7.3.1 Prime Numbers

The following program finds all the prime numbers up to 99999. It uses a word vector to store primes already found, and only tests new candidates for divisibility by these numbers:

```
 1 REM Prime Numbers
10 @=8;S=4;Z=0;J=TOP;G=J;!G=3;P=G+S
20 FORT=3TO99999STEP2
30cIFT%!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.

Program size: 155 bytes
Vector: as required.

7.3.2 Call by Reference

A major advantage of word vectors over the word arrays 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

100pREM 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
```

54

In this example subroutine p can be used to print any array 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 array, rather than the actual values in the array.

7.3.3 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=#3BFF          #3BFF
 10 H=(T-TOP)/3; DIM P(H),S(H),D(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";RUN
 60 M=Y;N=X;GOSUBp
 70 PRINT Y"^"X"="P!!P;IF!P<8 RUN
 90 F.L=!P-4TO4STEP-4
 95 IFL!P<1000P.0          < 100
100 IFL!P<100P.0           < 10
110 IFL!P<10P.0            < 1
120 P.L!P;N.;RUN
140*
200pJ=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.
280*
300m!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.
380*
400e!R=0;DO!R=!R+4;R!!R=J%H
410 J=J/H;U.J<1;R.
```

Description of Program:

5	Set T to top of lower text space.
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 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

Program size: 733 bytes.
Additional storage: as available.

Sample run:

>RUN

```
        POWER  PROGRAM
    COMPUTES Y^X, WHERE X>0 AND Y>0
    VALUE OF Y?16
    VALUE OF X?64
   16^64=115792089237316195423570985008687907853269984665640564039457
84007913129639936
```

7.3.4 Vectors of Vectors

A second way of representing two-dimensional arrays is possible using the ATOM's indirection operators '?' and '!'; 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 matrix called X with columns 0 to M, and rows 0 to N, the following statements will set up the vector of row addresses:

```
    DIM X(2*N-1)
    FOR J=0 TO N*2 STEP 2; DIM Q(M); X!J=Q; NEXT J
```

A word array is used to hold the base addresses. Q is a variable used to hold the base address temporarily. Now that the vector of row base addresses has been set up, the element with subscripts A,B is:

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

56

8 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 by means of
a DIM (dimension) statement. For example, for a string of up to 10
characters using the variable A the statement would be:

        DIM A(10)

Any number of strings can be dimensioned in one DIM statement.

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 DIM A(10) is to reserve 11 memory
locations in the area of free memory above the text of the BASIC
program, and to put the address of the first of those locations into
A. In other words, A is a pointer to that area of memory. After the
above assignment the contents of those locations are as follows:

A: | A | | S | T | R | I | N | G | ˜ | ? | ? |

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. The DIM statement allocates one extra location to hold this terminator character, although you will not normally be aware of its presence.

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
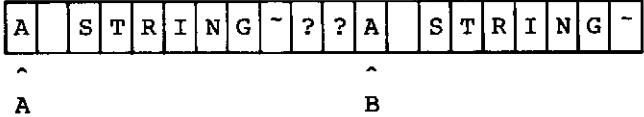
Suppose that a second string is dimensioned as follows:

      DIM B(8)

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:

| A | | S | T | R | I | N | G | ˜ | ? | ? | A | | S | T | R | I | N | G | ˜ |

      ^   ^
      A   B

8.2.5 String Equality

It is possible to test whether two strings are equal with the IF statement. For 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 dimensioned with a length of 64.

8.3 String Functions

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

58

8.3.1 Length of a String – LEN

The LEN function will return the number of characters in the string specified in 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""
```
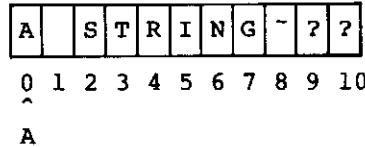
will be equal to 13.

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 invert all the characters in a string which is typed in:

```
 1 REM Invert String
 5 DIM Q(64)
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

Program size:
String storage: up to 64 bytes

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 you may like to
attempt to find the correct decoding of 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
```

60

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 DIM A(10),B(5)
    20 $A="ATOM"
    30 $B="BASIC"
    40 $A+LEN(A)=$B
    50 PRINT $A
    60 END

will print:

ATOMBASIC>

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 DIM A(10),B(5)
    20 $A="ATOMBASIC"
    30 $B=$A+4
    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 DIM A(10)
    20 $A="ATOMBASIC"
    30 $A+5=""; $A=$A+1
    40 END

then string A will have the value "TOMB".

61

8.5 Arrays of Fixed-Length Strings

The arrays AA to ZZ may be used as string variables, thus providing the ability to have arrays of strings. To allocate space for an array of strings the DIM statement can be incorporated into a FOR...NEXT loop. For example, the following program allocates space for 21 strings, AA(0) to AA(20), each capable of holding 10 characters:

```
25 DIM AA(20)
35 FOR N=0 TO 20
40   DIM J(10)
50   AA(N)=J
60 NEXT N
```

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

```
$AA(0)="ZERO"
$AA(10)="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 array.

```
1 REM Day of Week
10  DIM AA(6)
20  FOR N=0 TO 6; DIM B(10); AA(N)=B; NEXT N
30  $AA(0)="SUNDAY"; $AA(1)="MONDAY"
40  $AA(2)="TUESDAY";$AA(3)="WEDNESDAY"
50  $AA(4)="THURSDAY";$AA(5)="FRIDAY"
60  $AA(6)="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 " $AA(ABS(E%7)) ''
140 END
```

Description of Program:

10-20	Allocate space for string array
30-60	Set array elements
70	Input date
80-120	Calculate day
130	Print day of week.

Variables:
$AA(0...6) - String array to hold names of days
B - Temporary variable to hold base address of each string
D - Date in month
E - Expression which, modulo 7, gives day of week.
M - Month
N - Counter
Y - Year in 20th. century.

Program size: 458 bytes.
Array storage: 105 bytes.

Total memory: 563 bytes.
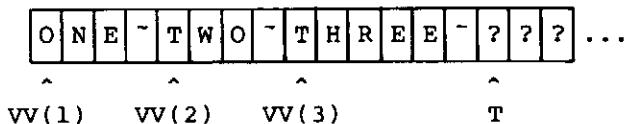
8.6 Arrays of Variable-Length Strings

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 VV(1) to VV(10):

```
10 DIM VV(10),T(-1)
20 FOR N=1 TO 10
30 INPUT $T
40 VV(N)=T
50 T=T+LEN(T)+1
60 NEXT N
70 INPUT "STRING NUMBER",N
80 PRINT $VV(N),'
90 GOTO 70
```

The statement DIM T(-1) sets T 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:

VV(1)   VV(2)   VV(3)   T

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 ATOM 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 DIM A(40); L=CH"t"
25 GOSUB f
30 FOR J=1 TO 20; GOSUB r
40 PRINT $A '
50 NEXT J
60 END
500fREM 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*
600rREM read next entry into A
```

63

```
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*
800tONE,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

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

Program size: 511 bytes
String storage: 41 bytes
Total memory: 552 bytes.

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:

```
800tl,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 following table gives the control codes, characters, and graphics symbols corresponding to the different codes:

64

Hex:	Decimal:	Character Printed:
#00 - #1F	0 - 31	Control codes
#20 - #5F	32 - 95	ASCII characters
#60 - #9F	96 - 159	Inverted ASCII characters
#A0 - #DF	160 - 223	~~White~~ graphics symbols GREY
#E0 - #FF	224 - 255	~~Grey~~ graphics symbols WHITE

Note that only half of the 64 possible grey graphics symbols can be obtained in this way.

The most useful control codes are specified in the following sections; for a full list of control codes see section 18.1.3.

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. Note that the entire screen memory is modified by scrolling; every line is shifted up one line, and the bottom line is filled with spaces.

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 white graphics 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, $(#A0+ABS(RND)%#40), $8
30 UNTIL 0
```

65

9 Reading and Writing Data

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

1. Words i.e. numbers between -2000 million and 2000 million (approximately).
Storage required: 4 bytes
e.g. variables A to Z
     arrays AA(1) ... etc.
     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.q. quoted string "A STRING"
     string variable $A ...etc.

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

The ATOM BASIC functions and statements for cassette input and output are designed to be fully compatible with the disk operating system, should that be added at a later stage. 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 system, the file handle is still required for compatibility.

9.2 Output

To output a word to cassette 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 cassette 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 cassette 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 cassette 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 Find Input and Find Output

The functions FIN (find input) and FOUT (find output) can optionally be called before inputting from, or outputting to, cassette. The functions are called with a null string as the argument, and they return the value 13; when used with a disk system the argument is the file name, and the value returned is the file handle.

The FOUT function is called as follows:

       A=FOUT""

and it will cause the message:

RECORD TAPE

to be printed, and the program will wait for a key to be pressed before continuing execution.

The FIN function is called as follows:

       A=FIN""

and it causes the message:

PLAY TAPE

to be printed, and again the program will wait for a key to be pressed. A dummy variable, such as A in this example, should be used to hold the file handle.

9.4.1 Data on Cassette

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

```
    1 REM Data to Cassette
   10 DIM VV(20)
   20 N=0
   30 DO INPUT J
   40 VV(N)=J; N=N+1
   50 UNTIL J=0 OR N>20
   60 A=FOUT""
```

68

```
      70 BPUT A,(N-1)
      80 FOR M=0 TO N-1
      90 PUT A,VV(M)
     100 NEXT M
     110 END
```

Description of Program:

30-50	Input numbers
60	Warn user to start tape
70	Output number of bytes
80-100	Save values on cassette

Variables:
A - Dummy file handle
J - Temporary variable for values input
M - Counter
N - Counter for number of values
VV(0...20) - Array of numbers

The next program reads the values back in and plots a histogram of the values. The program automatically scales the values if they are too large to fit onto the screen.

```
       1 REM Plot Histogram from Cassette
      10 DIM VV(20)
      20 A=FIN""; N=BGET A
      30 FOR M=0 TO N
      40 VV(M)= GET A
      50 NEXT M
      60 REM X=Maximum, Y=Minimum
      70 X=VV(0); Y=VV(0)
      80 FOR M=1 TO N
      90 IF X<VV(M) THEN X=VV(M)
     100 IF Y>VV(M) THEN Y=VV(M)
     110 NEXT M
     120 S=(X-Y+63)/64
     130 REM Plot Histogram
     135 CLEAR 0
     140 FOR M=0 TO N
     150 MOVE 0,M
     160 DRAW ((VV(M)-Y)/S),M
     170 NEXT M
     180 GOTO 180
```

Description of Program:

20-50	Read values into array
70-110	Find maximum and minimum values in array
120	Calculate scaling factor
140-170	Plot scaled histogram
180	Wait for ESC key.

Variables:
A - Dummy file handle
M - Counter
N - Number of values in array
S - Scale factor for array
VV(0...20) - Array of values
X - Maximum value
Y - Minimum value

9.5 Reading and Writing Speed

When writing data to the cassette 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.

9.6 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 'Animals'; 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 cassette input/output statements to load the database, or tree, from cassette 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 cassette. 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 cassette. 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 F=FIN""
 23 DO UNTIL BGET F=#AA
 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 " $X
 80 GOSUB q
 85 IF Q=4 PRINT "HO-HO";UNTIL 0
 90 DO INPUT"WHAT WERE YOU THINKING OF"$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
    1000qINPUT $T
    1010  IF ?T=CH"Y"THEN Q=4; RETURN
    1020  IF ?T=CH"Q"THEN END
    1030  Q=0; RETURN
    2000j$T=$R; A=1
    2010  DO A=A+1
    2020  V=T?(A+4); $T+A+4=""
    2030  IF $T+A=" IT " UNTIL 1; GOTO k
    2035  T?(A+4)=V
    2040  UNTIL A=LEN T-5
    2100  PRINT"WHAT WOULD THE ANSWER BE"'
    2110  PRINT"FOR " $X
    2120  RETURN
    2150kT?(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  F=FOUT ""
    9112  BPUT F,#AA; WAIT
    9115  PUT F,(T-TOP-1)
    9120  FOR N=TOP TO T-1
    9130  BPUT F, ?N
    9140  NEXT N
    9150  RETURN
```

Description of Program:
```
20-30     Load previous tree
23        Look for start flag
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

Program size: 1254 bytes
Additional storage: as required for 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
>
```

10 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.

10.1 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 ATOM 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 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.

10.1.1 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. The following abbreviations are possible:

Name:	Abbreviation:
ABS	A.
AND	A.
BGET	B.
BPUT	B.
CH	
CLEAR	
COUNT	C.
DIM	
DO	
DRAW	
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.
MOVE	
NEW	N.
NEXT	N.

```
OLD
OR
PLOT
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.
WAIT
```

10.1.2 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 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.

10.1.3 LET

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

    LET A=B

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

10.1.4 THEN

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

10.1.5 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, AA(3) may be written AA3, ABS(RND) may be written ABSRND, but AA(B+2) cannot be abbreviated.

74

10.1.6 Commas

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

10.1.7 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.

10.1.8 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.

10.2 Maximising Execution Speed

ATOM 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, and so that real-time graphics programs are feasible.
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 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 shorter than 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. For example:

```
FOR J=1 TO 10
FOR K=1 TO 10
VV(K)=VV(J)*2+K
NEXT K
NEXT J
```

could be written as:

```
FOR J=1 TO 10
Q=VV(J)*2
FOR K=1 TO 10
VV(K)=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. For 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. For example:

```
FOR N=0 TO 9
DRAW AA(2*N), AA(2*N+1)
NEXT N
```

could be rewritten as the faster:

```
FOR N=0 TO 18 STEP 2
DRAW AA(N),AA(N+1)
NEXT N
```

8. Use word operations rather than byte operations where possible. For example, to clear the graphics screen to white it is faster to execute:

76

```
        FOR N=#8000 TO #9800 STEP 4; !N=-l; NEXT N
```

than the following:

```
        FOR N=#8000 TO #9800; ?N=-l; 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.

11 Advanced Graphics

The ATOM provides nine different graphics modes, up to a resolution of 256x192 in black and white, and 128x192 in four selectable colours. The graphics modes use the BASIC statements PLOT, DRAW, and MOVE in an identical way. All the black-and-white graphics commands are present in the unexpanded ATOM, although extra memory will be required for the higher-resolution graphics modes. Colour plotting requires the addition of an assembler routine, or the COLOUR statement provided in the extension ROM.
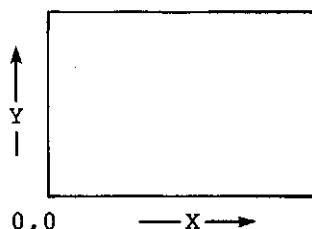
11.1 Graphics Modes

The nine graphics modes are listed below:

Mode:	Resolution: X:	Y:	Memory:
0	64	48	0.5 K
1a	64	64	1 K
1	128	64	1 K
2a	128	64	2 K
2	128	96	1.5 K
3a	128	96	3 K
3	128	192	3 K
4a	128	192	6 K
4	256	192	6 K

11.2 CLEAR

This statement clears the screen and puts it into graphics mode. It is followed by a number, or expression in brackets, to specify the mode. The graphics screen is labelled as follows:

The smallest square which can be plotted on the display is referred to as a 'pixel' (or 'picture element').

11.3 PLOT

The graphics statements include a versatile 'PLOT K,X,Y' statement, the value of K determining whether to draw or move, plot lines or points, whether to set, clear, or invert, and whether to take the parameters X and Y as the absolute screen position, or as a displacement from the last point. The values K, X, and Y can be arbitrarily-complicated expressions.

```
       K:    Function:
       0    Move relative to last position
       1    Draw line in white relative to last position
       2    Invert line relative to last position
       3    Draw line in black relative to last position

       4    Move to absolute position
       5    Draw line in white to absolute position
       6    Invert line to absolute position
       7    Draw line in black to absolute position

       8    Move relative to last position
       9    Plot point in white relative to last position
      10    Invert point relative to last position
      11    Plot point in black relative to last position

      12    Move to absolute position
      13    Plot point in white at absolute position
      14    Invert point at absolute position
      15    Plot point in black at absolute position
```

11.4 DRAW and MOVE

In addition DRAW and MOVE statements are provided as convenient
aliases for drawing a line and moving to an absolute X,Y position.
    MOVE X,Y is equivalent to PLOT 12, X, Y.
    DRAW X,Y is equivalent to PLOT 5, X, Y.

11.4.1 Random Rectangles

The following program illustrates the use of relative plotting using
the PLOT statement, and draws random rectangles on the display. The
program will work in any of the graphics modes.

```
    10 REM Random Rectangles
    13 S=20
    16 Z=1;B=0
    17 W=64;H=48
    18 E=W-S;F=H-S
    20 CLEARB
    30 FORQ=0TO7
    32 MOVE(ABSRND%E),(ABSRND%F)
    35 C=ABSRND%S+1;D=ABSRND%S+1;GOSUBs
    37 NEXTQ;FOR Q=0TO20000;NEXTQ
    38 GOTO20
   100sPLOTZ,C,0
   110 PLOTZ,0,D
   120 PLOTZ,-C,0
   130 PLOTZ,0,-D
   140 RETURN
```

Description of Program:
```
13-18      Set up constants
20         Initialise graphics
30         Draw 41 rectangles
32         Move to random point, leaving margin for size of largest
           rectangle.
35         Choose random rectangle
37         Wait; then repeat.
100-140    s: Draw rectangle.
```

Variables:
C,D - Dimensions of rectangle
E,F - Dimensions of safe part of screen to start drawing rectangle.
H - Screen height
Q - Counter
S - Size of squares
W - Screen width
Z - Plot mode; draw relative.

Program size: 278 bytes

11.5 Advanced Graphics Examples

The following examples are designed for use with the higher-resolution graphics modes, and illustrate some of the applications that are possible using the ATOM's graphics facilities.

11.5.1 The Sierpinski Curve

This curve is of interest to mathematicians because it has the property that it encloses every interior point of a square, and yet it is a closed curve whose area is less than half that of the square. This program draws successive generations to illustrate how the Sierpinski curve, which is the limit of these polygonal drawings, is constructed.

```
   1 REM Sierpinski Curve
  10 INPUT"MODE"O
  15 INPUT"SIZE"K
  20 CLEARO
  30 S=5
  40 J=1
  50 FOR I=1 TO 5
  60 J=J*2;D=K/J/4
  70 X=K-5*D; Y=K-2*D
  80 T=1; MOVE X,Y
  90 X=X+D; A=J; B=J; GOTO s
 100aIF A=J AND B=J GOTO z
 110sP=J; Q=A; R=B
 120vIF P<2 GOTO z
 130 IF P=2 GOSUB o; GOTO a
 140 P=P/2
 150 IF Q<P OR P+1<Q GOTO n
 170 IF R<P OR P+1<R GOTO n
 190 GOSUB c; GOTO a
 200nIF Q>=P THEN Q=Q-P
 210 IF R>=P THEN R=R-P
 220 GOTO v
 230zREM end of loop
 240 FOR N=1 TO 1000;NEXT
 250 CLEARO
 260 NEXT I
 270 END
1000cGOTO(1000+100*T)
1100 X=X+D
1105 PLOTS,X,Y
1110 X=X+D;Y=Y+D;PLOTS,X,Y
1120 Y=Y+D;B=B+1;T=4;RETURN
1200 Y=Y-D
1205 PLOTS,X,Y
1210 X=X+D;Y=Y-D;PLOTS,X,Y
```

81

```
1220  X=X+D;A=A+1;T=1;RETURN
1300  X=X-D
1305  PLOTS,X,Y
1310  X=X-D;Y=Y-D;PLOTS,X,Y
1320  Y=Y-D;B=B-1;T=2;RETURN
1400  Y=Y+D
1405  PLOTS,X,Y
1410  X=X-D;Y=Y+D;PLOTS,X,Y
1420  X=X-D;A=A-1;T=3;RETURN
2000oGOTO(2000+100*T)
2100  X=X+D;PLOTS,X,Y
2110  X=X+D;Y=Y+D;PLOTS,X,Y
2120  X=X+D;Y=Y-D;GOTO 1305
2200  Y=Y-D;PLOTS,X,Y
2210  X=X+D;Y=Y-D;PLOTS,X,Y
2220  X=X-D;Y=Y-D;GOTO 1405
2300  X=X-D;PLOTS,X,Y
2310  X=X-D;Y=Y-D;PLOTS,X,Y
2320  X=X-D;Y=Y+D;GOTO 1105
2400  Y=Y+D;PLOTS,X,Y
2410  X=X-D;Y=Y+D;PLOTS,X,Y
2420  X=X+D;Y=Y+D;GOTO 1205
```
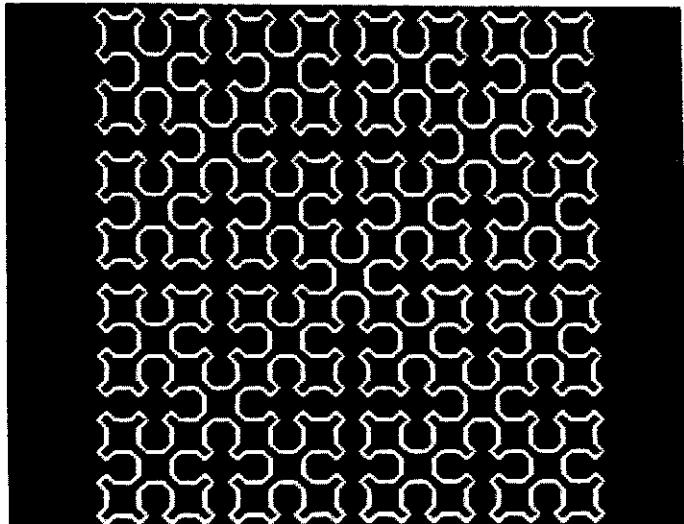
Description of Program:
```
50          Plot five generations
1000-1420 Plot centre square
2000-2420 Not a centre square
```

Variables:
A,B - Coordinates of current square
D - Number of cells in a quarter of a square
J - Number of squares in picture
K - Resolution of screen
O - Graphics mode
S - Argument for PLOT statement
T - Angle in units of 90 degrees.
X,Y - Current drawing position

Program size: 1047 bytes

Sample plot:

82

11.5.2 Three-Dimensional Plotting

The following program will plot a perspective view of a three-dimensional object or curve as viewed from any specified point in space. The program is simply provided with a subroutine giving the coordinates of the object to be drawn, or the equation of the curve.

The program below plots a perspective view of the curve $1/(1+x^2+y^2)$ for a range of values of x and y. The function has been scaled up by a factor of 300 to bring the interesting part of the curve into the correct range. The program is provided with an equation of the curve, specifying z (the vertical axis) in terms of x and y (the two horizontal axes), and the view position. It projects every point on the surface onto a plane perpendicular to the line joining the view position to the origin. The example given here draws lines of equal y, and the surface is drawn as if viewed from the point x=30, y=40, z=8; i.e. slightly above the surface.

```
  1 REM Three-Dimensional Plotting
 50 L=30;M=40;N=8
110 Z=0;CLEAR4
120 A=#8000;B=#9800
130 FORJ=A TO B STEP4;!J=-1;N.
150 S=L*L+M*M;GOS.s;R=Q
160 S=S+N*N;GOS.s;S=L*L+M*M
170 T=L*L+M*M+N*N
200 F.U=-20TO20
210 V=-20;GOS.c;GOS.b
220 F.V=-19TO20;GOS.c;GOS.a;N.;N.
230 END
400sQ=S/2
410 DOQ=(Q+S/Q)/2
415 U.(Q-1)*(Q-1)<S AND(Q+1)*(Q+1)>S
420 R.
500 REM DRAWTO(U,V,W)
510aZ=3
520bO=T-U*L-V*M-W*N
530 C=T*(V*L-U*M)*4/(R*O)+128
540 D=96+3*Q*(W*S-N*(U*L+V*M))/(R*O)
560 PLOT(Z+4),C,D;Z=0;R.
600cW=300/(10+U*U+V*V)-10;R.
```

Description of Program:

50	Set up view position
110	Set move mode, and clear screen
120-130	Invert screen
150-170	Calculate constants for linear projection
200-230	Scan X,Y plane evaluating function and plotting projected lines.
400-420	s: Square root routine (see Section 5.2.2).
500-560	a: Calculate projected position of next point and move to it (Z=0) or draw to it (Z=3)
600	c: Function for evaluation
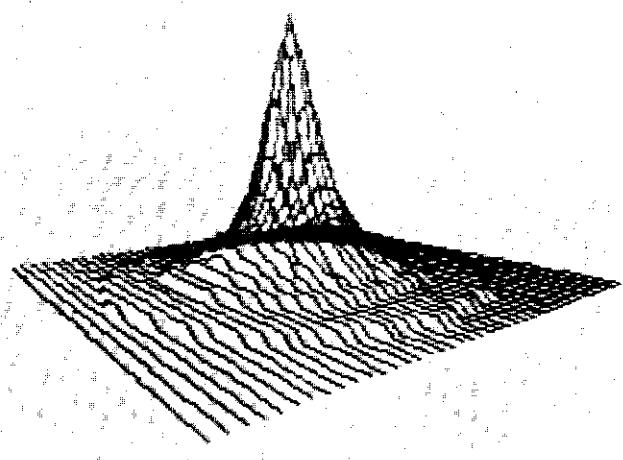
Variables:
A - Display area start
B - Display area end
C,D - Coordinates of projected point
J - Display location to be cleared
Q,R,S,T - Constants for projection
U,V - Scan variables

W - Function value

Program size: 491 bytes

Sample Plot:

11.6 Plotting Hex Characters

In the higher graphics modes, modes 1 to 4, characters cannot be plotted on the screen directly but it is fairly simple to draw characters using the graphics statements. The following simple routines will draw the hex characters 0 to F, with any desired scaling, and with an optional slope. The routines are useful for labelling graphs drawn in the higher-resolution graphics modes. Routine p plots a single hex character; routine q plots two hex characters. The routine is demonstrated by drawing random hex characters in a circle.

```
   1 REM Plotting Hex Characters
  10 N=TOP; !N=#6E3E4477; N!4=#467B6B4D
  12 N!8=#795F4F7F; N!12=#1B3B7C33
  20 V=2; H=2; S=0
  25 CLEAR 0
  30 X=30; Y=0
  40 MOVE (32+X),(24+Y)
  50 X=X+Y/6;Y=Y-X/6
  60 A=ABSRND&#F
  70 GOSUBp
  90 GOTO 40
1000qREM Plot B as 2 hex digits
1010 A=B/16; GOSUB p
1020 A=B&#F
2000pREM Plot A in hex
2001 REM uses:A,H,J,K,L,N,Q,V
2010 Q=N?A
2020 FOR J=1 TO 7
2030 K=(2-J%6)%2;L=(2-(J-1)%4)%2
2040 PLOT(Q&1),(L*H+K*S),(K*V)
2050 Q=Q/2; NEXT J
2060 PLOT0,((H+2)/2),0; RETURN
```

Description of Program:

10-12	Set up plotting statements for the 16 characters.
20	Scales for letters
30-50	Move X,Y around a circle
60-70	Plot random character
1000-1020	q: Plot low-order byte of B as two hex digits
2000-2060	p: Plot low-order hex digit of A in hex

Variables:
A - Hex digit to be plotted
B - Byte to be plotted
H - Horizontal scaling
N - Vector containing character plotting statements
Q - Next plot statement; low-order bit determines whether to draw or move.
S - Slope factor
V - Vertical scaling
X,Y - Coordinates of point on circle.

Program size: 457 bytes
Vector: 16 bytes

11.7 Animated Graphics

The graphics statements are optimised for speed. For example, to draw a diagonal across the screen using:

```
MOVE 0,0 ; DRAW 255,191
```

takes under 40 msec. The following program uses animated graphics to display a clock whose hands move to show the correct time. The hands are drawn using the statement PLOT 6, and the same statement is repeated to remove each hand's old position before drawing its new position. The clock keeps accurate time by executing the WAIT statement:

```
  1 REM Clock
 10 CLEAR4;E=128;F=96
 15 J=71;K=678;Q=100;R=#B001
 20 X=0;Y=8000;G=90
 30 MOVE(X/Q+E),(Y/Q+F)
 40 FORL=0 TO 59
 45 IF L%5<>0 GOTO c
 50 DRAW(X/G+E),(Y/G+F)
 55 MOVE(X/Q+E),(Y/Q+F)
 60cGOSUBi;GOSUBp
 68 NEXTL
 70 X=0;Y=5000;S=0
 72 DO A=0;B=6600
 80 FOR H=0 TO 4
 82 GOSUBh;C=X;D=Y;X=A;Y=B
 84 FOR M=0 TO 11
 85 GOSUBh;A=X;B=Y
 87 X=0;Y=7000
 88 IF ?R<>#FF GOTO b
 90 FOR L=0 TO 59
110 GOSUB s
120 FOR N=S TO 55;WAIT;NEXT N
130 S=0
140 GOSUBs;GOSUBi
150 NEXT L
155bX=A;Y=B
160 GOSUBh;GOSUBi
```

85

```
170  NEXT M
175  A=X;B=Y;X=C;Y=D
180  GOSUBh;GOSUBi
200  NEXT H;  UNTIL 0
399  REM
400hMOVE E,F
410  V=X/2/Q;U=Y/2/Q;W=V/5;T=U/5
415  WAIT
420  PLOT6,(V-T+E),(U+W+F)
430  PLOT6,(X/Q+E),(Y/Q+F)
440  PLOT6,(V+T+E),(U-W+F)
450  PLOT6,E,F;S=S+5;RETURN
500iWAIT;X=X+J*Y/K
510  Y=Y-J*X/K;S=S+1;RETURN
600sMOVE E,F
620pWAIT;PLOT6,(X/Q+E),(Y/Q+F)
630  S=S+1;RETURN
```

Description of Program:
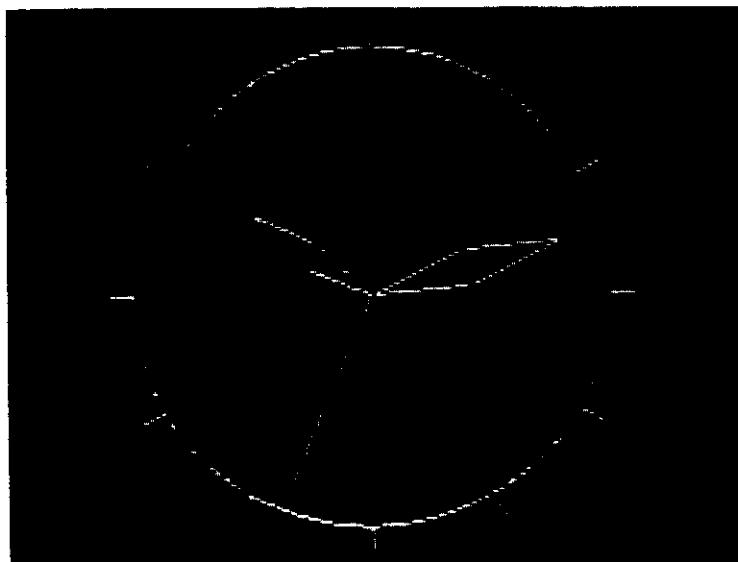
40-68	Draw clock face
80-84	Do hours and minutes
88	If shift key down miss out seconds
90-150	Do seconds
120	Use up remainder of each second
400-450	h: Draw hour/minute hand from centre of screen to X,Y
500-510	i: Increment X,Y one sixtieth of way around circle.
600	s: Draw second hand
620-630	p: Plot to point X,Y

Variables:
A,B - Coordinates of tip of minute hand
C,D - Coordinates of tip of hour hand
E,F - Coordinates of centre of screen
H - Twelves of minutes counter
J,K - Incremental variables; $J/K = 2*PI/60$ approx.
L - Seconds counter
M - Minutes counter
N - Counter
Q - Scaling factor
R - Address of shift key
S - Sixtieths of a second used out of current second
X,Y - Coordinates on screen scaled by Q

Program size: 806 bytes

Sample Plot:

To set the correct time hold the shift key down after typing RUN, and release it when the hour and minute hands are in the correct positions.

11.8 Plotting in BASIC

To illustrate how the plotting statements work, the following BASIC programs will plot points on the screen in the different graphics modes without using PLOT, DRAW, or MOVE.

11.8.1 Plotting and Testing Points in Mode 0

The following BASIC program will plot a point in the graphics mode 0; the main program sets up a vector V which contains bytes with a single bit set to denote the bit to be plotted. Subroutine p plots a point at the coordinates X and Y.

```
  1 REM Plot in Mode 0
 10 DIM V(5)
 20 !V=#04081020; V!4=#102

100 REM Plot point at X,Y
110 REM Changes: P; Uses V,X,Y
120pP=X/2+(47-Y)/3*32+#8000
130 ?P=?P◻V?(X&1+(47-Y)%3*2);RETURN
```

Using this method it is possible to determine the state of any point on the screen, as well as actually plotting points. For example, changing line 130 to:

```
130 Q=(?P&(V?(X&1+(47-Y)%3*2))<>0)
```

uses Q as a logical variable whose value is set to 'true' if the point X,Y is set, and to 'false' if the point is clear.

Note that the screen should be cleared by writing #40 in every location (or with the statement CLEAR 0) before plotting in graphics mode zero with this routine.

11.8.2 Plotting in Higher Graphics Modes

To set the ATOM to a higher graphics mode the following character should be stored in location #B000:

Mode:	Value:
0	#00
1a	#10
1	#30
2a	#50
2	#70
3a	#90
3	#B0
4a	#D0
4	#F0

This operation is performed automatically for modes 0, 1, 2, 3, and 4 by the CLEAR statement. Modes 1a, 2a, 3a, and 4a are colour graphics modes; see section 11.9 below.

To illustrate plotting in the higher modes the following BASIC program will plot a point on the screen at the coordinates X,Y in the highest-resolution graphics mode:

```
10 DIM V(7)
20 !V=#10204080; V!4=#1020408
30 ?#B000=#F0

100 REM Plot point at X,Y
110 REM Changes: P; Uses: V,X,Y
100pP=X/8+(191-Y)*32+#8000
102 ?P=?P\V?(X&7);RETURN
```

Again the program can be modified to test the state of points of the screen.

11.9 Colour Graphics

The ATOM provides three additional graphics modes which provide graphics in four selectable colours up to a maximum definition of 128x192. These modes are known as 1a, 2a, 3a, and 4a. The BASIC's PLOT, DRAW, and MOVE statements can be used in the 4-colour modes provided that a point-plotting routine, written in assembler, is provided to replace the black-and-white point plotting routines. Alternatively the COLOUR statement, provided in the extension ROM, can be used; see Section 22.2. The address of the point-plotting routine used by PLOT, MOVE, and DRAW is stored in RAM at #3FE and #3FF. The following information is passed down to the point-plotting routine in zero page:

Location:	Function:
5A	X coordinate – low byte
5B	" " high byte
5C	Y coordinate – low byte
5D	" " high byte
5E	1: set bit, 2: invert bit, else, clear bit.
5F	Free for workspace
60	" "

The following BASIC program demonstrates how an assembler point-plotting routine can be provided to give four-colour plotting in graphics mode 4a, the highest-resolution colour graphics mode:

```
 10 REM 4-Colour Plot
 12 GOSUB 400
 16 CLEAR4;?#B000=#D0
 18 ?#3FE=Q;?#3FF=Q&#FFFF/256
 30 FOR J=0 TO 64 STEP 2
 40 ?C=J%3*4;MOVE J,0
 50 DRAW 127,J;DRAW(127-J),191
 60 DRAW 0,(191-J);DRAW J,0
 70 NEXT J
 80 END
400 DIM V(11),C(0),P(-1),Q(-1)
420 !V=#01041040;V!4=#02082080;V!8=#030C30C0
430 P.$21
508[
510 LDA@0;STA #5F
520 LDA#5C;LSR A;ROR #5F
530 LSRA;ROR#5F;LSRA;ROR#5F
540 STA#60;LDA#5A;LSRA;LSRA
550 CLC;ADC#5F;STA#5F
560 LDA#60;ADC@#80;STA#60
570\#5F AND #60 CONTAIN ADDRESS
580 LDA#5A;AND@3;CLC;ADCC;TAY
590 LDX@0;LDAV,Y;ORA(#5F,X)
600 STA(#5F,X);RTS
610]
620 P.$6
630 RETURN
```

Description of Program:

12	Assemble point plotting routine
16	Clear display; set mode 3a
18	Change point plotter vector
30-70	Demonstration program; curve stitching in 4 colours
400	Set up variable space
420	Vectors for three colours
430	Disable assembler listing
508-610	Assembler point-plotter program
620	Turn screen back on

Variables:
C - Colour: 0, 4, or 8.
P - Location counter
Q - Address of point-plotting routine
V - Vectors for setting bits

Program size: 558 bytes
Vectors: 13 bytes

Note that the routine only sets bits, and plots in three colours - the fourth colour being the background colour. It would be a simple matter to modify the routine so that it was able to set or unset bits; i.e. plot in the background colour.

12 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.

12.1 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 27 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.:

    FOR N=1 TO 3

2. Mistyping a digit '1' for a letter 'I', and vice-versa. E.g.:

    lF J=2 PRlNT "TWO"

3. Forgetting to enclose an expression in brackets when it is used as a parameter in a statement. E.g.:

    MOVE X+32,Y

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.:

    MOVE O,O

was intended to move to the origin, but in fact moves to some coordinates dependent on the value of the variable O.

12.2 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 forseen 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, dimensioned by a DIM 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. Location counter P not set up when assembling. The value of P should be set before assembling a program to the address of an unused area of memory large enough to receive the machine code, and P should not be used for any other purpose in the program.

4. Graphics statements used without initialising graphics. The CLEAR statement must precede use of any graphics statements.

5. Assigning to a string variable and exceeding the allocated space. Care should be taken that enough space has been allocated to string variables, with DIM, to receive the strings allocated to them.

6. Assigning outside the bounds of an array or vector. Assigning to array or vector elements above the range dimensioned in the DIM statement will overwrite other arrays, vectors, or strings.

12.3 Suspected Hardware Faults

This section deals with faults on an ATOM 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 Technical Manual; this section describes only those hardware problems that can be tested by running software diagnostics.

12.3.1 RAM Memory Faults

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

```
 1 REM MEMORY TEST
10 INPUT"FROM"A,"  TO"B
20 DO ?12=0; R=!8
30 FOR N=A TO B STEP4;!N=RND; NEXT N
35 ?12=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 the screen memory enter:

```
>RUN
FROM?#8000
  TO?#81FC
```

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

12.3.2 ROM Memory Faults

The BASIC interpreter, operating system, and assembler, are all contained in a single 8K ROM, and as all ROMs are thoroughly tested before despatch it is very unlikely that a fault could 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. The program should be run for each 4K half of the ROM.

92

```
  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
```

60 [line too long.]

Program size: 213 bytes

Sample run:

```
>RUN
PROM ADDRESS?#C000
SIGNATURE IS    D67D
>RUN
PROM ADDRESS?#F000
SIGNATURE IS    E386
>
```

The program takes about 6 minutes to run, and if these signatures are
obtained the ROM is correct.

The Atom extension ROM, described in chapter 22, can be tested by
giving the reply #D000 to the prompt. It should give a signature of
AAA1.

12.3 Programming Service

If all else fails, owners of an ATOM may make use of the free
Programming Service provided by Acorn. To ensure a rapid reply to any
queries the special Programming Service Forms, supplied with the ATOM,
must be used to submit the problem. New forms will be supplied with
the reply to any queries, or on request.

All reports should be accompanied by a full description of the
problem or fault, and the occasions when it occurs. Please also
enclose a stamped addressed envelope for the reply. A program should
be supplied which illustrates the problem or suspected fault. This
program should preferably be only four or five lines long, and should
be written in the space provided on the Programming Service Form, with
any spaces in the original carefully included. If the problem or fault
is only exhibited by a longer program the report form should be
accompanied by a cassette tape recording of the program, and the title
of the file on the tape should be entered on the form. The cassette
will be returned with the reply.

13 Assembler Programming

In BASIC there are operators to perform multiplication, division, iteration etc., but in assembler the only operations provided are far more primitive and require a more thorough understanding of how the inside of the machine works. The ATOM is unique in that it enables BASIC and assembler to be mixed in one program. Thus the critical sections of programs, where speed is important, can be written in assembler, but the body of the program can be left in BASIC for simplicity and clarity.

The following table gives the main differences between BASIC and assembler:

BASIC	Assembler
26 variables	3 registers
4-byte precision	1 byte precision
Slow - assignment takes over 1 msec.	Fast - assignments take 10 usec.
Multiply and divide	No multiply or divide
FOR...NEXT and DO...UNTIL loops	Loops must be set up by the programmer
Language independent of computer	Depends on instruction set of chip
Protection against overwriting program	No protection

However, do not be discouraged; writing in assembler is rewarding and gives you a greater freedom and more ability to express the problem that you are trying to solve without the constraints imposed on you by the language. Remember that, after all, the BASIC interpreter itself was written in assembler.

A computer consists of three main parts:

1. The memory
2. The central processing unit, or CPU.
3. The peripherals.

In the ATOM these parts are as follows:

1. Random Access Memory (RAM) and Read-Only Memory (ROM).
2. The 6502 microprocessor.
3. The VDU, keyboard, cassette interface, speaker interface...etc.

When programming in BASIC it is not necessary to understand how these parts are working together, and how they are organised inside the computer. However in this section on assembler programming a thorough understanding of all these parts is needed.

13.1 Memory

The computer's memory can be thought of as a number of 'locations', each capable of holding a value. In the unexpanded ATOM there are 2048 locations, each of which can hold one of 256 different values. Only 512 of these locations are free for you to use for programs; the remainder are used by the ATOM operating system, and for BASIC's variables.

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, being the number stored in the location of that address.

13.2 Hexadecimal Notation

Having been brought up counting 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 ATOM can do it for you.

13.2.1 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.

13.2.2 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

96

hexadecimal'. Thus writing:

      PRINT &#123

will print 123.

13.3 Examining Memory Locations - '?'

We can now look at the contents of some memory locations in the ATOM'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 ?#E1

will look at the location whose address is #E1, and print out its value, which will be 128 (the cursor flag). 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).

13.4 Changing Memory Locations

A word of caution: although it is quite safe to look at any memory location in the ATOM, care should be exercised when changing memory locations. The examples given here specify locations that are not used by the ATOM 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 ATOM with BREAK.
    First print the contents of #80. The value there will be whatever was in the memory when you switched on, because the ATOM 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?

13.5 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.

13.6 The Byte

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.

13.7 The CPU

The main part of this chapter will deal with the ATOM's brain, the Central Processing Unit or CPU. In the ATOM this is a 6502, a processor designed in 1975 and the best-selling 8-bit microprocessor in 1979. Much of what you learn in this chapter is specific to the 6502, and other microprocessors will do things more or less differently. However, the 6502 is an extremely popular microprocessor with a modern instruction set, and a surprisingly wide range of addressing modes; furthermore it uses pipelining to give extremely fast execution times; as fast as some other microprocessors running at twice the speed.
    The CPU is the active part of the computer; although many areas of memory may may remain unchanged for hours on end when a computer is being used, the CPU is working all the time the machine is switched on, and data is being processed by it at the rate of 1 million times a second. The CPU's job is to read a sequence of instructions from memory and carry out the operations specified by those instructions.

13.8 Instructions

The instructions to the CPU are again just values in memory locations, but this time they are interpreted by the CPU to represent the different operations it can perform. For example, the instruction #18 is interpreted to mean 'clear carry flag'; you will find out what that means in a moment. The first byte of all instructions is the operation code, or 'op code'. Some instructions consist of just the op code; other instructions specify data or an address in the bytes following the op code.

13.9 The Accumulator

Many of the operations performed by the CPU involve a temporary location inside the CPU known as the accumulator, or A for short (nothing to do with BASIC's variable A). For example, to add two numbers together you actually have to load the first number into the accumulator from memory, add in the second number from memory, and then store the result somewhere. The following instructions will be needed:

Mnemonic	Description	Symbol
LDA	Load accumulator with memory	A=M
STA	Store accumulator in memory	M=A
ADC	Add memory to accumulator with carry	A=A+M+C

We will also need one extra instruction:

CLC	Clear carry	C=0

The three letter names such as LDA and STA are called the instruction mnemonics; they are simply a more convenient way of representing the

98

instruction than having to remember the actual op code, which is just a number.

13.10 The Assembler

The ATOM automatically converts these mnemonics into the op codes. This process of converting mnemonics into codes is called 'assembling'. The assembler takes a list of mnemonics, called an assembler program, and converts them into 'machine code', the numbers that are actually going to be executed by the processor.

13.10.1 Writing an Assembler Program

Enter the following assembler program:
```
   10 DIM P(-1)
   20[
   30 LDA #80
   40 CLC
   50 ADC #81
   60 STA #82
   70 RTS
   80]
   90 END
```

The meaning of each line in this assembler program is as follows:

10. The DIM statement is not an assembler mnemonic; it just tells the assembler where to put the assembled machine code; at TOP in this case.

20. The '[' and ']' symbols enclose the assembler statements.

30. Load the accumulator with the contents of the memory location with address #80. (The contents of the memory location are not changed.)

40. Clear the carry flag.

50. Add the contents of location #81 to the accumulator, with the carry. (Location #81 is not changed by this operation.)

60. Store the contents of the accumulator to location #82. (The accumulator is not changed by this operation.)

70. The RTS instruction will usually be the last instruction of any program; it causes a return to the ATOM BASIC system from the machine-code program.

80. See 20.

90. The END statement is not an assembler mnemonic; it just denotes the end of the text.

Now type RUN and the assembler program will be assembled. An 'assembler listing' will be printed out to show the machine code that the assembler has generated to the left of the corresponding assembler mnemonics:

```
RUN
   20 824D
   30 824D A5 80      LDA #80
   40 824F 18         CLC
   50 8250 65 81      ADC #81
   60 8252 85 82      STA #82
   70 8254 60         RTS
       ↑    ↑  ↑  ↑
       │    │  │  │    ↑
       │    │  │  │    mnemonic statement
       │    │  │  instruction data/address
       │    │  instruction op code
       │    location counter
       statement line number.
```

The program has been assembled in memory starting at #824D,
immediately after the program text. This address may be different when
you do this example if you have inserted extra spaces into the
program, but that will not affect what follows. All the numbers in the
listing, except for the line numbers on the left, are in hexadecimal;
thus #18 is the op code for the CLC instruction, and #A5 is the op
code for LDA. The LDA instruction consists of two bytes; the first
byte is the op code, and the second byte is the address; #80 in this
case.

   Typing RUN assembled the program and stored the machine code in
memory directly after the assembler program. The address of the end of
the program text is called TOP; type:

    PRINT &TOP

and this value will be printed out in hexadecimal. It will correspond
with the address opposite the first instruction, #A5. The machine code
is thus stored in memory as follows:

```
┌──┬──┬──┬──┬──┬──┬──┬──┐
│A5│80│18│65│81│85│82│60│
└──┴──┴──┴──┴──┴──┴──┴──┘
 ^
 TOP
```

So far we have just assembled the program, generated the machine code,
and put the machine code into memory.

13.10.2 Executing a Machine-Code Program

To execute the machine-code program at TOP, type:

    LINK TOP

What happens? Nothing much; we just return to the '>' prompt. But the
program has been executed, although it only took 17 microseconds, and
the contents of locations #80 and #81 have indeed been added together
and the result placed in #82.

   Execute it again, but first set up the contents of #80 and #81 by
typing:

    ?#80=7; ?#81=9

If you wish you can also set the contents of #82 to 0. Now type:

    LINK TOP

and then look at the contents of #82 by typing:

    PRINT ?#82

100

The result is 16 (in decimal); the computer has just added 7 and 9 and obtained 16!

13.11 Carry Flag

Try executing the program for different numbers in #80 and #81. You might like to try the following:

```
?#80=140; ?#81=150
LINK TOP
```

What is the result?

The reason why the result is only 34, and not 290 as one might expect, is that the accumulator can only hold one byte. Performing the addition in hexadecimal:

Decimal	Hexadecimal
140	8C
+150	+96
290	122

Only two hex digits can fit in one byte, so the '1' of #122 is lost, and only the #22 is retained. Luckily the '1' carry is retained for us in, as you may have guessed, the carry flag. The carry flag is always set to the value of the carry out of the byte after an addition or subtraction operation.

13.12 Adding Two-Byte Numbers

The carry flag makes it a simple matter to add numbers as large as we please. Here we shall add two two-byte numbers to give a two-byte answer, although the method can be extended to any number of bytes. Modify the program already in memory by retyping lines 50 to 120, leaving out the lower-case comments, to give the following program:

```
10 DIM P(-1)
20 [
30 LDA #80 low byte of one number
40 CLC
50 ADC #82 low byte of other number
60 STA #84 low byte of result
70 LDA #81 high byte of one number
80 ADC #83 high byte of other number
90 STA #85 high byte of result
100 RTS
110 ]
120 END
```

Assemble the program:

RUN

```
20 826E
30 826E A5 80     LDA #80
40 8270 18        CLC
50 8271 65 82     ADC #82
60 8273 85 84     STA #84
70 8275 A5 81     LDA #81
80 8277 65 83     ADC #83
90 8279 85 85     STA #85
```

101

```
     100 827B 60        RTS
```

Now set up the two numbers as follows:

```
     ?#80=#8C;  ?#81=#00
     ?#82=#96;  ?#83=#00
```

Finally, execute the program:

     LINK TOP

and look at the result, printing it in hexadecimal this time for convenience:

     PRINT &?#84, &?#85

The low byte of the result is #22, as before using the one-byte addition program, but this time the high byte of the result, #1, has been correctly obtained. The carry generated by the first addition was added in to the second addition, giving:

$0+0+carry = 1$

     Try some other two-byte additions using the new program.

13.13 Subtraction

The subtract instruction is just like the add instruction, except that there is a 'borrow' if the carry flag is zero. Therefore to perform a single-byte subtraction the carry flag should first be set with the SEC instruction:

```
SBC    Subtract memory from accumulator with borrow        A=A-M-(1-C)
SEC    Set carry flag                                              C=1
```

13.14 Printing a Character

The ATOM 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. 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. These routines are executed with the following instruction:

JSR   Jump to subroutine

A detailed description of how the JSR instruction works will be left until later.

     The following program outputs the contents of memory location #80 as a character to the VDU, using a call to the subroutine OSWRCH:

```
     10 DIM P(-1)
     20 W=#FFF4
     30 [
     40 LDA #80
     50 JSR W
     60 RTS
     70 ]
     80 END
```

102

The variable W is used for the address of the OSWRCH routine. Assemble the program, and then set the contents of #80 to #21:

    ?#80=#21

Then execute the program:

    LINK TOP

and an exclamation mark will be printed out before returning to the ATOM's prompt character, because #21 is the code for an exclamation mark. Try executing the program with different values in #80.

13.15 Immediate Addressing

In the previous example the instruction:

    LDA #80

loaded the accumulator with the contents of location #80, which was then set to contain #21, the code for an exclamation mark. If at the time that the program was written it was known that an exclamation mark was to be printed it would be more convenient to specify this in the program as the actual data to be loaded into the accumulator. Fortunately an 'Immediate' addressing mode is provided which achieves just this. Change the instruction to:

    LDA @#21

where the '@' (at) symbol specifies to the assembler that immediate addressing is required. Assemble the program again, and note that the instruction op-code for LDA @#21 is #A9, not #A5 as previously. The op-code of the instruction specifies to the CPU whether the following byte is to be the actual data loaded into the accumulator, or the address of the data to be loaded.

103

14 Jumps, Branches, and Loops

All the assembler programs in the previous section have been executed instruction by instruction following the sequence specified by the order of the instructions in memory. The jump and branch instructions enable the flow of control to be altered, making it possible to implement loops.

14.1 Jumps

The JMP instruction is followed by the address of the instruction to be executed next.

JMP   Jump

14.2 Labels

Before using the JMP instruction we need to be able to indicate to the assembler where we want to jump to, and to do this conveniently 'labels' are needed. In the assembler labels are variables of the form AA to ZZ followed by a number (0, 1, 2 ... etc). If you are already familiar with ATOM BASIC you will recognise these as the arrays.

First the labels to be used in an assembler program must be declared in the DIM statement. Note that we still need to declare P(-1) as before, and this must be the last thing declared. For example, to provide space for four labels LL0, LL1, LL2, and LL3 we would declare:

DIM LL(3), P(-1)   *Do not use array AA [bug in Assembler]*

Labels used in a program are prefixed by a colon ':' character. For example, enter the following assembler program:
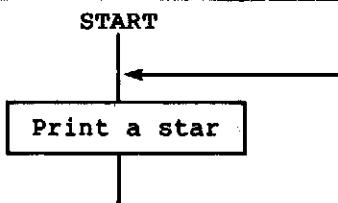
```
10 DIM LL(3),P(-1)
20 W=#FFF4
30[
40:LL0 LDA @#2A
50:LL1 JSR W
60 JMP LL0
70]
80 END
```

To execute the program the procedure is slightly different from the previous examples, because space has now been assigned at TOP for the labels. When using labels in an assembler program you should place a label at the start of the program, as with LL0 in this example, and LINK to that label. So, in this example, execute the program with:

LINK LL0

The program will output an asterisk, and then jump back to the previous instruction. The program has become stuck in an endless loop! If you know BASIC, compare this program with the BASIC program in section 4.6 which has the same effect.

A flowchart for this program is as follows:

```
               START
                 │
                 ▼◄──────────┐
        ┌─────────────────┐  │
        │  Print a star   │  │
        └─────────────────┘  │
                 │           │
                 └───────────┘
```

Try pressing ESCAPE. ESCAPE will not work; it only works in BASIC programs, and here we are executing machine code instructions so ESCAPE is no longer checked for. Fortunately there is one means of salvation: press BREAK, and then type OLD to retrieve the original program.

14.3 Flags

The carry flag has already been introduced; it is set or cleared as the result of an ADC instruction. The CPU contains several other flags which are set or cleared depending on the outcome of certain instructions; this section will introduce another one.

14.3.1 Zero Flag

The zero flag, called Z, is set if the result of the previous operation gave zero, and is cleared otherwise. So, for example:

        LDA #80

would set the zero flag if the contents of #80 were zero.

14.4 Conditional Branches

The conditional branches enable the program to act on the outcome of an operation. The branch instructions look at a specified flag, and then either carry on execution if the test was false, or cause a branch to a different address if the test was true. There are 8 different branch instructions, four of which will be introduced here:

BEQ   Branch if equal to zero (i.e. Z=1)
BNE   Branch if not equal to zero (i.e. Z=0)

BCC   Branch if carry-flag clear (i.e. C=0)
BCS   Branch if carry-flag set (i.e. C=1)

The difference between a 'branch' and a 'jump' is that the jump instruction is three bytes long (op-code and two-byte address) whereas the branch instructions ase only two bytes long (op-code and one-byte offset). The difference is automatically looked after by the assembler.

The following simple program will print an exclamation mark if #80 contains zero, and a star if it does not contain zero; the comments in lower-case can be omitted when you enter the program:
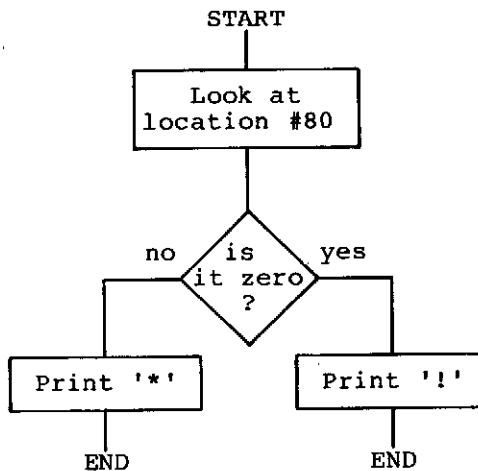
```
   10 DIM BB(3),P(-1)
   20 W=#FFF4
   30 [
   40 :BB0 LDA #80
   50 BEQ BB1      if zero go to BB1
   60 LDA @#2A      star
   70 JSR W         print it
   80 RTS           return
   90 :BB1 LDA @#21 exclamation mark
  100 JSR W         print it
 106
```

```
110 RTS          return
120 ]
130 END
```

A flowchart for this program is as follows:

```
                    START
                      |
            +-------------------+
            |     Look at       |
            |  location #80     |
            +-------------------+
                      |
                     / \
          no       /is \       yes
        +---------<it zero>---------+
        |          \  ? /           |
        |           \ /             |
        |                           |
  +-----------+              +-----------+
  | Print '*' |              | Print '!' |
  +-----------+              +-----------+
        |                           |
       END                         END
```

Now assemble the program with RUN as usual. You will almost certainly get the message:

OUT OF RANGE:

before the line containing the instruction BEQ BB1, and the offset in the branch instruction will have been set to zero. The message is produced because the label BB1 has not yet been met when the branch instruction referring to it is being assembled; in other words, the assembler program contains a forward reference. Therefore you should assemble the program a second time by typing RUN again. This time the message will not be produced and the correct offset will be calculated for the branch instruction.

Note that whenever a program contains forward references it should be assembled twice before executing the machine code.

Now execute the program by typing:

     LINK BB0

for different values in #80, and verify that the behaviour is as specified above.

14.5 X and Y registers

The CPU contains two registers in addition to the accumulator, and these are called the X and Y registers. As with the accumulator, there are instructions to load and store the X and Y registers:

LDX	Load X register from memory	X=M
LDY	Load Y register from memory	Y=M
STX	Store X register to memory	M=X
STY	Store Y register to memory	M=Y

However the X and Y registers cannot be used as one of the operands in arithmetic or logical instructions like the accumulator; they have their own special uses, including loop control and indexed addressing.

107

14.6 Loops in Machine Code

The X and Y registers are particularly useful as the control variables in iterative loops, because of four special instructions which will either increment (add 1 to) or decrement (subtract 1 from) their values:

INX	Increment X register	X=X+1
INY	Increment Y register	Y=Y+1
DEX	Decrement X register	X=X-1
DEY	Decrement Y register	Y=Y-1
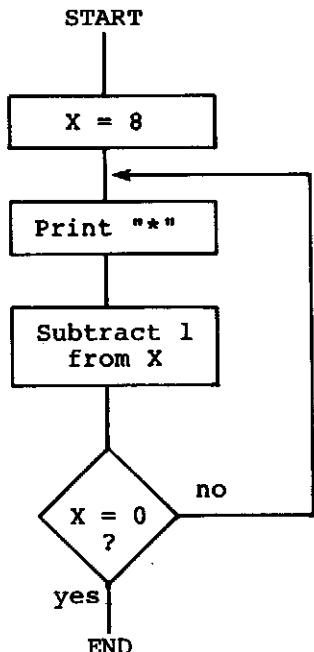
Note that these instructions do not affect the carry flag, so incrementing #FF will give #00 without changing the carry bit. The zero flag is, however, affected by these instructions, and the following program tests the zero flag to detect when X reaches zero.

14.6.1 Iterative Loop

The iterative loop enables the same set of instructions to be executed a fixed number of times. For example, enter the following program:

```
10 DIM LL(4),P(-1)
20 W=#FFF4
30[
40:LL0 LDX @8     initialise X
50:LL1 LDA @#2A   code for star
60:LL2 JSR W      output it
70 DEX            count it
80 BNE LL2        all done?
90 RTS
100]
110 END
```

A flowchart for the program is as follows:

START

X = 8

Print "*"

Subtract 1
from X

X = 0
?

no

yes

END

Assemble the program by typing RUN. This program prints out a star, decrements the X register, and then branches back if the result after decrementing the X register is not zero. Consider what value X will have on successive times around the loop and predict how many stars will be printed out; then execute the program with LINK LL0 and see if your prediction was correct. If you were wrong, try thinking about the case where X was initially set to 1 instead of 8 in line 40.

How many stars are printed if you change the instruction on line 40 to LDX @0 ?

14.7 Compare

In the previous example the condition X=0 was used to terminate the loop. Sometimes we might want to count up from 0 and terminate on some specified value other than zero. The compare instruction can be used to compare the contents of a register with a value in memory; if the two are the same, the zero flag will be set. If they are not the same, the zero flag will be cleared. The compare instruction also affects the carry flag.

CMP	Compare accumulator with memory	A–M
CPX	Compare X register with memory	X–M
CPY	Compare Y register with memory	Y–M

Note that the compare instruction does not affect its two operands; it just changes the flags as a result of the comparison.

The next example again prints 8 stars, but this time it uses X as a counter to count upwards from 0 to 8:

```
10 DIM LL(2),P(-1)
20 W=#FFF4
30[
40:LL0 LDX @0      start at zero
50:LL1 LDA @#2A    code for star
60 JSR W           output it
70 INX             next X
80 CPX @8          all done?
90 BNE LL1
100 RTS            return
110]
120 END
```

In this program X takes the values 0, 1, 2, 3, 4, 5, 6, and 7. The last time around the loop X is incremented to 8, and the loop terminates. Try drawing a flowchart for this program.

14.8 Using the Control Variable

In the previous two examples X was simply used as a counter, and so it made no difference whether we counted up or down. However, it is often useful to use the value of the control variable in the program. For example, we could print out the character in the X register each time around the loop. We therefore need a way of transferring the value in the X register to the accumulator so that it can be printed out by the OSWRCH routine. One way would be to execute:

```
STX #82
LDA #82
```

where #82 is not being used for any other purpose. There is a more convenient way, using one of four new instructions:

| TAX | Transfer accumulator to X register | X=A |
| TAY | Transfer accumulator to Y register | Y=A |

109

```
TXA    Transfer X register to accumulator                      A=X
TYA    Transfer Y register to accumulator                      A=Y
```

Note that the transfer instructions only affect the register being transferred to.

The following example prints out the alphabet by making X cover the range #41, the code for A, to #5A, the code for Z.

```
 10 DIM LL(2),P(-1)
 20 W=#FFF4
 30[
 40:LL0 LDX @#41     start at A
 50:LL1 TXA          put it in A
 60 JSR W            print it
 70 INX              next one
 80 CPX @#5B         done Z?
 90 BNE LL1          if so - continue
100 RTS              else - return
110]
120 END
```

Modify the program to print the alphabet in reverse order, Z to A.

All these examples could have used Y as the control variable instead of X in exactly the same way.

15 Logical Operations, Shifts, and Rotates

So far we have considered each memory location, or memory byte, as being capable of holding one of 256 different numbers (0 to 255), or one of 256 different characters. In this section we examine an alternative representation, which is closer to the way a byte of information is actually stored in the computer's memory.

15.1 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).

Verify the following facts about binary numbers:

1. Shifting a binary number left, and inserting a zero after it, is the same as multiplying its value by 2.

e.g. 7 is 111
    14 is 1110.

2. Shifting a binary number right, removing the last digit, is the same as dividing it by 2 and ignoring the remainder.

15.2 Bytes

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.

15.3 Logical Operations

Many operations in the computer's instruction set are easiest to think of as operations between two bytes represented as two 8-bit numbers. This section examines three operations called 'logical' operations

112

which are performed between the individual bits of the two operands. One of the operands is always the accumulator, and the other is a memory location.

AND   AND accumulator with memory   A=A&M

The AND operation sets the bit of the result to a 1 only if the bit of one operand is a 1 AND the corresponding bit of the other operand is a 1. Otherwise the bit in the result is a zero. For example:

```
Hexadecimal:    Binary:
    A9          1 0 1 0 1 0 0 1
    E5          1 1 1 0 0 1 0 1
    ──          ───────────────
    A1          1 0 1 0 0 0 0 1
```

One way of thinking of the AND operation is that one operand acts as a 'mask', and only where there are ones in the mask do the corresponding bits in the other operand 'show through'; otherwise, the bits are zero.

ORA   OR accumulator with memory   A=A\M

The OR operation sets the bit of the result to a 1 if the corresponding bit of one operand is a 1 OR the corresponding bit of the other operand is a 1, or indeed, if they are both ones; otherwise the bit in the result is zero. For example:

```
Hexadecimal:    Binary:
    A9          1 0 1 0 1 0 0 1
    E5          1 1 1 0 0 1 0 1
    ──          ───────────────
    ED          1 1 1 0 1 1 0 1
```

EOR   Exclusive-OR accumulator with memory   A=A:M

The exclusive-OR operation is like the OR operation, except that the corresponding bit in the result is 1 only if the corresponding bit of one operand is a 1, or if the corresponding bit of the other operand is a 1, but not if they are both ones. For example:

```
Hexadecimal:    Binary:
    A9          1 0 1 0 1 0 0 1
    E5          1 1 1 0 0 1 0 1
    ──          ───────────────
    4C          0 1 0 0 1 1 0 0
```
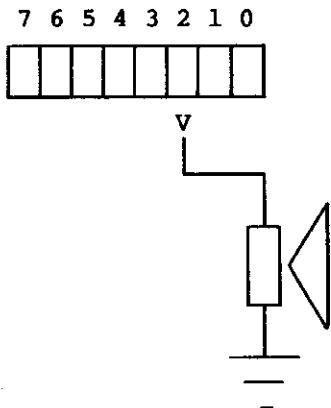
Another way of thinking of the exclusive-OR operation is that the bits of one operand are inverted where the other operand has ones.

15.4 Music

Music is composed of vibrations of different frequencies that stimulate our ears to give the sensations of tones and noise. A single tone is a signal with a constant rate of vibration, and the 'pitch' of the tone depends on the frequency of the vibration: the faster the vibration, or the higher the frequency of vibration, the higher is the perceived pitch of the tone. The human ear is sensitive to frequencies from about 10 Hz (10 vibrations per second) up to about 16 kHz (16000 vibrations a second). Since the ATOM can execute up to 500000 instructions per second in machine code, it is possible to generate tones covering the whole audible range.

    The ATOM contains a loudspeaker which is controlled by an output

line. The loudspeaker is connected to bit 2 of the output port whose
address is #B001:

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


To make the loudspeaker vibrate we can exclusive-OR the location
corresponding to the output port with the binary number 00000100 so
that bit 2 is changed each time. To make the ATOM generate a tone of a
particular frequency we need to make the output driving the
loudspeaker vibrate with the required frequency. Try the following
program:

```
 10 DIM VV(4),P(-1)
 20 L=#B002
 30[
 40:VV0 LDA L
 50:VV1 LDX #80
 60:VV2 DEX
 70 BNE VV2
 80 EOR @4
 90 STA L
100 JMP VV1
110]
120 END
```

The immediate operand 4 in line 80 corresponds to the binary number
00000100. The program generates a continuous tone, and can only be
stopped by pressing BREAK. (To get the program back after pressing
BREAK, type OLD.) The inner loop, lines 60 and 70, gives a delay
depending on the contents of #80; the greater the contents of #80, the
longer the delay, and the lower the pitch of the tone in the
loudspeaker.

15.4.1 Bleeps

To make the program generate a tone pulse, or a bleep, of a fixed
length, we need another counter to count the number of iterations
around the loop, and to stop the program when a certain number of
iterations have been performed. The following program is based on the
previous example, but contains an extra loop to count the number of
cycles. The only lines you need to enter are 45, 95, 100, and 105:

```
 5 REM Bleep
10 DIM VV(4),P(-1)
20 L=#B002
30[
```

114

```
 40:VV0 LDA L
 45 LDY #81
 50:VV1 LDX #80
 60:VV2 DEX
 70 BNE VV2
 80 EOR @4
 90 STA L
 95 DEY
100 BNE VV1
105 RTS
110 ]
120 END
```

Now the program generates a tone pulse whose frequency is determined by the contents of #80, and whose length is determined by #81.

To illustrate the operation of this program, the following BASIC program calls it, running through tones of every frequency it can generate:

```
200 ?#81=255
210 FOR N=1 TO 256
220 ?#80=N
230 LINK VV0
240 NEXT N
250 END
```

This program should be entered into memory with the previous example, and the END statement at line 120 should be deleted so that the BASIC program will execute the assembled Bleep program.

Try changing the statement on line 220 to:

```
220 ?#80=RND
```

to give something reminiscent of certain modern music!
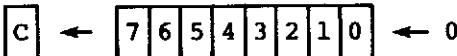
One disadvantage of this program, which you may have noticed, is that the length of the bleep gets progressively shorter as the frequency of the note gets higher; this is because the program generates a fixed number of cycles of the tone, so the higher the frequency, the less time these cycles will take. To give bleeps of the same duration it is necessary to make the contents of #81 the inverse of #80. For an illustration of how to achieve this, see the Harpsichord program of section 17.2.

15.5 Rotates and Shifts
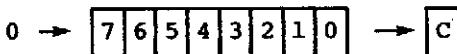
The rotate and shift operations move the bits in a byte either left or right. The ASL instruction moves all the bits one place to the left; what was the high-order bit is put into the carry flag, and a zero bit is put into the low-order bit of the byte. The ROL instruction is identical except that the previous value of the carry flag, rather than zero, is put into the low-order bit.

The right shift and rotate right instructions are identical, except that the bits are shifted to the right:
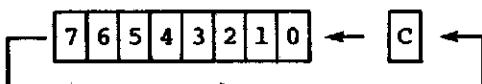
ASL   Arithmetic shift left one bit (memory or accumulator)
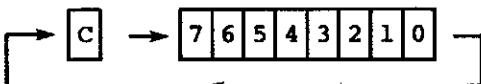
LSR   Logical shift right one bit (memory or accumulator)

115

ROL Rotate left one bit (memory or accumulator)

```
 ┌─────────────────────────────────────┐
 │  ┌─┬─┬─┬─┬─┬─┬─┬─┐      ┌─┐          │
─┴──│7│6│5│4│3│2│1│0│ ◄── │C│ ◄────────┘
    └─┴─┴─┴─┴─┴─┴─┴─┘      └─┘
 └──────────────────────►
```

ROR Rotate right one bit (memory or accumulator)

```
 ┌──► ┌─┐     ┌─┬─┬─┬─┬─┬─┬─┬─┐
 │    │C│ ──► │7│6│5│4│3│2│1│0│ ──┐
 │    └─┘     └─┴─┴─┴─┴─┴─┴─┴─┘   │
 └──────────────◄─────────────────┘
```

15.6 Noise

It may seem surprising that a computer, which follows an absolutely
determined sequence of operations, can generate noise which sounds
completely random. The following program does just that; it generates
a pseudo-random sequence of pulses that does not repeat until 8388607
have been generated. As it stands the noise it generates contains
components up to 27kHz, well beyond the range of hearing, and it takes
over 5 minutes before the sequence repeats.

The following noise program simulates, by means of the shift and
rotate instructions, a 23-bit shift register whose lowest-order input
is the exclusive-OR of bits 23 and 18:

```
 10 REM Random Noise
 20 DIM L(2),NN(1),P(-1)
 30 C=#B002
 40[
 50:NN0 LDA L; STA C
 60 AND @#48; ADC @#38
 70 ASL A; ASL A
 80 ROL L+2; ROL L+1; ROL L
 90 JMP NN0
100]
110 LINK NN0
```

Incidentally, the noise generated by this program is an excellent
signal for testing high-fidelity audio equipment. The noise should be
reproduced through the system and listened to at the output. The noise
should sound evenly distributed over all frequencies, with no
particular peak at any frequency revealing a peak in the spectrum, or
any holes in the noise revealing the presence of dips in the spectrum.

16 Addressing Modes and Registers

16.1 Indexed Addressing

So far the X and Y registers have simply been used as counters, but their most important use is in 'indexed addressing'. We have already met two different addressing modes: absolute addressing, as in:

    LDA ＄U

where the instruction loads the accumulator with the contents of location U, and immediate addressing as in:

    LDA @#21

where the instruction loads the accumulator with the actual value #21.

In indexed addressing one of the index registers, X or Y, is used as an offset which is added to the address specified in the instruction to give the actual address of the data. For example, we can write:

    LDA S,X

If X contains zero this instruction will behave just like LDA S. However, if X contains 1 it will load the accumulator with the contents of 'one location further on from S'. In other words it will behave like LDA S+1. Since X can contain any value from 0 to 255, the instruction LDA S,X gives you access to 256 different memory locations. If you are familiar with BASIC's byte vectors you can think of S as the base of a vector, and of X as containing the subscript.

16.1.1 Print Inverted String

The following program uses indexed addressing to print out a string of characters inverted. Recall that a string is held as a sequence of character codes terminated by a #D byte:

```
 10 DIM LL(2),S(64),P(-1)
 20 W=#FFF4
 30[
 40:LL0 LDX @0
 50:LL1 LDA S,X
 60 CMP @#D
 70 BEQ LL2
 80 ORA @#20
 90 JSR W
100 INX
110 BNE LL1
120:LL2 RTS
130]
140 END
```

Assemble the program by typing RUN twice, and then try the program by entering:

    $S="TEST STRING"
    LINK LL0

16.1.2 Index Subroutine

Another useful operation that can easily be performed in a machine-code routine is to look up a character in a string, and return its position in that string. The following subroutine reads in a character, using a call to the OSRDCH read-character routine, and saves in ?F the position of the first occurrence of that character in $T.

```
   1 REM Index Routine
  10 DIM RR(3),T(25),F(0),P(-1)
  20 R=#FFE3; $T="ABCDEFGH"
  30 [
 160 \Look up A in T
 165 :RR1 STX F; RTS
 180 :RR0 JSR R; LDX @LEN(T)-1
 190 :RR2 CMP T,X; BEQ RR1
 210 DEX; BPL RR2; BMI RR0
 220 ]
 230 END
```

The routine is entered at RR0, and as it stands it looks for one of the letters A to H.

16.2 Summary of Addressing Modes
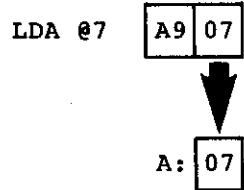
The following sections summarise all the addressing modes that are available on the 6502.

16.3 Immediate

When the data for an instruction is known at the time that the program is being written, immediate addressing can be used. In immediate addressing the second byte of the instruction contains the actual 8-bit data to be used by the instruction.

The '@' symbol denotes an immediate operand.

Instruction:

LDA @7 | A9 | 07 |

A: | 07 |
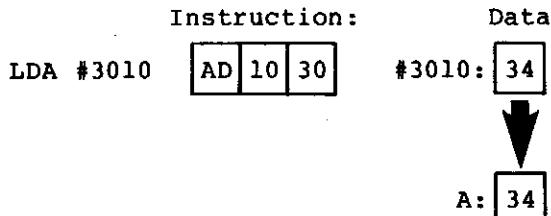
Examples: LDA @M
          CPY @J+2

16.4 Absolute
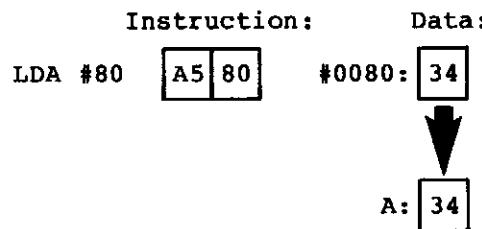
Absolute addressing is used when the effective address, to be used by the instruction, is known at the time the program is being written. In absolute addressing the two bytes following the op-code contain the 16-bit effective address to be used by the instruction.

```
                Instruction:        Data:

LDA #3010       │AD│10│30│      #3010: │34│
                                         │
                                         ▼
                                A:  │34│
```

Examples: LDA K
           SBC #3010

16.5 Zero Page

Zero page addressing is like absolute addressing in that the
instruction specifies the effective address to be used by the
instruction, but only the lower byte of the address is specified in
the instruction. The upper byte of the address is assumed to be zero,
so only addresses in page zero, from #0000 to #00FF, can be addressed.
The assembler will automatically produce zero-page instructions when
possible.

```
            Instruction:       Data:

LDA #80     │A5│80│       #0080: │34│
                                  │
                                  ▼
                         A:  │34│
```

Examples: BIT #80
           ASL #9A

16.6 Indexed Addressing
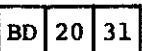
Indexed addressing is used to access a table of memory locations by
specifying them in terms of an offset from a base address. The base
address is known at the time that the program is written; the offset,
which is provided in one of the index registers, can be calculated by
the program.
     In all indexed addressing modes one of the 8-bit index registers,
X and Y, is used in a calculation of the effective address to be used
by the instruction. Five different indexed addressing modes are
available, and are listed in the following section.

16.6.1 Absolute,X - Absolute,Y

The simplest indexed addressing mode is absolute indexed addressing.
In this mode the two bytes following the instruction specify a 16-bit
address which is to be added to one of the index registers to form the
effective address to be used by the instruction:

```
                    Instruction:

LDA #3120,X    |BD|20|31|
                         \
                          \
                           + = #3132: |78|   Data:
                          /
                         /                     ↓
               X: |12|
                                        A: |78|
```

Examples: LDA M,X
          LDX J,Y
          INC N,X

16.6.2 Zero,X

In Zero,X indexed addressing the second byte of the instruction
specifies an 8-bit address which is added to the X-register to give a
zero-page address to be used by the instruction.
    Note that in the case of the LDX instruction a zero,Y addressing
mode is provided instead of the zero,X mode.

```
                    Instruction:

LDA #80,X    |B6|80|
                    \
                     \                Data:
                      + = #0082: |78|
                     /
                    /                  ↓
            X: |02|
                                A: |78|
```

Examples: LSR #80,X
          LDX #82,Y

16.7 Indirect Addressing

It is sometimes necessary to use an address which is actually computed
when the program runs, rather than being an offset from a base address
or a constant address. In this case indirect addressing is used.
    The indirect mode of addressing is available for the JMP
instruction. Thus control can be transferred to an address calculated
at the time that the program is run.

Examples: JMP (#2800)
          JMP (#80)
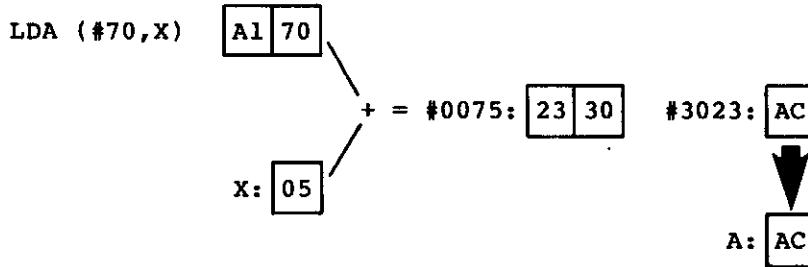
    For the dual-operand instructions ADC, AND, CMP, EOR, LDA, ORA,
SBC, and STA, two different modes of indirect addressing are provided:
pre-indexed indirect, and post-indexed indirect. Pure indirect
addressing can be obtained, using either mode, by first setting the
respective index register to zero.

120

16.7.1 Pre-Indexed Indirect

This mode of addressing is used when a table of effective addresses is provided in page zero; the X index register is used as a pointer to select one of these addresses from the table.

In pre-indexed indirect addressing the second byte of the instruction is added to the X register to give an address in page zero. The two bytes at this page zero address are then used as the effective address for the instruction.

```
                    Instruction:

LDA (#70,X)    A1 70

                         + = #0075: 23 30    #3023: AC

               X: 05

                                            A: AC
```
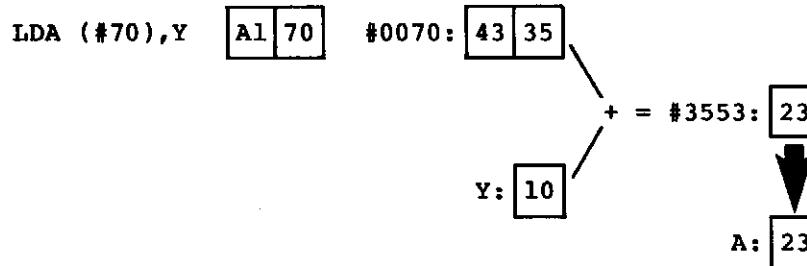
Examples: STA (J,X)
          EOR (#60,X)

16.7.2 Post-Indexed Indirect

This indexed addressing mode is like the absolute,X or absolute,Y indexed addressing modes, except that in this case the base address of the table is provided in page zero, rather than in the bytes following the instruction. The second byte of the instruction specifies the page-zero base address.

In post-indexed indirect addressing the second byte of the instruction specifies a page zero address. The two bytes at this address are added to the Y index register to give a 16-bit address which is then used as the effective address for the instruction.

```
                    Instruction:

LDA (#70),Y    A1 70    #0070: 43 35

                                    + = #3553: 23

                         Y: 10

                                            A: 23
```

Examples: CMP (J),Y
          ADC (#66),Y

16.8 Registers

This section gives a short description of all the 6502's registers:

Accumulator - A

8-bit general-purpose register, which forms one operand in all the arithmetic and logical instructions.

Index Register - X

8-bit register used as the offset in indexed and pre-indexed indirect addressing modes, or as a counter.

Index Register - Y

8-bit register used as the offset in indexed and post-indexed indirect addressing modes.

Status Register - S

8-bit register containing status flags and interrupt mask:

Bit 0 - Carry flag (C). Set if a carry occurs during an add operation; cleared if a borrow occurs during a subtract operation; used as a ninth bit in the shift and rotate instructions.

Bit 1 - Zero flag (Z). Set if the result of an operation is zero; cleared otherwise.

Bit 2 - Interrupt disable (I). If set, disables the effect of the IRQ interrupt. Is set by the processor during interrupts.

Bit 3 - Decimal mode flag (D). If set, the add and subtract operations work in binary-coded-decimal arithmetic; if clear, the add and subtract operations work in binary arithmetic.

Bit 4 - Break command (B). Set by the processor during a BRK interrupt; otherwise cleared.

Bit 5 - Unused.

Bit 6 - Overflow flag (V). Set if a carry occurred from bit 6 during an add operation; cleared if a borrow occurred to bit 6 in a subtract operation.

Bit 7 - Negative flag (N). Set if bit 7 of the result of an operation is set; otherwise cleared.

Stack Pointer - SP

8-bit register which forms the lower byte of the address of the next free stack location; the upper byte of this address is always #01.

Program Counter - PC

16-bit register which always contains the address of the next instruction to be fetched by the processor.

17 Machine-Code in BASIC

Machine-code subroutines written using the mnemonic assembler can be incorporated into BASIC programs, and several examples are given in the following sections.

17.1 Replace Subroutine

The following machine-code routine, 'Replace', can be used to perform a character-by-character substitution on a string. It assumes the existence of three strings called R, S, and T. The routine looks up each character of R to see if it occurs in string S and, if so, it is replaced with the character in the corresponding position in string T.

For example, if:

    $S="TMP"; $T="SNF"

then the sequence:

    $R="COMPUTER"
    LINK RR0

will change $R to "CONFUSER".

```
 10 REM Replace
 20 DIM LL(4),R(20),S(20),T(20)
 40 FOR N=1 TO 2; DIM P(-1)
 50[
 60:LL0 LDX @0
 70:LL1 LDY @0; LDA R,X
 80 CMP @#D; BNE LL3; RTS finished
 90:LL2 INY
100:LL3 LDA S,Y
110 CMP @#D; BEQ LL4
120 CMP R,X; BNE LL2
130 LDA T,Y; STA R,X replace char
140:LL4 INX; JMP LL1 next char
150]
160 NEXT N
200 END
```

The routine has many uses, including code-conversion, encryption and decryption, and character rearrangement.

17.1.1 Converting Arabic to Roman Numerals

To illustrate one application of the Replace routine, the following program converts any number from Arabic to Roman numerals:

```
 10 REM Roman Numerals
 20 DIM LL(4),Q(50)
 30 DIM R(20),S(20),T(20)
 40 FOR N=1 TO 2; DIM P(-1)
 50[
 60:LL0 LDX @0
 70:LL1 LDY @0; LDA R,X
```

```
  80 CMP @#D; BNE LL3; RTS finished
  90:LL2 INY
 100:LL3 LDA S,Y
 110 CMP @#D; BEQ LL4
 120 CMP R,X; BNE LL2
 130 LDA T,Y; STA R,X replace char
 140:LL4 INX; JMP LL1 next char
 150]
 160 NEXT N
 200 $S="IVXLCDM"; $T="XLCDM??"
 210 $Q=""; $Q+5="I"; $Q+10="II"
 220 $Q+15="III"; $Q+20="IV"; $Q+25="V"
 230 $Q+30="VI"; $Q+35="VII"
 240 $Q+40="VIII"; $Q+45="IX"
 250 DO $R="";D=10000
 255 INPUT A
 260 DO LINK LL0
 270 $R+LEN(R)=$(Q+A/D*5)
 280 A=A%D; D=D/10; UNTIL D=0
 290 PRINT $R; UNTIL 0
```

Description of Program:

20-30	Allocate labels and strings
40-160	Assemble Replace routine.
200	Set up strings of Roman digits
210-240	Set up strings of numerals for 0 to 9.
255	Input number for conversion
260	Multiply the Roman string R by ten by performing a character substitution.
270	Append string for Roman representation for A/D to end of R.
280	Look at next digit of Arabic number.
290	Print Roman string, and carry on.
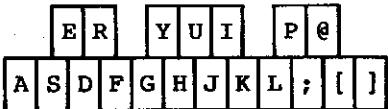
Variables:
A - Number for conversion
D - Divisor for powers of ten.
LL(0..4) - Labels for assembler routine.
LL0 - Entry point for Replace routine.
N - Counter for two-pass assembly.
P - Location counter.
Q - $(Q+5*X) is string for Roman numeral X.
$R - String containing Roman representation of A.
$S - Source string for replacement.
$T - Target string for replacement.

Program size: 579 bytes.

17.2 Harpsichord

The following program simulates a harpsichord; it uses the central section of the ATOM's keyboard as a harpsichord keyboard, with the keys assigned as follows:

where the S key corresponds to middle C. The space bar gives a 'rest', and no other key on the keyboard has any effect.
    The tune is displayed on a musical stave as it is played, with the

black notes designated as sharps. Pressing RETURN will then play the music back, again displaying it as it is played.

The program uses the Index routine, described in Section 16.3, to look up the key pressed, and a version of the Bleep routine in Section 15.4.1.

```
  1 REM Harpsichord
 10 DIM S(23),T(26),F(0)
 15 DIM WW(2),RR(2),Z(128)
 20 DIM P(-1)
 30 PRINT $21
100[\GENERATE NOTE
110:WW0 STA F; LDA @0
120:WW2 LDX F
130:WW1 DEX; NOP; NOP; BNE WW1
140 EOR @4; STA #B002
150 DEY; BNE WW2; RTS
160\READ KEY & LOOK UP IN T
165:RR1 STX F; RTS
170:RR0 JSR #FFE3
180 LDX @25
190:RR2 CMP T,X; BEQ RR1
210 DEX; BPL RR2; BMI RR0
220]
230 PRINT $6
380 X=#8000
390 D=256*#22
393 S!20=#01016572
395 S!16=#018898AB
400 S!12=#01CBE401
410 S!8=#5A606B79
420 S!4=#8090A1B5
430 S!0=#C0D7F2FF
450 $T="ASDFGHJKL;[]?ER?YUI?P@? ?"
460 T?24=#1B; REM ESCAPE
470 CLEAR 0
480 DO K=32
500 FOR M=0 TO 127; LINK RR0
505 IF ?F<>25 GOTO 520
508 IF M<>0 Q=M
510 M=128; GOTO 540
520 Z?M=?F
530 GOSUB d
540 NEXT M
780 K=32
800 FOR M=0 TO Q-1; WAIT; WAIT
810 ?F=Z?M; GOSUB d
820 NEXT M
825 UNTIL 0
830dREM DRAW TUNE
840 IF K<31 GOTO e
850 CLEAR 0
860 FOR N=34 TO 10 STEP -6
870 MOVE 0,N; DRAW 63,N
880 NEXT N
890 K=0
900eIF ?F=23 GOTO s
910 IF ?F>11 K?(X+32*(27-?F))=35; K=K+1
920 K?(X+32*(15-?F%12))=15
930 K=K+1
```

125

```
960 A=S?(?F); Y=D/A
970 LINK WW0
980 RETURN
990sFOR N=0 TO 500;NEXT N
995 K=K+l; RETURN
```

Description of Program:

100-150	Assemble bleep routine
160-210	Assemble index routine
393-430	Set up note values
450-460	Set up keyboard table
480-825	Main program loop
500-540	Play up to 128 notes, storing and displaying them.
800-820	Play back tune
830	d: Draw note on staves and play note
840-880	If first note of screen, draw staves
900-920	Plot note on screen
960-970	Play note
990-995	Wait for a rest

Variables:
A - Note frequency
D - Duration count
?F - Key index
K - Column count on screen
M - Counter
N - Counter
P - Location counter
Q - Number of notes entered
RR(0..2) - Labels in index routine
RR0 - Entry point to read routine
S?0..S?23 - Vector of note periods
T?0..T?26 - Vector of keys corresponding to vector S
WW(0..2) - Labels in note routine
WW0 - Entry point to note routine
X - Screen address
Y - Number of cycles of note to be generated
Z(0..128) - Array to store tune.

Program size: 1049 bytes
Extra storage: 205 bytes
Machine code: 41 bytes
Total size: 1295 bytes

17.3 Bulls and Cows or Mastermind

Bulls and Cows is a game of logical deduction which has become very
popular in the plastic peg version marketed as 'Mastermind'. In this
version of the game the human player and the computer each think of a
'code', consisting of a string of four digits, and they then take
turns in trying to guess the other player's code. A player is given
the following information about his guess:

The number of Bulls - i.e. digits correct and in the right position.

The number of Cows - i.e. digits correct but in the wrong position.

Note that each digit can only contribute to one Bull or one Cow. The
human player specifies the computer's score as two digits, Bulls
followed by Cows. For example, if the code string were '1234' the
score for guesses of '0004', '4000', and '4231' would be '10', '01',
and '22' respectively.

    The following program plays Bulls and Cows, and it uses a

combination of BASIC statements to perform the main input and output operations, and assembler routines to speed up sections of the program that are executed very frequently; without them the program would take several minutes to make each guess.

```
  10 REM Bulls & Cows
  20 DIM M(3),N(3),C(0),B(0),L(9)
  23 DIM GG(10),RR(10)
  25 DIM LL(10)
  50 GOSUB z; REM Assemble code
  60 GOSUB z; REM Pass Two
1000 REM MASTERMIND *****
1005 Y=1; Z=1
1007 @=2
1010 GOSUB c
1015 G=!M ;REM MY NUMBER
1020 GOSUB c; Q=!M
1030 I=0
1040 DO I=I+1
1050 PRINT "(" I ")" '
1100 IF Y GOSUB a
1150 IF Z GOSUB b
1350 UNTIL Y=0 AND Z=0
1400 PRINT "END"; END
1999********************************
2000 REM Find Possible Guess
2010fGOSUB c; F=!M
2160wLINK LL7
2165 IF !M=F PRINT "YOU CHEATED"; END
2170 X=1
2180v!N=GG(X)
2190 LINK LL2
2200 IF !C&#FFF<>RR(X) THEN GOTO w
2210 IF X<I THEN X=X+1; GOTO v
2220 Q=!M; RETURN
3999********************************
4000 REM Choose Random Number
4005cJ=ABSRND
4007 REM Unpack Number
4010uFOR K=0 TO 3
4020 M?K=J%10
4030 J=J/10
4040 NEXT
4050 RETURN
4999********************************
5000 REM Print Guess
5010gFOR K=0 TO 3
5020 P. $(H&15+#30)
5030 H=H/256; NEXT
5040 RETURN
5999********************************
6000 REM Your Turn
6040aPRINT "YOUR GUESS"
6045 INPUT J
6050 GOSUB u
6060 !N=G
6065 LINK LL2
6070 P.?B" BULLS, " ?C" COWS"'
6075 IF!C<>#400 RETURN
6080 IF Z PRINT"...AND YOU WIN"'
```

127

```
6083 IF Z:1 PRINT" ABOUT TIME TOO!"'
6085 Y=0
6090 RETURN
6999**********************************
7000 REM My Turn
7090bPRINT " MY GUESS: "
7100 H=Q; GOSUB g
7110 PRINT '
7120 INPUT "REPLY" V
7140 RR(I)=(V/10)*256+V%10
7150 GG(I)=Q
7225 IF V<>40 GOSUB f; RETURN
7230 IF Y PRINT"...SO I WIN!"'
7235 Z=0
7240 RETURN
7999**********************************
9000zREM Find Bulls/Cows
9035 PRINT $#15 ;REM Turn off screen
9045 DIM P(-1)
9050[
9055\ find bulls & cows for m:n
9060:LL2 LDA @0; LDX @13   ZERO L,B,C
9065:LL3 STA C,X; DEX; BPL LL3
9100 LDY @3
9105:LL0
9120 LDA M,Y
9130 CMP N,Y is bull?
9140 BNE LL4 no bull
9150 INC B count bull
9160 BPL LL1 no cows
9165:LL4
9170 TAX  not a bull
9180 INC L,X
9190 BEQ LL6
9200 BPL LL5 not a cow
9210:LL6 INC C
9220:LL5 LDX N,Y; DEC L,X
9225 BMI LL1; INC C
9260:LL1 DEY; BPL LL0 again
9350 RTS
9360\ increment M
9370:LL7 SED; SEC; LDY @3
9380:LL9 LDA M,Y; ADC @#90
9390 BCS LL8; AND @#0F
9400:LL8 STA M,Y; DEY
9410 BPL LL9; RTS
9500]
9900 PRINT $#6 ;REM Turn Screen on
9910 RETURN

Description of Program:
20-25     Declare arrays and vectors
50-60     Assemble machine code
1010      Computer chooses code
1020      Choose number for first guess
1040-1350 Main program loop
1050      Print turn number
1100      If you have not finished - have a turn
1150      If I have not finished - my turn
1350      Carry on until we have both finished
```

```
1999      Lines to make listing more readable.
2000-3999 f: Find a guess which is compatible with all your replies to
          my previous guesses.
4000-4999 c: Choose a random number
4007-4050 u: Unpack J into byte vector M, one digit per byte.
5000-5040 g: Print guess in K as four digits.
6000-6090 a: Human's guess at machine's number; print score.
7000-7240 b: Machine's guess at human's code.
9000-9910 z: Subroutine to assemble machine-code routines
9055-9350 Find score between numbers in byte vectors M and N; return in
          ?B and ?C.
9360-9500 Increment number in vector M, in decimal, one digit per byte.
```

Variables:
?B - Number of Bulls between vectors M and N
?C - Number of Cows between vectors M and N
GG(1..10) - List of human's guesses
H - Computer's number
I - Turn number
J - Human's guess as 4-digit decimal number
K - Counter
L - Vector to count occurrences of digits in numbers
LL(0..10) - Labels in assembler routines
LL2 - Entry point to routine to find score between 2 codes
LL7 - Entry point to routine to increment M
!M, !N - Code numbers to be compared
P - Location counter
Q - Computer's guess, compatible with human's previous replies.
RR(1..10) - List of human's replies to guesses GG(1..10)
Y - Zero if human has finished
Z - Zero if computer has finished.

Program size: 1982 bytes
Additional storage: 152 bytes
Machine-code: 223 bytes
Total storage: 2357 bytse

Sample run:

```
>RUN
( 1)
YOUR GUESS?1122
 0 BULLS,   0 COWS
 MY GUESS: 6338
REPLY?10
( 2)
YOUR GUESS?3344
 0 BULLS,   0 COWS
 MY GUESS: 6400
REPLY?20
( 3)
YOUR GUESS?5566
 0 BULLS,   0 COWS
 MY GUESS: 6411
REPLY?10
( 4)
YOUR GUESS?7788
 1 BULLS,   1 COWS
 MY GUESS: 6502
REPLY?40
...SO I WIN!
( 5)
```

YOUR GUESS?

18 ATOM Operating System

18.1 Keyboard

18.1.1 Teletype/Typewriter Modes

After switching on, or typing BREAK, the ATOM is in teletype mode. In this mode all the alphabetic keys produce upper case letters, and the SHIFT key is used to obtain the lower-case letters. This mode is most convenient for normal operation of the ATOM because all commands are typed in upper case.

When entering documents which contain mixed lower and upper case it is convenient to have the ATOM keyboard behave like a standard typewriter; i.e. for the alphabetic keys to produce lower case, and upper case when shifted. This state may be obtained by typing the LOCK key. The mode is cancelled by typing LOCK a second time. Note that the LOCK key only affects the alphabetic keys, A – Z.

18.1.2 SHIFT Key

All but one of the 128 ASCII codes are available from the ATOM keyboard. The code which cannot be obtained appears as a back-arrow on the display.

The codes which can be obtained, but which are not marked on the keyboard, are as follows:

SHIFT +	Displayed as	ASCII character	Code in hex
@	Inverted @	\	#60
A	Inverted A	a	#61
.	.	.	.
.	.	.	.
Z	Inverted Z	z	#7A
[	Inverted [	{	#7B
\	Inverted \	|	#7C
]	Inverted]	}	#7D
^	Inverted ^	~	#7E

18.1.3 Control Codes

The following list gives all the control codes that perform special functions on the ATOM. They are all available from the keyboard, by typing CTRL with the specified key, or from programs.

STX (CTRL-B, 2) Start printer

This code, which is not sent to the printer, starts the printer output stream. All further output is sent to the printer as well as the VDU until receipt of an ETX code.

ETX (CTRL-C, 3) End printer

Ends the printer output stream.

ACK (CTRL-F, 6) Start screen

131

Starts the output stream to the VDU screen, and resets the VDU to character mode. This code is sent to the VDU on BREAK.

BELL (CTRL-G, 7) Bleep

Causes the output stream to make a 1/2 second bleep on the internal speaker.

BS (CTRL-H, 8) Backspace

Moves the cursor back one position.

HT (CTRL-I, 9) Horizontal tab

Moves the cursor forward one position.

LF (CTRL-J, 10) Linefeed

Moves the cursor down one line.

VT (CTRL-K, 11) Vertical tab

Moves the cursor up one line.

FF (CTRL-L, 12) Formfeed

Clears the screen, moves the cursor to the top left-hand corner, and sets the VDU to character mode.

CR (CTRL-M, 13) Return

Moves the cursor to the start of the current line.

SO (CTRL-N, 14) Page mode on

Turns on paged mode, and resets the line count to zero. Every time the screen in scrolled the line count is incremented. In paged mode the VDU will wait for a key to be typed every time the line count reaches 16.

SI (CTRL-O, 15) Page mode off

Turns off paged mode. This is the mode set on BREAK and on power-up.

NAK (CTRL-U, 21) End screen

Ends the output stream to the VDU; the only code recognised when in this condition is ACK.

CAN (CTRL-X, 24) Cancel

Deletes the line currently being typed, and returns the cursor to the start of the following line. Only happens in BASIC's input modes.

ESC (CTRL-[, 27) Escape

Causes an escape from an executing BASIC program. If typed twice, resets the VDU to character mode.

RS (CTRL-^, 30) Home cursor

Moves the cursor to the top left-hand corner of the screen.

18.4 Screen Editing

Three keys on the ATOM keyboard have special functions, and are used in conjunction with the SHIFT key for screen editing. Their functions

are:

```
        ▲
        ▼        Cursor up
SHIFT ▲▼         Cursor down

        ◄►       Cursor right
SHIFT ◄►         Cursor left

COPY             Read character under cursor
```

Pressing the first four key combinations move the cursor around the screen but do not send any new characters down the input channel. They may be typed at any time and will have no effect on the ATOM, or on programs; they just determine where the cursor is positioned.

The COPY key will read the character under the cursor, and transmit that character to the input stream; the effect is the same as if that character had been typed at the keyboard. After reading a character the cursor is automatically moved one place to the right.

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

```
>LIST
   10 PIECE OF TEXTUAL MATERIAL
>█
```

After listing the program the cursor is positioned after the prompt, as shown. First move the cursor vertically upwards, using the ▲▼ key, until it is opposite the line we wish to edit:

```
>LIST
█   10 PIECE OF TEXTUAL MATERIAL
>
```

Now use the COPY key to read the correct part of the line:

```
>LIST
   10 PIECE OF█TEXTUAL MATERIAL
>
```

Note that the cursor inverts every character it passes over. If any inverted characters are present in the text, these will be un-inverted by the cursor.

Now type in the corrected part of the line:

```
>LIST
   10 PIECE OF CAKE█AL 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 ◄► key may be used to omit parts of the old line that are no longer required, and SHIFT ◄► 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.

18.5 The VDU

The character display shows the contents of memory from #8000 to #81FF, mapped one character cell per byte. The address of the top

133

left-hand cell is #8000, and the address of the Cth column in the Lth line is simply:

$$#8000+32*L+C$$

where $0<=C<=31$ and $0<=L<=15$, and L=0, C=0 corresponds to the top left-hand character position.

The value stored in the memory cell determines the character displayed. All 256 different possible codes produce different displayed characters (with two exceptions), and the codes are assigned as follows:

Hex Code:	Characters:
#00 - #1F	@ to <- (including alphabet)
#20 - #3F	space to ? (including digits)
#40 - #7F	white graphics symbols
#80 - #9F	inverted @ to <-
#A0 - #BF	inverted space to ?
#C0 - #FF	grey graphics symbols

The complete character set is displayed by executing:

```
FOR N=0 TO 255; N?#8000=N; NEXT N
```

which will generate the display shown below:

The graphics symbols consist of a block divided into 6 pixels, the state of each pixel being determined by the lower 6 bits of the byte, as follows:

5	4
3	2
1	0

If the bit is set, the corresponding pixel is grey or white; if the bit is clear the pixel is black. Note that #20 and #40, and #7F and #A0 give the same graphics patterns.

Note that in all cases except #20 to #3F the code stored in the cell differs from the ASCII code for the character displayed. If C is the ASCII code for the character to be displayed, the code to be stored in the cell is obtained by:

```
C=C+#20; IF C<#80 THEN C=C:#60
```

Similarly, to obtain the ASCII code for a character from the value V stored in the screen memory, execute:

```
IF V<#80 THEN V=V:#60
V=V-#20
```

18.6 Changing Text Spaces

The 'text space' is the region of memory used by the ATOM for storing the text of programs. On switching on, or pressing BREAK, the ATOM is initialised with a fixed text space at address #8200 in the unexpanded ATOM, or at #2900 in the ATOM with extra memory in the lower text space. 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 use like this, so its use as a memory space allocator and pointer to the end of text in the line editor must be watched carefully.

18.6.1 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=#82
    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=#83
    NEW
 10 REM CALL SUBROUTINE IN #82
 20 ?18=#82
 30 GOSUB 10
 40 REM PROVE YOU'RE BACK
 50 PRINT"TEXT AREA TWO"'
 60 GOSUB 10
 70 ?18=#83;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=#82

will not change the value of TOP:

```
    PRINT & TOP'
   8398
```

To reset TOP, type:

```
     END
     PRINT & TOP'
8225
```

18.7 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.

18.7.1 Renumbering in the Expanded ATOM

In an expanded ATOM, with the default text space at #2900, the renumber routine can conveniently be entered at #8200 by typing:

```
     ?18=#82
     NEW
```

before loading it from tape, or entering it from the keyboard.

```
  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
```

The program to be renumbered should be in the default text space, #29. Then RUN the program, and reply to the prompts as follows:

```
TEXT SPACE TO RENUMBER ?#29
START AT?10
STEP?10
```

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

18.7.2 Renumbering Using the Screen Memory

In an unexpanded ATOM there may be no space in the upper text space to load the renumber program. However, with care, it can be loaded from tape, or typed in, and executed in the area of memory that is displayed on the VDU. The size of the program is about #A0 bytes, which will occupy memory corresponding to about 6 lines of the display. Provided that the cursor is kept below the sixth line of the display, and is not allowed to reach the bottom line of the display when it will cause scrolling, the VDU memory can be safely used as a temporary text space in this way.

First type:

```
     ?18=#80
```

to set the text space to the screen area of memory. Move the cursor to the 6th. line of the display using the ↕ edit key, and type:

```
     LOAD "RENUMBER"
```

Alternatively, enter the program from the keyboard in the usual way. The top few lines of the display will be filled with strange

136

characters, corresponding to the text of the program stored directly in the screen memory. Now type:

    RUN

and reply to the prompts of the renumber program as follows (or, as desired):

TEXT SPACE TO RENUMBER?#82
START AT?10
STEP?10

When the program has run the screen can be allowed to scroll, corrupting the renumber program, and you can list the renumbered program.

18.8 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:

    @=1;P.$6$7'"ERROR "?0;@=8;IF?1☐?2P." 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.

18.8.1 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 DIM A(8)
    20 $A="GOTO 170"
    30 ?16=A; ?17=A&#FFFF/256

18.8.2 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. The program uses integer BASIC statements, but could easily be modified to use the floating-point extension:

    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

19 Cassette Operating System

The Cassette Operating System, or COS, saves and loads data to and from tape using the Computer Users Tape Standard (CUTS), which is also known as Kansas City Standard. Data is coded as audio tones on the tape. A logic 0 consists of 4 cycles of a 1.2 kHz tone, and a logic 1 consists of 8 cycles of a 2.4 kHz tone. Each byte of data is preceeded by a logic zero start bit, and is terminated by a logic 1 stop bit. Each bit lasts for 3.33 mS, giving an operating speed of 300 bits/second.

19.1 Named Files

Named files are stored as a number of blocks, each of which is 256 bytes or less, and includes a checksum over all the bytes in the block. Each block is identified by a name header, and includes the start address for loading that block, the execution address for that block, and the number of bytes in that block minus one.

19.2 Unnamed Files

Unnamed files are stored as a two-byte start address, a two-byte end address, and end minus start bytes of data. An unnamed file could have no name at all (when using *LOAD and *SAVE), or it may have a zero length name denoted by "". Unnamed files may thus be used anywhere that named files could be used. The format of an unnamed file on tape corresponds to the format of an Acorn System One computer.

19.3 Commands

All COS commands start with an asterisk to distinguish them from BASIC commands. Note the difference between SAVE and *SAVE, and LOAD and *LOAD:

SAVE creates text files from the ATOM's text space. No start address is specified. The execution address is automatically set to #C2B2, the entry point to BASIC.
*SAVE saves a block of memory whose start and end addresses must be specified.

LOAD loads text files to the current text space.
*LOAD loads a block of memory to a fixed address, or to an address specified in the command.

*CAT   Catalogue tape   *.

The *CAT command gives a catalogue of a tape. Each block of a named file will appear in the catalogue as follows:

FILENAME   SSSS EEEE NNNN BB

where FILENAME is the name of the file
      SSSS is the start address of the block
      EEEE is the execution address of the file (used by RUN)
      NNNN is the block number, starting at zero
and   BB is the number of data bytes in the block, minus one.
All the numbers are in hexadecimal.

139

Unnamed files will appear in the catalogue as:

        SSSS LLLL

where SSSS is the start address
and   LLLL is the last address, plus one. Again, both numbers are in
hexadecimal.

*LOAD   Load file   *L.

To load a named file the syntax is:

       *LOAD "FILENAME" XXXX

where XXXX is a hexadecimal address specifying where the data is to be
loaded. If XXXX is omitted the data will be loaded back to the address
from which it was originally saved. On pressing RETURN the system will
reply:

PLAY TAPE

The cassette recorder should be played, and the ATOM's space bar
pressed to indicate that this has been done.   .
     The COS will display the names of any files that are encountered
on the tape before the specified file is found. When the file is found
it will be loaded and on completion the '>' prompt will reappear.
     If the file to be loaded is part way past the tape heads the COS
will display:

REWIND TAPE

The tape should then be rewound and the space bar pressed again, to
which the COS will reply:

PLAY TAPE

and the loading process can be repeated.
To load an unnamed file the syntax is:

       *LOAD "" XXXX   or
       *LOAD XXXX

where XXXX is again the optional, hexadecimal, start address. Since
there is no name search the space bar should only be pressed during
the high-tone leader, and the first file encountered will be loaded.
Unnamed files consist of a single block, and there is no error
checking; however they provide the fastest way of saving and loading
data or programs.

CTRL and SHIFT

During loading and *CAT:

CTRL   will cause a return to the ATOM '>' prompt. If pressed during
loading an error message will be given to indicate that part of the
file being loaded was lost.

SHIFT will override the search for the high-tone leader, and can thus
be used to load and catalogue files with very short periods of
high-tone leader.

Note that there is no way to exit from SAVE or *SAVE except by BREAK.

*SAVE   Save file   *S.

To save a named file on tape the syntax is:

       *SAVE "FILENAME" SSSS LLLL EEEE

where FILENAME is the filename of up to 16 characters

140

```
              SSSS is the start address
              LLLL is the end address plus one
              EEEE is the optional execution address
```
The execution address is used by the RUN command, and if omitted will default to the start address.

On pressing return the COS will respond with:

RECORD TAPE

The tape recorder should now be started in record mode, and the space bar pressed to indicate that this has been done. Once started, SAVE cannot be aborted except by BREAK.

To save an unnamed file the syntax is:

```
    *SAVE "" SSSS LLLL    or
    *SAVE SSSS LLLL
```

where SSSS and LLLL are as above, and the data will be saved as one continuous block.

*MON   Enable messages   *M.

The usual condition after switch-on and BREAK is for the messages:

PLAY/RECORD/REWIND TAPE

to be produced. The MON command may be used to enable messages if they have been disabled.

*NOMON Disable messages   *N.

This command turns off messages produced by the COS.

*FLOAD Finish loading   *F.

The normal LOAD command demands that files are loaded from the start of the first block, and will request that the tape be rewound if started in the middle of the file. FLOAD allows loading to commence from the start of any block in the file, and the syntax of the command is:

```
    FLOAD "FILENAME" SSSS
```

where SSSS is an optional start address specifying the address to which the start of the first block is loaded if relocation is required.

FLOAD is useful after a checksum error has been encountered. The tape may be stopped and rewound to any point before the block that produced the error. FLOAD is then used to allow loading to continue, and the block headers will ensure that the blocks are being loaded in the correct place.

*RUN   Load and execute machine code file   *R.

The syntax of this command is:

```
    RUN "FILENAME" SSSS
```

The function is exactly as for LOAD, but on completion of loading execution is transferred to the execution address specified when the file was created. The optional start address SSSS may be used to relocate the file. The execution address is not affected by relocation.

This command initialises the Disk Operating System, if present, by
linking to #E000.   *ERROR 12.*

19.4 Errors

The following error messages are given for errors in commands to the
COS; i.e. for commands starting with '*':

SUM
ERROR 6   Checksum error

COM?
ERROR 48   Command error

NAME
ERROR 118   Name error

SYN?
ERROR 135   Syntax error

ERROR 165   Premature exit from loading

19.5 Appending Text from Several Files

A BASIC or Assembler subroutine may often be required for several
different programs. In this case it is possible to store the
subroutine text on a separate file, and append this text to the text
in memory every time the subroutine is needed in a program.

   The subroutine text should be entered in memory on its own, and
should be written with fairly high line numbers, such as 9000-9999.
The subroutine is then saved as usual; e.g.:

        SAVE "SUB9"

At a later date a program is written which needs a copy of this
subroutine. First check that the program does not use any line numbers
above the first line of the program. Then find the address of the end
of the program by typing:

        PRINT &TOP-2

Remember that this address will be in hexadecimal. Now, using *LOAD,
load the subroutine into memory starting at the address printed out in
the above step:

        *LOAD "SUB9" XXXX

where XXXX is the address that was printed out. Finally, to reset TOP
to the end of the subroutine, type:

        END

   Any number of text files can be appended in this way, but note
that, unless the resulting text is to be renumbered, the parts
appended should use line numbers which are larger than any line number
in the text file already in memory.

20 BASIC Statements, Functions, and Commands

All the ATOM 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 26:

\<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), AA(3), !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's 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's 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

CLEAR   Clear graphics screen   **CLEAR**

This statement clears the screen and initialises the display for the
graphics mode specified its argument:

CLEAR 0 : Screen is 64*48 (semi-graphics mode)
CLEAR 1 : Screen is 128*64
CLEAR 2 : Screen is 128*96
CLEAR 3 : Screen is 128*192
CLEAR 4 : Screen is 256*192

In graphics modes 1 to 4 an error will be caused if the text space and
graphics area conflict.

144

COUNT   Count of characters printed   C.

This function returns the number of characters printed since the last
return, and is thus 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=20
=====================>

DIM   Dimension statement   DIM

This statement automatically allocates space after the end of the text
for arrays or strings. DIM causes an error if used in direct mode.
Associated with DIM is a 16 bit location referred to as the 'free
space pointer'. The RUN statement sets this pointer to the value of
TOP. A declaration:

        DIM A(Q)

sets A to the current value of the free space pointer, and the pointer
is moved up by (Q+1) bytes. A declaration:

        DIM AA(Q)

allocates space for an array AA with elements AA(0) to AA(Q), and
moves the value of the free space pointer up by 4*(Q+1) bytes.
        A special use of DIM is to set the value of P for assembling:

        DIM P(-1)

sets P to the current value of the free space pointer, without
changing the pointer's value. Several items may be dimensioned in one
DIM statement:

        DIM A(2),AA 45,BB(67),CC(F)

DRAW   Draw line to absolute position   DRAW

The statement DRAW A,B is equivalent to PLOT 5,A,B.

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's 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 a 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 the message PLAY TAPE will be printed, and the system
will wait for any key to be pressed.

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 -20*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. Zero will be returned 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) file already in use as an input file
        (e) insufficient memory space

The number returned is only a byte long (1-255) and can be stored in
variables or using ! or ?. Usage of a number not given by the

146

operating system will result in an error.

In the COS the message RECORD TAPE will be printed, and any key waited for. 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's 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's 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   GOS.

This statement gives the ability for programs to call sub programs. 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 an <factor> whose value is a line number, or by a label. No more than 14 GOSUB statements without RETURNs are allowed. Example:

```
 10 GOSUB a
 20 GOSUB a
 30 END
100a PRINT"THIS IS A SUB PROGRAM"'
200 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 !
```

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

148

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   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"

MOVE   Move to absolute position   MOVE

The statement MOVE A,B is equivalent to PLOT 4,A,B.

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

149

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
>
```

OLD   Recover text area   OLD

This statement executes ?(?18*256+1)=0;END to recover a text space after typing NEW. If the first line number in the text area is greater than 255 it will be changed by the OLD statement.

OR   Relational OR   OR

This symbol provides the logical OR operation between two <relational expressions>. 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"'

PLOT   Plot statement   PLOT

This statement takes three arguments: a parameter that determines how to plot, and a pair of relative or absolute cartesian coordinates. The first parameter is as follows:

```
 0 plot line relative to last point with no change in pixels
 1 as 0 but set pixels
 2 as 0 but invert pixels
 3 as 0 but clear pixels

 4 plot line to absolute position with no change in pixels
 5 as 4 but set pixels
 6 as 4 but invert pixels
 7 as 4 but clear pixels

 8 plot point relative to last point with no change in pixel
 9 as 8 but set pixel
10 as 8 but invert pixel
11 as 8 but clear pixel

12 plot point at absolute position with no change in pixel
13 as 12 but set pixel
14 as 12 but invert pixel
15 as 12 but clear pixel
```

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 'newline'.

150

(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's 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's 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's 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's 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's 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's 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 is ignored.

152

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's 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's 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 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

WAIT   Wait statement   **WAIT**

This statement waits until the next 60 Hz vertical sync pulse from the CRT controller. The statement has two uses: to give a delay of one

153

sixtieth of a second, and to wait until flyback so that a subsequent graphics command will not cause noise on the screen. Examples:

```
FOR Z=1 TO 60; WAIT; NEXT; REM wait a second.
MOVE 0,0; WAIT; DRAW 8,8; REM noise-free plotting
```

21 BASIC Characters and Operators

This section lists all the ATOM BASIC 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 26.

21.1 Special Characters

Line terminator {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. Typing escape when in direct mode resets the screen to character mode.

The ESC key can be disabled from a program by executing:

    ?#B000=10

Separator {space}

This character is stored intact to allow formatting 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 8. 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
```

21.2 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.

156

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

157

```
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 truth 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 truth 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:

1. Arithmetic   Example:
    <variable>=<expression>   A=2
    <variable>!<factor>=<expression>   A!J=3
    <variable>?<factor>=<expression>   A?J=4
    !<factor>=<expression>   !J=5
    ?<factor>=<expression>   ?J=6
    <array element>=<expression>   AA(1)=7

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 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

159

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

Ｎ   OR   {inverted backslash}

This character is the binary operation of bitwise logical OR between two 32 bit <term>s. Its form is <term a>Ｎ<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 truth 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 truth 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 truth 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.

22 Extending the ATOM

22.1 Floating-Point Extension to BASIC

The ATOM's BASIC can be extended to provide floating-point arithmetic, and many scientific functions, simply by inserting an extra 4K ROM chip into a socket on the ATOM board (see Technical Manual). The floating-point extension adds 27 new variables, %@ and %A to %Z, 27 floating-point arrays %@@ and %AA to %ZZ, and the following special statements and functions to the existing integer BASIC, including a statement for plotting in the ATOM's four-colour graphics modes:

Floating-Point Statements

COLOUR, FDIM, 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 extension ROM does not in any way alter the operation of the existing BASIC statements, functions, or operators, and floating-point arithmetic may be mixed with integer arithmetic in the same line.

All the extension-ROM statements and functions, except COLOUR and FLT, and all the extension-ROM 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)
```
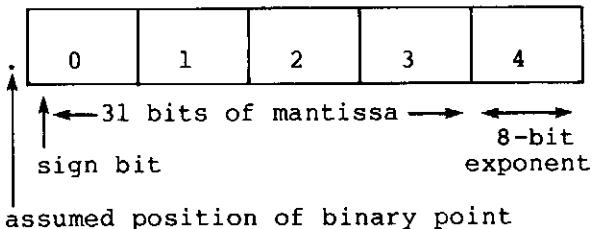
will print 1.00000000. The floating-point function may be obtained in an integer context by prefixing it with the '%' operator. Thus:

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

will print 1.

22.1.1 Floating-Point Representation

Each floating-point number occupies five bytes; a four-byte mantissa and a one-byte exponent:

assumed position of binary point

The mantissa is stored in sign and magnitude form. Since it will always be normalized, it logically always has a '1' as its top bit. This position is therefore used to store the sign. The exponent is an ordinary 8-bit signed number. A higher precision is used for internal calculations to preserve accuracy. The representation provides about 9.5 significant figures of accuracy, and allows for numbers in the range 1E-38 to 1E+38 approximately. All the possible 32-bit integers in the standard integer BASIC can be floated without loss of accuracy.

22.1.2 Floating-Point Statements

FDIM   Floating-point dimension

Allocates space after the end of text for the floating-point arrays %@@ and %AA to %ZZ. Example:

```
      DIM %JJ(5)
```

allocates space for elements %JJ(0) to %JJ(5), a total of 30 bytes.

FIF   Floating-point IF

Same syntax as IF, but connectives such 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:

162

```
                    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, except no connectives (OR or AND) are 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
```

22.1.3 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:

163

```
     FPRINT COS 1
5.40302306E-1
```

DEG   Radians to degrees   D.

Converts its argument from radians to degrees. Example:

```
     FPRINT DEG PI
180.000000
```

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 an angle in radians. 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
```

RAD   Degrees to radians   R.

Converts its argument from degrees to radians. Example:

```
     FPRINT RAD 90
1.57079632
```

SGN   Sign

Returns -1, 0, or 1 depending on whether its floating-point argument is negative, zero, or positive respectively.

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

22.1.4 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:

    DIM A(14); %!A=PI; %!(A+5)=3; %!(A+10)=4

%   Convert to integer   {percent}

The unary % operator converts its floating-point argument to an
integer. For 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>

22.1.5 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

165

by executing:

```
FOR J=0 TO 26*5 STEP 5
%!(#2800+J)=0
NEXT J
```

22.1.6 Examples

The following program plots curves of the sine and tangent functions,
using the floating-point routines.

```
  1 REM Sine and Tangent
  5 PRINT $30 ; CLEAR 0
  7 PRINT"PLOT OF SIN AND TAN FUNCTIONS"
  9 %I=2*PI/64
 10 %V=0
 12 FOR Z=0 TO 64
 15 %V=%V+%I
 20 PLOT13,Z,(22+%(22*SIN%V))
 25 PLOT13,Z,(22+TAN%V)
 30 NEXT
100 END
```

Program size: 206 bytes

The following program plots a cycloid curve:

```
  1 REM Cycloid
 10 %Z=60
 20 CLEAR2
 30 FORQ=0TO359
 40 %S=RAD Q
 50 %R=%Z*SIN(%S*2)
 60 PLOT13,%(%R*SIN%S+64.5),%(%R*COS%S+48.5)
 70 NEXT
 80 END
```

Program size: 142 bytes

22.1.7 Three-Dimensional Plotting

The following program plots a perspective view of a saddle curve, with
any desired viewing point. The program is a floating-point version of
the program in Section 11.5.2.

```
  1 REM Saddle Curve
100 FINPUT"CHOOSE VIEW POSITION"'"X="%L,"Y="%M,"Z="%N
110 FINPUT"LOOKING TOWARDS"'"X="%A,"Y="%B,"Z="%C
115 %L=%L-%A;%M=%M-%B;%N=%N-%C
120 W=4;CLEAR4
150 %S=%L*%L+%M*%M;%R=SQR%S
160 %T=%S+%N*%N;%Q=SQR%T
200 FORX=-10TO10
210 Y=-10;GOS.c;GOS.m
220 FORY=-9TO10;GOS.c;GOS.p;N.;N.
230 FORY=-10TO10
240 X=-10;GOS.c;GOS.m
250 FORX=-9TO10;GOS.c;GOS.p;N.;N.
260 END
400pW=5
410m%U=%X-%A;%V=%Y-%B;%W=%Z-%C
420 %O=(%T-%X*%L-%Y*%M-%Z*%N)*%R
```

166

```
425 FIF %O<0.1 W=4
430 G=%(400*(%Y*%L-%X*%M)*%Q/%O)+128
440 H=%(500*(%Z*%S-%N*(%X*%L+%Y*%M))/%O)+96
460 PLOTW,G,H;W=4;R.
600c%Y=Y;%X=X
610 %Z=.05*(%Y*%Y-%X*%X);R.
```

Description of Program:

100-110	Input view position and shifted origin.
115	Shift view position for new origin.
120	Clear screen and get ready to move.
150-160	Set up constants for plot projection.
200-250	Scan X,Y plane.
400	p: Entry for drawing.
410	m: Entry for moving; also shift coordinates for new origin.
420	Calculate how far away X,Y,Z is from eye.
425	Avoid plotting too close.
430-440	Project image onto plane.
460	Move or draw and return.
600	c: Define function to be plotted.

Variables:
G,H - Plot position on screen
W - 4 for move, 5 for draw.
X,Y - Used to scan X,Y plane.
%A,%B,%C - Position centred on screen.
%L,%M,%N - View position.
%O - Distance of point from eye.
%Q,%R,%S,%T - Constants for projection.
%U,%V,%W - 3D coordinates referred to new origin.
%X,%Y,%Z - 3D coordinates of point being plotted

Program size: 594 bytes

22.2 Colour Graphics Extension - COLOUR

The extension ROM also contains routines for plotting in the colour graphics modes. The following colour graphics modes are available:

Mode:	Resolution:		Memory:
	X:	Y:	
1a	64	64	1 K
2a	128	64	2 K
3a	128	96	3 K
4a	128	192	6 K

The graphics modes are obtained by specifying the CLEAR statement followed by the mode number (without the 'a'), and the COLOUR statement to determine which colour is to be plotted. The parameter to the COLOUR statement determines the colour as follows; on a black and white television or monitor the colours will be displayed as shades of grey:

Value:	Colour:	Grey scale:
0	Green	Grey
1	Yellow	White
2	Blue	Black
3	Red	Black

COLOUR 0 corresponds to the background colour.
    When a colour has been specified, all subsequent DRAW statements will draw lines in that colour. The PLOT statement will 'set' lines

and points in that colour, will always 'clear' to the background colour, and will always 'invert' to a different colour, irrespective of the current COLOUR.

22.2.1 Random Coloured Lines

The following simple program illustrates the use of the COLOUR command by drawing coloured lines between randomly-chosen points on the screen.

```
10 REM Random Coloured Lines
20 CLEAR 4
30 DO COLOUR RND
40 DRAW(ABSRND%128),(ABSRND%192)
50 UNTIL 0
```

22.3 Memory Expansion

The ATOM's memory can be expanded, on the same board, in units of 1K bytes (1024 bytes) up to a maximum on-board memory capacity of 12K bytes. Refer to the Technical Manual for details of how to insert the extra memory devices.   The unexpanded ATOM contains 1K of Block 0 memory, from #0000 to #0400, and 1K of VDU and text-space memory, occupying between #8000 and #8400. The lower half is used by the VDU and graphics mode 0, and the upper half forms the BASIC text-space starting at #8200 and giving 512 free bytes for programs. The three different areas of RAM that can be fitted on the main circuit board are referred to as follows:

Addresses:	Area:
#0000-#0400	Block zero RAM
#2800-#3C00	Lower text space
#8000-#9800	Graphics space/Upper text space

The following stages in expansion are recommended:

22.3.1. Lower Text Space

Extra memory can be added starting at #2800 in the lower text space. If memory is present in this text space BASIC will automatically be initialised using this region as its text space. The text space starts at #2900 to allow space between #2800 and #2900 for the floating-point variables, but if the floating-point scientific package is not being used the extra memory between #2800 and #2900 can be used for the text space by typing:

```
?18=#28
NEW
```

A total of 5K of memory can be added in the extra text space. There are two advantages in using the lower text space for programs:

1. Whenever the graphics memory is accessed noise will be generated on the screen. Although this noise is slight under most circumstances, it can become annoying when running machine-code programs assembled in the upper text area, which is shared with the graphics area. Moving to the lower text area will eliminate this noise.

2. When the upper text area is used it is only possible to use the lower graphics modes. The lower text area permits all graphics modes to be used.

168

22.3.2 Graphics Space

Memory can be added in the graphics area from #8400 up to #9800, providing a total of 6K of graphics memory. This will make the higher graphics modes available, or can be used for programs in the graphics space.

22.4 Versatile Interface Adapter

A Versatile Interface Adapter, or VIA, can be added to the ATOM 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 Acorn Bus connector.

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). The VIA registers occur in the following memory addresses:

Register:	Address:	Name:
Data Register B	#B800	DB
Data Register A	#B801	DA
Data Direction Register B	#B802	DDRB
Data Direction Register A	#B803	DDRA
Timer 1 low counter, latch	#B804	T1CL
Timer 1 high counter	#B805	T1CH
Timer 1 low latch	#B806	T1LL
Timer 1 high latch	#B807	T1LH
Timer 2 low counter, latch	#B808	T2CL
Timer 2 high counter	#B809	T2CH
Shift Register	#B80A	SR
Auxiliary Control Register	#B80B	ACR
Peripheral Control Register	#B80C	PCR
Interrupt Flag Register	#B80D	IFR
Interrupt Enable Register	#B80E	IER
Data Register A	#B80F	DA

On BREAK 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.

22.4.1 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 18.1.3.

22.4.2 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 ordinary I/O, the printer software driver should be removed from the output stream by setting the vector WRCVEC (address #208) to WRCVEC+3; e.g.:

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

22.4.3 Writing to a Port

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

```
10 !#208=!#208+3
20 ?#B80C=0
30 ?#B802=#FF
40 INPUT J
50 ?#B800=J
60 GOTO 40
```

Description of Program:
10	Remove printer drive from port B.
20	Remove all handshaking.
30	Program all lines as outputs.
50	Output byte.

22.4.4 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=#B808
30 !B=65535
40 X=Y
50 B?3=32; Q=!B&#FFFF
60 PRINT 65535-Q-1755 " 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.

170

23 Mnemonic Assembler

The ATOM mnemonic assembler is a full 6502 assembler; by virtue of its close relationship with the BASIC interpreter the mnemonic assembler provides many facilities found only on assemblers for much larger computers, including conditional assembly and macros.

23.1 Location Counter – P

The assembler uses the BASIC variable P as a location counter to specify the next free address of the program being assembled. Before running the assembler P should be set to the address of a free area of memory. This will normally be the free space above the program, and may be conveniently done with the statement:

        DIM P(-1)

which sets P to the address of the first free location in memory after the program, effectively reserving zero bytes for it. Note that P should be the last variable dimensioned.

The location counter may also appear in the operand field of instructions. For example:

        LDX @0
        DEX
        BNE P-1
        RTS

will cause a branch back to the DEX instruction. The program gives a 1279-cycle delay.

23.2 Assembler Delimiters '[' and ']'.

All assembler statements are enclosed inside square brackets '[' and ']'. When RUN is typed each assembler statement is assembled, the assembled code is inserted directly in memory at the address specified by P, the value of P is incremented by the number of bytes in the instruction, and a line of the assembler listing is printed out. A typical line of the listing might be:

    120 2A31 6D 34 12   :LL1 ADC #1234
     ↑   ↑   ↑ ↑   ↑   ↑   ↑
     | | | | | | mnemonic statement
     | | | | | assembler label
     | | | | instruction data/address
     | | location counter
     | statement line number.

Note that '#' denotes a hexadecimal number.

23.3 Labels

Any of the array variables AA-ZZ may be used as labels in the assembler. The label is specified by preceding the array element by a

171

colon ':'. Note that the brackets enclosing the array subscript may be omitted. The labels must be declared in a DIM statement.

The effect of a label is to assign the value of the location counter, P, at that point to the label variable. The label can then be used as an argument in instructions. For example the following program will assemble a branch back to the DEX instruction::

```
10 DIM ZZ(2)  ← 15 DIM P-1   10 DIM ZZZ, P-1
20[
30 LDX @0
40:ZZ1 DEX
50 BNE ZZ1
60 RTS
70]
80 END
```

23.4 Comments

Assembler instructions may be followed by a comment, separated from the instruction by a space:

    101 LDA @7 bell character

Alternatively a statement may start with a '\' backslash, in which case the remainder of the statement is ignored:

    112 \ routine to multiply two bytes

23.5 Backward References

When an assembler program is assembled, by typing RUN, backward references are resolved automatically the first time the assembler is RUN, because the associated labels receive their values before their value is needed by the instruction.

23.6 Forward References

In a forward reference the label appears as the argument to an instruction before its value is known. Therefore two passes of the assembler are required; one to assign the correct value to the label, and the second to use that value to generate the correct instruction codes.

On the first pass through the assembler branches containing forward references will give the warning message:

OUT OF RANGE:

indicating that a second pass is needed. The second byte of the branch will be set to zero.

23.7 Two-Pass Assembly

A two-pass assembly can be achieved simply by typing RUN twice before executing the machine code program. Alternatively it is possible to make the two-pass assembly occur automatically by incorporating the statements to be assembled within a FOR...NEXT loop. The following program assembles instructions to perform a two-byte increment:

```
10 REM Two-Pass Assembly
20 DIM M(3),JJ(2)
30 FOR N=1 TO 2
40 PRINT '"PASS "N '
50 DIM P(-1)
55[
60:JJ0 INC M
```

172

```
    70 BNE JJ1
    80 INC M+1
    90:JJ1 RTS
   100]
   110 NEXT N
   120 INPUT L
   130 !M=L
   140 LINK JJ0
   150 P. &!M
   160 END
```

Note that the statement DIM P(-1) is enclosed within the loop so that
P is reset to the correct value at the start of each pass.
        The listing produced by this program is as follows; note that the
first pass is unable to resolve the reference to JJ1 in the
instruction of line 70:

```
PASS          1
    55 29DE
    60 29DE EE CE 29 :JJ0 INC M
OUT OF RANGE:
    70 29E1 D0 00      BNE JJ1
    80 29E3 EE CF 29   INC M+1
    90 29E6 60         :JJ1 RTS

PASS          2
    55 29DE
    60 29DE EE CE 29 :JJ0 INC M
    70 29E1 D0 03      BNE JJ1
    80 29E3 EE CF 29   INC M+1
    90 29E6 60         :JJ1 RTS
```

23.8 Suppression of Assembly Listing

The assembly listing may be suppressed by disabling the output stream
with a NAK character, and enabling it again with an ACK at the end of
the assembly. The codes for NAK and ACK are 21 and 6 respectively. The
following program assembles instructions to print an "X" using a call
to the operating-system write-character routine OSWRCH at #FFF4:

```
    10 REM Turn off Assembly Listing
    20 DIM P(-1)
    30 PRINT $21; REM TURN OFF
    40[LDA @#58; JSR #FFF4; RTS;]
    50 PRINT $6 ; REM TURN ON
    60 LINK TOP
    70 END
```

23.9 Executing Programs

The LINK statement should be used to transfer control from a BASIC
program to a machine-code program. The operation of the LINK statement
is as follows:

1. The low-order bytes of the BASIC variables A, X, and Y are
transferred to the A, X, and Y registers respectively.

2. Control is transferred to the address given after the LINK
statement.

        The argument to the LINK statement will normally either be TOP,
when no arrays have been declared in the space after the program, or a

173

label corresponding to the entry-point in the assembler program (which need not be the first instruction in the program). For examples see the example programs in this chapter, and in Chapter 17.

23.10 Breakpoints

During debugging of a machine-code program it may be convenient to discover whether sections of the program are being executed. A convenient way to do this is to insert breakpoints in the program. The BRK instruction (op-code #00) is used as a breakpoint, and execution of this instruction will return control to the system, with the message:

ERROR XX LINE LL

where XX is two greater than the lower byte of the program counter, in decimal, where the BRK occurred, and the line number is the last BASIC line executed before the BRK occurred. Any number of BRK instructions may be inserted, and the value of the program counter in the ERROR message will indicate which one caused the break.

To provide more information on each BRK, such as the contents of all the processor's registers, the break vector can be altered to indirect control to a user routine, as shown in the following section.

23.10.1 Breakpoint Routine

The BRK instruction can be used to show which parts of a machine-code routine are being executed. By adding a small assembler program it is possible to keep a record of the register contents when the BRK occurred, and, if required, print these out.

The memory locations #202 and #203 contain the address to which control is transferred on a BRK instruction. This address can be redefined to point to a routine which will save the register contents in a vector K. The registers are saved as follows:

PCL	PCH	A	X	Y	S	P
K:0	1	2	3	4	5	6

After the registers have been saved in the vector K, the routine jumps to the standard BRK handler, the address previously in locations #202 and #203:

```
 10 REM Print Registers on BRK
 30 DIM K(6),AA(1),A(8),P(-1)
 35 B=?#202+256*?#203
 40 ?16=A;?17=A&#FFFF/256;$A="GOTO150"
 45[
 50:AA0 STA K+2; STX K+3
 60 PLA; STA K+6; PLA; STA K
 80 PLA; STA K+1
 90 STY K+4; TSX; STX K+5
100 JMP B
110]
120 REM INSTALL BRK ROUTINE
130 ?#202=AA0; ?#203=AA0&#FFFF/256
135 GOTO 200
140 REM PRINT REGISTERS
150 @=5
160 PRINT"  PC   A  X  Y  S  P"'
170 PRINT&!K&#FFFF-2;FORN=2TO6
```

174

```
      175 @=3
      180 PRINT&K?N;N.
      190 PRINT'; END
      200 REM DEMONSTRATE USE
      210[
      220:AA1 LDA @#12; LDX @#34
      230 LDY @#56; BRK
      240]
      250 REM EXECUTE TEST PROGRAM
      260 LINK AA1
```

Description of Program:
30	Declare vectors and array
35	Set B to BRK handler address
40	Point error line handler to "GOTO 150"
50-100	Assemble code to save registers in vector K
130	Point BRK handler to register-save routine.
150-190	Print out vector K, with heading.
220-240	Assemble test program to give a BRK
260	Execute test program.

Variables:
$A - String to contain BASIC line.
AA(0..1) - Labels for assembler routines.
AA0 - Entry point to routine to save registers in vector K.
AA1 - Entry point to test program.
B - Address of BRK routine.
K?0..6 - Vector to hold registers on BRK.

If this program is compiled, the following will be printed out after the assembler listing:

```
 PC   A  X  Y  S  P
2B60 12 34 56 FD 35
```

23.11 Conditional Assembly

The simplest facility is conditional assembly; the assembler source text can contain tests, and assemble different statements depending on the outcome of these tests. This is especially useful where slightly different versions of a program are needed for many different purposes. Rather than creating a different source file for each different version, a single variable can determine the changes using conditional assembly. For example, two printers are driven from a parallel port. They differ as follows:

1. The first printer needs a 12 microsecond strobe, and true data.

2. The second printer needs an 8 microsecond strobe and inverted data.

The variable V is used to denote the version number (1 or 2). H contains the address of the 8-bit output port, and the top bit of location H+1 is the strobe bit; D is the address of the data to be output.

```
   10 DIM P(-1)
   20 H=#B800; D=#80
  300[ LDA D;]
  310 IF V=2 [ EOR #FF invert;]
  320[ STA H to port
  330 LDA @#80
  340 STA H+1
  360 NOP strobe delay;]
```

```
     370 IF V=1 [ NOP; NOP extra delay;]
     380[ LDA @0
     390 STA H+1
     400]
     410 END
```

If this segment of the program is first executed with V=1 the
assembled code is as required for printer 1:

```
>V=1;RUN
   300 29BB A5 80        LDA D
   320 29BD 8D 00 B8     STA H to port
   330 29C0 A9 80        LDA @#80
   340 29C2 8D 01 B8     STA H+1
   360 29C5 EA           NOP strobe delay
   370 29C6 EA           NOP
   370 29C7 EA           NOP extra delay
   380 29C8 A9 00        LDA @0
   390 29CA 8D 01 B8     STA H+1
```

Extra NOP instructions have been inserted to give the required strobe
delay. If now the program is executed with V=2 the code generated is
suitable for printer 2:

```
>V=2;RUN
   300 29BB A5 80        LDA D
   310 29BD 45 FF        EOR #FF invert
   320 29BF 8D 00 B8     STA H to port
   330 29C2 A9 80        LDA @#80
   340 29C4 8D 01 B8     STA H+1
   360 29C7 EA           NOP strobe delay
   380 29C8 A9 00        LDA @0
   390 29CA 8D 01 B8     STA H+1
```

An instruction to invert the data has been added before writing it to
the port.
    Conditional assembly is also useful for the insertion of extra
instructions to print out intermediate values during debugging; these
statements will be removed when the program is finally assembled. To
do this a logical variable, D in the following example, is given the
value 1 (true) during debugging and the value 0 (false) otherwise. If
D=1 a routine to print the value of the accumulator in hex is
assembled, and calls to this routine are inserted at two relevant
points in the test program:

```
    10 REM Print Hex Digits
    20 DIM GG(3),P(-1)
    30 IF D=0 GOTO m
    50[
    55 \ print hex digit
    60:GG1 AND @#F
    70 CMP @#A; BCC P+4
    80 ADC @6; ADC @#30
    90 JMP #FFF4
    95 \ print A in hex
   100:GG2 PHA; PHA; LSRA; LSRA
   110 LSRA; LSRA; JSR GG1
   120 PLA; JSR GG1; PLA; RTS
   130]
   140mREM main program
```

176

```
150[
170:GG0 CLC; ADC @#40;]
190 IF D [ JSR GG2;]
200[
210 BEQ GG3; SBC @#10;]
220 IF D [ JSR GG2;]
230[
240:GG3 RTS;]
250 END
```

For debugging purposes this program is assembled by typing:

```
>D=1
>RUN
>RUN
```

The program can then be executed for various values of A by typing:

```
A=#12; LINK GG0
```

The final version of the program is assembled, without the debugging aids, by typing:

```
>D=0
>RUN
>RUN
```

23.12 Macros

Macros permit a name to be associated with a number of assembler instructions. This name can then be used as an abbreviation for those instructions; whenever the macro is called, the effect is as if the corresponding lines of assembler had been inserted at that point.

In their simplest form macros just save typing. For example, the sequence:

```
LSR A; LSR A; LSR A; LSR A
```

occurs frequently in assembler programs (to shift the upper nibble of the accumulator into the lower nibble), but it is not worth making the instructions into a subroutine. A macro, with the name s in the following example, can be set up as follows:

```
1000s[LSR A; LSR A; LSR A; LSR A;]
1010 RETURN
```

Then the above four instructions can be replaced by the following call to the macro s:

```
GOSUB s
```

23.12.1 Macro Parameters

The great power of macros lies in the ability to pass parameters to them so that the assembler lines they generate will be determined by the values of the parameters.

The simplest type of parameter would simply be an address; for example, the macro r below will rotate right any location, zero page or absolute, whose address is passed over in L:

```
2000r[ROR L: ROR L; ROR L; ROR L:]
2010 RETURN
```

A typical call in a program might be:

```
L=#80; GOSUB r
```

177

The following program illustrates the use of two macros. Macro i increments a 16-bit number in locations J and J+1. Macro c performs an unsigned compare between two 16-bit numbers in J,J+1 and K,K+1. The program uses these two macros to move a block of memory from one starting address to a lower starting address.

```
  10 REM Block Move
' 20 DIM LL(2),P(100)
  30 F=#80; L=#82; T=#84
  40[:LL0 LDY @0
  45:LL1 LDA (F),Y; STA (T),Y;]
  50 J=T; GOSUB i
  60 J=F; GOSUB i
  70 K=L; GOSUB c
  80[ BNE LL1; RTS;]
  90
 100 REM TRY IT OUT
 110 REM F=first address
 112 REM L=last address
 114 REM T=address moved to (T<F)
 120 !F=#500;!L=#800;!T=#400
 130 LINK LL0
 140 END
8000
8100 REM MACRO - INC J,J+1
8105i[INC J; BNE P+4+(J>254)&1
8110 INC J+1;]
8120 RETURN
8130
8140 REM MACRO - CMP J,J+1 WITH K,K+1
8145c[LDA J+1; CMP K+1
8150 BNE P+6+(J>255)&1+(K>255)&1
8160 LDA J; CMP K;]
8170 RETURN
```

Note that both macros are designed to work whether J and K are absolute addresses or zero-page addresses; to avoid the need for labels in these macros they test for the size of the address, and generate the correct argument for the branch instruction. The expression:

(J>255)&1

has the value 1 if J is greater than 255, and the value 0 if J is 255 or less.

23.12.2 In-Line Assembly

In critical sections of programs, where speed is important, it may be necessary to code repetitative calculations by actually repeating the instructions as many times as necessary, rather than using a loop, thereby avoiding the overhead associated with the loop calculations. The following macro compiles a routine to multiply a 7-bit number in the A register by a fractional constant between 0/256 and 255/256. The numerator of the constant is passed to the macro in C:

```
  1 REM Fractional Multiplication
  5 J=#80; DIM P(-1)
 10 C=#AA
 20 GOSUBm
 30 [STA J;RTS;]
```

```
   40 INPUT A
   50 LINK TOP
   60 P.&A,&?J
   70 END
 2000mREM macro - multiply by constant
 2010 REM A = A * C/256
 2020 REM uses J
 2030 B=#80
 2040 [STA J;LDA @0;]
 2050 DO [LSR J;]
 2060 IF C&B<>0 [CLC;ADC J;]
 2070 C=(C*2)&#FF; UNTIL C=0
 2080 RETURN
```

The macro is tested with C=#AA. In this case the code produced will be:

```
2040 2A42 85 80     STA J
2040 2A44 A9 00     LDA @0
2050 2A46 46 80     LSR J
2060 2A48 18        CLC
2060 2A49 65 80     ADC J
2070 2A4B 46 80     LSR J
2070 2A4D 46 80     LSR J
2060 2A4F 18        CLC
2060 2A50 65 80     ADC J
2070 2A52 46 80     LSR J
2070 2A54 46 80     LSR J
2060 2A56 18        CLC
2060 2A57 65 80     ADC J
2070 2A59 46 80     LSR J
2070 2A5B 46 80     LSR J
2060 2A5D 18        CLC
2060 2A5E 65 80     ADC J
2080 2A60 85 80     STA J
2080 2A62 60        RTS
```

24 Assembler Mnemonics

The following section lists all the instruction mnemonics in alphabetical order. Each instruction is accompanied by a description of the instruction, a symbolic representation of the action performed by the instruction, a diagram showing the status-register flags affected by the instruction, and a list of the permitted addressing modes for the instruction.

The following symbols are used in this section:

Symbol:	Definition:
+	Addition
–	Subtraction
&	Logical AND
Ṉ	Logical OR
:	Logical Exclusive-OR
!	Push onto hardware stack
^	Pull from hardware stack
=	Assignment
M	Memory location
(PC+1)	Contents of location after op-code
@	Immediate addressing mode
~	No change to flag
%	Change to flag
1	Set
0	Cleared
A	Accumulator
X	X Index Register
Y	Y Index Register
PC	Program Counter
PCL	Low byte of Program Counter
PCH	High byte of Program Counter

ADC   Add memory to accumulator with carry   **ADC**

A,C=A+M+C

N Z C I D V
% % % ~ ~ %

Addressing	Assembler Format		Bytes	Cycles
Immediate	ADC	@ Oper	2	2
Zero Page	ADC	Oper	2	3
Zero Page,X	ADC	Oper,X	2	4
Absolute	ADC	Oper	3	4
Absolute,X	ADC	Oper,X	3	4*
Absolute,Y	ADC	Oper,Y	3	4*
(Indirect,X)	ADC	(Oper,X)	2	6
(Indirect),Y	ADC	(Oper),Y	2	5*

* Add 1 if page boundary crossed.
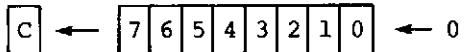
AND   AND memory with accumulator   **AND**

A=A&M

```
                                                      N Z C I D V
                                                      % % - - - -
```

Addressing	Assembler Format		Bytes	Cycles
Immediate	AND	@ Oper	2	2
Zero Page	AND	Oper	2	3
Zero Page,X	AND	Oper,X	2	4
Absolute	AND	Oper	3	4
Absolute,X	AND	Oper,X	3	4*
Absolute,Y	AND	Oper,Y	3	4*
(Indirect,X)	AND	(Oper,X)	2	6
(Indirect),Y	AND	(Oper),Y	2	5*

* Add 1 if page boundary crossed.

ASL   Arithmetic shift left one bit (memory or accumulator)   **ASL**

```
                                                      N Z C I D V
 +-+     +-+-+-+-+-+-+-+-+                             % % % - - -
 |C| <-- |7|6|5|4|3|2|1|0| <-- 0
 +-+     +-+-+-+-+-+-+-+-+
```

Addressing	Assembler Format		Bytes	Cycles
Accumulator	ASL	A	1	2
Zero Page	ASL	Oper	2	5
Zero page,X	ASL	Oper,X	2	6
Absolute	ASL	Oper	3	6
Absolute,X	ASL	Oper,X	3	7

BCC   Branch on carry clear   **BCC**

Branch if C=0

```
                                                      N Z C I D V
                                                      - - - - - -
```

Addressing	Assembler Format		Bytes	Cycles
Relative	BCC	Oper	2	2*

* Add 1 if branch occurs to same page
  Add 2 if branch occurs to different page

BCS   Branch on carry set   **BCS**

Branch if C=1

```
                                                      N Z C I D V
                                                      - - - - - -
```

Addressing	Assembler Format		Bytes	Cycles
Relative	BCS	Oper	2	2*

* Add 1 if branch occurs to same page
  Add 2 if branch occurs to different page

BEQ   Branch on result zero   **BEQ**

Branch if Z=1

```
                                                      N Z C I D V
                                                      - - - - - -
```

Addressing	Assembler Format		Bytes	Cycles
Relative	BEQ	Oper	2	2*

* Add 1 if branch occurs to same page
  Add 2 if branch occurs to different page

BIT   Test bits in memory with accumulator   **BIT**
$A\&M$, $N=M_7$, $V=M_6$

N	Z	C	I	D	V
M_7	%	~	~	~	M_6

Bit 6 and 7 are transferred to the status register. If the result of
A&M is zero then Z=1, otherwise Z=0.

Addressing	Assembler Format		Bytes	Cycles
Zero Page	BIT	Oper	2	3
Absolute	BIT	Oper	3	4

BMI   **Branch on result minus**   **BMI**
Branch if N=1

N	Z	C	I	D	V
~	~	~	~	~	~

Addressing	Assembler Format		Bytes	Cycles
Relative	BMI	Oper	2	2*

* Add 1 if branch occurs to same page
  Add 2 if branch occurs to different page

BNE   **Branch on result not zero**   **BNE**
Branch if Z=0

N	Z	C	I	D	V
~	~	~	~	~	~

Addressing	Assembler Format		Bytes	Cycles
Relative	BNE	Oper	2	2*

* Add 1 if branch occurs to same page
  Add 2 if branch occurs to different page

BPL   **Branch on result plus**   **BPL**
Branch if N=0

N	Z	C	I	D	V
~	~	~	~	~	~

Addressing	Assembler Format		Bytes	Cycles
Relative	BPL	Oper	2	2*

* Add 1 if branch occurs to same page
  Add 2 if branch occurs to different page

BRK   **Force break**   **BRK**
Forced interrupt; PC+2 ! P !

N	Z	C	I	D	V
~	~	~	1	~	~

Addressing	Assembler Format	Bytes	Cycles
Implied	BRK	1	7

A BRK command cannot be masked by setting I.

BVC   **Branch on overflow clear**   **BVC**
Branch if V=0

N	Z	C	I	D	V
~	~	~	~	~	~

Addressing	Assembler Format		Bytes	Cycles
Relative	BVC	Oper	2	2*

* Add 1 if branch occurs to same page
  Add 2 if branch occurs to different page

183

BVS   **Branch on overflow set**   **BVS**
Branch if V=1

	N	Z	C	I	D	V
	~	~	~	~	~	~

Addressing	Assembler Format	Bytes	Cycles
Relative	BVS Oper	2	2*

* Add 1 if branch occurs to same page
  Add 2 if branch occurs to different page

CLC   **Clear carry flag**   **CLC**
C=0

	N	Z	C	I	D	V
	~	~	0	~	~	~

Addressing	Assembler Format	Bytes	Cycles
Implied	CLC	1	2

CLD   **Clear decimal mode**   **CLD**
D=0

	N	Z	C	I	D	V
	~	~	~	~	0	~

Addressing	Assembler Format	Bytes	Cycles
Implied	CLD	1	2

CLI   **Clear interrupt disable bit**   **CLI**
I=0

	N	Z	C	I	D	V
	~	~	~	0	~	~

Addressing	Assembler Format	Bytes	Cycles
Implied	CLI	1	2

CLV   **Clear overflow flag**   **CLV**
V=0

	N	Z	C	I	D	V
	~	~	~	~	~	0

Addressing	Assembler Format	Bytes	Cycles
Implied	CLV	1	2

CMP   **Compare memory and accumulator**   **CMP**
A-M

	N	Z	C	I	D	V
	§	§	§	~	~	~

Addressing	Assembler Format	Bytes	Cycles
Immediate	CMP @ Oper	2	2
Zero Page	CMP Oper	2	3
Zero Page,X	CMP Oper,X	2	4
Absolute	CMP Oper	3	4
Absolute,X	CMP Oper,X	3	4*
Absolute,Y	CMP Oper,Y	3	4*
(Indirect,X)	CMP (Oper,X)	2	6
(Indirect),Y	CMP (Oper),Y	2	5*

* Add 1 if page boundary crossed.

CPX   **Compare memory and index register X**   **CPX**
X-M

	N	Z	C	I	D	V
	§	§	§	-	~	~

Addressing	Assembler Format	Bytes	Cycles
Immediate	CPX @ Oper	2	2
Zero Page	CPX Oper	2	3
Absolute	CPX Oper	3	4

CPY Compare memory and index register Y   **CPY**

Y-M

| | | | | N Z C I D V |
| | | | | | % % % ~ ~ ~ |

Addressing	Assembler Format		Bytes	Cycles
Immediate	CPY	@ Oper	2	2
Zero Page	CPY	Oper	2	3
Absolute	CPY	Oper	3	4

DEC Decrement memory by one   **DEC**

M=M-1

| | | | | N Z C I D V |
| | | | | | % % ~ ~ ~ ~ |

Addressing	Assembler Format		Bytes	Cycles
Zero Page	DEC	Oper	2	5
Zero Page,X	DEC	Oper,X	2	6
Absolute	DEC	Oper	3	6
Absolute,X	DEC	Oper,X	3	7

DEX Decrement index register X by one   **DEX**

X=X-1

| | | | N Z C I D V |
| | | | | % % ~ ~ ~ ~ |

Addressing	Assembler Format	Bytes	Cycles
Implied	DEX	1	2

DEY Decrement index register X by one   **DEY**

Y=Y-1

| | | | N Z C I D V |
| | | | | % % ~ ~ ~ ~ |

Addressing	Assembler Format	Bytes	Cycles
Implied	DEY	1	2

EOR Exclusive-OR memory with accumulator   **EOR**

A=A:M

| | | | | N Z C I D V |
| | | | | | % % ~ ~ ~ ~ |

Addressing	Assembler Format		Bytes	Cycles
Immediate	EOR	@ Oper	2	2
Zero Page	EOR	Oper	2	3
Zero Page,X	EOR	Oper,X	2	4
Absolute	EOR	Oper	3	4
Absolute,X	EOR	Oper,X	3	4*
Absolute,Y	EOR	Oper,Y	3	4*
(Indirect,X)	EOR	(Oper,X)	2	6
(Indirect),Y	EOR	(Oper),Y	2	5*

* Add 1 if page boundary crossed.

INC Increment memory by one   **INC**

M=M+1

| | | | | N Z C I D V |
| | | | | | % % ~ ~ ~ ~ |

Addressing	Assembler Format		Bytes	Cycles
Zero Page	INC	Oper	2	5
Zero Page,X	INC	Oper,X	2	6
Absolute	INC	Oper	3	6
Absolute,X	INC	Oper,X	3	7

INX   **Increment index register X by one**   **INX**

X=X+1

```
                                              N Z C I D V
                                              % % ~ ~ ~ ~
```

Addressing	Assembler Format	Bytes	Cycles
Implied	INX	1	2

INY   **Increment index register Y by one**   **INY**

Y=Y+1

```
                                              N Z C I D V
                                              % % ~ ~ ~ ~
```

Addressing	Assembler Format	Bytes	Cycles
Implied	INY	1	2

JMP   **Jump to new location**   **JMP**

PCL=(PC+1)

PCH=(PC+2)

```
                                              N Z C I D V
                                              ~ ~ ~ ~ ~ ~
```

Addressing	Assembler Format	Bytes	Cycles
Absolute	JMP Oper	3	3
Indirect	JMP (Oper)	3	5

JSR   **Jump to subroutine saving return address**   **JSR**

PC+2 !, PCL=(PC+1)

       PCH=(PC+2)

```
                                              N Z C I D V
                                              ~ ~ ~ ~ ~ ~
```

Addressing	Assembler Format	Bytes	Cycles
Absolute	JSR Oper	3	6

LDA   **Load accumulator with memory**   **LDA**

A=M

```
                                              N Z C I D V
                                              % % ~ ~ ~ ~
```

Addressing	Assembler Format	Bytes	Cycles
Immediate	LDA @ Oper	2	2
Zero Page	LDA Oper	2	3
Zero Page,X	LDA Oper,X	2	4
Absolute	LDA Oper	3	4
Absolute,X	LDA Oper,X	3	4*
Absolute,Y	LDA Oper,Y	3	4*
(Indirect,X)	LDA (Oper,X)	2	6
(Indirect),Y	LDA (Oper),Y	2	5*

* Add 1 if page boundary crossed.

LDX   **Load index register X with memory**   **LDX**

X=M

```
                                              N Z C I D V
                                              % % ~ ~ ~ ~
```

Addressing	Assembler Format	Bytes	Cycles
Immediate	LDX @ Oper	2	2
Zero Page	LDX Oper	2	3
Zero Page,Y	LDX Oper,Y	2	4
Absolute	LDX Oper	3	4
Absolute,Y	LDX Oper,Y	3	4*

* Add 1 when page boundary crossed

LDY   **Load index register Y with memory**   **LDY**
Y=M   N Z C I D V
                                                                      % % ~ ~ ~ ~
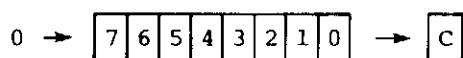
Addressing	Assembler Format	Bytes	Cycles
Immediate	LDY @ Oper	2	2
Zero Page	LDY Oper	2	3
Zero Page,X	LDY Oper,X	2	4
Absolute	LDY Oper	3	4
Absolute,X	LDY Oper,X	3	4*

* Add 1 when page boundary crossed

LSR   **Logical shift right one bit (memory or accumulator)**   **LSR**
                                                                      N Z C I D V
                                                                      % % % ~ ~ ~

$$0 \rightarrow \boxed{7\,6\,5\,4\,3\,2\,1\,0} \rightarrow \boxed{C}$$

Addressing	Assembler Format	Bytes	Cycles
Accumulator	LSR A	1	2
Zero Page	LSR Oper	2	5
Zero page,X	LSR Oper,X	2	6
Absolute	LSR Oper	3	6
Absolute,X	LSR Oper,X	3	7

NOP   **No operation**   **NOP**
                                                                      N Z C I D V
                                                                      ~ ~ ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	NOP	1	2

ORA   **OR memory with accumulator**   **ORA**
A=A∨M   N Z C I D V
                                                                      % % ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Immediate	ORA @ Oper	2	2
Zero Page	ORA Oper	2	3
Zero Page,X	ORA Oper,X	2	4
Absolute	ORA Oper	3	4
Absolute,X	ORA Oper,X	3	4*
Absolute,Y	ORA Oper,Y	3	4*
(Indirect,X)	ORA (Oper,X)	2	6
(Indirect),Y	ORA (Oper),Y	2	5*

* Add 1 if page boundary crossed.

PHA   **Push accumulator on stack**   **PHA**
A !   N Z C I D V
                                                                      ~ ~ ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	PHA	1	3

PHP   **Push processor status on stack**   **PHP**
P !   N Z C I D V
                                                                      ~ ~ ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	PHP	1	3

PLA Pull accumulator from stack   **PLA**

A ^

N	Z	C	I	D	V
%	%	~	~	~	~

Addressing	Assembler Format	Bytes	Cycles
Implied	PLA	1	4

PLP Pull processor status from stack   **PLP**
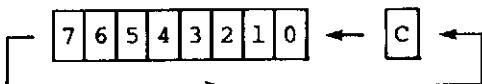
P ^

N	Z	C	I	D	V
from	stack				

Addressing	Assembler Format	Bytes	Cycles
Implied	PLP	1	4

ROL Rotate left one bit (memory or accumulator)   **ROL**

N	Z	C	I	D	V
%	%	%	~	~	~
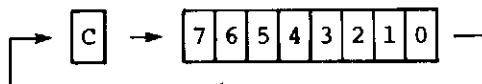
Addressing	Assembler Format		Bytes	Cycles
Accumulator	ROL	A	1	2
Zero Page	ROL	Oper	2	5
Zero page,X	ROL	Oper,X	2	6
Absolute	ROL	Oper	3	6
Absolute,X	ROL	Oper,X	3	7

ROR Rotate right one bit (memory or accumulator)   **ROR**

N	Z	C	I	D	V
%	%	%	~	~	~

Addressing	Assembler Format		Bytes	Cycles
Accumulator	ROR	A	1	2
Zero Page	ROR	Oper	2	5
Zero page,X	ROR	Oper,X	2	6
Absolute	ROR	Oper	3	6
Absolute,X	ROR	Oper,X	3	7

RTI Return from interrupt   **RTI**

P^ PC^

N	Z	C	I	D	V
from	stack				

Addressing	Assembler Format	Bytes	Cycles
Implied	RTI	1	6

RTS Return from subroutine   **RTS**

PC^

N	Z	C	I	D	V
~	~	~	~	~	~

Addressing	Assembler Format	Bytes	Cycles
Implied	RTS	1	6

SBC   Subtract memory from accumulator with carry   **SBC**
A,C=A-M-(C-1)   N Z C I D V
                                                                     % % % ~ ~ %

Addressing	Assembler Format	Bytes	Cycles
Immediate	SBC @ Oper	2	2
Zero Page	SBC Oper	2	3
Zero Page,X	SBC Oper,X	2	4
Absolute	SBC Oper	3	4
Absolute,X	SBC Oper,X	3	4*
Absolute,Y	SBC Oper,Y	3	4*
(Indirect,X)	SBC (Oper,X)	2	6
(Indirect),Y	SBC (Oper),Y	2	5*

* Add 1 if page boundary crossed.

SEC   Set carry flag   **SEC**
C=1   N Z C I D V
                                                                     ~ ~ 1 ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	SEC	1	2

SED   Set decimal mode   **SED**
D=1   N Z C I D V
                                                                     ~ ~ ~ ~ 1 ~

Addressing	Assembler Format	Bytes	Cycles
Implied	SED	1	2

SEI   Set interrupt disable bit   **SEI**
I=1   N Z C I D V
                                                                     ~ ~ ~ 1 ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	SEI	1	2

STA   Store accumulator in memory   **STA**
M=A   N Z C I D V
                                                                     ~ ~ ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Zero Page	STA Oper	2	3
Zero Page,X	STA Oper,X	2	4
Absolute	STA Oper	3	4
Absolute,X	STA Oper,X	3	5
Absolute,Y	STA Oper,Y	3	5
(Indirect,X)	STA (Oper,X)	2	6
(Indirect),Y	STA (Oper),Y	2	6

STX   Store index register X in memory   **STX**
M=X   N Z C I D V
                                                                     ~ ~ ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Zero Page	STX Oper	2	3
Zero Page,Y	STX Oper,Y	2	4
Absolute	STX Oper	3	4

STY   Store index register Y in memory   **STY**
M=Y

N Z C I D V
~ ~ ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Zero Page	STY Oper	2	3
Zero Page,X	STY Oper,X	2	4
Absolute	STY Oper	3	4

TAX   Transfer accumulator to index register X   **TAX**
X=A

N Z C I D V
§ § ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	TAX	1	2

TAY   Transfer accumulator to index register Y   **TAY**
Y=A

N Z C I D V
§ § ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	TAY	1	2

TSX   Transfer stack pointer to index register X   **TSX**
X=S

N Z C I D V
§ § ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	TSX	1	2

TXA   Transfer index register X to accumulator   **TXA**
X=A
A=X

N Z C I D V
§ § ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	TXA	1	2

TXS   Transfer index register X to stack pointer   **TXS**
X=S

N Z C I D V
§ § ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	TXS	1	2

TYA   Transfer index register Y to accumulator   **TYA**
Y=A

N Z C I D V
§ § ~ ~ ~ ~

Addressing	Assembler Format	Bytes	Cycles
Implied	TYA	1	2

25 Operating System Routines and Addresses

25.1 Input/Output Routines

The ATOM operating system contains several routines which can be called by user programs to provide input and output facilities. The routines are defined so that they are compatible with the other Acorn operating systems; in particular, if the ATOM is expanded to include a Disk Operating System the same routines will automatically function with the disk.

OSCLI   Command line interpreter

This subroutine interprets a string of characters at address #0100 and terminated by carriage return as an operating system command. Detected errors are met with a BRK. All processor registers are used, and the decimal-mode flag is set to binary on exit.

OSWRCH   Write character

This subroutine sends the byte in the accumulator to the output channel. Control characters are normally recognised as detailed in Section 18.1.3. All registers are preserved.

OSCRLF   Carriage return - line feed

This subroutine generates a line feed followed by a carriage return using OSWRCH. On exit A will contain #0D, N and Z will be 0, and all other registers are preserved.

OSECHO   Read character with echo

This subroutine reads a byte using OSRDCH and then writes it out using OSWRCH. The routine converts carriage returns to a line feed followed by a carriage return. On exit A will contain the byte read, N, Z, and C are undefined, and all other registers are preserved.

OSRDCH   Read character

This subroutine reads a byte from the input channel and returns it in A. The state of N, Z, and C is undefined; all other registers are preserved.

OSLOAD   Load file

This subroutine loads a complete file into a specified area of memory. On entry X must point to the following data in zero page:

X+0 address of string of characters, terminated by #0D, which is the file name.
X+2 Address in memory of the first byte of the destination.
X+4 Flag byte: if bit 7 = 0 use the file's start address.

All processor registers are used. A break will occur if the file cannot be found. In interrupt or DMA driven systems a wait until completion should be performed if the carry flag was set on entry.

OSSAVE   Save file

This subroutine saves all of an area of memory to a specified file. On entry X must point to the following data in zero page:

X+0 Address of string of characters, terminated by #0D, which is the file name.
X+2 Address for data to be reloaded to.
X+4 Execution address if data is to be executed
X+6 Start address of data in memory
X+8 End address + 1 of data in memory

The data is copied by the operating system without being altered. All registers are used. In interrupt or DMA driven operating systems a wait until completion should be performed if the carry flag was set on entry. A break will occur if no storage space large enough can be found.

OSBPUT   Put byte

This subroutine outputs the byte in the accumulator to a sequential write file. Registers X and Y are saved. In the ATOM operating system interrupts are disabled during OSBPUT but interrupt status is restored on exit. In the Disk Operating System the file's sequential file pointer will be incremented after the byte has been saved.

OSBGET   Get byte

The subroutine returns, in A, the next byte from a sequential read file. Registers X and Y are retained. In the ATOM operating system interrupts are disabled during OSBGET but interrupt status is restored on exit. In the Disk Operating System the file's sequential file pointer will be incremented after the byte has been read.

OSFIND   Find file

This subroutine returns, in A, a 'handle' for a file. The X register points to zero page locations containing the address of the first character of the file name; the file name is terminated by a #0D byte. The 'handle' is zero if the file does not exist; otherwise it is a byte uniquely specifying the file. If the file is to be used for sequential input the carry should be set, or if for sequential output the carry should be clear. In the ATOM operating system the file handle is set to 13, and the message "PLAY TAPE" or "RECORD TAPE" is produced. In the Disk Operating System the file's sequential pointer is set to zero.

OSSHUT   Shut file

This subroutine removes a reference to a file whose handle is in the Y register. If a handle of zero is supplied, all files are shut. In the ATOM operating system the call does nothing.

The following subroutines are not used in the cassette system, and cause an error if called:

OSRDAR   Read file's arguments
OSSTAR   Store file's arguments

25.2 Operating System Calls

The following table gives the addresses of all the operating system calls:

192

```
             Address:    Subroutine:    Instruction:
             #FFCB       OSSHUT         JMP  (SHTVEC)
             #FFCE       OSFIND         JMP  (FNDVEC)
             #FFD1       OSBPUT         JMP  (BPTVEC)
             #FFD4       OSBGET         JMP  (BGTVEC)
             #FFD7       OSSTAR         JMP  (STRVEC)
             #FFDA       OSRDAR         JMP  (RDRVEC)
             #FFDD       OSSAVE         JMP  (SAVVEC)
             #FFE0       OSLOAD         JMP  (LODVEC)
             #FFE3       OSRDCH         JMP  (RDCVEC)
             #FFE6       OSECHO         JSR  OSRDCH
             #FFE9       OSASCI         CMP  @#0D
             #FFEB                      BNE  OSWRCH
             #FFED       OSCRLF         LDA  @#0A
             #FFEF                      JSR  OSWRCH
             #FFF2                      LDA  @#0D
             #FFF4       OSWRCH         JMP  (WRCVEC)
             #FFF7       OSCLI          JMP  (COMVEC)
```

The operating system calls are all indirected via addresses held in
RAM, and these addresses may be changed to the addresses of
user-supplied routines. The addresses are initialised on reset as
follows:

```
             Address:    Subroutine:    Function:
             #0200       NMIVEC         NMI service routine
             #0202       BRKVEC         BRK service routine
             #0204       IRQVEC         IRQ service routine
             #0206       COMVEC         Command line interpreter
             #0208       WRCVEC         Write character
             #020A       RDCVEC         Read character
             #020C       LODVEC         Load file
             #020E       SAVVEC         Save file
             #0210       RDRVEC         Error
             #0212       STRVEC         Error
             #0214       BGTVEC         Get byte from tape
             #0216       BPTVEC         Put byte to tape
             #0218       FNDVEC         Print message
             #021A       SHTVEC         Dummy
```

A call to one of the routines OSRDAR or OSSTAR will cause the message:

COM?

to be output, followed by a BRK.

25.3 Interrupts
The following action is taken on interrupts:

```
NMI       PHA
          JMP (NMIVEC)

IRQ/BRQ   STA #FF
          PLA
          PHA
          AND @#10   which interrupt was it
          BNE BRK
          LDA #FF
          PHA
          JMP (IRQVEC)

BRK       LDA #FF
```

193

```
      PLP
      PHP
      JMP (BRKVEC)
```

Note that the accumulator is pushed before the jump occurs.

25.4 Block Zero RAM Locations

Hexadecimal:	Decimal:	Function:
#0	0	Error number
#1, #2	1, 2	BASIC line number.
#8 - #C	8 - 12	Random number seed
#10, #11	16, 17	Pointer to BASIC error handler
#12	18	Text-space pointer
#00 - #6F	0 - 111 *127*	BASIC zero-page workspace
#70 - #7F	112 - ~~143~~	Floating-point workspace
#80 - #AF	*128* ~~144~~ - 175	Free
#B0 - #FF	176 - 255	Cassette system workspace
#FE	254	Character not sent to printer
#100 - #13F	256 - 319	Input line buffer
#140 - #17F	320 - 383	String processing & INPUT statement buffer
#180 - #1FF	384 - 511	Stack
#200 - #21B	512 - 539	Operating system vectors
#21C - #23F	540 - 575	Free
#240 - #3FF	576 - 1023	BASIC workspace
#3FE, #3FF	1022, 1023	Address of point-plotting routine

25.5 Input/Output Port Allocations

The 8255 Programmable Peripheral Interface Adapter contains three 8-bit ports, and all but one of these lines is used by the ATOM.

Port A - #B000

Output bits:	Function:
0 - 3	Keyboard row
4 - 7	Graphics mode

Port B - #B001

Input bits:	Function:
0 - 5	Keyboard column
6	CTRL key (low when pressed)
7	SHIFT keys (low when pressed)
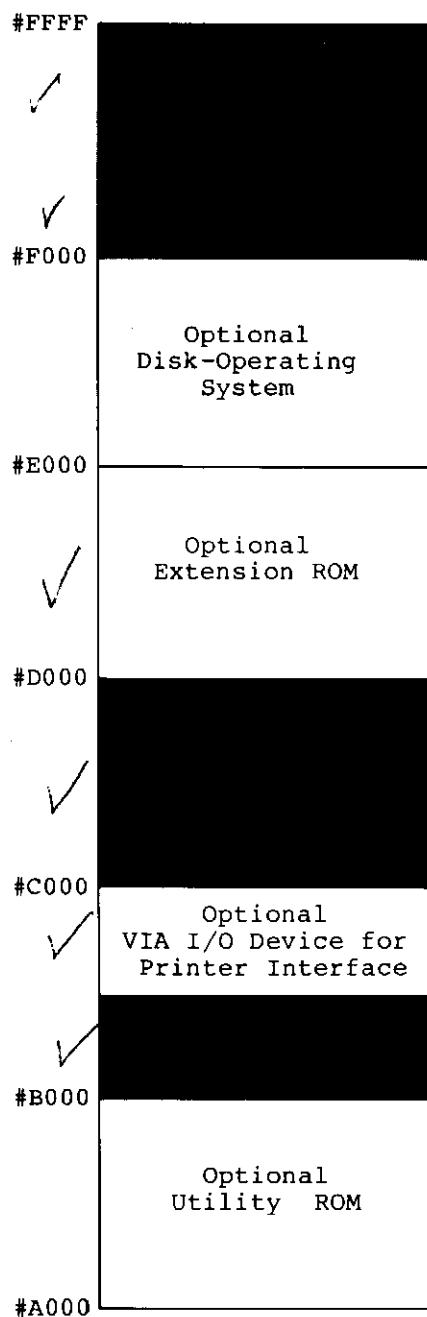
Port C - #B002
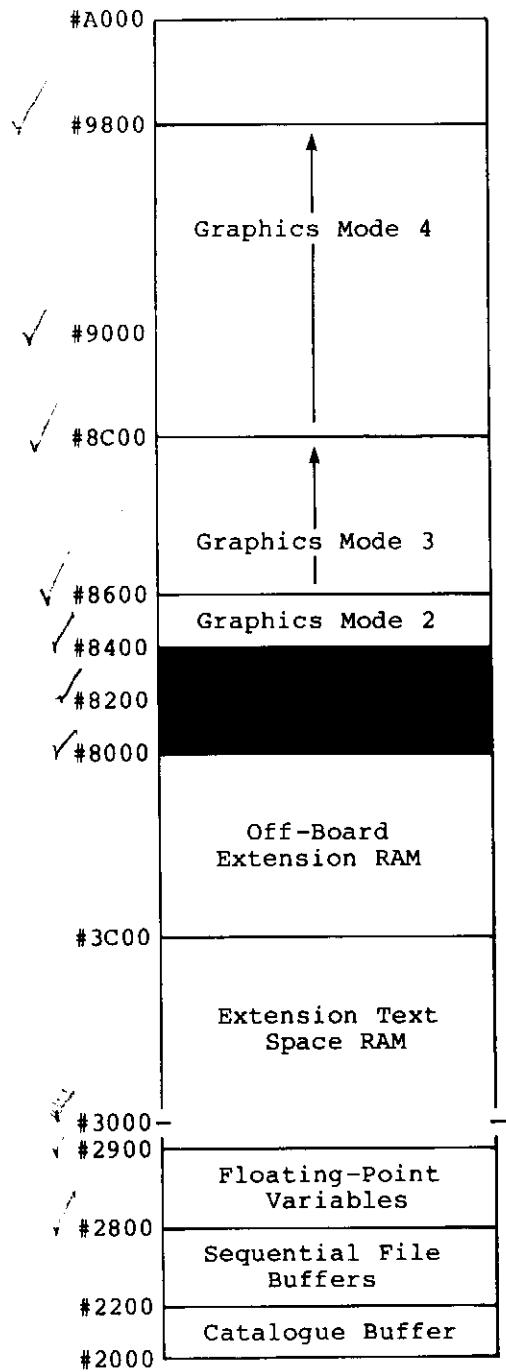
Output bits:	Function:
0	Tape output
1	Enable 2.4 kHz to cassette output
2	Loudspeaker
3	Not used

Input bits:	Function:
4	2.4 kHz input
5	Cassette input
6	REPT key (low when pressed)
7	60 Hz sync signal (low during flyback)
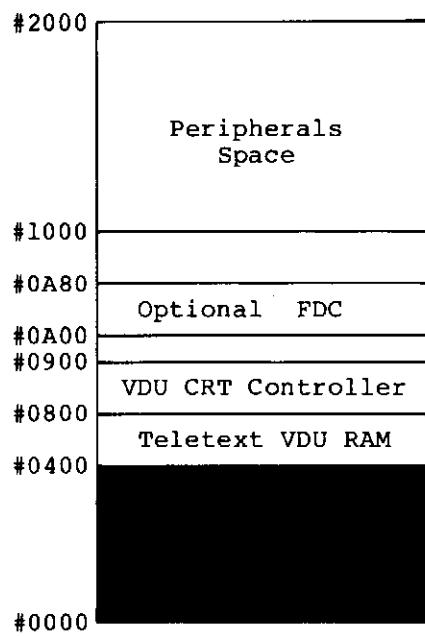
The port C output lines, bits 0 to 3, may be used for user applications when the cassette interface is not being used.

194

25.6 Memory Map

The following diagram shows how the ATOM's address space is allocated. Sections shown shaded are present in the minimal-system ATOM. The map includes the addresses of devices on the Acorn cards, which may be fitted inside the ATOM case.

```
#FFFF ┌──────────────────┐
      │██████████████████│
      │██████████████████│
      │██████████████████│
#F000 ├──────────────────┤
      │     Optional     │
      │  Disk-Operating  │
      │      System      │
#E000 ├──────────────────┤
      │                  │
      │     Optional     │
      │  Extension ROM   │
      │                  │
#D000 ├──────────────────┤
      │██████████████████│
      │██████████████████│
      │██████████████████│
#C000 ├──────────────────┤
      │     Optional     │
      │ VIA I/O Device for│
      │ Printer Interface │
      ├──────────────────┤
      │██████████████████│
#B000 ├──────────────────┤
      │     Optional     │
      │  Utility  ROM    │
      │                  │
#A000 └──────────────────┘
```

```
#A000 ┌─────────────────────────┐
      │                         │
#9800 ├─────────────────────────┤
      │              ▲          │
      │                         │
      │     Graphics Mode 4     │
      │                         │
#9000 │                         │
      │                         │
      │                         │
#8C00 ├─────────────────────────┤
      │              ▲          │
      │     Graphics Mode 3     │
      │              │          │
#8600 ├─────────────────────────┤
      │     Graphics Mode 2     │
#8400 ├─────────────────────────┤
      │█████████████████████████│
#8200 │█████████████████████████│
      │█████████████████████████│
#8000 ├─────────────────────────┤
      │                         │
      │       Off-Board         │
      │     Extension RAM       │
      │                         │
#3C00 ├─────────────────────────┤
      │                         │
      │     Extension Text      │
      │       Space RAM         │
      │                         │
#3000 ├─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─┤
#2900 ├─────────────────────────┤
      │     Floating-Point      │
      │       Variables         │
#2800 ├─────────────────────────┤
      │    Sequential File      │
      │        Buffers          │
#2200 ├─────────────────────────┤
      │    Catalogue Buffer     │
#2000 └─────────────────────────┘
```

```
#2000 ┌─────────────────────┐
       │                     │
       │     Peripherals     │
       │       Space         │
       │                     │
#1000  ├─────────────────────┤
       │                     │
#0A80  ├─────────────────────┤
       │   Optional   FDC    │
#0A00  ├─────────────────────┤
#0900  ├─────────────────────┤
       │ VDU CRT Controller  │
#0800  ├─────────────────────┤
       │  Teletext VDU RAM   │
#0400  ├─────────────────────┤
       │█████████████████████│
       │█████████████████████│
       │█████████████████████│
       │█████████████████████│
#0000  └─────────────────────┘
```

26 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.

26.1 BASIC Syntax Definition

26.1.1 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 [ 〈 <> <= >= @@ AA BB CC CH DD DO EE FF GG
HH II IF JJ KK LL MM NN OO OR PP QQ RR SS TO TT UU VV WW XX YY ZZ ABS
AND DIM END EXT FIN FOR GET LEN LET NEW OLD PTR PUT REM RND RUN TOP
BGET BPUT DRAW FOUT GOTO LINK LIST LOAD MOVE NEXT PLOT SAVE SGET SHUT
SPUT STEP THEN WAIT CLEAR 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 .}

<array name>::=@@|AA|BB|CC|DD|EE|FF|GG|HH|II|JJ|KK|LL|MM|NN|OO|PP|QQ
        |RR|SS|TT|UU|VV|WW|XX|YY|ZZ

```
<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>|
         <array name><factor>

<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>

<vector statement>::=<array name><factor>=<expression>

<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>

<old statement>::=OLD<sd>

<link statement>::=LINK<factor><sd>

<OS statement>::=*<asciic>^
```

200

```
<plot statement>::=PLOT<factor>,<factor>,<factor><sd>

<draw statement>::=DRAW<factor>,<factor><sd>

<move statement>::=MOVE<factor>,<factor><sd>

<clear statement>::=CLEAR<factor><sd>

<wait statement>::=WAIT<sd>

<go entity>::=<label>|<factor>

<goto statement>::=GOTO<go entity><sd>

<gosub statement>::=GOSUB<go entity><sd>

<end statement>::=END<sd>

<enter assembler statement>::=[

<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>

<dim section>::=<variable><factor>|<array name><factor>

<dim statement>::=DIM<dim section>{,<dim section>}^<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>
```

26.2 Assembler Syntax Definition

This uses the same syntax as Section 26.1, and refers to some of the
syntactic entities given there. Basic symbols may not be abbreviated;
spaces may be used freely to improve readability.

26.2.1 Basic Symbols

```
(  )  ,  :  @ A X Y \ ] ADC AND ASL BCC BCS BEQ BIT BMI BNE BPL BRK BVC
BVS CLC CLD CLI CLV CMP CPX CPY DEC DEX DEY EOR INC INX INY JMP JSR
LDA LDX LDY LSR NOP ORA PHA PHP PLA PLP ROL ROR RTI RTS SBC SEC SED
SEI STA STX STY TAX TAY TSX TXS TXA TXS TYA

<set label statement>::=<two chars>:<label name><assembler statement>

<comment statement>::=<two chars>\<comment field>

<back to basic>::=]

<empty statement>::=<two chars><sd>

<two chars>::=<asciic>|<asciic><asciic>|{no character at all}

<comment field>::={ascii until <sd>}

<immed>::=@<expression>

<indexX>::=<expression>,X

<indexY>::=<expression>,Y

<group1>::=<indexX>|<indexY>|(<indexX>)|(<expression>),Y|<expression>

<branch>::={BCC|BCS|BEQ|BMI|BNE|BPL|BVC|BVS}<expression>

<memory to A>::=ADC|AND|CMP|EOR|LDA|ORA|SBC{<group1>|<immed>}

<A to memory>::=STA<group1>

<single byte A>::={ASL|LSR|ROL|ROR}A

<single byte>::=<single byte A>|BRK|CLC|CLD|CLI|CLV|DEX|DEY|INX|INY
               |NOP|PHA|PHP|PLA|PLP|RTI|RTS|TAX|TAY|TSX|TXA|TXS|TYA

<read modify write>::={ASL|DEC|INC|LSR|ROL|ROR}{<indexX>|<expression>}

<bit>::=BIT<expression>

<cp>::={CPX|CPY}{<immed>|<expression>}

<jmp>::=JMP{<expression>|(<expression>)}

<jsr>::=JSR<expression>

<ldx>::=LDX{<immmed>|<indexY>|<expression>}
```

```
<ldy>::=LDY{<immmed>|<indexX>|<expression>}

<stx>::=STX{<indexY|<expression>}

<sty>::=STY{<indexX>|<expression>}

<assembler statement>::={<branch>|<memory to A>|<A to memory>
          |<single byte>|<read modify write>|<bit>|<cp>|<jmp>|<jsr>
          |<ldx>|<ldy>|<stx>|<sty>}<comment field>
```

27 Error Codes

The following list of errors includes BASIC errors, COS errors, and errors generated by the extension ROM. Note that it is possible to obtain errors not on this list by executing a BRK in a machine-code program.

2   Too many GOSUBs

The largest permitted depth of subroutine nesting is 14. This error means that more than 14 GOSUB statements have been executed without matching RETURN statements. Example:

```
10 GOSUB 10
20 END
```

6   SUM   Checksum error

When loading a named file from tape, each block is followed by a checksum byte; if the checksum does not agree with this byte, this error is given. The cause of checksum errors is usually a damaged tape, or incorrect volume on playback. The remaining blocks of a damaged tape can be retrieved using FLOAD.

18   Too many DO statements

The largest permitted number of nested DO...UNTIL loops is 11. This limit has been exceeded.

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)
```

30   Array too large in DIM statement

The DIM statement checks that there is valid memory at the last element of each array in the DIM statement. This error implies that there is no RAM at the end of the array being dimensioned.

31   RETURN without GOSUB

A RETURN was found in the main program. RETURN is only meaningful in a subroutine.

39   Attempt to use variable in LIST

The LIST command may only be used with constants as its arguments. Example:

```
LIST A,B
```

48 COM? Command error

The command following the '*' was not a legal COS command. Example:

    *MEM   (command does not exist)

69 Illegal FDIM statement

Only the floating-point array variables %AA to %ZZ may be dimensioned in an FDIM statement. Example:

    10 FDIM %A(2)

Attempt to use FDIM in direct mode.

76 Assembler label error

The characters following the ':' character are not a legal label. Legal labels are two letters followed by a number optionally in brackets. Example:

    10:LOOP JMP LOOP

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

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)

An array appears in an INPUT statement; only simple variables are permitted. Example:

    25 INPUT AA(2)

95 Floating-point item missing or malformed

An unexpected character was encountered during the interpretation of a floating-point statement. Example:

    10 FUNTIL 0   (argument must be a relational expression)
    20 FIF A PRINT "OK"   (logical variables not allowed)

109 Number too large

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.

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:

206

```
35 FOR CC(1)=1 TO 10
```

The maximum permitted number of nested FOR...NEXT loops is 11; this number has been exceeded.

118 NAME Name error

The filename specified in a LOAD, SAVE, *LOAD, *SAVE, or *FLOAD command was not a legal COS filename. Example:

```
SAVE "THIS FILENAME IS TOO LONG"
```

123 Illegal argument to floating-point function

Examples:

```
12 A=SQR(-1)    (square root of a negative number)
24 B=ASN(2)     (arcsine of number outside range -1 to 1)
```

127 Line number not found in GOTO or GOSUB

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)
```

128 Argument to SIN, COS or TAN too large

The largest angle that may be specified in the SIN, COS or TAN functions is about 8.3E6.

129 Division by zero; protected RAM in graphics mode

A number was divided by zero. Example:

```
10 J=J/(A-B)     (where A and B were equal)
```

A CLEAR command specified a graphics mode that would have destroyed BASIC's text space. Example:

```
10 ?18=#90 ;REM Move text space
20 CLEAR 4
```

134 Array subscript out of range

An array element was specified with a negative subscript, or has not been dimensioned before use. Example:

```
10 DIM AA(4)
20 AA(-2)=7
```

135 SYN? Syntax error

A COS command was recognised, but was followed by illegal parameters. Example:

```
*SAVE "FRED"    (start and end addresses omitted)
```

149 Floating-point array subscript out of range

A floating-point array element was specified with a negative subscript. Example:

```
10 %AA(-2)=0
```

152 GOSUB without RETURN; FOR without NEXT

The GOSUB statement, when used in direct mode, must be followed by a semicolon. Example:

207

The FOR statement was used in direct mode without a NEXT statement.

156 Assembler error: illegal type

The argument specified for the operation is illegal. Example:

```
30 LDA @300     (constant greater than 8 bits)
50 STA (J,Y)    (not a legal addressing mode)
70 BIT @23      (immediate addressing not available with BIT)
```

This error is also generated if a JMP or JSR is assembled with a zero-page address. This may occur, by chance, on the first pass of a forward-reference JMP or JSR; in this case the value of the label should be initialised to P before assembling. Example:

```
40 JMP #34     (jumps into page zero are not permitted)
```

157 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
```

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:""
```

165 Loading interrupted

The CTRL key will escape from a load-from-tape operation, with this error message being produced.

169 Floating-point result too large

The result of a floating-point calculation was larger than about 1.7E38. Example:

```
20 FPRINT TAN(PI/2)
```

174 Significant item missing or malformed

An unexpected character was encountered during the interpretation of a statement. Example:

```
10 GOTO 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)
40 LET AA(1)=2     (LET is illegal with arrays)
```

191 LOG or power of zero or a negative number

The argument to the floating-point function LOG, or the operator '^', must be greater than zero. Examples:

```
10 %A=-1^2
30 %B=LOG(0)
```

198 UNTIL with no DO

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)
```

200 Unmatched quotes in string

Strings appearing in a program should have an even number of '"' quotation marks.

208 Unrecognised mnemonic in assembler

The mnemonic is not a legal 6502 assembler operation. Example:

```
20 ADD @20      (only ADC instruction available)
30 .BYTE        (assembler directives are not available)
```

216 Illegal DIM statement

The list of variables in the DIM statement contained an illegal entry. Example:

```
20 DIM A(2,3)      (only one-dimensional arrays allowed)
30 DIM AA(-2)      (negative array size)
```

Attempt to use DIM in direct mode.

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:

```
50 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.

238 Argument to EXP too large

The calculation of the EXP function gave a result that was too large. Example:

```
10 FPRINT EXP(100)
```

248 Not enough room to insert line

The line just entered has used up all the available memory. More memory can be released by shortening all the command names if this has not already been done.

Index