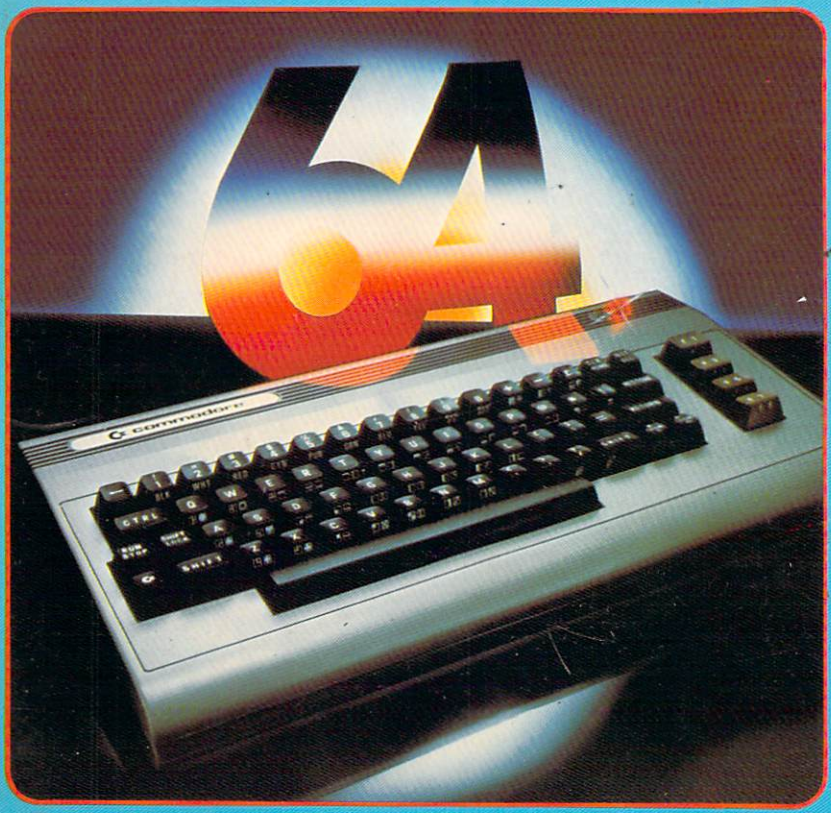
TEACH YOURSELF

Computer Programming

with the

COMMODORE 64



L.R. Carter and E. Huzan

COMPUTER PROGRAMMING WITH THE COMMODORE 64

Lionel Carter is a qualified Chartered Mechanical Engineer and Principal Lecturer in Management Science at Slough College of Higher Education.

Dr Eva Huzan is Head of the Computing Division at Slough College of Higher Education, a member of the Computing Science Advisory Panel ('O' and 'A' level), University of London, and a committee member of the British Computer Society Microcomputer Specialist Group.

The authors have collaborated before in writing *Learn Computer Programming with the Commodore VIC-20* and Teach Yourself books on *The Pocket Calculator, Microelectronics and Microcomputers* and *Computer Programming in BASIC*.

TEACH YOURSELF BOOKS

COMPUTER PROGRAMMING WITH THE COMMODORE 64

L. R. Carter and E. Huzan

TEACH YOURSELF BOOKS
Hodder and Stoughton

First published 1983

British Library Cataloguing in Publication Data

Carter, L. R.
Computer programming with the Commodore 64. – (Teach yourself books)
1. Commodore 64 (Computer) – Programming
I. Title II. Huzan, E.
001.64'24 QA76.8.c/

ISBN 0 340 34638 8

Printed and bound in Great Britain for Hodder and Stoughton Educational, a division of Hodder and Stoughton Ltd, Mill Road, Dunton Green, Sevenoaks, Kent, by Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk. Photoset by Rowland Phototypesetting Ltd, Bury St Edmunds, Suffolk

Contents

List of Figures

List of Tables

Acknowledgements

We would like to thank Commodore, especially John Baxter, Steve Beats, Adrian Butcher and Gail Wellington, for their helpful advice and assistance.

Introduction

The Commodore 64 computer has many facilities which you can use for a variety of applications in the home, at school, in offices and laboratories – wherever you have a mains point and a television set.

As soon as you switch on your Commodore 64, you will have access to the BASIC programming language; BASIC stands for Beginners All-purpose Symbolic Instruction Code. You will be able to write useful programs in BASIC very quickly by working through this book. No knowledge of computers or programming is assumed and the many exercises and problems (with answers in the appendices) will help you to progress at your own speed.

An attractive feature of the 64 is its colour capability. You will learn about colour and easy-to-use animation techniques early on in the book so that you can introduce these facilities in any of the programs. The 64 also allows you to program a variety of graphics and sound effects, including the music synthesiser. Sample routines are given in this book to illustrate how to program these effects, and you can include these in your own programs.

The applications dealt with in this book include:

colour effects, animation and sprite graphics;
sound effects and Morse trainer;
cost calculations and mortgage calculations;
stock file processing and simulation;
processing scientific and engineering data;
simple and more complex mathematical calculations.

These by no means exhaust the types of application for which you can use the 64. Once you have mastered these simple applications you will be able to start using your 64 for more sophisticated tasks. For example, the user port on the 64 can be used to control models and scientific equipment. You can obtain further attachments for your 64, such as joy sticks and paddles for games and control applications.

For more advanced business applications, a disk unit and printer can be plugged into the 64. The use of these units is discussed in later chapters in the book, and illustrated by a comprehensive stock recording program.

1

Introduction to Computers and Programming

1.1 Basic functions and units of a computer

An essential function of a computer is the ability to store the set of instructions required to process a particular task. This set of instructions (the *program*), which is prepared by a programmer, has to be held in the computer's store (its main *memory*) while the instructions are followed.

Each computer has a fixed instruction set which it can execute. A *control unit* selects the instructions, one at a time, from the memory, decodes or interprets them and causes the computer to carry out the instruction. If the instruction requires an arithmetic operation to be performed then the control unit transfers the necessary data between the memory and the *arithmetic and logic unit*.

The main memory, arithmetic and logic unit, and control unit comprise the central part of the computer, and together are known as the *central processor*.

Input and *output* peripheral devices, linked to the central processor, are used to insert programs and data into the computer's memory and to output results from there. Typical input devices read and decode patterns of punched holes on cards or paper tape, or sense marks optically or magnetically, and transmit this information electronically to the central processor. Alternatively, the information can be keyed in directly from a keyboard (similar to that of a typewriter). Results may be displayed on a television screen or visual display unit, or, if a 'hard' copy is required, printers are available which print either one character or a complete line at a

time across a page. Graph plotters may also be used as output peripherals.

Programs may be stored magnetically on *backing* or *secondary storage* devices, and these can then be read back into the computer's memory when required. On large (mainframe) computers, magnetic disks and tape are used for this purpose, while on the smaller microcomputers the equivalent devices are floppy disks and cassettes which hold less information and transfer it much more slowly. Other forms of secondary storage, such as magnetic bubble memories, are also available.

Secondary storage devices are also used to hold files of data records. These are transferred to and from the computer's memory under program control for file processing applications.

Figure 1.1 shows the basic units of a computer, the flow of data and control links.

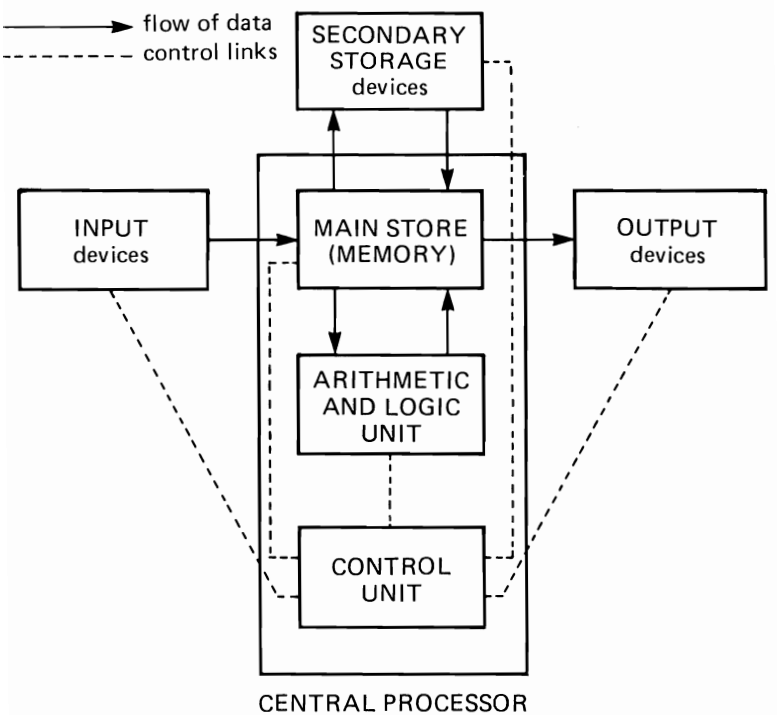


Figure 1.1 Basic units of a computer

1.2 How information is held

A computer is largely made up of a number of two-state devices. The 'off' state of the device may be considered to represent a 0 and the 'on' state a 1. A numbering system comprising only 0s and 1s is called a *binary system*. Different patterns of these binary digits (or *bits*) may be used to represent a character set and ranges of numbers.

Numbers are represented in the computer's memory as a combination of bits. The number of bits available to represent a number varies with the computer used. When using BASIC, most systems will work in floating point arithmetic, in which numbers are held as a *mantissa* and an *exponent*. For example, $6 \times 10^3 = 6000$, has a mantissa of 6 and an exponent of 3.

1.3 Programming a computer

Each family of processors has its own instruction set which is likely to differ from that of other processors. This means that a particular processor is only capable of understanding its own set of instructions in *binary code*.

The computer's memory can be considered as consisting of a number of cells capable of storing binary patterns representing program instructions or data. Each of these cells is uniquely numbered so that reference can be made to particular memory cells, either to select a program instruction or data, or to write data into a certain memory cell.

As an example of how programs are written in a computer's own code (machine code), it will be assumed that two numbers are held in memory cells 5 and 6, that these are to be added together, and the result stored in memory cell 8. The addition will be performed in a storage location called the *accumulator*, so the first instruction needs to load one of the numbers into the accumulator. The second instruction adds the other number to the number in the accumulator, which will then contain the sum of the two numbers. The third instruction stores the contents of the accumulator in the required memory cell.

The binary codes for these instructions for a typical processor are shown in Table 1.1.

Table 1.1 Machine code instructions

	Instruction	*Machine code*
1	Load number held in memory cell 5 into accumulator	10100101 0101
2	Add number held in memory cell 6 to number in accumulator	01100101 0110
3	Store number held in accumulator in memory cell 8	10000101 1000

In one program run, memory cells 5 and 6 could have been set to 70 and 25, respectively. After the three instructions in Table 1.1 have been obeyed, cells 5 and 6 would still contain 70 and 25 and cell 8 would now contain 70 + 25 i.e. 95. The same program could be run again with different data in cells 5 and 6 (say 43 and 12), which would result in cell 8 having its previous value of 95 replaced by the new value of 55.

1.4 Programming languages

As you have seen, programming in the computer's own machine code requires that the instructions and data are given to it in binary. Writing down and keying in a series of 1s and 0s is time-consuming and prone to error. An alternative way of expressing the instructions is to use *mnemonic codes*. For example, the command to load a number from a memory cell could be written as LDA instead of, say, 10100101. Also the memory cells could be given symbolic names instead of referring to them by their actual numeric (binary) addresses.

This type of programming language is used when it is necessary to have close control over the functions of the computer. Languages which use such mnemonic codes are known as *assembly languages*. Each assembly language instruction usually corresponds to an equivalent machine code instruction. The translation of the assembly language program into machine code is carried out by a machine code program called an *assembler*.

High-level languages have been devised which allow several machine code instructions to be expressed in one statement. BASIC is such a programming language as shown in the example below:

```
LET C = A + B
```

is a BASIC statement which causes the two numbers, held in memory cells A and B, to be added together and the sum stored in memory cell C. This is the same problem which previously required three machine code or assembly language instructions.

However, neither assembly language nor BASIC programs can be understood directly by the computer. BASIC programs need to be translated into machine code using a *compiler* or *interpreter*. The basic difference between these two is the stage at which the translation from BASIC into machine code is performed. Using a compiler, the translation is done *before* the program is executed; this gives speed advantages over an interpreter which performs the translation process as it executes the program. The Commodore 64 BASIC interpreter is particularly suitable for first-time users because it has been designed so that programs can easily be altered and corrected.

1.5 Microprocessors and microcomputers

The development of microtechnology has opened up the use of computers to many more people than was previously possible, and has changed the way many applications can be handled. In addition, the compactness and low cost of microcomputer-based systems are making possible new applications.

Microelectronic devices (integrated circuits) are made from wafer-thin pieces of semi-conductor material, such as silicon. A small chip of silicon, a few millimetres square, can contain a very large number of electronic components built into circuits.

Integrated circuits (ICs) which have a wide variety of processing and storage functions are available. Today it is possible to have all the circuits needed for a microcomputer on a single semi-conductor chip, which is about the same size as the early ICs that contained only a few components. Large Scale Integrated circuits (LSIs) contain many thousands of components.

In a microcomputer, the chip performing the functions of the central processor is called a microprocessor. The microprocessor performs the following functions:

Synchronisation of processing events and instruction decoding (control unit);

Temporary storage of addresses and data (registers);
Arithmetic and logic operations (arithmetic and logic unit).

Further information is given in *Microelectronics and Microcomputers*, by L. R. Carter and E. Huzan (Teach Yourself Books, 1981).

Your 64 contains Commodore's own 6510 microprocessor and three other Commodore special-function devices: two video interface controllers (devices 6566 and 6567), and a sound interface device, SID (device 6581). The video interface controllers inside your 64 are special devices which control information, such as colour, displayed on the screen. The sound interface device is a three-voice electronic music synthesiser/sound generator. These devices can be controlled by instructions written into programs.

2

Simple Input and Output Statements

2.1 READ, DATA, INPUT and PRINT statements

This chapter explains how you may enter information into your 64 computer *(input)*, and how the 64 may be programmed to supply information, for example on a printer or video screen *(output)*.

Each BASIC instruction (or statement) consists of a command to the computer to carry out a certain action, and a combination of variables, constants, separators (e.g. a comma) and operators (e.g. +) on which the action is to be performed. For example:

```
10 READ A,B,C
```

tells the 64 to read three numbers (numeric constants) from the DATA statement (see line 20 below) and store them in three cells in the 64's memory identified by the names A, B and C. A, B and C are called *variables* and refer to unique numeric addresses, as explained in Chapter 1. When A, B or C are referenced again in the *same* program, the 64 will obtain the current contents of these cells. In a *different* program, A, B and C may refer to cells with different actual numeric addresses but unique for *that* program. Single memory cells are referenced in BASIC programs by single letters of the alphabet, A–Z, followed optionally by a single number, 0–9, or another letter A–Z.

The 10 before READ in the statement above is the *line number*. Line numbers enable you to change particular lines in your program by retyping the line. Gaps may be left in the sequence of line

numbers for subsequent insertion of additional instructions. Three further instructions complete the program to read and output (PRINT) three numbers:

```
20 DATA 25,11,30
30 PRINT A,B,C
40 END
```

The END statement terminates execution of the BASIC program; that is, the processing of the program instructions is stopped.

Note that the following three statements have the same effect as the above READ statement. That is, after these three instructions have been executed with the DATA statement shown in line 20, A, B and C will contain 25, 11 and 30 respectively.

```
10 READ A
11 READ B
12 READ C
```

Type this program into your 64 exactly as shown, pressing the RETURN key once at the end of each line. After the complete program has been typed in, type LIST to check that this has been done correctly. If there are any mistakes, you can 'edit' the program by moving the cursor to the required position. There are special cursor control keys which allow you to move the cursor to the left and right and up and down. Once you have reached the position of the mistake on the screen, simply type over the mistake with the corrected character. After the correction has been made, press the RETURN key with the cursor still in position over the line which is being modified. On re-listing the program, the corrected version should appear on the screen.

To help you in editing, there is a special insert/delete key which allows insertion of spaces and deletion of characters. The LIST command can be used to list a particular line (e.g. LIST 30) or groups of lines (e.g. LIST 200–250).

When you have entered your program correctly, type in the command RUN and press the RETURN key to initiate execution of your program.

Note in particular how the contents of A, B and C (i.e. 25, 11 and 30) are output and the number of spaces between the numbers. Change the program so that the PRINT statement is as follows:

```
30 PRINT A;B;C
```

and note the spacing between the numbers when a semi-colon is used to separate the variables in a PRINT statement instead of a comma.

To change the data, you will need to alter the numbers in the DATA statement. Alternatively, you may use an INPUT statement instead of READ and DATA statements. Replace the READ and DATA statements in the program by the following statement:

```
10 INPUT A,B,C
```

(delete line 20 by typing 20 and pressing the RETURN key). When run, the 64 will output a question-mark (?) to indicate that data should be input from the keyboard.

A heading may be output at the beginning of the output from the computer by putting it in double quotation marks in a PRINT statement as in the following example:

```
28 PRINT "A", "B", "C"
30 PRINT A,B,C
```

(*Note*: In computer codes the *same* character is used for open and closed double quotation marks.)

An alternative method of identifying the three numbers is to output A = followed by the number. Experiment with the following statement to obtain the spacing you require (delete line 28):

```
30 PRINT "A =";A; "B =";B; "C =";C
```

The information in the double quotes is output as given in line 30, while A, B and C which are *not* in quotes refer to memory cells. If A, B and C contain 25, 11 and 30 respectively, line 30 will output:

$$A = 25 \quad B = 11 \quad C = 30$$

You may put extra spaces (∇ indicates a space) between the quotation marks. For example, substituting,

$$\text{"}\nabla\nabla\nabla\nabla\nabla A\nabla = \text{"}$$

in statement 30 above would result in A being output with five spaces before it and one space before the equals sign.

The TAB function may be used to output information in particular column positions on your video screen as illustrated in the following example:

```
30 PRINT TAB(5);"A =";A; TAB(15);
"B =";B;TAB(25);"C =";C
```

This will cause A = to be output in positions 6, 7 and 8, followed by the contents of cell A, then B = in positions 16, 17 and 18, followed by the contents of cell B, then C = in positions 26, 27 and 28, followed by the contents of cell C.

The TAB function will be discussed further in subsequent chapters.

2.2 String variables

For many problems it is necessary to input, store and output variable information which consists of a mixture of letters, numbers and special characters, including spaces. Such a series of symbols is called a *string*. Strings may be stored in *string variables*. These must be given a name consisting of one of the alphabetic letters A–Z (followed optionally by another letter A–Z or 0–9), followed finally by a dollar sign $ (e.g. A$, B$, C$, . . ., Z$).

The constant information given in double quotes previously is termed *a string constant.*

String variables are essential for reading in and manipulating files of information, particularly for business applications. Just a few examples of the use of string variables are given here to give you some initial practice.

Once a program has been written and proved correct, it may be used over and over again with different data, on different occasions and by different people. It is useful, therefore, to output the date the program has been run, and perhaps by whom. Two string variables may be used to input this information and to cause it to be output.

Change and insert statements in the program to output three numbers as follows:

```
10 INPUT A,B,C,D$,N$
26 PRINT
27 PRINT "DATE";D$;"      ";N$
28 PRINT
```

When this amended program is run, in reply to ? you will need to input three numbers separated by commas (for A, B and C), followed by the date and your name. For example, input data for the above program could be:

? 25,11,30, 26/04/83 , J.SMITH

Try running this program with different data, different dates and your name. Notice that the PRINT statements at lines 26 and 28 output blank lines.

2.3 Obtaining the required print layout

It is important to design suitable output so that this can be output in different formats for different purposes. Various ways of using name and address information will be used to illustrate this. The program given in Table 2.2 inputs a title (MR, MRS, MISS etc.), a name and an address, so that this is stored in memory cells referenced by string variables, and outputs a letter heading, notebook label and envelope labels to the screen.

The five INPUT statements shown in Table 2.2 will cause the computer to request five lines of data to be input. When working interactively, each line of data is entered in response to the ? output by the computer as shown in Table 2.1. The INPUT statements (lines 10 to 50) contain a 'message' to prompt the user to enter the correct data. For example, line 10 will cause TITLE? to be displayed on the screen.

Table 2.1 Data for program

```
TITLE? MR
NAME? J.SMITH
ADDRESS 1? 1 THE AVENUE
ADDRESS 2? KENSINGTON
ADDRESS 3? LONDON W8
```

Enter and run the program on your 64. You will find it convenient to save the program on a cassette as explained in the next section. Programs can also be saved on disk using the instructions given in Chapter 13.

The REM (remarks) statements at lines 5, 70, 150 and 290, in

Table 2.2 Name and address program

```
5 PRINT"□":REM CLEAR SCREEN
10 INPUT "TITLE";T$
20 INPUT "NAME";N$
30 INPUT "ADDRESS 1";A$
40 INPUT "ADDRESS 2";B$
50 INPUT "ADDRESS 3";C$
60 PRINT"□"
70 REM LETTER HEADING
80 PRINT TAB(19);A$
90 PRINT TAB(19);B$
100 PRINT TAB(19);C$
110 PRINT
120 PRINT
130 PRINT
140 END
150 REM NOTEBOOK LABEL
160 PRINT
170 PRINT
190 PRINT TAB(7);"**********************"
200 PRINT
210 PRINT
220 PRINT TAB(9);N$
230 PRINT
240 PRINT
250 PRINT TAB(7);"**********************"
260 PRINT
270 PRINT
280 PRINT
285 END
290 REM ENVELOPE LABELS
300 PRINT
310 PRINT
320 PRINT
330 PRINT T$;" ";N$;TAB(19);T$;" ";N$
340 PRINT A$;TAB(19);A$
350 PRINT B$;TAB(19);B$
360 PRINT C$;TAB(19);C$
370 END
```

Table 2.2, are only listed with the program to explain the program's actions. Note that you can put more than one statement separated by a colon (:) against one line number (as shown in line 5). This avoids using extra line numbers and is shown in several examples in this book. The 'clear screen' character, shown in quotes in line 5, is obtained by pressing shift and the CLR/HOME key. This will give you a clear screen so that the letter heading is displayed at the top of your screen. When the END instruction is reached at line 140,

execution of the program is terminated but you can continue the processing by typing the command CONT, and similarly at the end of the 'Notebook Label' part of the program to continue after line 285.

The notebook label will contain just the name of the owner in between two lines of asterisks. However, at this stage you will not be able to centralise the name, according to its length, for names of varying length. This will be dealt with in Chapter 13, which contains an enhanced version of the program that outputs letter headings, notebook labels and envelope labels to a printer.

2.4 Saving and loading programs using the cassette recorder

Start by positioning the cassette in the recorder at the place where you want to record the program. Then type SAVE"TABLE 2.2" and press the RETURN key. (TABLE 2.2 will be recorded as the name of the program in the 'header' at the beginning of the saved program on the cassette tape.) The 64 will prompt you to record your program by displaying a message specifying which keys to press on the cassette recorder. It is useful to make use of the counter on the recorder so that re-positioning of the tape for subsequent verifying or loading of the program is made easier.

You should write the program name, counter number and date on a card (stored in the cassette case) each time you save a program. Several small programs may be saved on each side of the tape.

You can verify that the recording has taken place successfully by rewinding the cassette tape, to a position before the program header (using the counter), typing VERIFY"TABLE 2.2" and pressing the RETURN key. The 64 will compare the program in its memory with the one recorded on the tape. If there is a discrepancy, an error message will appear on the screen, otherwise OK will be displayed. If there is a verification error, you will need to save and verify the program again.

To load the saved program into the 64's memory, position the tape in the cassette recorder at a place before the program header, type LOAD"TABLE 2.2" and press the RETURN key.

When you press the cassette keys to save or load a program, the screen goes blank while the 64 is communicating with the cassette unit. If a program is being loaded, the screen display returns each

time a program is found on the tape. The 64 will stop reading the tape for a few seconds to display the name of the program found. It will then continue to read the tape in a similar manner until the required program is found. Again after a short pause the program will be loaded, during which time the screen blanks out again. When the program has been loaded, the screen display returns to normal and displays the word READY.

The few seconds' pause while the 64 displays each program name can be terminated by pressing the CBM logo key on each occasion. Alternatively, at these times the cassette operation can be aborted by pressing the STOP key.

The screen also blanks out whenever the 64 is writing to the cassette unit during a 'save' operation.

2.5 Clearing a program from the memory

Before keying in a new program, it is advisable to clear the current program from the memory. This is done by typing NEW and pressing the RETURN key. Remember to SAVE the program first, before using NEW, if you want to use it at a later stage.

If NEW is not used, then lines in the previous program will appear in the current program, unless the *same* line numbers have been used, that is, they have been overwritten.

3

Arithmetic Operations

3.1 Constants and variables

Your 64 may be programmed to perform a variety of calculations by means of arithmetic assignment statements in which the result of the calculation is assigned to a memory cell. For example:

```
50 LET S = X + Y
```

causes your 64 to add the contents of memory cell X to that of memory cell Y and puts the result in a memory cell called S. X and Y will have had values assigned to them previously, either by an INPUT or READ + DATA statements or by another LET statement. The contents of cells X and Y are unchanged by the action of the LET statement. For example, Table 3.1 shows the contents of X, Y and S before and after execution of the above LET statement, in a program which contains the following statements in addition to line 50 above:

```
30 READ X, Y
40 DATA 123,56
```

Table 3.1 Contents of X, Y and S

Cell	*Before*	*After*
X	123	123
Y	56	56
S	?	179

Note the original, unknown, content of S has been overwritten by the new value 179, the sum of 123 and 56.

The variables on the right-hand side of the equals sign in a LET statement may be operated on by a number of different arithmetic operators, and may be mixed with constant values (constants). For example:

```
51 LET I = I - 1
```

subtracts 1 from the current value of I, so that after the LET statement has been obeyed I has a value one less than its previous value. Note that the word LET may be omitted.

The *numeric constants* that may be used are:

(*a*) whole numbers *(integers)* which do not contain a decimal point, for example, −45,360 (or +360);
(*b*) numbers containing a decimal point *(floating point)*, for example, 8.123, −97.5;
(*c*) numbers in exponential format, for example, 12.3E4, which represents $12.3 \times 10^4 = 123000$ (4 is called the exponent). The exponent may also be negative, for example, 12.3E −4, which is $12.3 \times 10^{-4} = 0.00123$.

Note that numbers are made *negative* by putting a minus sign (−) in front of them; a plus sign (+), or *no* sign, indicates the number is *positive*.

Any number that is used in the program, either as a constant or as the contents of a variable, must lie within the range of limits of your 64 (i.e. ±1.70141183E+38, ±2.93873588E−39 for largest and smallest numbers, respectively).

3.2 Arithmetic operators

The symbols on the right-hand side of the equals sign in a LET statement may consist of variable names, constants and arithmetic operators; this combination of symbols is called an *arithmetic expression*. The *arithmetic operators* indicate which arithmetic operation is to be carried out on the numbers in the arithmetic expression. The following list shows the order in which operations are performed unless changed by the use of brackets as explained in the next section.

Arithmetic operators	*Meaning*
↑	raise to a power
+	(exponentiation)
*, /	multiply, divide
+,−	add, subtract

3.3 Hierarchy of operations

It is possible to use brackets in an arithmetic expression to give the correct meaning. The contents of the brackets are evaluated first starting with the innermost pair of brackets and working outwards. For example, to evaluate

$$\frac{5+9}{4+3}$$

the top line *(numerator)* needs to be added first, then the bottom line *(denominator)* needs to be added, and finally the numerator is divided by the denominator. Brackets are used to ensure this order of evaluation.

Table 3.2 Program to illustrate order of evaluation

```
30 READ B,C,D,E
40 DATA 5,9,4,3
50 LET A=(B+C)/(D+E)
70 PRINT B;C;D;E;A
80 END
```

A program to illustrate the order of evaluation is given in Table 3.2; the larger gap in the sequence of line numbers between the LET and PRINT statements will allow the insertion of additional statements later. Table 3.3 shows the contents of the memory cells before and after the LET statement in line 50 has been obeyed.

Table 3.3 Contents of B, C, D, E and A

Cell	B	C	D	E	A
Before	5	9	4	3	?
After	5	9	4	3	2

Run this program on your 64 and then amend the LET statement as follows (i.e. remove the brackets):

```
50 LET A = B + C/D + E
```

A will now be 10.25 (i.e. $\frac{9}{4} + 5 + 3$). This is because the 64 evaluates the arithmetic expression in a certain order if there are no brackets, depending on the arithmetic operators in the expression.

If there are no brackets, then the 64 will perform the exponentiations first (if there are any), followed by multiplication and division of equal hierarchy, but in the order left to right, lastly addition and subtraction of equal hierarchy. Within brackets the same order of evaluation is carried out, innermost brackets being calculated first as previously stated.

Looking again at the last statement at line 50, you will see that the division, C/D, has been carried out first as it is of higher hierarchy than addition. This gives a completely different result from that calculated in the previous LET statement in Table 3.2 where brackets were used.

3.4 Arithmetic expressions and statements

Table 3.4 LET statements

```
50 LET A=(B+C)/(D+E)
51 LET G=C/E-B*D
52 LET H=C/(E-B)*D
53 LET J=G-H/E+E↑2
61 LET S=C*D-B↑A
62 LET T=(C*D-B)↑A
63 LET U=(E*(C-B)↑(D/A))
70 PRINT
71 PRINT
72 PRINT"B=";B;"C=";C;"D=";D;"E=";E
73 PRINT
74 PRINT
75 PRINT"A=";A;"G=";G;"H=";H;"J=";J
76 PRINT
77 PRINT
78 PRINT"S=";S;"T=";T;"U=";U
80 END
```

Insert all the LET and PRINT statements shown in Table 3.4 into your program and run it. Output all the results (A,G,H,J,S,T,U)

together with the variable names as identification. The data read and the final results are shown in Table 3.5.

Table 3.5 Data read and final results

B = 5 C = 9 D = 4 E = 3
A = 2 G = −17 H = −18 J = −2
S = 11 T = 961 U = 48

Notes

1 Line 51 could be replaced by

```
51 LET G = (C/E) − (B*D)
```

to give the same result, although the brackets are unnecessary in this case.

2 The brackets in line 52 are essential to give the correct answer, as can be seen by comparing the results of line 52 with that of line 51.

3 In line 53, H is used in the expression because a value was assigned to it in line 52. Instead of E ↑ 2, you can use E*E, which is a quicker operation. Amend line 53 to

```
53 LET J = G − H/E+E*E
```

and check that you get the same result for J.

4 In line 61, B ↑ A (i.e. 5^2) is evaluated first, then C*D (i.e. 9 × 4) before the subtraction (i.e. 36 − 25) is carried out. However, in line 62 the contents of the brackets are evaluated first (i.e. 21) before this is squared by A.

5 There are three pairs of brackets in line 63. The innermost pair is evaluated first from the left, that is, C − B (equals 4), then D/A is evaluated (equals 2). The exponentiation is carried out next to give 4^2, and finally this is multiplied by E (i.e. 3).

Table 3.6 Changes to arithmetic expressions

```
51 LET G=C/(E-B)*D
52 LET H=C/(E-B*D)
53 LET J=((G-H)/E+E)↑2
61 LET S=C*(D-B)↑A
62 LET T=C*(D-B↑A)
63 LET U=E*C-B↑D/A
```

As a further exercise, change the arithmetic expressions in lines 51 to 63 in Table 3.4 to those given in Table 3.6. Check your results with those given in Table 3.7. Your 64 has the facility for changing individual characters in a line; use this facility instead of typing whole lines again.

Table 3.7 Results of arithmetic operations

```
B = 5 C = 9 D = 4 E = 3
A = 2 G = -18 H = -.529412 J = 7.97232
S = 9 T = -189 U = -285.5
```

3.5 Problems

You are now ready to attempt some simple problems. For more complicated problems, it is advisable to express the logic in the form of a *flowchart* before coding it in BASIC, as explained in Chapter 4.

In your programs, use constants instead of variables for values that are not going to change during the execution of the program or from one run of the program to the next. Variable names should be meaningful: for example, use E for Expenses.

Write programs and run them on your 64 for each of the following problems. If you get errors, reading Chapter 4 will help you to correct them. Compare your programs with those given in Tables A1 and A2 in Appendix A; substituting actual values in place of the variables will help you understand the action of each instruction. The results from each program for the data given are shown in Tables 3.8 and 3.9. You should experiment with a variety of PRINT statements to give different outputs (e.g. underline answer with hyphens or asterisks, line up values).

Problem 1 – Number of £s required

On your proposed visit to the USA, you will need 150 dollars a night for accommodation and 125 dollars a day for food, travelling and incidental expenses. You intend to stay five nights and wish to take sufficient dollars to have 100 dollars to buy presents. How many pounds sterling will you need to exchange if the exchange rate is 1.5 dollars to the £? (Your program should be flexible enough to cope with changes in expenses, the length of stay and the exchange rate for subsequent visits.)

Table 3.8 Output from 'Number of £s required' program

```
LENGTH OF STAY (NIGHTS)                 : 5
ACCOMMODATION (PER NIGHT)               $: 150
EXPENSES (MEALS ETC.)                   $: 125
ALLOWANCE FOR PRESENTS                  $: 100
EXCHANGE RATE ($ TO THE £)              : 1.5

POUNDS STERLING REQUIRED                : 983.33
************************************************
```

Problem 2 – Cost of stationery

Calculate the cost of stationery for a course that is being run, given the following information:

Number of delegates attending	58
Cost of folders	14p each
Cost of paper	26p per pad
Cost of pens	12p each

Allow two pens per delegate (there is a quantity discount of 8% for orders over 100 pens). Write the program so that it may also be used on other occasions, when different numbers of delegates will be attending, and allow for changes in costs.

Table 3.9 Output from 'Cost of stationery' program

```
NO OF DELEGATES                 : 58
COST OF FOLDERS                 : 14P EACH
COST OF PAPER                   : 26P PER PAD
COST OF PENS LESS 8%            : 12P EACH

TOTAL COST OF STATIONERY = £36.00
************************************************
```

4

Program Development

4.1 The need for pre-planning

This chapter gives you guidance on developing a proposed program. If a program is written too hastily, valuable time may be lost subsequently in implementing the necessary changes. Time spent pre-planning is seldom wasted. Commercial systems designers and programmers are expected to conform to a specific formal procedure. In developing your own programs, you need to exercise self-discipline.

4.2 Understanding the problem

The first step is to ensure that you understand what you intend or are required to do. Are the terms of reference clear? This might mean that you need to check the meaning of any terminology or jargon used. You may also need to ensure you understand the mathematical notation used to specify any relationships involved. Thus, initially, some research or background reading may be necessary. Research may also be necessary when you know what you want to do, but are not sure of the method to be used.

4.3 Designing output

The starting point of designing a program should be the output. You need to consider and make decisions on the following aspects.

The output from a program may be printed and/or written to a

file. Is your output going to be solely printed, written to a file or a mixture of both? This leads on to deciding precisely what is to be printed and what is to be written to the file.

For example, your intention may be to write a program to read a stock data file and produce a list of items to reorder. Given, for the moment, that a program can be written to identify the items to be reordered, you need to consider: should the output be solely a printed list or should a reorder file be produced that can be the input to a purchase order program? If you are going to have a printed list of items to be reordered, what should it contain? Should it list the complete stock record of each item or, the other extreme, should it just be a list of stock code numbers?

A program of this nature is developed in Chapter 14, in that case the reorder list consists of stock code, stock description and order quantity. It was not necessary to output the whole record.

Having decided what is to be output, it is then necessary to consider the format and general layout. The considerations to be made are:

In which columns are the variables to be printed?
Should they be truncated or rounded?
Are column headings necessary?
Are main headings necessary?
What spacing is required between headings?
Should headings be underlined?

4.4 Input requirements

Once the output details have been decided you can then identify the necessary input. If a large amount of data is to be processed it may be advisable to read it from a data file; this is dealt with further in Chapter 14. If the data is solely associated with the one program it can be incorporated in DATA statements, while data that varies from run to run is best entered via INPUT statements.

You may not be the only person using the program and this is a factor to be considered. Values should be entered in their most usual form (i.e. 12.5 not .125 for interest rates – see, for example, the mortgage problem in Chapter 12). Ample print messages should be provided, giving guidance, if necessary, as to the input required.

A further aspect of the input design is the desirability of providing some form of control over the program during run time. For example, in the 'Heat of combustion' problem (Chapter 12), the user is asked whether any more data is to be processed and replies Y or N, i.e.

```
100 INPUT"ANY MORE DATA (Y = YES, N = NO)"; Y$
```

4.5 Flowcharting

Once you have a broad idea of your requirements the logical sequence of the program statements needs to be developed. This can be done by drawing a flowchart. The more common symbols used in flowcharts are shown in Figure 4.1.

An example of the use of the flowchart symbols is given in Figure 4.2, where it is required to calculate the average of three numbers. The purpose of a flowchart is to ensure the logic is correct before becoming involved with the detail of individual program statements. Further examples of flowcharts will be found elsewhere in this book accompanying the descriptions of programs.

On occasions it becomes apparent from the flowchart or analysis of the problem that a similar calculation will be repeated several times in the program. When a similar set of program statements is likely to be required in several parts of the program, this may indicate the possibility of writing them once only as a *subroutine* and using this routine several times over. A discussion of subroutines is the subject of Chapter 9.

Having drawn flowcharts, the next stage is writing the program. When the program has been written, you still have not finished. A very important part of producing useful programs is to ensure that they perform as intended, and the next section discusses the testing and documentation of your programs.

4.6 Program testing

If you make a mistake in the use of the BASIC language, your 64 will detect this and output a message to tell you that there is a *syntax error* in your program. Examples of typical syntax errors are: mistakes in spelling (e.g. IPUT instead of INPUT), wrong instruc-

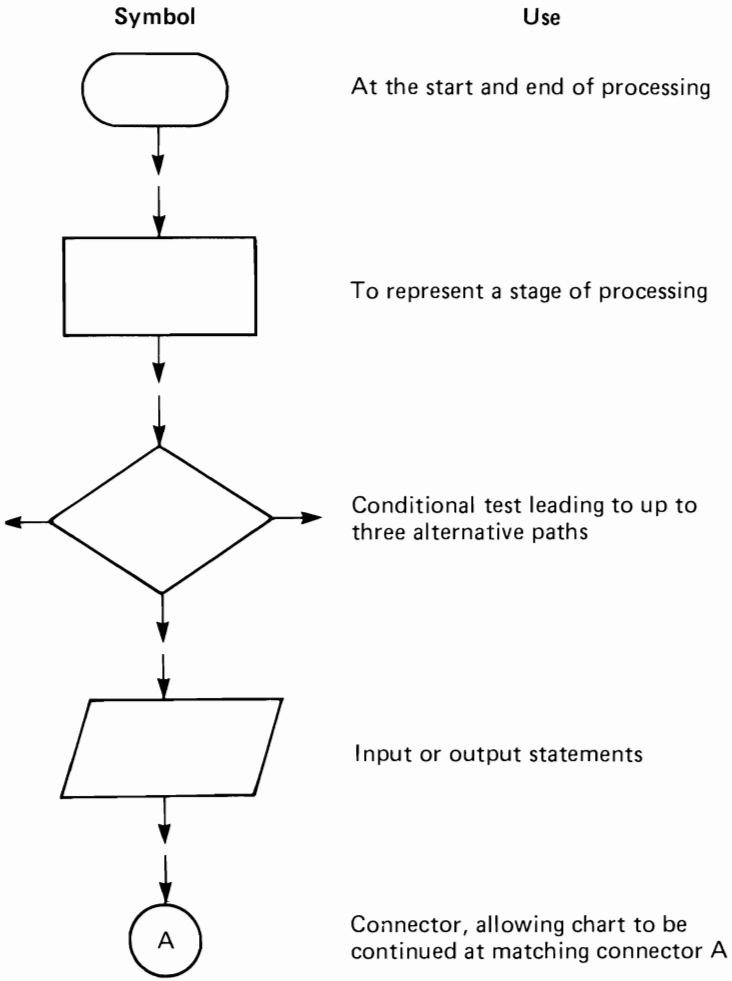


Figure 4.1 Some flowchart symbols

tion format (e.g. LET X+Y=S instead of LET S=X+Y), and unacceptable variable name (e.g. 2A instead of A2). You must clear all the syntax errors before you proceed.

Your program may still be incorrect after the syntax errors have been cleared. You may get an *execution error* caused by asking your 64 to perform an action which it cannot do. For example, if values

are calculated by your program which are either too small or too large you will get arithmetic overflow (this will happen when dividing by zero). An execution error will occur also if you try to assign a string to a numeric variable (e.g. using D instead of D$ for a date, 26/04/83).

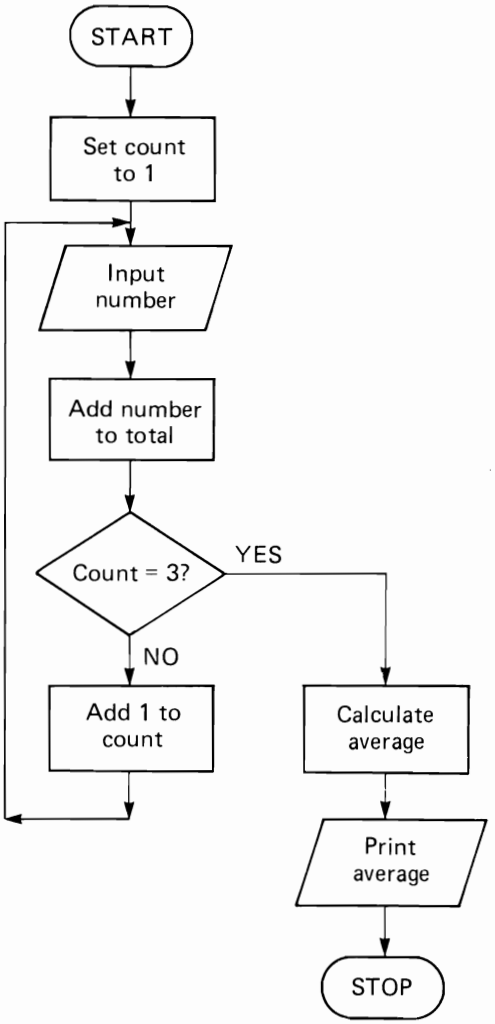


Figure 4.2 Flowchart for average of three numbers

A program which runs successfully, without an execution error occurring, may still give the wrong results because the logic of the program is incorrect. You should work through your flowchart and/or program instructions with typical data before running the program on your 64 (this is known as performing a dry-run). Then run the program on your 64 with this typical data; this should be designed to test every instruction path in the program (i.e. every branch in your flowchart).

It is important to write down details of the program and its use, for subsequent reference. You will find it useful to include the following sections in your documentation: Identification, Contents Page, Summary, Description of the Problem, Specification of the Problem, Input and Output Formats, Use of Program, Interpretation of Outputs, Modifications, Appendices.

5

Conditional and Unconditional Branching

5.1 Controlling the order in which instructions are obeyed

For most problems your 64 needs to be programmed to *repeat* a set of instructions and to execute different sets of instructions in the program according to the requirements for that particular run. This is done by means of *branch* (jump) instructions.

The GOTO instruction causes control to pass to the line number in the statement. That is, the computer will execute next the statement it has branched to and continue to execute the instructions following in sequence until it encounters another branch instruction. For example:

```
50 LET I = 1
60 PRINT I
70 LET I = I + 2
80 GOTO 60
```

will cause the odd numbers 1, 3, 5 etc. to be printed. When the computer executes the instruction at line 80, it will always branch to line 60 and obey that instruction followed by line 70. Therefore, the GOTO statement is an unconditional branch instruction, since it is always executed independently of any condition that exists.

However, you will notice that in the above section of a program, there is no instruction which stops the program being executed; it will go on for ever!

To stop the computer executing this set of instructions, you will

need to insert a conditional branch instruction. This will perform a test to see if a condition exists and pass control to a different part of the program according to the result of the test.

A conditional branch instruction that you may use in BASIC is the IF . . . THEN statement. For example, to stop the program which prints odd numbers, you could add the following instructions to those given above:

```
65 IF I = 21 THEN 90
90 END
```

Try running the program and see if it stops after 21 has been printed. If you replace 21 by an even number, say, 20 or 22, the program will not stop since I never has this value.

5.2 Loops and their control

This small program that you have just tested has a set of instructions, lines 60 to 80, which are performed repeatedly, thus forming a loop. The flowchart for this program shows the loop and the branch out of the loop more clearly (see Figure 5.1). Notice the GOTO 60 instruction is represented by an arrow from box 70 to box 60.

There are several alternative ways of exiting from a loop and for branching to different parts of a program. The format of the IF . . . THEN statement is:

line number IF *relational expression* THEN *different line number*

Notice that the line number following the THEN *must* be different from the line number preceding the IF, otherwise the IF statement itself will cause continuous looping.

The relational expression is the test that is to be performed. If this test is true (that is, the condition exists), then control passes to the line number following the THEN. If the test is false, then control passes to the line number following the IF statement, that is, the instructions will continue to be obeyed in sequence until another branch instruction is met.

The relational expression compares two expressions, so that its format is:

expression relational operator *expression*

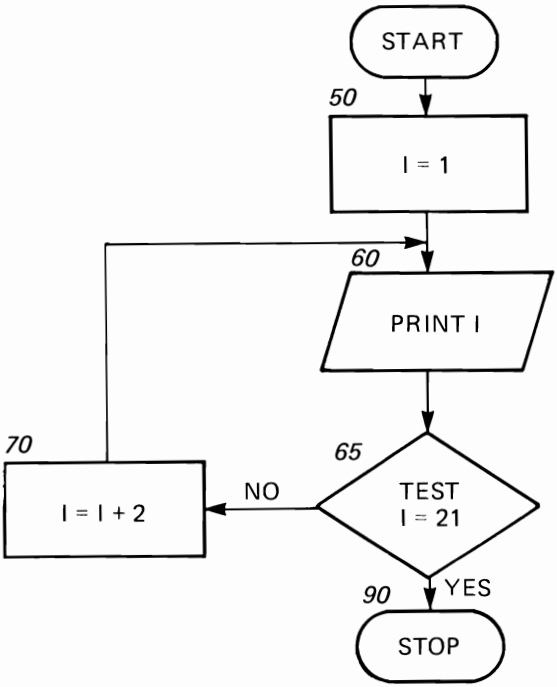


Figure 5.1 Flowchart to illustrate loop control

You have already used one relational operator in the previous example = (equal to). The full list is given in Table 5.1.

Table 5.1 Relational operators

relational operator	*meaning*
=	equal to
>	greater than
<	less than
> = or = >	greater than or equal to
< = or = <	less than or equal to
< > or > <	not equal to

The IF . . . THEN statement is useful for terminating the inputting of data, as it can be used to test for a final dummy value. This is a

value which indicates the end of the data list, but which is not used in the calculations in the program. This is illustrated in Table 5.2, which shows a program to add numbers. The numbers are entered one at a time in response to the INPUT statement in line 30.

Table 5.2 Terminating with a dummy value

```
10 PRINT"ADD NUMBERS"
11 PRINT
20 LET T=0
30 INPUT X
40 IF X<=0 THEN 70
50 LET T=T+X
60 GOTO 30
70 PRINT"TOTAL =";T
80 END
```

The program in Table 5.2 will stop when either a zero or a negative value is read into X. The IF . . . THEN statement must appear before the calculations involving X, so that the dummy value is not used in the calculations.

Another way to stop repetition of a set of instructions is to specify the number of times the loop has to be carried out, as shown in Table 5.3.

Table 5.3 Program to add N numbers

```
10 INPUT N
20 PRINT"ADD";N;"NUMBERS"
25 PRINT
30 LET I=0
35 LET T=0
40 INPUT X
45 LET T=T+X
60 LET I=I+1
70 IF I<N THEN 40
80 PRINT"TOTAL =";T
90 END
```

If line 30 in Table 5.3 read:

30 LET I = 1

then line 70 would need to be:

70 IF I< = N THEN 40

This is because the value of I, after line 60 has been obeyed, is one greater than the number of numbers when the loop has been executed N times, if I is set to 1 to start with. This means the loop is terminated when I = N + 1.

5.3 Comparing character strings

The IF . . . THEN statement may also be used to compare character strings, since each character is represented by a unique combination of binary digits when stored in the computer. For example, if P$ contains the character H, then:

```
25 IF P$ = "H" THEN 30
```

will be true and a branch will be made to line 30.

This facility is particularly useful for comparing names, addresses and similar information for business applications. You will need to refer to a list of codes used to represent characters in your computer's memory to find out which characters have a lower or higher value for greater than or less than tests (see Appendix C).

5.4 The FOR . . . NEXT statements

In section 5.2, the number of times a loop was executed was programmed by setting an initial value for the loop counter, testing for a final value, and incrementing the current value of the loop counter if the final value had not been reached. The FOR . . . NEXT statements have been designed to program these three operations in an easier way.

In the example to add N numbers, in Table 5.3, the variable I (used as the loop counter) was set to an initial value 0. 1 was added to I after the number had been read and added in, and finally a test was carried out (I < N) to determine whether the program should loop back or stop. FOR . . . NEXT statements will be used in an alternative version of the program. The FOR . . . NEXT statements consist of two lines of code. At the beginning of the loop the FOR statement is used to set up the initial conditions, the increment

or STEP to be made at the end of the loop and the final value as follows:

line number FOR *variable* = *expression 1* TO *expression* 2
STEP *expression* 3

where expression 1 sets the initial value of the loop counter also known as the index), expression 2 sets the final value of the loop counter, and expression 3 gives the increment to be added to the variable at the end of each pass through the set of instructions in the loop. If the STEP is equal to 1, both the word STEP and expression 3 may be omitted.

The final instruction in the loop has the format:

line number NEXT *variable*

where the variable has the same name as that given in the associated FOR statement.

The program in Table 5.3 can be amended as shown in Table 5.4. A number will be read into X N times as controlled by the FOR . . . NEXT statements. I is set to 1 initially in line 35, then in line 60 I is incremented by 1 and if it is greater than N the program will go to line 80 and print the total, otherwise it goes back to line 40.

Table 5.4 Alternative program to add N numbers

```
10 INPUT N
20 PRINT"ADD";N;"NUMBERS"
25 PRINT
30 LET T=0
35 FOR I=1 TO N
40 INPUT X
50 LET T=T+X
60 NEXT I
80 PRINT"TOTAL =";T
90 END
```

Insert the instruction:

```
70 PRINT I
```

so that you can see the value of I after the loop has been executed for the required number of times.

You may use I within the loop, but you should avoid changing I (that is, assigning a new value to I) within the loop as this changes

the conditions set up by the FOR . . . NEXT statements. The problem flowcharted in Figure 5.1 may be coded as follows:

```
50 FOR I = 1 TO 21 STEP 2
60 PRINT I
70 NEXT I
80 END
```

The value of the increment given in the expression following STEP may be negative (so that the loop counter is decremented) or fractional. Table 5.5 shows a program where you can input the start, end and step as variables (A, B and C). Try a number of different combinations, including negative and fractional values, and see what happens. Table 5.6 shows a similar program where expressions have been used in place of simple variables.

Table 5.5 Start, end and step variables

```
10 PRINT"START,END AND STEP VARIABLES"
16 PRINT
20 INPUT A,B,C
30 FOR I=A TO B STEP C
40 PRINT I
50 NEXT I
60 END
```

Table 5.6 Start, end and step expressions

```
10 PRINT"START,END AND STEP EXPRESSIONS"
16 PRINT
20 INPUT A,B,C
30 FOR I=A+1 TO B/2 STEP C-3
40 PRINT I
50 NEXT I
60 END
```

5.5 The ON . . . GOTO statement

The format of the ON . . . GOTO statement is

line number ON *expression* GOTO *two or more line numbers separated by commas*

The integer part of the evaluated expression must be a positive number not greater than the *number* of line numbers after the GOTO part of the statement.

Control will pass to the first, second, third, etc., line number after

the GOTO if the integral part of the expression is equal to 1, 2, 3, etc.

For example, different calculations may need to be carried out according to a code, as in the following problem. A number of sets of data are to be input. Each set consists of a code (1, 2, 3, 4 or 5) and values of X and Y. Calculations are to be performed on each set of data according to the rules shown in Table 5.7.

Table 5.7 Calculations for different codes

Code	*Calculation*
1	R = X + Y
2	R = X − Y
3	R = X*Y
4	R = X/Y
5	R = X ↑ Y

Problem 1 – Using the ON . . . GOTO statement
Write a program to tabulate the code, the X and Y values, and the results of the calculations shown in Table 5.7.

Use the ON . . . GOTO statement to control which calculation is to be carried out according to its associated code. Draw a flowchart for the program, prepare test data, code and run your BASIC program. Remember the test data must test every branch in your program. You may input the codes and data in any order, that is the first set of data may have a code of, say, 3, the next a code of 1, etc. Compare your program with the one listed in Table A3. Suitable test data and calculated values are given in Appendix B.

5.6 Further use of the TAB function and FOR loops

The TAB function may be used with a variable or expression in the brackets following TAB, for example TAB(I), TAB(P − 1). The program given in Table 5.8 outputs a rectangle of variable dimensions, consisting of L1 dashes for the two lines across and L2 ↑ s for the two vertical lines.

Notes on Table 5.8

1 Lines 20 and 40 output a message to the user asking for data to be entered.

Table 5.8 Program to output a rectangle

```
20 INPUT"START COLUMN POSITION";P
40 INPUT"LENGTH ACROSS & DOWN";L1,L2
60 LET K=1
70 PRINT TAB(P-1);
80 REM OUTPUT DASHES ACROSS
90 FOR I=1 TO L1
100 PRINT"-";
110 NEXT I
120 PRINT
130 IF K=2 THEN 240
140 REM OUTPUT "↑"S DOWN
150 FOR I=1 TO L2-2
160 PRINT TAB(P-1);"↑";
170 FOR J=1 TO L1-2
180 PRINT" ";
190 NEXT J
200 PRINT"↑"
210 NEXT I
220 LET K=K+1
230 GOTO 70
240 END
```

2 The PRINT statement in line 70 is terminated by a semi-colon (;); this will cause the *next* PRINT statement that is obeyed to output on to the *same* line.

3 Line 120 is necessary to cause the complete line of L1 dashes to be output. After passing through line 220, which sets K to 2, lines 90 to 120 are repeated to complete the rectangle and execution of the program is then terminated.

4 Lines 150 to 210 comprise a FOR loop which has another FOR loop (lines 170–190) wholly within it. The FOR loops are said to be *nested* and this will be discussed further in Chapter 8. For each pass through the outer FOR loop, the inner loop is executed L1–2 times, so that a ↑ is output followed by some spaces and then another ↑. When line 190 is reached another pass through the outer loop is executed until L2–2 lines, consisting of ↑ spaces ↑, have been output.

In this case, the use of nested FOR loops could be avoided by replacing lines 160 to 200 by the following statement:

160 PRINT TAB(P − 1); "↑"; TAB(L1 − 1 + P); "↑"

Figure 5.2 shows a rectangle output by the program when the following data was used:

7
20, 10

Figure 5.2 Rectangle output from program

that is, twenty dashes were output for the two lines across and eight ↑ s for the two vertical lines.

You should enter the program given in Table 5.8 into your 64 and run it with different input data.

Problem 2 – Centering a rectangle

Write another program to output a rectangle of variable width and depth in the centre of the screen, using graphics characters for the four corners and four sides. The program is listed in Table A4.

6

POKE, PEEK and Colour

6.1 Character codes

As explained in Chapter 1, characters and numbers are stored in a computer as binary patterns. Standard binary codes have been established by different organisations. The American Standard Code for Information Interchange (ASCII) has been widely adopted and Appendix C gives these codes as implemented on the 64. Characters can be converted into these codes and *vice versa* by the use of ASC and CHR$ string functions. These and further string functions are described below.

6.1.1 CHR$

This function returns the character corresponding to a specified ASCII code, i.e.

```
10 LET A$ = CHR$(66)
```

The ASCII code for the letter B is 66, so the above statement stores B in A$. Words can be built up by concatenation, i.e.

```
10 LET A$ = CHR$(66) + CHR$(69)
```

results in A$ = BE, where 69 is the ASCII code for E.

Note that as CHR$ returns the ASCII code, variables can be set if required to various control characters (i.e. Return).

6.1.2 ASC

This function is the opposite of CHR$ in that it returns the ASCII code number for a specified character, i.e.

```
10 LET X = ASC("E")
```

results in X = 69.

If the argument is a string variable the ASCII code of the first character is returned, i.e.

```
 5 LET T$ = "TOTAL"
10 LET X = ASC(T$)
```

results in X = 84.

6.1.3 LEN

This function returns the length of a string. For example, changing line 10 to:

```
10 LET L = LEN(T$)
```

would set L = 5.

6.1.4 LEFT$, RIGHT$

These functions return the leftmost or rightmost specified number of characters from a string, for example:

```
10 LET B$ = LEFT$(T$,2)
```

returns the leftmost two characters from the string T$, i.e. B$ = TO, similarly,

```
10 LET E$ = RIGHT$(T$,3)
```

leaves E$ = TAL

6.1.5 MID$

This function returns a substring of n characters starting with the ith character, i.e.

```
10 LET C$ = MID$(T$,2,3)
```

results in C$ = OTA where $n = 3$ and $i = 2$.

6.1.6 STR$

This function converts a numeric argument to the string equivalent of its PRINTed form, i.e.

```
10 LET N = 64
20 LET X$ = STR$(N)
```

results in X$ containing " 64" as a string, thus

```
30 LET Y$ = "CBM" + X$
40 PRINT Y$
```

results in,

```
CBM 64
```

being printed. Note that the string version of the numeric contains a leading blank (the suppressed + sign).

6.1.7 VAL

This function is the opposite of STR$. The string is examined, left to right and the *first* recognisable number format is returned, i.e.

X = VAL("CBM 64") results in X = 64
X = VAL("−78.97.65") results in X = −78.97

6.2 POKE and PEEK

The POKE command allows you to place any specified value directly into a required memory location. Its general form is, POKE *x,y* where the value of *x* specifies the memory location and *y* the value to be POKEd, for example:

```
POKE 53281, 5
```

The above command places the value 5 into memory location 53281. In this case location 53281 happens to be the one that determines the colour of the screen. The value 5 causes the screen (i.e. the background to the text) to go green. A more detailed explanation of the control of colour is given in section 6.3.

The most common use for POKE on the 64 is to control the colour, sound and position of characters on the screen. Advanced programming techniques can involve PEEKing and POKEing to a special area of computer memory known as zero page, in order to

achieve special effects that are not available from BASIC. Although PEEKing and POKEing cannot in any way damage your 64, you should first master simple applications, such as POKEing the 64's screen memory and the 64's control registers, before trying out more sophisticated techniques. If at any time the 64 'crashes' (e.g. the cursor disappears, etc.), then pressing the STOP and RESTORE keys simultaneously will reset the 64 without the loss of your program.

The PEEK function allows access to the contents of a specified memory location. PEEK(x) returns the value of the contents of location x. The value returned can be assigned to a variable in the usual way, for example:

```
10 POKE 1024, 83
20 LET S = PEEK(1024)
30 PRINT S
```

Running the above program causes the value 83 to be stored in memory location 1024. When this location is PEEKed in line 20, the variable S is set to the value found in location 1024, namely 83. Line 30 therefore prints out the value 83.

PEEKing may be used in games and animation to determine the 'status' of parts of the screen, for example whether a missile is now occupying the target's location, thereby implying a hit.

Variables can be used with both PEEK and POKE so that all the following are examples of valid syntax:

```
POKE X,Z
POKE X + 40*L, Z
A = PEEK (X + 40*L)
IF A$ = CHR$(PEEK(K)) THEN 50
```

6.3 The 64 colour system

The 64 allows independent control of the colour of the screen, its border and any individual character.

In each case one of sixteen colours can be chosen. The border colour is set by POKEing a value ranging from 0 to 15 to memory location 53280. For example, to set the border colour to green:

```
POKE 53280, 5
```

The screen colour (also called background colour) can be set in a similar way by POKEing 53281. Appendix D gives a list of the colours available and their appropriate POKE values.

Control over the colour of characters can be achieved in several ways. The colour of characters can be programmed by the use of the colour keys, or by the use of specific character codes, or by POKEing direct to the screen.

The colour of characters output to the screen from the current position of the cursor onwards can be set by keying CTRL simultaneously with one of the labelled colour keys, or the CBM logo simultaneously with a labelled colour key. CTRL and 1 to 8 gives you the first eight colours as labelled, while CBM and 1 to 8 gives you the remaining eight colours. This keying sequence can also be incorporated within quotes in a PRINT statement, for example:

```
10 PRINT "CTRL and Red key THIS PRINTS RED"
```

When used within a program each colour control instruction will be displayed as a unique symbol. These symbols are shown in Appendix E.

The keying of CTRL (or CBM logo) and a colour key can be assigned to a string variable which is then incorporated as required in PRINT statements, for example:

```
10 C$ = "CTRL and Red key"
20 PRINT C$; "THIS PRINTS RED"
```

A third way of PRINTing in a specified colour is to use the appropriate character code and CHR$. The values to be used with CHR$ are given in Appendix D, for example:

```
10 PRINT CHR$(28); "THIS PRINTS RED"
```

This method of changing the colour of characters is useful for printing headings.

The program in Table 6.1 demonstrates changing the colours of the border, screen and text. When the program is run, the sixteen colours are listed on the screen and you are requested to input an S, B, or T to change the colour of the screen, border or text, respectively. You are then asked to input the number of the colour (from 0 to 15). Having made the colour change, the program loops back so that you can continue making further changes. You will find that

some text-screen colour combinations make it difficult to read the text and if the text is made the same colour as the screen it will not show up at all. Note that the program uses arrays which are dealt with in detail in Chapter 8.

Table 6.1 Demonstration of screen, border and text colours

```
35 DIM CC(15)
40 FOR I=0TO15:READ CC(I):NEXT
50 PRINT"■"
60 PRINT" 0  BLACK"
62 PRINT" 1  WHITE"
64 PRINT" 2  RED"
66 PRINT" 3  CYAN"
68 PRINT" 4  PURPLE"
70 PRINT" 5  GREEN"
72 PRINT" 6  BLUE"
74 PRINT" 7  YELLOW"
76 PRINT" 8  ORANGE"
78 PRINT" 9  BROWN"
80 PRINT"10  LIGHT RED"
82 PRINT"11  GREY 1"
84 PRINT"12  GREY 2"
86 PRINT"13  LIGHT GREEN"
88 PRINT"14  LIGHT BLUE"
90 PRINT"15  GREY 3"
100 PRINT
105 PRINT"■S■REEN, ■B■ORDER OR ■T■EXT ";
110 INPUT A$
120 PRINT
130 INPUT"COLOUR NUMBER";N
135 IF N<0 OR N>15 THEN 50
140 IF A$="S" THEN POKE 53281,N
150 IF A$="B" THEN POKE 53280,N
160 IF A$<>"T" THEN 50
165 PRINT
170 PRINT CHR$(CC(N));"COLOURED TEXT"
180 INPUT"PRESS RETURN TO CONTINUE";Z
190 GOTO 50
200 DATA 144,5,28,159,156,30,31,158
210 DATA 129,149,150,151,152,153,154,155
220 END
```

Line 35 dimensions a character code array CC in readiness to store the 16 CHR$ values. These values are read into the array CC within the loop at line 40 from DATA lines 200 and 210. The screen is cleared in line 50, and lines 60 to 90 list the colours and their associated numbers onto the screen. Line 105 requests the user to

enter S, B or T as required, the response being stored in A$. The colour number is requested in line 130 and stored in the variable N. Line 135 checks that N is within limits. If A$ contains S then the screen colour is changed in line 140. If A$ contains B then the background colour is changed in line 150. Line 160 ensures that the program returns to line 50 unless a text colour change was selected (A$ = "T"). When a text colour change is requested the program passes to line 170 where the appropriate CHR$ code is used to print the message COLOURED TEXT. If the program then immediately looped back to line 50 (which clears the screen) this message would be lost, therefore line 180 is necessary to allow the user time to read the message.

It should be noted that any changes to the colours persist and get carried over into other programs. Using the STOP and RESTORE keys will cause the 64 to revert to its 'normal colour' display of blue and light blue.

The space bar can be used to produce 'bars' of colour by use of the RVS ON and RVS OFF keys. If the reverse character set is turned on by the RVS ON key, a space becomes a solid block of whatever text colour is in operation. This is also programmable. For example:

10 PRINT "*CTRL and RVS ON key CTRL and Red key* ▽▽▽▽▽"

produces a red bar five spaces long. Note that RVS ON and RVS OFF both have character codes (18 and 146 respectively) that can be used in programs in the same way as the codes for colours. Thus the previous example could also be:

10 PRINT CHR$(18);CHR$(28); "▽▽▽▽▽"

A further method of producing coloured characters is to use the POKE command and is discussed in the next section.

6.4 POKEing coloured characters

The screen of the 64 is memory mapped; that is, each possible character position on the screen has an associated memory location. For example, the top left-hand corner position corresponds to memory location 1024 and by POKEing this address with the required character value, the character will appear on the screen in

white (for this character to be visible the screen colour should be set for the time being to another colour). The screen codes for each key are given in Appendix F. Note that these codes do not correspond to the character codes of Appendix C, and are only relevant for POKEing to the screen. Thus

```
POKE 1024, 83
```

will produce a white heart in the 'home position'. By adding 128 to the code its equivalent reverse character is obtained.

For each screen memory location there is a corresponding colour code memory location (e.g. 55296 is the colour location corresponding to the screen position 1024). It is important to ensure that the correct matching colour location is used with the required screen memory location. The colour of the character POKEd to a location can be controlled by POKEing the values given in Appendix D, for example 2 for red. Thus

```
POKE 1024, 83 : POKE 55296, 2
```

will produce a red heart in the 'home position'. Note that the colour values, for the first eight colours, are one less than the number shown on the colour keys.

The screen memory locations together with their associated colour locations are given in Appendix G. It is convenient, when writing programs that PEEK or POKE to the screen, to use expressions that relate to the 'home position' and the required offset. These expressions can also use the required row and column numbers to calculate the memory location, i.e.

```
100 SM=1024: CM=55296
110 POKE SM + 40*R + C, 83
120 POKE CM + 40*R + C, 2
```

This routine will POKE a red heart to row R, column C. Note that the top row and the leftmost column are row zero and column zero, respectively.

6.5 Histogram example

The program in Table 6.2 illustrates the use of colours to produce a histogram. This routine plots thirty vertical lines, in green, over a scale from 0 to 20.

Table 6.2 Coloured histogram example

```
8 POKE 53280,5:POKE53281,6:PRINT CHR$(5)
9 SM=1024+40*21+5:CM=55296+40*21+5
10 PRINT"■" ;
20 PRINT"              ***HISTOGRAM***"
30 FOR L=20 TO 1 STEP -1
40 L$="  "+STR$(L)
50 PRINT RIGHT$(L$,4);" ┐"
60 NEXT L
70 PRINT"   0 └┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┴┘"
80 FOR P=1 TO 30
90 READ D
100 IF D=0 THEN 150
110 FOR H=1 TO D
120 POKE SM-(H*40)+P,160
130 POKE CM-(H*40)+P,5
140 NEXT H
150 NEXT P
160 DATA 5,7,8,3,5,2,6,0,1,6
170 DATA 7,9,3,5,8,10,12,7,9,12
180 DATA 10,15,11,17,13,18,15,13,10,12
190 END
```

Line 8 sets the border colour to green, the screen colour to blue and the text to white (PRINT CHR$(5)). The screen is cleared in line 10 and the main heading printed by line 20. Lines 30 to 60 form a loop that produces the vertical scale. The string handling in these lines causes the numbers obtained from the loop count to be aligned correctly. The graphics character in line 50 forms part of the scaled vertical axis. Line 70 produces the horizontal axis.

In this example, data is contained in DATA lines 160 to 180 and is read as required for each column in line 90 into the variable D. The histogram is built up by POKEing green reverse characters vertically over the thirty columns as required. Lines 80 to 150 form the outer loop that repeats the inner loop for each column. If the histogram has a zero entry then line 100 causes the inner loop to be bypassed.

The inner loop (lines 110–140) POKEs a reverse character space (code 32 + 128 = 160) vertically D times. The correct POKE position is determined by the screen and colour memory offsets, set in line 9, modified by the current value of H and P.

6.6 Coloured mosaic example

The ability of the 64 to hold colour character codes in string variables allows the colours to be manipulated by string handling techniques within a program. An example of this approach is given by the program in Table 6.3. The program produces a mosaic pattern of coloured squares. An initial sequence of twelve colours, generated at random, are used to form a symmetrical pattern 24 by 24.

Table 6.3 Coloured mosaic patterns

```
10 DIM O$(24,24),C$(12),D$(12)
15 POKE 53280,12:POKE53281,12:PRINT"■"
20 PRINT"□"
25 PRINT"           PLEASE WAIT WHILE"
26 PRINT"           COLOURS ARE SET UP"
30 A$="■[illegible]"
40 PRINT"■";
50 FOR I=1 TO 12
60 N=INT(RND(1)*16+1)
70 C$(I)=MID$(A$,N,1)
80 D$(I)=C$(I)
90 NEXT I
100 FOR L=1 TO 12
110 D$(1)=C$(L)
120 FOR P=12 TO 1 STEP -1
130 O$(P,L)=D$(P)
140 O$(25-P,L)=D$(P)
150 O$(P,25-L)=D$(P)
160 O$(25-P,25-L)=D$(P)
170 D$(P)=D$(P-1)
180 NEXT P
190 NEXT L
200 FOR L=1 TO 24
210 PRINT "         ";
220 FOR P=1 TO 24
230 PRINT O$(P,L)"■ ";
240 NEXT P
250 PRINT
260 NEXT L
300 GOTO 40
310 END
```

Line 10 dimensions the arrays used in the program. The border and screen colours are set to Grey 2 in line 15 and the text to Black. Line 20 clears the screen and lines 25 and 26 display a message. The message is positioned near the centre of the line so that it becomes overwritten by the pattern. In line 30 the variable A$ has a string of

sixteen colour codes assigned to it. Line 40 moves the cursor to the 'home position'. In lines 50 to 90, a random number from 1 to 16 is generated; this is used as the basis for selecting one of the colour codes from string A$. This is repeated twelve times so that the array C$ ends up containing a random sequence of colours. The same sequence is also assigned to D$. During the program run C$ remains unchanged to provide a reference string, while D$ is manipulated as required for each successive line.

The pattern for the top half (first twelve lines) is developed within the loop 100 to 190. As the line number (L) is increased by one, so the first colour to be displayed (D$(1)) is set to C$(L). The loop lines, 120 to 180, then allocate the display codes contained in the array D$ to specific positions in the output array O$. Four assignment statements are used to place the contents of D$ into four 'mirror image' positions.

Line 170 advances the display codes along one position, leaving D$(1) containing a null (that is, empty) string. On looping back to 110, this first position has a colour code assigned to it, according to the current line number (L), from C$.

The final part of the program, lines 200 to 260, prints out the array O$ within two loops. Line 210 starts each line eight spaces in from the left-hand side of the screen. Line 230 prints a reverse space character in the colour represented by the code in O$(P,L). Having printed the whole pattern, line 300 causes the program to repeat from line 40, which has the effect of overlaying the existing pattern on the screen with a new randomly generated pattern.

6.7 Animation

The building up of the bars in the histogram example represents a simple form of animation. However, more elaborate animation, such as that used in games, requires characters to move up, down and across the screen. The illusion of movement is obtained by POKEing the character into a suitable adjacent screen location and then POKEing a space into the preceding position.

A simple example of this is given by the program in Table 6.4. This program causes an asterisk to move across the screen from left to right in row 8 (note that the top row is row zero). Lines 10 to 30 set the initial screen and colour locations to the first position of row 8.

Table 6.4 An example of simple animation

```
5 PRINT"□"
10 SM=1024:CM=55296
20 S=SM+40*8
30 C=CM+40*8
40 FOR I=0 TO 39
50 POKE S+I,42
60 POKE C+I,5
70 POKE S+I-1,32
80 FOR D=1 TO 100:NEXT D
90 NEXT I
100 POKE S+39,32
110 END
```

The loop (lines 40–90) moves the required character to each column along that row and colours it green (line 60). Line 70 within the loop puts a space into the preceding column. A delay loop in line 80 stops the asterisk moving too fast. Line 100 puts a space into the final position of the asterisk on exiting the loop.

Problem

Use graphics characters to build up a simple picture of a face on the screen. By POKEing to the appropriate part of the picture, introduce animation that causes the face alternately to sulk and smile.

An example of such a routine is given in Table A5. This program is based upon the face design shown in Figure 6.1.

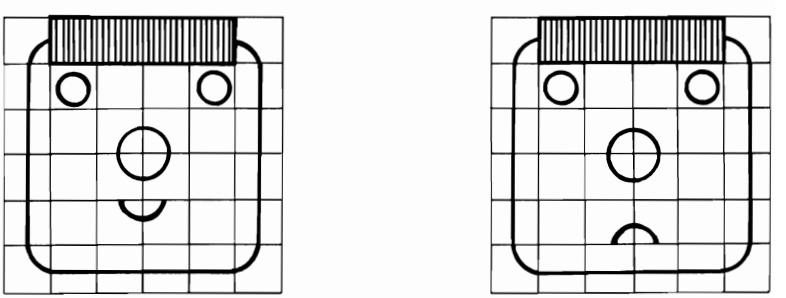


Figure 6.1 Animated face

7

Other Functions

7.1 Mathematical functions

Commonly used routines, such as those required for obtaining the integer part of a number (INT), the logarithm and antilogarithm of a number (LOG and EXP), and trigonometric functions (e.g. SIN) are available as library functions in BASIC. Examples of a variety of these functions will be given in the following sections.

Further background on mathematical functions and problems involving these are given in *The Pocket Calculator* by L. R. Carter and E. Huzan (Teach Yourself Books, 1979).

7.2 Arguments

Each function name is followed by an expression (the argument) in brackets. The function operates on the argument, that is, the value of the expression is used in the standard routine represented by the function name. For example:

```
100 LET S = SQR(B*B - 4*A*C)
```

will evaluate the square root of the expression in brackets, i.e. $B^2 - 4AC$, and put the result in cell S.

There may be restrictions regarding the values of the argument associated with a function. For example, it is not possible to take the square root of a negative number, therefore the argument used with SQR must not have a negative value. The TAB function followed by a semicolon causes characters to be output in the column following

the argument value; therefore, this value must correspond to a possible column position. A comma in place of the semicolon will have a different effect.

7.3 Using library functions

Library functions are used in LET or PRINT statements on their own or in expressions of any complexity. These expressions may contain further library functions. The evaluation is, as usual, working from the innermost brackets outwards.

7.4 Truncation

You have already used the library function INT to obtain the integer part of a number which has decimal places. The INT function gives the largest integer which is not greater than the argument. Therefore, if the argument is a *positive* number, the decimal places are dropped and the number is said to be truncated after INT has been used. For example:

```
110 LET B = INT(A)
```

puts 15 into B if A is 15.36. Remember, A will remain unchanged after the LET statement has been obeyed, so it will still contain 15.36.

However, if A contains −15.36 then the integer placed into B is *not* −15 (since this is larger than −15.36) *but* −16; in this case, B does not contain the truncated value of A.

To obtain the truncated value of a *negative* number, the sign must be removed from the number before the INT function is applied, using the function ABS which takes the absolute value of its argument (i.e. the sign is ignored), and the function SGN used. SGN gives the value of 1 if its argument has a positive value, −1 if its argument has a negative value, and a zero if the value of its argument is zero. For example:

```
120 LET B = SGN(A)*INT(ABS(A))
```

multiplies the integer part of the absolute value of A by its sign, so that B will contain the truncated value of A when A is positive or negative. Assuming A contains −15.36, as in the previous ex-

ample, then ABS(A) gives 15.36, INT(ABS(A)) gives 15, and SGN(A)*INT(ABS(A)) multiplies 15 by −1 giving −15.

7.5 Rounding

Numbers often need to be rounded to a nearest number of decimal places or to a nearest value in general. Adding 0.5 to a number before truncating it will cause the number to be rounded to the nearest integer (whole number). For example:

```
130 LET B = INT(A + 0.5)
```

puts 24 in B if A contains, say 24.3, and 25 in B if A contains, say 24.5 or 24.6. The program shown in Table 7.1 illustrates this method of rounding; angles input in decimals of a degree are output in degrees and minutes, rounded to the nearest minute.

Table 7.1 Rounding to nearest minute

```
20 PRINT"  ANGLE    DEGS    MINS"
30 PRINT
40 INPUT A
50 IF A=0 THEN 110
60 LET D=INT(A)
70 REM ROUND
80 LET M=INT((A-D)*60+0.5)
90 PRINT A;TAB(10);D;TAB(17);M
100 GOTO 40
110 END
```

To round a number to a certain number of decimal places, you need to divide the number by a scaling factor before adding 0.5, truncating, and finally multiplying by the scaling factor. For example, to round to three decimal places the scaling factor is 0.001:

```
140 LET P2 = INT(P1/0.001 + 0.5)*0.001
```

puts 3.142 into P2 when P1 contains 3.14159.

In general, if the scaling factor is contained in F then an expression may be rounded by using:

INT((*expression*)/F + 0.5)*F

This will work also if, for example, you wish to round a number to the nearest 10; in this case, F = 10.

7.6 Square roots

The library function for obtaining a square root is SQR. Remember the argument must not have a negative value. You can use SGN to test the sign, as shown in Table 7.2.

Table 7.2 Use of SGN and SQR

```
70 INPUT N
80 FOR I=1 TO N
90 INPUT A,B,C:PRINT
100 LET R=B*B-4*A*C
110 IF SGN(R)=-1 THEN 150
120 LET R=SQR(R)
130 PRINT"SQUARE ROOT OF R =";R;"FOR";A;B;C
140 GOTO 155
150 PRINT"RESULT NEGATIVE FOR";A;B;C
155 PRINT
160 NEXT I
170 END
```

The program given in Table 7.3 calculates and outputs the diameter in metres (rounded to two decimal places) of cylindrical tanks, given the volume V (in litres of water) and three standard heights in metres. The formula for the volume of a cylinder of height h, and radius r is:

$$V = \pi r^2 h$$

Therefore, the diameter d is given by:

$$d = 2r = 2\sqrt{\frac{V}{\pi h}}$$

Table 7.3 Calculation of diameter of cylindrical tanks

```
20 PRINT"   VOLUME      HEIGHT    DIAMETER"
30 PRINT"    LTRS.        M.          M."
40 PRINT"   ------      ------    --------"
50 DATA 1,1.25,1.75
60 INPUT V
80 IF V=0 THEN 160
85 PRINT
90 FOR I=1 TO 3
100 READ H
110 LET D=INT(SQR(V/(1000*π*H))*200+0.5)/100
120 PRINT TAB(4);V;TAB(14);H;TAB(24);D
130 NEXT I
140 RESTORE
150 GOTO 60
160 END
```

In the problem, the three standard heights are given in a DATA statement. For each volume V the diameter D is calculated and output using each of the three standard heights in turn. Every time the READ H statement (line 100) is obeyed, the next value in the DATA statement is taken. That is, the first time through the FOR loop H is taken to be 1, the second time 1.25, and the third time 1.75. The DATA pointer then needs to be reset to the beginning of the DATA values ready for a further three passes through the FOR loop with the next value of V. This is achieved by the RESTORE statement in line 140. Use the program to find the diameter of tanks which have volumes of 500 and 1000 litres (1000 litres = 1 m^3). The answers are shown in Table 7.4.

Table 7.4 Output from program given in Table 7.3

VOLUME LTRS.	HEIGHT M.	DIAMETER M.
500	1	.8
500	1.25	.71
500	1.75	.6
1000	1	1.13
1000	1.25	1.01
1000	1.75	.85

7.7 Trigonometric functions

The sine, cosine and tangent of angles are obtained by using the function names SIN, COS and TAN respectively, followed by the angle in brackets (expressed in radians). For example:

```
100 LET X=COS(B)
```

will put the cosine of B(radians) in cell X.

Only the inverse tangent (arctangent) is available as the function ATN. This has as the argument the tangent of the required angle. The angle obtained will be in radians.

The following BASIC statements may be used to find angle A in radians given that the sine of the angle is S or the cosine of the angle is C:

```
110 LET A=ATN(S/SQR(1 - S*S))
120 LET A=π/2 - ATN(C/SQR(1 - C*C))
```

Note: You must avoid using the above formulae when S=1 (required angle is $\pi/2$) or C=1 (required angle is 0).

π is available as a library function with your 64.

You may need to use a combination of these functions and you can do this in one statement. For example, you may be given the three sides of a triangle *(a,b,c)* and need to find the sines of the angles.

You could use the formula

$$\cos B = (a^2 + c^2 - b^2)/2ac$$

to find the cosine of angle B (and similarly for angles A and C).

If you then want to find the sine of the angles, you could use the following formula

$$\sin B = \sin\left(\tan^{-1}\left(\frac{\sqrt{1-\cos^2 B}}{\cos B}\right)\right)$$

since $\sin^2 B + \cos^2 B = 1$ and $\tan B = \dfrac{\sin B}{\cos B}$.

Therefore, the following BASIC statements will result in sin B being stored in variable S:

```
40 LET X=(A*A+C*C-B*B)/(2*A*C)
50 LET S=SIN(ATN(SQR(1-X*X)/X))
```

where the *sides* of the triangle *(a,b,c)* are stored in the variables A, B and C.

7.8 Logarithms and antilogarithms

The logarithms and antilogarithms of expressions are given by the functions LOG and EXP, respectively. For example, the x^{th} root of a number may be found by dividing the log of the number *(y)* by *x* and taking the antilog; this may be expressed as shown in the following BASIC statement:

```
100 LET R = EXP(LOG(Y)/X)
```

After this statement has been obeyed, R will contain the required root.

The function LOG gives the logarithm of its argument to base e; these are known as Naperian (or natural) logarithms. Since,

$$\log_{10} y = \frac{\log_e y}{\log_e 10}$$

the following BASIC statement finds the log of a number (Y) to base 10:

```
110 LET T = LOG(Y)/LOG(10)
```

Similarly, the antilog is found by multiplying the log to base 10 by $\log_e 10$ and taking the antilog of the result as follows:

```
120 LET A = EXP(T*LOG(10))
```

e has the value 2.7182818 to 8 significant figures. The function EXP raises e to the X^{th} power, where X is its argument. That is, $EXP(X) = e^x$; the use of EXP is illustrated further in the next section.

7.9 Hyperbolic functions

Hyperbolic functions may be expressed in terms of e^x. For example:

$$\sinh x = \frac{1}{2}(e^x - e^{-x})$$

$$\cosh h = \frac{1}{2}(e^x + e^{-x})$$

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The sinh of the number held in cell X will be placed into cell Y by the following LET statement:

```
110 LET Y = (EXP(X) - EXP(-X))/2
```

7.10 TAB function

The TAB function has already been used in several examples. The definition of TAB is summarised below. The TAB function may

only be used in PRINT statements to give the next output column position.

If TAB(P) is followed by a semicolon, then the variable or expression following the semicolon will be output starting at column INT(P+1). P may be any expression whose value lies between 0 and 255.

7.11 Random numbers

Pseudo random numbers may be obtained by the use of the function RND. This function chooses a number at random between 0 and 1. This facility can be used in programs to form the basis of chance outcome in games, and to simulate randomness in scientific and business applications.

The function is RND(X), where X is a dummy number having any value. The value of X determines the starting point of the string of numbers generated, that is, RND(7) will generate a difference sequence from RND(3).

The random numbers generated will usually need to be manipulated. For example, to represent the throw of a die, integer values between 1 and 6 need to be randomly generated. This may be done with the following instruction:

```
100 LET T = INT(6*RND(3) + 1)
```

The +1 is required as otherwise the truncated integer would lie between 0 and 5.

When it is required to generate numbers to represent a sample from a uniform distribution a single statement similar to the above will be sufficient. In more advanced cases of simulation, it is often required to sample from a given frequency distribution. A subroutine suitable for these circumstances is described in Chapter 9.

7.12 User defined functions

You may define your own functions by using a DEF FN*x* statement, which has the following format:

line number DEF FN*x*(*variable*) = *expression*

Each user-defined function must have a unique name within the program as given by FNx, where x is a variable name.

Each function has a dummy argument given by the variable in brackets above. The actual argument used when the function is subsequently referenced in the program will be different from the dummy argument in the function definition. For example, the previous expression used to round a number can now be defined as a function as follows:

```
50 DEF FNR(A) = INT(A/F + 0.5)*F
```

This can be used subsequently in the same program to round a number to, say, the nearest 100 and to one decimal place as shown in Table 7.5.

Table 7.5 Program to round numbers

```
50 DEF FNR(A)=INT(A/F+0.5)*F
60 READ B,C
70 DATA 650,32.55,649,32.54,651,32.56,0,0
80 IF B = 0 THEN 190
100 REM ROUND B TO NEAREST 100
110 LET F=100
120 LET B1=FNR(B)
130 REM ROUND C TO 1 DECIMAL PLACE
140 LET F=0.1
150 LET C1=FNR(C)
160 PRINT "B =";B;"B1 =";B1,"C =";C;"C1 =";C1
170 PRINT
180 GOTO 60
190 END
```

7.13 Problems

Problem 1 – Radius of circumcircle

Write a program to find the radius of a circular track passing through points which form a triangle. The radius *(r)* of the circumcircle of a triangle is given by:

$$r = \frac{a}{2\sin A} = \frac{b}{2\sin B} = \frac{c}{2\sin C}$$

where a, b, c and A, B, C are the sides and angles of the triangle.

The program is listed in Table A6, and the answer for $a = 452$ metres, $b = 386$ metres and $c = 739$ metres is given in Appendix B. (*Note:* $\cos B = (a^2 + c^2 - b^2)/2ac$.)

Problem 2 – Volumes of solids

The volume of a solid of uniform cross-sectional area *(A)* and height *(H)* is given by:

$$V = A \times H$$

The uniform cross-sectional areas of some common solids are given in Table 7.6 together with their codes.

Write a program to calculate the volumes of the solids given in Table 7.6. All dimensions are in cm. The name of the solid is to be held in a DATA statement. Output the name of the solid and its volume. Code 0 can be used to terminate execution of the program. Suitable test data is given in Table 7.7, but include some extra data of your own.

Table 7.6 Some solids with uniform cross-sectional areas

Code	Solid	Cross-sectional area
1	cuboid	$L \times W$
2	cylinder	$\pi \times R^2$
3	hexagonal bar	$\frac{1}{2}\sqrt{27} \times D^2$

L = length W = width R = radius D = length of side

Table 7.7 Data for 'Volumes of solids' problem

Code	*1st dimension (L or R or D)*	*2nd dimension (W or zero)*	*Height (H)*	*Required no. of decimal places*
2	4.5	0	1.75	2
3	12.6	0	250	0
1	5.3	7.0	4.2	1

The program is listed in Table A7, and the answers are given in Appendix B. The program presented in Table A7 allows the user to round the answer using a scaling factor (F) to give the required number of decimal places. Note that the scaling factors for 2, 0 and 1 decimal places are .01, 1 and .1, respectively (see section 7.5).

8

Arrays

8.1 Lists and tables

So far single memory cells have been referenced by single variable names.

Many problems involve processing a number of variables in exactly the same way. In these programs, it is much more convenient to use the same name to reference a number of memory cells whose contents are processed by the same set of instructions in the program; a subscript is used in association with the variable name to identify uniquely each particular memory cell. For example, the program given in Table 8.1 inputs a list of N numbers and outputs a list of numbers that are greater than 10, and a list of numbers that are negative, using two passes through the stored data.

If N is equal to 9, then the list of 9 numbers is input into memory cells A(1), A(2), . . ., A(9), since in the FOR loop (lines 20–40) I takes the values 1–9. I is the subscript and A is the name of an array of nine elements. Each element of the array may be referenced by the array name and the subscript referring to its position in the array. That is, A(4) refers to the fourth number input into the array A, which is the memory cell between those occupied by A(3) and A(5).

Your BASIC system starts numbering the elements of an array at 0, that is, the first element of the array is referenced by A(0). The program given in Table 8.1 may be amended so that it can be used to input and process nine numbers starting at A(0) by changing the

Table 8.1 Program to output numbers >10 and negative numbers

```
10 INPUT N
20 FOR I=1 TO N
30 INPUT A(I)
40 NEXT I
50 PRINT"NUMBERS > 10"
60 PRINT
70 FOR I=1 TO N
80 IF A(I) <= 10 THEN 100
90 PRINT A(I)
100 NEXT I
110 PRINT
120 PRINT
130 PRINT"NEGATIVE NUMBERS"
135 PRINT
140 FOR I=1 TO N
150 IF A(I) >= 0 THEN 170
160 PRINT A(I)
170 NEXT I
180 PRINT
190 END
```

initial value of I to 0 in each FOR statement, lines 20, 70 and 140. N would need to be input as 8 instead of 9 in this case.

Try running the program given in Table 8.1 with the following nine numbers:

$$6, 12, -30, 10, -4, 47, 9, 0, 58$$

The output for this data is shown in Table 8.2.

Table 8.2 Output from program given in Table 8.1

```
NUMBERS > 10
  12
  47
  58
NEGATIVE NUMBERS
-30
-4
```

Array A, in the previous example, is called a one-dimensional array because it has one subscript. A one-dimensional array is a list, and a two-dimensional array is a table. A three-dimensional array is more difficult to visualise; an example would be to have the page number of a book as the third dimension (subscript), and the lines and columns on a page forming a table referenced by the other two

subscripts. The subscripts are separated by commas within the brackets following the name of the array, so that T$(3,2,8) could refer to the third line and second column on the eighth page of a book.

8.2 Naming arrays

Arrays used for holding numbers must be called by a variable name followed by the subscripts in brackets.

Array names which are identical to single variable names may be used in the same program. That is, the BASIC system will distinguish between A used as a single variable and A (subscript(s)) used as an array element for storing numbers, and A$ used as a single string variable and A$ (subscript(s)) used for storing character strings.

8.3 Subscripts

The subscripts that may be used with array names may consist of any expression. However, since the subscripts refer to unique positions in the array which is stored in memory cells in the computer, the individual subscripts must have positive values which the system will truncate to integer values (zero is a possible value for a subscript as explained previously).

The integer values of the subscripts must lie within the bounds of the array. For example, in the program given in Table 8.1 the BASIC system automatically allocates eleven memory cells (subscripts 0–10) in the absence of a DIM statement, which will be explained in the next section. If N were input as, say, 20, then elements referenced in the FOR statement beyond the A(10) element would be outside the defined storage of the array (i.e. outside the bounds of the array); in this case an execution error would occur.

8.4 The DIM statement

The DIM statement is used to define storage for arrays which have subscripts whose values are greater than ten. Although the DIM statements can appear anywhere in the program (before the array is accessed) it is better to place it at the beginning of the program so that it is separate from the main logic.

The format of the DIM statement is:

line number DIM list of array variables separated by commas

The array variables in the list may be ordinary or string variables; each variable name is followed by subscripts, separated by commas, in brackets.

For example, array X is to be used to store a list of up to fifty numbers, and array T$ is to be used to hold a table comprising a maximum of 5 rows and 7 columns.

The DIM statement to define storage for these two arrays is:

```
30 DIM X(50), T$(5,7)
```

X will have fifty-one memory cells of storage reserved, referenced by X(0), X(1), X(2), . . ., X(50). T$ will have a total of forty-eight cells reserved, the first cell being referenced by T$(0,0) and the last cell by T$(5,7).

Notice that storage is reserved for the *maximum* array size in each case. A particular run of your program may require less storage than the maximum; this is acceptable, or alternatively you can input values for the variable subscripts in the DIM statement before it is used (this is known as dynamic dimensioning). Note that the dimensioning may only be done *once* during the program run. For example, the DIM statement below requires K, L and M to be input at run time:

```
40 DIM X(K),T$(L,M)
```

More than one DIM statement may be used in a program, but the *same* array name may *not* appear in more than one DIM statement in a program. For example:

```
50 DIM B$(30,8),A$(60),A(20,20)
60 DIM D(100), C$(5,7,6)
```

correct

is correct.

```
70 DIM B$(30,8),A$(60),A(20,20)
80 DIM D(100),A(20,20),C$(5,7,6)
```

incorrect

will produce an error because A(20,20) appears in the DIM statements in line 70 *and* in line 80.

It is important to note that the DIM statement may be used to

override the automatic storage allocation for small arrays. For example, if array A is to contain a maximum of six cells and array B a maximum of four cells, then DIM A(5),B(3) will cause the *exact* storage required to be allocated, thus *saving* storage compared with the automatic allocation of eleven cells for each array.

8.5 Nested FOR loops

A FOR loop may lie wholly within another FOR loop, as was shown in Table 5.8 (page 36). This facility is particularly useful for manipulating arrays. For example, the data given in Table 8.3 is to be input into a two-dimensional array called A and output in the form shown in Table 8.4. The program given in Table 8.5 uses nested FOR loops to achieve this; try running this program.

Table 8.3 Input data for 'Nested FOR loops' program

1	2	3	4
5	6	7	8
9	10	11	12

Table 8.4 Table to be output

1	5	9
2	6	10
3	7	11
4	8	12

Table 8.5 Program using nested FOR loops

```
10 DIM A(3,4)
20 DATA 1,2,3,4,5,6,7,8,9,10,11,12
30 FOR I=1 TO 3
40 FOR J=1 TO 4
50 READ A(I,J)
60 NEXT J
70 NEXT I
80 FOR I=1 TO 4
90 FOR J=1 TO 3
100 PRINT A(J,I);
110 NEXT J
120 PRINT
130 NEXT I
140 END
```

8.6 Problems

Write programs for the following problems.

Problem 1 – Copying an array
Copy an array A comprising N elements into an array B, of the same size as A, in reverse order. For example, if N is 20, A(20) will go into B(1), A(19) into B(2), etc. Assume N is always a multiple of 5, and output array B in rows of 5 columns.

The program is listed in Table A8.

Problem 2 – Sum of elements
Sum the elements on the diagonals of an M × M array. Allow for M to be odd (as well as even) when the central element must be added in only *once*. Test your program in one run with an odd *and* an even value of M. Output the array and the sum of the elements on the diagonals in each case.

The program is listed in Table A9.

Problem 3 – Sorting a list of numbers
Sort a list of N numbers, held in array A, into ascending numerical order. Use only *one* array which is *just* large enough to hold the maximum number of numbers that may be input. The logic of the method is shown in Figure 8.1. This involves pushing the highest number to the end of the list by exchanging the higher number of each pair working through the list. That is, if element A(1) is greater than element A(2) then their contents are exchanged so that the higher value is in A(2); then the value in A(2) is compared with that in A(3) and exchanged if necessary. The second pass through the list is shorter since at the end of the first pass A(N) contained the highest value in the list and does not need to be compared again. If no exchanges take place during a pass (i.e. E = 0) then the list is in the required sorted order and no further passes are necessary.

Output the list of numbers in its original order and after each pass of the sort so that you can see how this sorting method works. Use the following data, and create your own data, to provide a variety of different lists to be sorted.

Data:

15, 3, 20, 22, 22, 9, 4, 23, 2, 0, −25

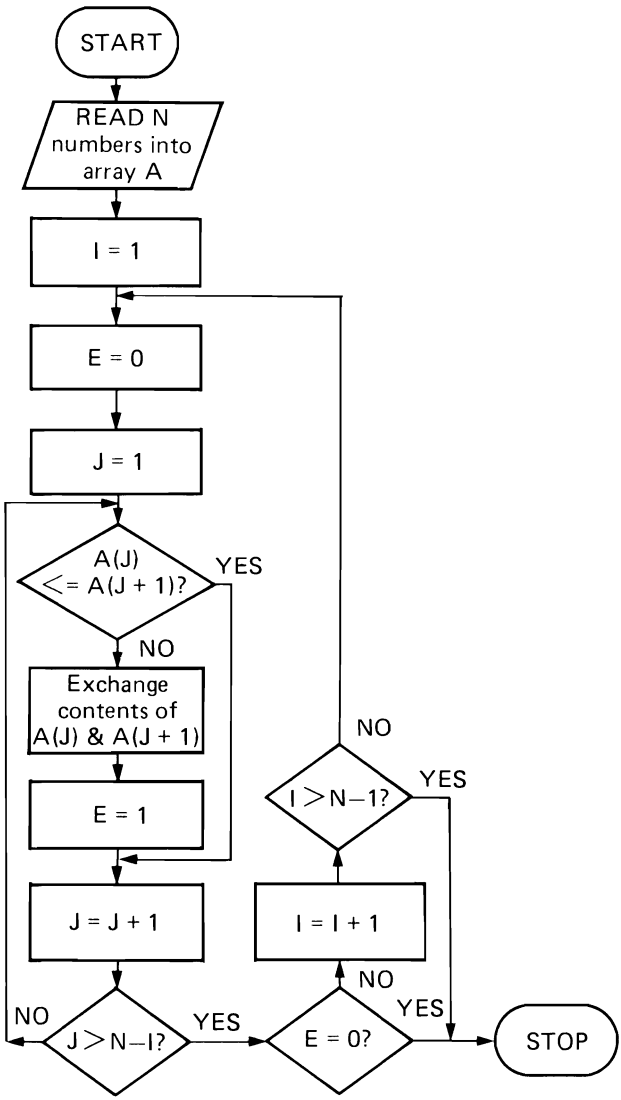


Figure 8.1 Sorting a list of numbers into ascending order

The program is listed in Table A10.

The changes in the order of the numbers in the list after each pass are shown in Table 8.6, starting with the list in its original order and finishing with the numbers sorted in ascending sequence.

Table 8.6 Sorting a list of numbers

15	3	20	22	22	9	4	23	2	0	−25
3	15	20	22	9	4	22	2	0	−25	23
3	15	20	9	4	22	2	0	−25	22	23
3	15	9	4	20	2	0	−25	22	22	23
3	9	4	15	2	0	−25	20	22	22	23
3	4	9	2	0	−25	15	20	22	22	23
3	4	2	0	−25	9	15	20	22	22	23
3	2	0	−25	4	9	15	20	22	22	23
2	0	−25	3	4	9	15	20	22	22	23
0	−25	2	3	4	9	15	20	22	22	23
−25	0	2	3	4	9	15	20	22	22	23

9

Subroutines

9.1 Purpose of subroutines

A subroutine is a sequence of instructions designed to perform one or more specific tasks. The routine may be required more than once in different places in the program. When a routine is written as a subroutine it is incorporated in the main program once. During execution the statement, GOSUB *linenumber*, causes control to pass to the line number specified. Execution continues until a RETURN statement is encountered. Control then passes back to the statement *following* the originating GOSUB statement.

A subroutine can be entered as many times as required and therefore can save the writing of similar instructions in several parts of the program. Apart from the extra program writing, the program usually becomes longer if subroutines are not used. A longer program requires more computer storage, and takes longer to translate into machine code; using subroutines wherever possible generally makes a program more efficient.

Once a subroutine has been developed and tested it may be used in quite different programs, either as it stands or with modifications. If possible, subroutines should be designed to allow them to be used in many different ways without modification. This may be done by building in flexibility.

9.2 Independent development

Another advantage of using subroutines is that they may be developed and tested independently from the program(s) in which they are to be used.

By testing subroutines independently a complex program may be built up more quickly using *proved* subroutines. In addition, if a subroutine has been developed for one program, then it can be tested with suitable test data for use in a different program *before* it is incorporated. However, the final program will need to be tested as a *whole* to ensure that the linkages – i.e. statements between the subroutines (as well as the subroutines) – give correct results for every branch of the program. The test data must be comprehensive enough to test *every* instruction in the program, as discussed in Chapter 4.

9.3 Graphs and histograms

If you use a computer to analyse data it is almost certain that at some time you will want to plot the data, or maybe group it into a frequency table. The following sections describe a series of subroutines that allow you to do this. To allow the subroutines to be compatible we need to standardise some of the variable names. The routines have been written to allow up to 100 data values to be processed. These values will be held in the array V. There is therefore the need for a DIM V(100) in the main program. If the data is to be grouped into a frequency table before, say, printing out a histogram, the variable will be stored in array X and the frequency in array F. A frequency table having a maximum of fifteen class intervals should be adequate for most purposes. Therefore the main program will need a DIM statement containing X(15), F(15).

9.4 Serial plotting routine

A routine to plot a series of values sequentially is useful for time series based data. The examination of the graph might confirm that it is not worthwhile using more elaborate analyses to seek non-existent trends, or seasonal patterns.

Table 9.1 Serial plotting routine

```
1010 PRINT
1020 PRINT"NO OF DATA POINTS =";N
1030 PRINT
1040 INPUT"ENTER NO OF POINTS TO PLOT";Q
1060 IF Q>N THEN 1010
1070 LET B=1
1080 IF Q=N THEN 1110
1085 PRINT
1090 INPUT"START PLOT AT DATA POINT NO";B
1105 LET Q = Q + B - 1
1110 LET M=0
1120 FOR I=B TO Q
1130 IF V(I)<M THEN 1150
1140 LET M=V(I)
1150 NEXT I
1160 LET S=1+INT(M/30)
1170 PRINT
1180 PRINT"ONE PLOT POSITION =";S;"UNITS"
1190 PRINT
1200 PRINT" N  DATA";
1210 PRINT TAB(9);"0";TAB(17);10*S;TAB(27);20*S
1220 PRINT TAB(9);"I....:....I....:....I....:...."
1230 FOR I = B TO Q
1240 LET K=INT(V(I)/S+.5)
1250 IF K>0 THEN 1280
1260 PRINT I;TAB(5);V(I);TAB(9);"*"
1270 GOTO 1290
1280 PRINT I;TAB(5);V(I);TAB(9);"I";TAB(K+9);"*"
1290 NEXT I
1300 PRINT TAB(9);"I....:....I....:....I....:...."
1310 PRINT
1320 RETURN
```

A plotting routine is given in Table 9.1. The largest value to be plotted (M) is found (lines 1110–1150) and from this the scaling factor is calculated. Line 1160 has been written to fit the maximum value within thirty print positions. The first ten print positions are taken up by the data point number (N) and the actual value V(I).

Problem 1 – Plot of percentage pastureland

Write a program using the above routine to plot the reducing percentage pastureland, as given in Table 9.2. The program is given in Table A11, and the output in Table 9.3.

Table 9.2 Pastureland in a parish

Year	*% Pastureland*
1	82
2	72
3	63
4	60
5	57
6	54
7	50
8	45
9	38
10	35

Table 9.3 Percentage pastureland output

```
ENTER NO OF YEARS? 10

% PASTURELAND, YR 1 ? 82
% PASTURELAND, YR 2 ? 72
.
.
.
% PASTURELAND, YR 10 ? 35

NO OF DATA POINTS = 10

ENTER NO OF POINTS TO PLOT? 10

ONE PLOT POSITION = 3 UNITS

 N   DATA 0         30         60
          I....:....I....:....I....:....
 1     82 I                          *
 2     72 I                       *
 3     63 I                    *
 4     60 I                   *
 5     57 I                  *
 6     54 I                 *
 7     50 I                *
 8     45 I              *
 9     38 I             *
 10    35 I            *
          I....:....I....:....I....:....
```

9.5 Frequency grouping subroutines

Before producing a histogram, or carrying out other forms of analysis, it is often required to group individual data points into class intervals and note the total number of values falling into each interval (i.e. the frequency).

A subroutine to do this is given in Table 9.4. The frequency table so constructed is composed of fifteen class intervals. Any data not included as a result of this constraint is printed out by line 2100. A new run can then be undertaken with the class interval parameters respecified accordingly.

Table 9.4 Frequency grouping routine

```
2000 INPUT"ENTER SIZE OF CLASS INTERVAL";C
2015 PRINT
2020 INPUT"ENTER LOWER BOUND OF 1ST. INTERVAL";L
2040 FOR I=1 TO N
2050 FOR J=1 TO 15
2060 IF V(I) >= (L+(C*J)) THEN 2090
2070 LET F(J)=F(J)+1
2080 GOTO 2110
2090 NEXT J
2100 PRINT V(I);"NOT COUNTED"
2110 NEXT I
2120 FOR J=1 TO 15
2130 LET X(J)=L+((J-.5)*C)
2140 NEXT J
2150 RETURN
```

The reason for designing the program in this manner is that a completely automatic parameter setting routine may disguise the presence of a 'rogue' value which, once pointed out, you are happy to ignore.

A subroutine to print out a frequency table is given in Table 9.5. As this subroutine is intended to be independent of the grouping subroutine, the class interval and lower bounds are calculated from the array values of X.

The third subroutine in this set outputs the frequency table in histogram form. The program is given in Table 9.6. As with the plotting routine it has been written to use thirty print positions for the histogram.

Table 9.5 Frequency table routine

```
3000 PRINT
3010 LET C = X(2) - X(1)
3020 LET L = X(1) - (.5*C)
3030 PRINT"------------------"
3040 PRINT TAB(4);"X";TAB(14);"F"
3050 PRINT"------------------"
3060 FOR I=1 TO 15
3070 LET B = L + C*(I-1)
3080 PRINT B;TAB(4);"-";TAB(14);F(I)
3090 NEXT I
3100 PRINT"------------------"
3110 RETURN
```

Table 9.6 Histogram routine

```
4000 LET M = F(1)
4010 PRINT
4020 PRINT
4210 FOR I = 2 TO N
4220 IF F(I) < M THEN 4240
4230 LET M = F(I)
4240 NEXT I
4250 IF M > 30 THEN 4300
4260 LET U = INT(30/M)
4265 LET S = 1/U
4285 PRINT U;" STARS = 1 UNIT"
4290 GOTO 4330
4300 LET S = INT(M/30)+1
4320 PRINT "ONE STAR = ";S;" UNITS"
4330 PRINT
4350 PRINT"CLASS";
4351 PRINT TAB(7);"I....:....I....:....I....:....I"
4400 FOR I = 1 TO 15
4410 LET K = INT((F(I)/S)+.5)
4412 IF K > 0 THEN 4420
4414 PRINT I;TAB(7);"I"
4416 GOTO 4470
4420 PRINT I;TAB(7);"I";
4430 FOR J = 1 TO K
4440 PRINT "*";
4450 NEXT J
4460 PRINT
4470 NEXT I
4475 PRINT TAB(7);"I....:....I....:....I....:....I"
4480 PRINT
4490 RETURN
```

Problem 2 – Pastureland histogram

Write a program incorporating these subroutines to process the data shown in Table 9.7. The output required is a frequency table and histogram of the percentage pastureland.

Table 9.7 Parish data

Parish	*% amount of pastureland*
1	46
2	47
3	63
4	74
5	76
6	26
7	37
8	39
9	35
10	43
11	52
12	59

The program, which gives the output shown in Tables 9.8 and 9.9, is listed in Table A12.

Table 9.8 Frequency table for Problem 2

X	F
20 –	1
30 –	3
40 –	3
50 –	2
60 –	1
70 –	2
80 –	0
90 –	0
100 –	0
110 –	0
120 –	0
130 –	0
140 –	0
150 –	0
160 –	0

Table 9.9 Histogram for Problem 2

```
     10 STARS = 1 UNIT
CLASS     |....:....|....:....|....:....|
  1       |**********
  2       |******************************
  3       |******************************
  4       |********************
  5       |**********
  6       |********************
  7       |
  8       |
  9       |
 10       |
 11       |
 12       |
 13       |
 14       |
 15       |
          |....:....|....:....|....:....|
```

9.6 Sampling from a frequency distribution

A subroutine is described below that allows a value to be sampled from a frequency distribution. The frequency distribution is contained in the two-dimensional array X. The first dimension contains the variable value, the second dimension contains the cumulative percentage frequency.

To allow the subroutine to be used generally in a variety of programs some standardisation of the array containing the frequency distribution is necessary. The number of class intervals has been set at 10 resulting in the dimensions for the array X being (10,2). Note that, for convenience, the existence of 0 subscripts have been ignored. If a required distribution contains less than ten rows (i.e. class intervals), the final entries in the array will be identical. For example, the data to be sampled, shown in Table 9.10, would be contained in the array X(R,I) as shown in Table 9.11.

Table 9.10 Data to be sampled

Variable	*Cumulative % frequency*
5	10
10	27
15	42
20	65
25	80
30	100

Table 9.11 Contents of X(R,I)

		Column subscript, I	
		(,1)	*(,2)*
	(1,)	5	10
	(2,)	10	27
	(3,)	15	42
	(4,)	20	65
	(5,)	25	80
Row subscript, R	(6,)	30	100
	(7,)	30	100
	(8,)	30	100
	(9,)	30	100
	(10,)	30	100

Any distributions to be used from a main program are established in a similar (10,2) format and array X can be equated to them before entering the subroutine.

9.7 Description of subroutine

A flowchart for the subroutine is shown in Figure 9.1 and the listing is given in Table 9.12.

Table 9.12 Sampling routine

```
900 REM SAMPLING SUB
910 LET Z=100*RND(3)
920 FOR R=1 TO 10
930 LET V=X(R,1)
940 IF Z<=X(R,2) THEN 980
950 NEXT R
960 PRINT"ERROR:RN NOT PROPERLY ALLOCATED"
970 END
980 RETURN
```

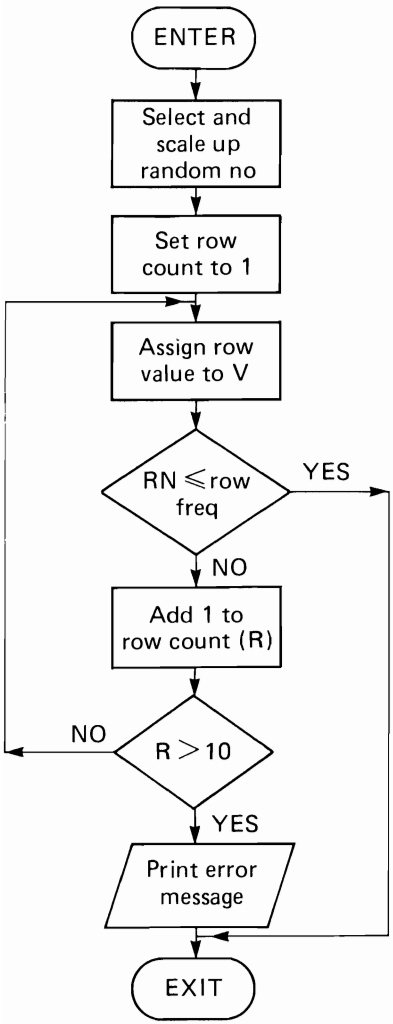


Figure 9.1 Sampling flowchart

The random number generated is scaled to lie between 0 and 100 (line 910). Within the FOR loop the array X is inspected row by row. The value of the current row variable is assigned to V (line 930) and the value of the scaled random number Z is compared with the current cumulative frequency (line 940). If Z is greater than the frequency the process is repeated for the next row (line 950). When, eventually, the random value Z falls within the current class interval the subroutine is left, carrying back the current value of the variable V. If, due to errors in setting up the distribution, the random value Z cannot be associated with any particular row then lines 960 and 970 are encountered, giving rise to the error message.

A simulation program using this subroutine is given in Chapter 12, section 12.5.

Problem 3 – Input subroutine

Write a subroutine to allow the details of Table 9.10 to be entered into a two-dimensional array D. Make provision for up to ten rows to be entered.

A subroutine to meet the above requirements is shown in Table A13.

9.8 Protected INPUT routine

Input statements can cause the inexperienced user problems. If the return key is pressed in response to an input statement without entering 'data' the 64 accepts a null return and continues through the program, usually with unintended results. The routine to be described (Table 9.13) will not accept solely the return key, thereby preventing 'null' inputs.

This subroutine makes use of the GET statement which accepts a single character from the keyboard without waiting for the return character. The GET statement allows any entry from the keyboard to be examined by the 64. You can use this technique, together with the conditional branch statements given in Chapter 5, for writing programs that allow the user to choose a course of action from a 'menu' of choices. Because the GET statement will also accept a 'null' response the user does not have time to respond unless a loop is built into the program. A common method is:

```
200 GET A$: IF A$ = "" THEN 200
```

Table 9.13 Protected input routine

```
5000 REM PROTECTED INPUT
5010 ZI$="":ZJ$=""
5020 PRINT"■";:FOR D=1 TO 300:NEXT D
5030 PRINT" ";:FOR D=1 TO 300:NEXT D
5040 GET ZI$:IF ZI$="" THEN 5020
5050 IF ZI$<>CHR$(20) THEN 5080
5055 ZL=LEN(ZJ$):IF ZL<1 THEN 5020
5060 ZJ$=LEFT$(ZJ$,ZL-1)
5070 PRINT ZI$;:GOTO 5020
5080 IF ZI$=CHR$(13) THEN 5120
5090 PRINT ZI$;
5100 ZJ$=ZJ$+ZI$
5110 GOTO 5020
5120 IF ZJ$="" THEN 5020
5130 PRINT
5140 RETURN
```

Line 200 forms a closed loop which is only broken by a key being pressed. While the loop in line 200 is being executed no cursor will be displayed.

The principle of the subroutine is that a series of characters keyed in response to a GET loop are concatenated into a string to give the equivalent of the response to an INPUT statement. Line 5010 sets the GET variable ZI$ and the concatenated variable ZJ$ to null strings. A flashing cursor is simulated in lines 5020 and 5030. Lines 5020 to 5040 form therefore a more elaborate GET loop (i.e. 5020 – cursor on; 5030 – cursor off; 5040 – GET character, else cursor on, etc.). When a character is entered this loop is left and the character tested.

The first test (line 5050) is to check for the delete key (character code 20). If the delete key has been pressed, the length of the current string ZJ$ is calculated (line 5055). If ZJ$ is reduced to a null string the entry loop is re-entered (i.e. control is passed to line 5020). Providing ZJ$ does contain characters, line 5060 removes the rightmost character. Line 5070 'prints' the delete character, i.e. causes the cursor to move left and then the GET loop is re-entered.

The second test (line 5080) checks for the pressing of the return key (character 13). Providing the return key has *not* been pressed, the acceptable character is printed and concatenated to ZJ$ in lines 5090 and 5100 before 5110 returns control to the GET loop.

When the return key is pressed, line 5080 causes execution to pass to line 5120 where the status of ZJ$ is checked. If ZJ$ is still a null

string then depression of the return key is not acceptable and the GET loop is re-entered. When ZJ$ does contain a response, depression of the return key is 'accepted' and the final PRINT is executed to terminate the action of the trailing semicolons in previous print statements.

10

Sprite Graphics and Customised Characters

10.1 Introduction to sprites

Sprites are graphics characters designed by the user within a block of 24 dots across by 21 rows deep. Eight sprites can be displayed simultaneously on the screen and moved around as required. For this reason sprites are sometimes referred to as MOBS (mobile object blocks). The colour of a sprite can be changed when required and collisions between sprites, or between sprites and 'text', can be detected within a program. When sprites meet other sprites or 'text', priorities set within the program determine which pass in front. Finally, any sprite can be expanded to twice its size on the screen in the horizontal (X axis) and/or vertical (Y axis) direction.

This chapter contains simple programs that demonstrate each of the above features. Each program has been developed to illustrate just one or two of the features at a time as there are, in all, over thirty locations that could be POKEd! Appendix H summarises the appropriate locations (also called registers) according to the sprite number (0 to 7) and the feature to be controlled. The locations in the Appendix are offsets from the memory location 53248. That is, the required memory location is found by adding the offset to 53248. In practice, this is done within the program as demonstrated by the examples in this chapter.

Several of the locations are common for all eight sprites. The intended sprite is indicated by setting the appropriate bit within the byte. It is therefore necessary to understand how bits can be set (i.e.

changed from zero to one) by POKEing appropriate decimal values to the memory location. These preliminary aspects to the use of sprites are discussed in the next section.

10.2 Setting bit patterns by POKEing

A memory location (or register) consists of eight bits (0s or 1s) which together constitute one byte (see Figure 10.1). The bit positions are numbered from *right to left*, starting at 0 and ending at 7. Each bit position can contain a 0 or 1, and when it contains a 1 that bit is said to be 'set' or 'on'. Each bit position represents a decimal value, as shown in Figure 10.1. This figure illustrates that the bit pattern 01011001 is the equivalent of the decimal value 89. The maximum decimal value that can be represented is 255 (i.e. 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1). The decimal value is sometimes called the denary value, as decimal parts of a number do not occur.

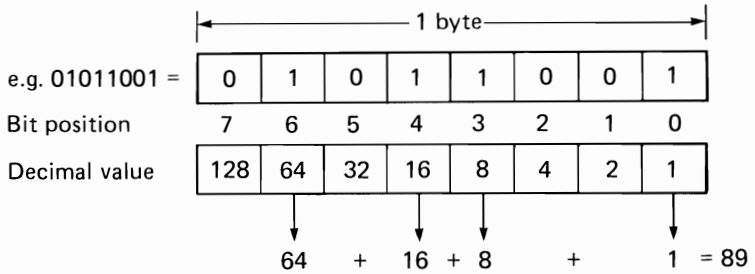


Figure 10.1 Decimal (denary) values of bit positions showing the denary equivalent of 01011001

Referring to Figure 10.1, to set bits 0,3,4 and 6 the value 89 would be POKEd to the required register. Thus

```
POKE 53248+21, 89
```

would turn on sprites 0,3,4, and 6 (see Appendix H).

On some occasions it is necessary to convert the denary number found in a register to its bit equivalent to interpret its meaning. For example, having POKEd the above register if it was subsequently PEEKed by:

```
PRINT PEEK(53248+21)
```

the value 89 would be displayed. The method of converting 89 to its binary equivalent is shown in Table 10.1. The value is successively divided by 128, 64, 32, etc., the remainder being carried forward.

Table 10.1 Converting a denary number to binary

$\frac{89}{128}$	= 0 remainder	89
$\frac{89}{64}$	= 1	25
$\frac{25}{32}$	= 0	25
$\frac{25}{16}$	= 1	9
$\frac{9}{8}$	= 1	1
$\frac{1}{4}$	= 0	1
$\frac{1}{2}$	= 0	1
$\frac{1}{1}$	= 1	

i.e. denary 89 = 0 1 0 1 1 0 0 1 binary

During the course of a program it may be necessary to set a particular bit in a register in addition to any bits already set. Although the register can be PEEKed and the result worked out, this is inconvenient during the running of a program. The use of the logical operators AND and OR allow the required action to be built into the program:

to set the Nth bit of a register R to 1, without changing any other bit,

```
POKE R, PEEK(R) OR (2 ↑ N)
```

to set the Nth bit of register R to 0, without changing any other bit,

```
POKE R, PEEK(R) AND (255 – (2 ↑ N))
```

The above two expressions also illustrate how the denary value of a bit can be calculated from the bit position (e.g. 128 = 2 ↑ 7, 64 = 2 ↑ 6, etc.).

Note on AND and OR

The bit pattern in a memory location can be changed by ANDing or ORing it with a number or another memory location. The *bit pattern* of the memory location to be changed is matched against the second memory location or number. Using AND will change '1' bits to '0' if the matching bit is *not* '1'. Using OR will change '0' bits to '1' in *every* position where there is a matching bit set to '1'.

10.3 Turning sprites on and off

Although the object of the program shown in Table 10.2 is quite simple – to turn a sprite on and off at a predetermined screen location – some fundamental sprite programming methods are used that are required in virtually every sprite program.

Table 10.2 Turning a sprite on and off

```
10 SM=53248
40 POKE 2040,13
50 FOR N=0 TO 62
52 POKE 832+N,255
54 NEXT N
60 POKE SM+0,160
62 POKE SM+1,120
70 POKE SM+21,1
75 FOR I=1 TO 1000:NEXT I
80 POKE SM+21,0
85 FOR I=1 TO 1000:NEXT I
90 GOTO 70
100 END
```

Firstly, in line 10, the variable SM is set to the start of sprite memory (53248). This allows all other sprite locations to be referenced by adding the appropriate offset to SM.

The sprite to be displayed needs to be defined as 63 data values. As one line of a sprite is 24 dots wide, the bit pattern takes up 3 bytes ($3 \times 8 = 24$) and therefore can be specified by three equivalent denary values. The 21 rows of a sprite therefore requires $21 \times 3 = 63$ data values in total to be specified. These data values need to be stored in memory and their location specified within the program.

There are eight special locations, 2040 to 2047, that are used to point to the location of the data values. Location 2040 is used to specify the source of sprite 0's data values, 2041 is used for sprite 1, etc., up to 2047 for sprite 7. Line 40 is therefore specifying that the

data for sprite 0 will lie in the 13th block of memory. (A block of memory being defined as 64 bytes.) The data will therefore be found starting at memory location 13 × 64 = 832.

As the memory is used by the 64 for many other aspects of operation and execution, only certain parts of the memory may be used to hold sprite data. The 'first' choice would be blocks 13 to 15 inclusive, as these are only used when the cassette unit is actually operating. Blocks 16 to 31 should *not* normally be used, as these are where the screen memory is located. However, blocks 32 to 63 could be used after some memory adjustment. Blocks 32 to 63 normally contain the current BASIC program, but it can be moved by using the following POKEs *before* a BASIC program is loaded:

```
POKE 8192,0
POKE 44,32
POKE 43,1
```

These POKEs reposition BASIC to start at 8193 instead of the normal 1025.

Having specified that the data for sprite 0 will be stored in block 13 (line 40), the FOR . . . NEXT loop over lines 50 to 54 POKEs values into the 63 memory locations starting at location 832 (i.e. 13 × 64).

To keep the example as simple as possible, the value POKEd into all locations is 255. A byte value of 255 means all the bits have been set. Therefore, in this case, the effect is to 'switch on' every dot in every byte to show the sprite as a solid block.

Line 60 specifies the horizontal position of the sprite as being 160. The screen is 320 dots wide, starting at position 24 going across to position 343. Line 62 specifies the vertical position of the sprite as being 120. The screen is 200 dots high, starting at position 50 at the top and going down to position 249. The position of the sprite is related to its top left-hand corner. For example, changing lines 60 and 62 to:

```
60 POKE SM+0,24
62 POKE SM+1,50
```

will display the sprite, in full, at the top left-hand corner of the screen. To display the sprite at the bottom left-hand corner use:

```
60 POKE SM+0,24
62 POKE SM+0,229
```

because 249 + 1 − 21 = 229.

As the maximum value that can be POKEd to any memory location is 255, the positioning of the sprite across the screen beyond position 255 needs further programming. This aspect is dealt with in a later demonstration program.

Line 70 turns the sprite on; line 75 creates a delay before line 80 turns the sprite off. After a further delay in line 85, the jump in line 90 back to 70 causes the sprite to turn on and off repeatedly.

Note that, in this program, the POKE location in line 40 and the sprite memory offsets in lines 60, 62, 70 and 80 all relate to sprite number 0.

10.4 Changing a sprite character

In the previous program, all the dots in sprite 0 were switched on. To demonstrate how the data values POKEd into the selected block of memory determine which dots are 'switched on', the program shown in Table 10.3 changes the values while the sprite is displayed on the screen.

Table 10.3 Changing a sprite character

```
10 SM=53248
40 POKE 2040,13
50 FOR N=0 TO 62
52 POKE 832+N,255
54 NEXT N
60 POKE SM+0,160
62 POKE SM+1,120
70 POKE SM+21,1
80 FOR B=1 TO 255
81 PRINT"■";B
82 FOR N=0 TO 62
84 POKE 832+N,B
86 NEXT N
87 POKE SM+21,1
90 NEXT B
100 END
```

The program is identical to the previous one over lines 40 to 70. Having displayed the full sprite, lines 80 to 90 form a loop that progressively displays the effects of POKEing all values from 1 to 255 into the 63 data positions in the 13th block of memory.

Line 80 sets the value of B and line 81 displays the current value of B in the top left-hand corner of the screen. Lines 82 to 86 POKE this value of B into each of the 63 data positions. After they are POKEd, the sprite displayed on the screen will change.

The sprite will appear to consist of three identical columns. This is because, in this example, each row of the sprite will be made up from three identical bytes. When sprite 0 is fully on, it appears as a white block. As the POKEd value of B changes, the sprite may appear to be multicoloured. The extent of the colour fringing experienced will depend upon the match between the 64's modulator and the television set.

10.5 Expanding the size of a sprite

The program shown in Table 10.4 demonstrates the facilities for expanding a sprite along either axis. In this program, two sprites will be displayed.

Table 10.4 Expanding the size of sprites

```
10 SM=53248
40 POKE 2040,13
46 POKE 2041,13
50 FOR N=0 TO 62
52 POKE 832+N,255
54 NEXT N
60 REM SPRITE 0 POSITION
62 POKE SM+0,160
64 POKE SM+1,120
66 REM SPRITE 1 POSITION
67 POKE SM+2,80
68 POKE SM+3,200
70 POKE SM+21,3
72 FOR I=1 TO 500:NEXT I
74 REM CHANGE SIZES OF SPRITES
80 POKE SM+29,1
82 FOR I=1 TO 500:NEXT I
84 POKE SM+23,2
86 FOR I=1 TO 500:NEXT I
88 POKE SM+29,3
90 FOR I=1 TO 500:NEXT I
92 POKE SM+23,3
94 FOR I=1 TO 500:NEXT I
96 POKE SM+29,2
98 FOR I=1 TO 500:NEXT I
99 POKE SM+23,0
100 END
```

The start of the program is similar to the previous examples. In addition to line 40 stipulating that sprite 0 will obtain its data from the 13th block, line 46 specifies that sprite 1 will use the same data block. Lines 50 to 54 set up the data values in the 13th block and lines 62 and 64 position sprite 0 while lines 67 and 68 position sprite 1.

Line 70 turns sprites 0 *and* 1 on by setting bits 0 and 1 (denary 1 + 2). The following changes are then made:

```
80 POKE SM+29,1   Expand sprite 0 on X axis
84 POKE SM+23,2   Expand sprite 1 on Y axis
```

Note that the action of line 80 remains: sprite 0 remains expanded.

```
88 POKE SM+29,3   Expand both sprites on X axis
```

As sprite 0 is already expanded, the visual effect is to expand sprite 1 on the X axis.

```
92 POKE SM+23,3   Expand both sprites on Y axis as well
96 POKE SM+29,2   Expand sprite 1 only on X axis
```

There is a delay loop between each of the above lines to allow time for the effect to be viewed.

Note that sprite 1 is displayed in red. This is sprite 1's default colour; it can be changed by POKEing offset 40, as is demonstrated in the next program.

Try listing the program, then running it. Notice that the sprites are displayed 'on top' of the text. Also note that if the clear screen key is used, the program listing is cleared but the sprites remain. To clear the sprites the STOP and RESTORE keys should be used.

10.6 Moving sprites around

The program in Table 10.5 illustrates how sprites can be moved across the full width of the screen, and how priorities can be changed so that a sprite passes under instead of over text.

Moving a sprite across the full width of the screen involves the use of a further sprite register (the msb X register, offset 16 in Appendix H).

When the X position is POKEd, for example with sprite 0 by POKE SM+0,X, the highest acceptable value for X is 255 (i.e. a byte, full of 1s). To move beyond the 255 co-ordinate on the X axis,

Table 10.5 Moving sprites around

```
6 DEF FNR(Z)=PEEK(SM+16) OR Z
7 DEF FNL(Z)=PEEK(SM+16) AND (255-Z)
10 SM=53248
40 POKE 2040,13
46 POKE 2041,13
50 FOR N=0 TO 62
52 POKE 832+N,255
54 NEXT N
60 POKE SM+21,3
70 FOR X=0 TO 341
71 X0=X:X1=X
72 Y=INT(255*X/341)
73 REM TEST X POSITION OF SPRITE 0
74 IF X0<=255 THEN POKE SM+16,FNL(1)
75 IF X0>255 THEN X0=X-255:POKE SM+16,FNR(1)
76 REM RE-POSITION SPRITE 0
77 POKE SM+0,X0
78 POKE SM+1,Y
80 REM SPRITE 1 UNDER TEXT AT X > 100
81 IF X>100 THEN POKE SM+27,2
83 REM TEST X POSITION OF SPRITE 1
84 IF X1<=255 THEN POKE SM+16,FNL(2)
85 IF X1>255 THEN X1=X-255:POKE SM+16,FNR(2)
86 REM RE-POSITION SPRITE 1
87 POKE SM+2,X1
88 POKE SM+3,160
90 NEXT X
100 END
```

the appropriate sprite bit has to be set in offset 16; the co-ordinates then continue from zero. In setting the bit in offset 16, the rest of the contents of the offset remain unchanged. It is convenient to create a user-defined function for this purpose, as follows:

DEF FNR(Z)=PEEK(SM+16) OR Z

where Z = the bit value to be set.

To set a bit value of 1 (for sprite 0) in offset 16, the syntax would be:

POKE(SM+16),FNR(1)

To set a bit value of 2 (for sprite 1):

POKE(SM+16),FNR(2)

When a sprite moves from the right-hand side of the screen past the 'boundary' at 255, the appropriate bit in offset 16 has to be

switched off. A corresponding user-defined function for switching off a specified bit without affecting the rest of the byte is:

```
DEF FNL(Z)=PEEK(SM+16) and (255–Z)
```

These two user-defined functions are defined in lines 6 and 7 of the program. Line 10 sets SM to the start of sprite memory, and lines 40 and 46 specify that sprite 0 and sprite 1 will both be defined by the data values in block 13. The loop (lines 50–54) sets up the data values in the 13th block. Line 60 turns on both sprites (bits 0 and 1).

The movement of both sprites across the screen is carried out within the loop, line 70 to line 90, where X (the horizontal position) is progressively increased by 1 from 0 to 341. Line 71 transfers the X value into X0 (for sprite 0) and X1 (for sprite 1) so that either sprite can be tested for position independently.

Sprite 0 is required to move downwards and to the right across the screen, while sprite 1 moves horizontally. Line 72 converts the current value of X into a proportionate vertical position (Y).

Lines 74 to 78 test and position sprite 0. If X is less than or equal to 255, then line 74 uses the user-defined function that ensures bit 0 is *not set* to '1' in offset 16. If X is greater than 255, then line 75 ensures that bit 0 is set to '1'. Lines 77 and 78 then move sprite 0 to the new position.

Line 81 tests the intended position of sprite 1; if X is greater than 100, then bit 1 is set to '1' in offset 27 to give text priority over the sprite.

Lines 84 and 85 set bit 1 in offset 16 according to the X axis position of sprite 1 and lines 87 and 88 position the sprite.

To see the effect of giving text priority over sprite 1, list the program before running it so that the sprites move over the program listing on the screen. Note that if this program is run immediately after running the program in Table 10.4, sprite 1 will still be expanded on its X axis. The latest settings of the sprite offsets persist until the STOP and RESTORE keys are pressed. Thus if the program in Table 10.5 is run twice in succession, on the second occasion sprite 1 will start off already 'under' the text.

10.7 Changing sprite colours and detecting collisions

The colour of a sprite can be set by POKEing the appropriate offsets 39 to 46 for sprites 0 to 7 respectively. The value to be POKEd is that given in Appendix D (e.g. 5 for green). If the following line is added to the previous program (Table 10.5), then sprite 1 will change from red to green when the X axis position value reaches 150:

```
82 IF X=150 THEN POKE(SM+40),5
```

(As the setting persists, X=150 can be used rather than X>=150.)

Collisions between sprites are detected by PEEKing offset 30. When a sprite collides, its bit is set in the offset. To detect whether any collisions have occurred, add the following line to the program:

```
89 IF PEEK(SM+30) > 0 THEN PRINT X
```

When a collision occurs the X axis position value will be printed on the screen. On running the program, a zero will be printed before the sprites appear, because when they are first turned on (line 60) they are in the same offscreen position. As the two sprites move across the screen, 188 will be the first value printed and then each value of X will be successively printed until the two sprites separate (at X = 243).

In this particular case, as there are no other sprites being used, a collision specifically between sprite 0 and sprite 1 can be detected by testing to see if bits 0 *and* 1 have been set, i.e.

```
89 IF PEEK(SM+30)=3 THEN PRINT X
```

10.8 Sprites designed from DATA statements

In all the demonstration programs used in this chapter a simple loop has been used to POKE identical values to every sprite byte. Customised sprite shapes can be displayed by reading from a calculated set of DATA values.

Figure 10.2 shows the 63 DATA values necessary to display an umbrella design. To use the umbrella, modify the program given in Table 10.5 by adding lines 82 and 89 (from section 10.7) and make the following changes:

1 Change line 52 to read DATA statements, i.e.

```
52 READ D:POKE 832+N,D
```

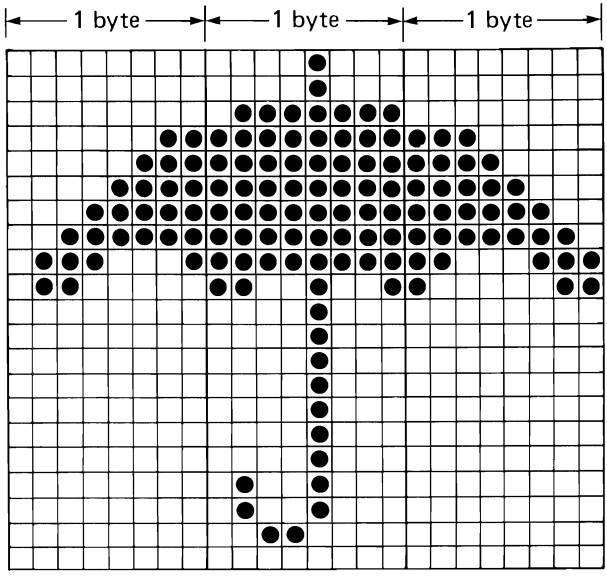


Figure 10.2 Sprite design for umbrella

2 Add the following DATA statements:

```
200 DATA 0,8,0,0,8,0,0,127,0
202 DATA 3,255,224,7,255,240,15,255,248
204 DATA 31,255,252,63,255,254,113,255,199
206 DATA 96,201,131,0,8,0,0,8,0
208 DATA 0,8,0,0,8,0,0,8,0
210 DATA 0,8,0,0,8,0,0,72,0
212 DATA 0,72,0,0,48,0,0,0,0
```

In creating DATA statements for sprite designs, it is convenient to have complete rows of a sprite on one DATA line.

A final modification to Table 10.5 is to expand sprite 0 in both directions, i.e. to add

```
56 POKE SM+23,1:POKE SM+29,1
```

When the above modifications have been made to Table 10.5, list the program and RUN. On this occasion, a large white umbrella should 'float' down and across the screen while a smaller red umbrella moves across. If all the modifications are incorporated

(i.e. lines 52, 56, 82, 89, 200–212), then the red umbrella starts above the program listing and then goes 'under' it. A little later the red umbrella changes to green and, while both umbrellas touch, the X axis position values are displayed on the screen. Note that, once run, sprite 1 will continue to be displayed in green.

10.9 Multicolour sprites

In addition to specifying an individual sprite colour, it is possible to specify two other common colours that can be used with any sprite. For example, if the sprite 0 colour is white and the sprite 1 colour is green and then red and yellow are specified for multicolour use, sprite 0 can be designed in white, red and yellow (and the current background, i.e. screen, colour) and sprite 1 can be designed in green, red, yellow and the background colour.

Sprites are specified as multicolour by setting their appropriate bit in register SM+28. For example:

```
POKE SM+28,3
```

will specify both sprites 0 and 1 as being multicolour.

The two additional common colours that can be used, called multicolour 0 and multicolour 1, are specified in registers SM+37, SM+38 respectively. For example:

```
POKE SM+37,2
POKE SM+38,7
```

sets multicolour 0 to red and multicolour 1 to yellow.

The four possible colours to be displayed are specified in the sprite design by 'bit pairs', i.e.

00	transparent (i.e. screen colour)
01	multicolour 0
10	sprite's individual colour
11	multicolour 1

Since two bits of data are required for each colour, the resolution of a line of a sprite is reduced from twenty-four to twelve, each dot being expanded to twice its horizontal size so that the overall size of the sprite does not change.

A multicoloured sprite line therefore still consists of a 24-bit

pattern (12 × 2) which is coded into three 8-bit denary values as before. For example, if the following line of twelve coloured double dots was required:

YYYYGGGGRRRR

where

Y = yellow, multicolour 1
G = green, sprite's individual colour
R = red, multicolour 0

then the bit pair pattern would be:

11 11 11 11 10 10 10 10 01 01 01 01

and the three denary values would be:

255, 170, 85

A multicoloured sprite mosaics example

The program described in section 10.8 can be changed to a multicolour sprite program by the addition of a short subroutine and a re-specification of the DATA values. Line 58 needs to be changed so as to call the new subroutine:

```
58 GOSUB 300
```

The subroutine is shown in Table 10.6. The DATA statements, given in Table 10.7, make the sprites appear as 3-by-3 mosaics.

Table 10.6 Multicolour sprite subroutine

```
300 REM MULTICOLOUR MODE
310 POKE SM+28,3
320 POKE SM+37,2
330 POKE SM+38,7
340 POKE SM+39,1
350 POKE SM+40,5
360 RETURN
```

Before running the program, list it on the screen. When the sprites appear, the centre mosaic, which has been specified as the background colour, will appear transparent; that is, the sprite has a hole in the middle. The background colour can therefore only be used as a 'colour' provided the sprite does not pass in front of anything.

Table 10.7 Revised DATA statements for multicolour sprite mosaics

```
200 DATA 85,255,85,85,255,85
202 DATA 85,255,85,85,255,85
204 DATA 170,0,255,170,0,255
206 DATA 170,0,255,170,0,255
208 DATA 255,170,85,255,170,85
210 DATA 255,170,85,255,170,85
212 DATA 0,0,0,0,0,0,0,0,0
214 DATA 0,0,0,0,0,0,0,0,0
216 DATA 0,0,0,0,0,0,0,0,0
220 END
```

Problem

Redesign the multicoloured sprites in the previous program as goldfish. A suggested colouring is, body—orange, markings—yellow, eye—black. Table A14 gives a suitable set of DATA values and the revised multicolour subroutine values.

10.10 Customised characters

The characters used by the Commodore 64 are defined in a similar way to the sprites. Each character is composed of eight lines of 8 bits, thus eight denary numbers can be used to define a character.

The character set is predefined in ROM but the contents of this ROM can be read and copied over into the 64's RAM. Once in RAM, individual data values can be altered to change the 'shape' of the characters. The 64 will need to be informed as to where to find the new character set by a suitable POKE instruction.

The 64 provides a great deal of flexibility for moving its internal instructions around in memory, and internal clashes can occur. For this reason, the program given in this section is intended only to demonstrate the principle. A detailed explanation is beyond the scope of this book.

The program in Table 10.8 transfers the character set to RAM and then changes the capital S to a Greek sigma symbol.

Line 10, by means of a POKE, instructs the 64 to refer to a character set starting at memory location 14336. (The normal character set is obtained by POKE 53272,21 and the lower case set by POKE 53272,23.) Memory location 14336 has been chosen for the demonstration because it will be beyond the end of a BASIC program up to 8K in length. An alternative approach is to position the character set at the beginning of the BASIC area and then

Table 10.8 Customised character (sigma) program

```
10 POKE 53272,31
20 PRINT"WATCH THIS CHANGE"
90 REM MOVE CHARACTER SET
100 FOR J=0 TO 2047
105 POKE 56333,127: POKE 1,51
110 X=PEEK(53248+J)
115 POKE 1,55: POKE 56333,129
120 POKE(14336+J),X
130 NEXT J
500 REM DEFINE S AS SIGMA
510 FOR N=0 TO 7
520 READ X
530 POKE(14336+N+8*19),X
540 NEXT N
600 DATA 126,96,48,24,48,96,126,0
610 END
```

change a series of pointers to move the start of BASIC. However, in this case, the simplest method of programming has been adopted.

Line 20 prints a message on the screen. After execution of line 10, when the 64 is looking for characters in the new location and they have not yet been set up, this message will be displayed as nonsense on the screen. As the character set is copied over, the message will gradually become intelligible.

The character set is copied over within the loop (lines 100–130). The number of characters to be copied over is 256 which, at eight data values per character, means the loop count for J is set from 0 to 2047. Line 105 is necessary to allow the characters at the ROM address to be used. Line 110 sets X to the value PEEKed at the ROM address and then line 115 restores locations 1 and 56333 to their original values. The value in X is then POKEd into the new address location by line 120.

When all the characters have been established in their new positions their design can be changed. The routine, lines 500 to 540, changes the capital S to a sigma. Lines 510 to 540 form a loop to POKE eight new values. The values are held in the DATA statement at line 600. These values are derived from the eight lines of 8 bits forming the character as shown in Figure 10.3.

The start of the character table is now 14336 and the letter S is the nineteenth letter of the alphabet so the eight data values for the letter S start at $14336 + 8 \times 19$. Line 530 POKEs the eight successive data values.

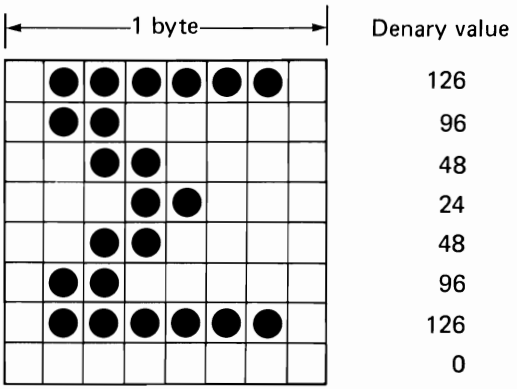


Figure 10.3 A customised character

When the program is run, any screen display will become unintelligible; as the characters are copied across, the screen will gradually begin to return to 'normal'. It takes about 1½ minutes for the character set to be completely formed. Once this has been done, all the capital Ss will then change to sigmas and the letter S key should now be defined as a sigma.

Note that using the RESTORE key will have the usual action and will force a POKE 53272,21. That is, the letter S will become 'normal'. However, the character set is still set up in RAM and the program does not need to be run again. To obtain the revised characters simply POKE 53272,31. This character set will be available via this POKE until the 64 is switched off.

11

Generating Sound

11.1 Introduction

The Commodore 64 has three voices, each of which are controlled by POKEing to their seven registers. In addition, the volume is controlled by POKEing to one common register. Appendix I gives details of the registers involved. As with the sprites, it is convenient to use register values that are offsets from the first value (54272).

Each voice is turned 'on' and 'off' by POKEing the waveform register. The value POKEd, for 'on', is selected according to the waveform required (i.e. 17 for triangular). To turn the waveform 'off' a value of one less is used (i.e. 16 for triangular).

The characteristics of the voice must be set before turning the waveform 'on'. These characteristics fall under two headings: the shape of the sound (the sound envelope) and the frequency of the sound (i.e. note).

The shape, or sound envelope, is determined by four parameters – the attack period, decay period, sustain level and release period – as shown in Figure 11.1. These four parameters are often referred to as the ADSR envelope.

The attack period determines how quickly the note reaches its peak volume (as determined by the volume setting). This period can range from 2 milliseconds to 8 seconds with the 64. A decay period (from 6 milliseconds to 24 seconds) then follows, during which the volume reduces to the sustain level. The sustain level can be set in increments from 15 (full volume) down to 0. The sustain level is therefore a proportionate part of the current volume setting. Final-

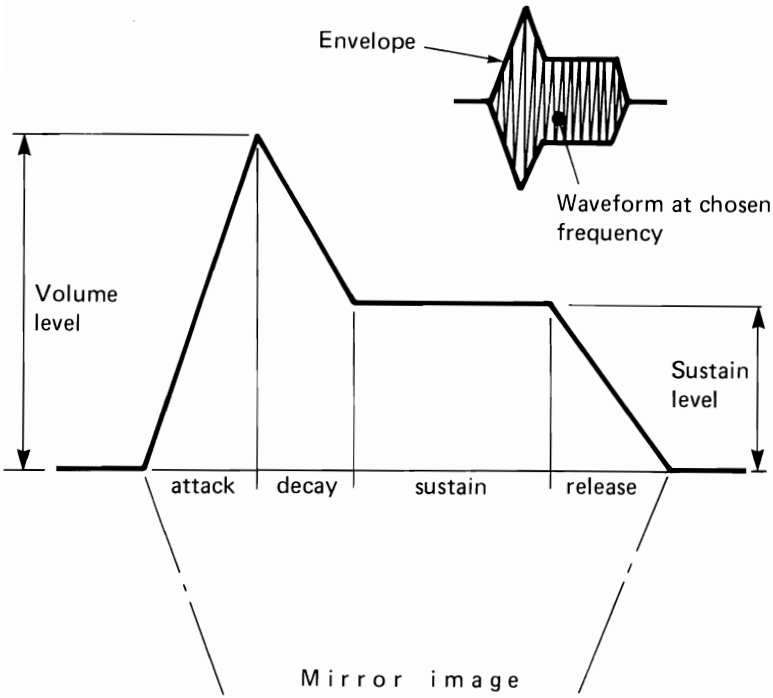


Figure 11.1 Elements of an ADSR sound envelope

ly, the release period is similar to the decay period but reduces the sound from the sustain level to zero.

The attack and decay are set by POKEing one register, and the sustain and release by POKEing another register. Reference to Appendix I will show that the attack/decay register for voice 1 is offset 5. The attack value is set in bits 4, 5, 6 and 7, while the decay value is set in bits 0, 1, 2 and 3. A value of 0 to 15 can therefore be used in each case.

As the attack rate is set in the top four bits of the register, the chosen value (from 0 to 15) needs to be multiplied by 16 to obtain the correct positional denary equivalent. The decay value is then added to give the complete POKE value. For example, to implement an attack period of 68 milliseconds requires an attack value of 6 (see Appendix J). If the decay period was to be 2.4 seconds (decay value = 11) then the composite POKE value would be:

$$(6 \times 16) + 11 = 107$$

A similar calculation is made for sustain and release values because the sustain value is set in the top four bits of the sustain/release register.

The effect of different attack, decay, sustain and release values are best judged by using them, and the next section gives a program that will allow this experimentation.

The frequency of the note produced by the envelope designed, as described above, is determined by POKEing to two other registers (for each voice). These registers are referred to as the 'High frequency' and 'Low frequency' registers, values in these two registers interacting to produce a note. Appendix K gives the values to be used in this book for music notes over eight chromatic octaves. These are not absolute values, and alternative lists may be developed depending on the pitch required for a key note (e.g. concert pitch, etc.).

If the 'pulse' waveform is to be used (POKE value 65) then two additional registers need to be POKEd. These registers are called the 'HI' pulse and the 'LO' pulse. Values of 0 to 15 can be POKEd to the HI pulse register and values of 0 to 255 to the LO pulse register. The pulse waveform is a square wave and can be visualised as the battlements of a castle. The HI pulse and LO pulse values used determine the relative proportions of the spacing on the 'battlements'.

The duration of a note (from the beginning of the attack to the end of the sustain) is determined by the program. POKEing the waveform 'on' starts the note and it is sustained for the required time by a delay loop in the program. Note that, if the sustain *level* is set to zero, the effective duration of a note is only the attack period plus the decay period. The note may therefore appear to stop before the end of a sustain loop or before the waveform is POKEd 'off'. In contrast, a note may carry on *after* the waveform has been POKEd 'off', because the release period has not expired.

The above considerations give the programmer great flexibility and control in developing sounds. However, because of the number of controls that need to be set, it is essential to experiment when developing programs using the sound facilities.

In summary, the general sequence of instructions required within a program is as follows:

1 Set volume level (0 to 15)
2 Set attack/decay value
3 Set sustain/release value
4 Set waveform 'on'
5 Set High frequency
6 Set Low frequency
7 Sustain note by program delay
(7*a* Branch back to step 5, if required)
8 Turn waveform 'off'
(8*a* Branch back to step 1, 2, 3 or 4, if required)
9 Continue with program (set volume level to zero, if not required)

The above sequence may not be followed in every instance as the program logic may be the deciding factor. In general, the volume, attack/decay and sustain/release must be set *before* the waveform is set. Having created the desired sound effect, it is good practice to 'turn off' the waveform and perhaps the volume because any settings will remain operative and will be carried over into the next program that is run. For this reason, it is useful to start a sound program with a routine that POKEs all sound registers to zero, i.e.

```
10 FOR L=5427 TO 54272+24
20 POKE L,0
30 NEXT L
```

11.2 ADSR envelope program

The best way to understand the effects of changing A, D, S or R values is to try alternative values in a program. Table 11.1 gives a program that allows different envelopes to be tried. As the emphasis is on changing A, D, S or R values, the program uses a single waveform (triangle) and a single note for the comparisons.

Table 11.1 ADSR envelope program

```
8 GOSUB 500
10 PRINT"ENTER VALUES BETWEEN 0 AND 15"
15 PRINT
20 INPUT"ATTACK ";A
30 INPUT"DECAY  ";D
40 INPUT"SUSTAIN";S
```

```
50 INPUT"RELEASE";R
60 AD=A*16+D:SR=S*16+R
70 GOSUB 500
72 PRINT"ATTACK  =";A;TAB(14);"DECAY   =";D;
73 PRINT TAB(29);"A/D =";AD
80 PRINT"SUSTAIN =";S;TAB(14);"RELEASE =";R;
82 PRINT TAB(29);"S/R =";SR
83 PRINT
90 SS=54272
92 FOR L=SS TO SS+24
100 POKE L,0
110 NEXT L
120 POKE SS+24,15
130 POKE SS+05,AD
140 POKE SS+06,SR
150 POKE SS+01,17
160 POKE SS+00,37
170 POKE SS+04,17
180 FOR T=1 TO 600
190 PRINT"SUSTAIN COUNT";T
195 PRINT"□";
200 NEXT T
210 POKE SS+04,16
220 END
500 PRINT"□";
510 PRINT"VALUE      A TIME    D/R TIME   % SUSTAIN"
515 PRINT"----------------------------------------"
520 FOR I=0 TO 15
522 READ L$
524 PRINT L$;INT(100*I/15+.5)
526 NEXT I
530 RESTORE
540 PRINT"----------------------------------------"
620 DATA "  0          2 MS        6 MS       "
621 DATA "  1          8 MS       24 MS       "
622 DATA "  2         16 MS       48 MS      "
623 DATA "  3         24 MS       72 MS      "
624 DATA "  4         38 MS      114 MS      "
625 DATA "  5         56 MS      168 MS      "
626 DATA "  6         68 MS      204 MS      "
627 DATA "  7         80 MS      240 MS      "
628 DATA "  8        100 MS      300 MS      "
629 DATA "  9        250 MS      750 MS      "
630 DATA " 10        500 MS      1.2 S       "
631 DATA " 11        800 MS      2.4 S       "
632 DATA " 12          1 S         3 S       "
633 DATA " 13          3 S         9 S       "
634 DATA " 14          5 S        15 S       "
635 DATA " 15          8 S        24 S      "
640 RETURN
```

Line 8 branches to a subroutine which causes the times associated with different A, D, S and R values to be displayed on the screen. These details are in DATA statements (lines 620–635). The sustain levels are calculated as a percentage of full volume in line 524 (within the loop, lines 520–526). When all sixteen DATA statements have been read, the RESTORE in line 530 causes the DATA pointer to point to the *beginning* of the DATA list, so that the next time the subroutine is called (in line 70) the table can be displayed again.

Values for the attack rate, decay rate, sustain level and release rate are requested in lines 20 to 50. The values are over the scale 0 to 15. Line 60 then converts the attack rate and the decay rate into a common value (AD) to be POKEd to the attack/decay register. The second statement in line 60 similarly converts the sustain level and release rate into a common value (SR).

Line 70 refreshes the screen with the ADSR table of values again by calling subroutine 500. The values input and the calculated values to be POKEd to the AD and SR registers are then displayed directly under the table.

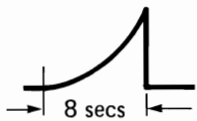
The sound waveform is specified over lines 90 to 170. Line 90 assigns the first sound register, 54272, to the variable SS so that the other registers can be specified as offsets. The loop (lines 92–110) clears all the sound registers by POKEing zero into each one. The volume level is then set in line 120, the attack/decay rate in line 130 and the sustain/release in line 140. Lines 150 and 160 POKE the high and low frequency values required to sound the note. The triangular waveform is then turned on in line 170.

The note is sustained by the loop from line 180 to line 200. A loop count of 600, which is a rather long sustain period, has been chosen so that the effect of slow attack or decay rates can be heard. As the loop is incremented, the loop counter (variable T) is printed as a 'sustain count'. This allows the duration due to different attack, decay and release rates to be roughly compared. Line 195 within the loop moves the cursor up a line (using a 'cursor up' control character) to overprint continually the message 'sustain count', and the value of T, on the screen.

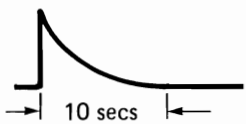
At the end of the program, line 210 turns the waveform off.

11.3 Some experiments with the ADSR program

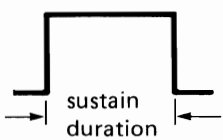
(*a*) attack = 15 decay = 0
sustain = 0 release = 0
i.e. A/D = 240 and S/R = 0



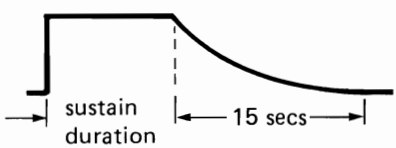
(*b*) attack = 0 decay = 13
sustain = 0 release = 0
i.e. A/D = 13 and S/R = 0



(*c*) attack = 0 decay = 0
sustain = 15 release = 0
i.e. A/D = 0 and S/R = 240



(*d*) attack = 0 decay = 0
sustain = 15 release = 15
i.e. A/D = 0 and S/R = 255



Note: Times shown are approximate

Figure 11.2 Some sound envelopes

Run the program shown in Table 11.1 and enter the following inputs:

(*a*) Attack = 15 Decay = 0
Sustain = 0 Release = 0

This gives an envelope that takes about eight seconds to reach peak volume ('sustain count' approximately 400). The note then abruptly ceases. An illustration of the envelope is given in Figure 11.2a.

(*b*) Attack = 0 Decay = 13
Sustain = 0 Release = 0

This envelope, shown in Figure 11.2b, starts at full volume and slowly decays away. The note has effectively disappeared after a 'sustain count' of 420.

(*c*) Attack = 0 Decay = 0
Sustain = 15 Release = 0

This envelope demonstrates, in effect, a square wave. The note commences immediately at full volume and continues at full volume throughout the sustain loop. A diagram of the envelope is shown in Figure 11.2c. In practice, this envelope is more likely to be used with very short sustain periods to create bleeps.

(*d*) Attack = 0 Decay = 0
Sustain = 15 Release = 15

This envelope demonstrates the effect of the sustain and release characteristics. Due to the sustain valve, the note continues at full volume for the sustain duration. At the end of the sustain duration, when the note is turned off, the high release value causes the note to die away slowly over a period of approximately fifteen seconds. An illustration of the envelope is shown in Figure 11.2d.

11.4 A two-tone siren

This program (Table 11.2) demonstrates the use of the envelope tested out in (*c*) above. The program alternates between two notes to produce a siren effect.

Line 30 sets the variable SS to 54272 for use with the sound offsets. Line 50 sets up the volume register as V, the attack/decay register as AD, the sustain release as SR and the waveform as W.

The volume is turned 'full on' in line 70 and line 90 sets the sound to the envelope shown in Figure 11.2c. The value of 240 for the SR register comes from:

sustain level of 15 × 16 + release rate of 0

A sawtooth waveform is turned in line 110.

The loop, lines 120 to 190, is used to sound the two notes ten times. Line 130 POKEs a value to sound a note and line 140

Table 11.2 A two-tone siren

```
30 SS=54272
40 REM VOL,AD,SR,W/FORM POKE VALUES
50 V=SS+24:AD=SS+5:SR=SS+6:W=SS+4
60 REM SET VOL TO MAX
70 POKE V,15
80 REM SET AD & SR VALUES
90 POKE AD,0: POKE SR,240
100 REM SET W/FORM TO SAWTOOTH
110 POKE W,33
120 FOR N=1 TO 10
130 POKE SS+0,75:POKE SS+1,34
140 FOR D=1 TO 300
150 NEXT D
160 POKE SS+0,52:POKE SS+1,43
170 FOR D=1 TO 300
180 NEXT D
190 NEXT N
200 POKE W,32
210 POKE V,0
220 END
```

determines the sustain duration. The note is changed in line 160 and is sounded for a similar duration (lines 170 and 180).

When the two notes have been repeated ten times, the loop is exited and line 200 turns the waveform off. Finally, line 210 turns the volume off.

The notes produced by the program sound rather harsh, similar to an electronic siren. If the waveform is changed to triangular by:

```
110 POKE W,17
200 POKE W,16
```

then the siren sounds more mellow.

If the noise waveform is chosen by:

```
110 POKE W,129
200 POKE W,128
```

then the sound is somewhat similar to a steam engine.

By changing line 90 to:

```
90 POKE AD,253:POKE SR,0
```

the engine appears to come near and pass into the distance. This

effect is solely due to setting the attack rate value to 15 and the decay value to 13. If the value of N is printed out by:

```
185 PRINT N
```

it will be seen that the long attack time means that the 'steam engine' does not reach full volume until about the fourth loop (i.e. when N = 4) and the decay lasts from N = 5 until the end when N = 10.

11.5 Two sound subroutines

As every program involving sound requires several sound registers to be referenced, it is convenient to develop and use standard subroutines.

Table 11.3 Sound initialisation subroutine

```
1000 REM SOUND INITIALISATION
1010 SS=54272
1020 VOL=SS+24
1030 WT=17:WS=33:WP=65:WN=129
1040 LF(1)=SS+0:HF(1)=SS+1
1050 LP(1)=SS+2:HP(1)=SS+3
1060 WF(1)=SS+4
1070 AD(1)=SS+5:SR(1)=SS+6
1080 FOR I=2 TO 3
1090 LF(I)=LF(I-1)+7
1100 HF(I)=HF(I-1)+7
1110 LP(I)=LP(I-1)+7
1120 HP(I)=HP(I-1)+7
1130 WF(I)=WF(I-1)+7
1140 AD(I)=AD(I-1)+7
1150 SR(I)=SR(I-1)+7
1160 NEXT I
1170 RETURN
```

The subroutine shown in Table 11.3 sets up all the sound registers in easily remembered variables, for example SS for the start of sound register and VOL for volume. Note however that only the first two letters of a variable name are recognised by the Commodore 64, so these must be unique within the program – having used VOL, in line 1020, you then cannot use other variable names within the program starting with VO.

Line 1010 assigns the first sound register (54272) to the variable SS. VOL is then set to 54296 in line 1020. The four waveforms are

assigned in line 1030: WT for triangular waveform, WS for sawtooth waveform, WP for pulse and WN for noise.

The register locations associated with the three voices are assigned to array variables so that the array element represents the voice number. Voice 1 is first set up over lines 1040 to 1070 where the low frequency (LF(1)), high frequency (HF(1)), low pulse (LP(1)), high pulse (HP(1)), waveform (WF(1)), attack/decay (AD(1)) and sustain/release (SR(1)) are assigned the appropriate register value.

As corresponding registers for each successive voice are seven units greater than the previous voice, the registers for voices 2 and 3 are set within the loop, lines 1080 to 1160.

This subroutine is required only to be used once in any program. However, when sound programs are run, it may be necessary to clear the sound registers prior to further use. It is convenient to use a 'clear sound registers' subroutine as shown in Table 11.4.

Table 11.4 Clear sound registers subroutine

```
2000 REM CLEAR SOUND REGISTERS
2010 FOR L=SS TO SS+24
2020 POKE L,0
2030 NEXT L
2040 RETURN
```

Line 2010 assumes that the previous subroutine has been used and therefore the variable SS has already been assigned the value 54272. The loop 2010 to 2030 POKEs a zero into every sound register.

11.6 A ringing phone routine

This program (Table 11.5) demonstrates the use of the two subroutines just described and also illustrates how a sound routine itself can be developed as a subroutine.

Line 10 calls the initialisation subroutine (Table 11.3) and then line 20 calls the ringing phone subroutine that starts at line 5000.

The sound registers are cleared by the subroutine given in Table 11.4, which is called at line 5005. The volume, attack/decay rate, sustain/release rate and waveform are set by lines 5010 to 5040.

The loop, lines 5050 to 5150, causes the basic ringing tone to be repeated five times. A further loop, lines 5060 to 5130, causes each

Table 11.5 Ringing phone program

```
10 GOSUB 1000
20 GOSUB 5000
30 END


5000 REM RINGING PHONE
5005 GOSUB 2000
5010 POKE VOL,15
5020 POKE AD(1),9
5030 POKE SR(1),220
5040 POKE WF(1),WT
5050 FOR R=1 TO 5
5060 FOR N=1 TO 2
5070 FOR T=1 TO 15
5071 POKE WF(1),WT
5080 POKE HF(1),68:POKE LF(1),149
5090 FOR S=1 TO 5:NEXT S
5100 POKE HF(1),0: POKE LF(1),0
5110 NEXT T
5120 FOR D=1 TO 100: NEXT D
5130 NEXT N
5140 FOR P=1 TO 500: NEXT P
5150 NEXT R
5160 POKE WF(1),WT-1
5190 RETURN
```

'ring' to consist of two short 'trills'. The 'trill' is obtained by turning a note on and off very quickly fifteen times, within the loop 5070 to 5110.

Line 5071 POKEs a triangular waveform and line 5080 POKEs the two frequency values for the note. A very short sustain is provided by the loop in line 5090 and then line 5100 turns the note off.

After a delay, caused by line 5120, the value of N is incremented at line 5130 and the loop from 5060 is repeated to give a second 'trill'. Line 5140 causes a longer pause before R is incremented (line 5150) and the two-trill pattern is repeated from line 5050.

After five 'rings', line 5160 turns the waveform off and the subroutine returns to the main program at line 30 to end.

11.7 Playing tunes

The program shown in Table 11.6 plays the tune Frere Jacques, allowing you to choose one of the following instruments: piano, xylophone or organ. It has been written as a series of subroutines so that, for example, the tune or the instruments can easily be changed.

Table 11.6 Frere Jacques tune

```
10 GOSUB 1000
20 GOSUB 1200
30 GOSUB 1400
40 GOSUB 2000
50 GOSUB 1700
60 GOSUB 1900
70 GOSUB 3000
80 GOTO 40
90 END



1200 REM SET UP INSTRUMENTS
1210 FOR I=1 TO 3
1220 READ W(I),A(I),S(I),HI(I),LO(I)
1230 NEXT I
1240 DATA 65,9,0,0,255:REM PIANO
1250 DATA 17,9,0,0,0:REM XYLOPHONE
1260 DATA 17,0,240,0,0:REM ORGAN
1270 RETURN



1400 REM NOTES TO ARRAY
1410 DIM H(40),L(40),D(40)
1420 FOR I=1 TO 40
1430 READ H(I),L(I),D(I)
1440 IF H(I)<>-1 THEN 1470
1450 N=I-1
1460 I=40
1470 NEXT I
1480 DATA 25,177,1,28,214,1,32,94,1,25,177,1
1490 DATA 25,177,1,28,214,1,32,94,1,25,177,1
1500 DATA 32,94,1,34,75,1,38,126,2
1510 DATA 32,94,1,34,75,1,38,126,2
1520 DATA 38,126,.5,43,52,.5,38,126,.5
1530 DATA 34,75,.5,32,94,1,25,177,1
1540 DATA 38,126,.5,43,52,.5,38,126,.5
```

```
1550 DATA 34,75,.5,32,94,1,25,177,1
1560 DATA 25,177,1,19,63,1,25,177,2
1570 DATA 25,177,1,19,63,1,25,177,2
1580 DATA -1,-1,-1
1590 RETURN
1700 REM DISPLAY MENU
1710 PRINT"■";"        ■CHOOSE AN INSTRUMENT■"
1720 PRINT
1730 PRINT"            1  PIANO"
1740 PRINT"            2  XYLOPHONE"
1750 PRINT"            3  ORGAN"
1760 PRINT
1770 INPUT"ENTER INSTRUMENT NO ";Z
1780 IF Z<1 OR Z>3 THEN 1700
1790 RETURN
1900 REM SET UP REGISTERS
1910 POKE VOL,15
1920 POKE AD(1),A(Z):POKE SR(1),S(Z)
1930 POKE LP(1),LO(Z):POKE HP(1),HI(Z)
1940 RETURN



3000 REM PLAY TUNE
3010 FOR I=1 TO N
3020 POKE HF(1),H(I):POKE LF(1),L(I):POKE WF(1),W(Z)
3030 FOR T =1 TO D(I)*150:NEXT T
3040 POKE WF(1),W(Z)-1
3050 FOR T =1 TO 50:NEXT T
3060 NEXT I
3070 RETURN
```

The main program is contained in lines 10 to 90. Line 10 calls the sound initialisation subroutine previously discussed (see Table 11.3) and then line 20 calls a subroutine at line 1200 to set up the POKE values for the three instruments.

This subroutine reads three sets of DATA values into arrays within the loop from line 1210 to line 1230. Array W holds the waveform values, array A the attack/decay values, array S the sustain/release values, array HI the high pulse values and array LO the low pulse values. The associated DATA values are in the subroutine at lines 1240 to 1260.

The main program at line 30 next calls a subroutine at line 1400 to set up the required notes and their durations into arrays.

The subroutine starting at 1400 is designed to read three DATA values for each note, the high frequency, the low frequency and

duration, and store them in arrays H, L and D respectively. Line 1410 dimensions the arrays to 40. This is sufficient for most simple tunes. The intention is that up to this limit of 40 the subroutine is general purpose and only the DATA lines (1480–1580) will need changing for another tune. The end of the tune is indicated in the DATA lines by $-1,-1,-1$.

A loop, lines 1420 to 1470, reads in the DATA values into the three arrays. The high and low frequency values are established by converting the musical note to a POKE value from Appendix K. The note duration is relative; for example, a crotchet = 1.

When the note is 'played', this duration is multiplied by a factor of 150 (line 3030) to create a sustain duration. To play the tune quicker the factor can be decreased or vice versa.

Line 1440 in the loop tests the value of H(I) for the end of the data, when it will equal -1. If H(I) is not equal to -1, the reading and storing of DATA values continue. When the end of the data is reached, line 1450 sets N to the number of notes to be played and then I is set to 40 in line 1460 so that the loop is exited. The end of the subroutine is immediately after the DATA statements at line 1590.

Having set up the notes ready to be played, the sound registers are cleared by line 40 of the main program calling the subroutine previously given in Table 11.4. Line 50 then calls a subroutine at line 1700 which displays the menu.

The menu subroutine clears the screen and displays the title in line 1710. The choice of three instruments is then given over lines 1730 to 1750, and line 1770 requests an instrument number be entered. Line 1780 tests that the number entered, stored in variable Z, lies within 1 and 3 otherwise the subroutine is restarted. Once Z has been set to 1, 2 or 3, the subroutine returns from line 1790 to the main program so that the appropriate sound registers can be set.

Line 60 of the main program calls a subroutine starting at line 1900 to set up the sound registers for the chosen instrument. Line 1910 turns the volume 'full on'. Line 1920 POKEs the attack/decay register for voice 1 with the appropriate value for instrument Z, and then POKEs the sustain/release register for voice 1. The appropriate high pulse and low pulse values are POKEd for voice 1 in line 1930. The subroutine returns in line 1940 to the main program.

Line 70 of the main program finally calls subroutine 3000 to 'play' the tune. This subroutine consists of a loop, lines 3010 to 3060,

which processes each note of the tune. Line 3020 POKEs the appropriate high frequency and low frequency and then turns the waveform on. Line 3030 sustains the note according to the value of the current D(I) value and then the note is turned off by line 3040. Line 3050 provides a slight delay before the next note is played by repeating the loop. When all the notes have been played (from 1 to N) the subroutine returns to the main program.

The main program, line 80, then jumps back to line 40 so that the tune can immediately be played again, with a different choice of instrument.

Note that in order to do this, all 'setting up' subroutines involving the reading of DATA must precede line 40, otherwise the DATA pointer could be wrong. It is also important, in any sound programs that recycle, to ensure that the sound registers are cleared to avoid the possibility of previous settings being carried over.

Problem

Write a program to act as a timer alarm. The time duration should be entered in minutes and the screen should display the elapsed time in minutes and seconds. As the timing count proceeds a ticking noise should be heard. When the required elapsed time is reached an alarm is sounded. A suitable program is listed in Table A15.

Hints: The variable TI is used by the Commodore 64 as a timer and is automatically incremented by 1 every 1/60 of a second from the moment the 64 is switched on. Therefore, if B is set equal to TI at the beginning of the count, then the duration, D, in seconds can be calculated from

$$D = (TI - B)/60$$

12

Range of Applications

12.1 Series

A series consists of a number of terms, each term having a constant relationship to the next term. When devising computer programs for evaluating series, a procedure needs to be designed which allows the next term in the series to be calculated from the previous term.

For example, the exponential series may be evaluated as follows:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

where $2! = 1 \times 2$ $3! = 1 \times 2 \times 3$, etc.
The steps in the repetitive process to calculate e^x to n terms are:

Step 1 (initialisation), set first term (T) to x, e^x to 1 + T, and I to 2
Step 2 calculate next term by multiplying previous term by x/I, and add this new term to the old value of e^x
Step 3 repeat step 2 a further $n - 2$ times.

The BASIC routine to calculate e^x is shown in Table 12.1.

Table 12.1 Program to calculate e^x

```
20 INPUT"NO OF TERMS FOR E↑X";N
40 INPUT"VALUE OF X";X
60 LET T=X
70 LET E=1+T
80 FOR I=2 TO N
90 LET T=T*X/I
100 LET E=E+T
110 NEXT I
115 PRINT
120 PRINT"E↑";X;"=";E
125 PRINT"**********"
130 END
```

Problem 1 – Evaluation of cos *x*

Write a BASIC program for evaluating $\cos x$, given that:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

The program is listed in Table A16, and the value of cos 30° to five terms is given in Appendix B.

12.2 Processing experimental data

The program given in Table 12.2 illustrates the use of the computer to process data which is entered at run time from the keyboard in response to messages output from the program. This method of working is applicable to, say, a class of students where each group is carrying out similar experiments. The results of the experiments are prepared for input to the computer program, and the program is used to output the final answer for each group. Similarly, a scientist may repeat the same experiments for different substances and the series of results may then be processed by one computer program.

The output messages, replies (data input) and results are shown in Table 12.3. S is the mass of substance burnt and W the mass of water heated by the substance in grammes, T is the rise in temperature of the water in °C, and R is the relative molecular mass.

Table 12.2 'Heat of combustion' problem

```
10 PRINT"HEAT OF COMBUSTION"
15 PRINT"------------------"
20 PRINT
30 INPUT"NAME OF SUBSTANCE";N$
40 INPUT"ENTER S,W,T,R";S,W,T,R
60 LET H=INT(W*4.2*T*R*0.001/S+0.5)
70 PRINT
80 PRINT"RESULT FOR ";N$;" = ";H;" KJ/MOL"
85 PRINT"************************************"
90 PRINT
100 INPUT"ANY MORE DATA (Y=YES,N=NO)";Y$
120 PRINT
130 IF Y$ = "Y" THEN 20
140 END
```

Table 12.3 Output from Table 12.2 and data input

```
HEAT OF COMBUSTION
- - - - - - - - - - - - - - - - - - - -
NAME OF SUBSTANCE? ETHANOL
ENTER S,W,T,R? .36,100,23.5,46
RESULT FOR ETHANOL = 1261 KJ/MOL
 ************************************
ANY MORE DATA (Y = YES, N = NO)?Y
NAME OF SUBSTANCE? METHANOL
ENTER S,W,T,R? .39,99,21.2,32
RESULT FOR METHANOL = 723 KJ/MOL
 ************************************
ANY MORE DATA (Y = YES, N = NO)?Y
NAME OF SUBSTANCE? PROPANOL
ENTER S,W,T,R? .31,101,24.8,60
RESULT FOR PROPANOL = 2036 KJ/MOL
 ************************************
ANY MORE DATA (Y = YES, N = NO)?N
```

Problem 2 – Roots of quadratic equations

Write a program to calculate the values of the roots of any number of quadratic equations ($ax^2 + bx + c = 0$), given the coefficients a, b and c. If $b^2 - 4ac > 0$, output the message 'REAL ROOTS' and the two roots. If $b^2 - 4ac = 0$, output the message 'COINCIDENT ROOTS', and the value $= -b/2a$. If $b^2 - 4ac < 0$, output the message 'COMPLEX ROOTS'. Allow for interactive entry of a, b and c during run time and stop the execution of the program by zeros being entered for a, b and c. The program is listed in Table A17, and the answers for the equations shown in Table 12.4 are given in Appendix B.

Table 12.4 Quadratic equations

$$
\begin{aligned}
3x^2 + 9x + 2 &= 0 \\
7x^2 - 5x + 3 &= 0 \\
x^2 - 8x + 16 &= 0 \\
2x^2 + 3x - 4 &= 0 \\
-3x^2 - 2x + 1 &= 0 \\
x^2 + 2x + 3 &= 0 \\
4x^2 + 4x + 1 &= 0
\end{aligned}
$$

12.3 Tabulation of results and averaging

Measurements, intermediate calculations and final results of experiments may need to be tabulated so that a permanent record is available in an easily readable form. The final answer is often obtained by averaging the results of more than one experiment.

Problem 3 – Width of a slit

The collimator of a spectrometer was used to provide a parallel beam of light from a sodium flame. The beam of light was allowed to fall on a slit placed vertically at the centre of the table of the spectrometer. When appropriate adjustments had been made, parallel bands were seen on looking through the telescope. These were made as sharp as possible by adjusting the slit of the collimator. The crosswires of the eyepiece of the telescope were set on corresponding minima on either side of the centre and the vernier readings were noted; this gave a value of 2A for each of the fringes 1 to 6.

The width of the slit W (cm) = $N\lambda/A$ where N is the fringe number, $\lambda = 5.893 \times 10^{-5}$ cm (wavelength of sodium light), and A is in radians.

Write a program to tabulate the measurements taken and values of A and W as shown in Table 12.5, and output the average value of the width of the slit. There are two pairs of vernier readings for each fringe number. Values of 2A are found by subtracting the second vernier reading from the first vernier reading. The average value of A is then calculated for each fringe number. The program is listed in Table A18, and the average value of W is given in Appendix B.

12.4 Linear regression

Often straight line graphs may be obtained by manipulating the formula which defines the relationship between the variables. The equation of a straight line may be written as,

$$y = mx + c$$

where

x = the independent variable
y = the dependent variable
m = the slope of the line
c = the intercept of the line on the y axis

Table 12.5 Tabulation of results for Problem 3

FRINGE NUMBER	VERNIER DEG	VERNIER MIN	READINGS DEG	READINGS MIN	A MIN	WIDTH OF SLIT CM
1	41	26	41	20	3	.0675
	221	25	221	19		
2	41	29	41	17		
	221	27	221	15		
3	41	33	41	15		
	221	31	221	13		
4	41	35	41	11		
	221	34	221	10		
5	41	39	41	9		
	221	38	221	8		
6	41	41	41	5		
	221	40	221	4		

A line of 'best fit' can be calculated for a series of data points from,

$$m = \frac{n\Sigma xy - \Sigma x \Sigma y}{n\Sigma x^2 - (\Sigma x)^2}$$

and

$$c = \frac{\Sigma y - m\Sigma x}{n}$$

where

x and y are the co-ordinates of each data point and

n = number of data points.

There are many equivalent forms of the above expression; some are more suited to manual calculation than programming. A measure of how closely the data follows the calculated straight line is given by the coefficient of correlation *(r)*. If the data lies on a perfectly straight line then r will be $+1$ (for positive slope) or -1 (for negative slope). In the extreme case of no correlation whatsoever (i.e. the points are scattered randomly) r will equal zero. The acceptable level of correlation (i.e. value of r) for the number of readings involved can be found from statistical tables.

Again, the formulae for r can be presented in different ways. The expression given below is in a convenient form for programming when the slope is already evaluated.

$$r = \sqrt{\frac{m(\Sigma xy - \Sigma x \Sigma y/n)}{\Sigma y^2 - (\Sigma y)^2/n}}$$

It should be noted that m, c and r require similar preliminary calculations and that it is convenient to calculate initially and store,

$$\Sigma x, \Sigma y, \Sigma x^2, \Sigma y^2, \Sigma x \Sigma y$$

A program to perform linear regression and calculate r is given in Table 12.6.

Table 12.6 Linear regression routine

```
50 DIM X(20),Y(20)
60 INPUT"ENTER NO OF PAIRS OF READINGS";N
80 FOR I = 1 TO N
90 INPUT"ENTER X,Y PAIR ";X(I),Y(I)
110 NEXT I
120 GOSUB 4000
130 END
4000 LET S1=0
4010 LET S2=0
4020 LET S3=0
4030 LET S4=0
4040 LET S5=0
4100 FOR I = 1 TO N
4110 LET S1=S1 + X(I)
4120 LET S2=S2 + Y(I)
4130 LET S3=S3 + X(I)↑2
4140 LET S4=S4 + Y(I)↑2
4150 LET S5=S5 + X(I)*Y(I)
4160 NEXT I
4170 LET M=(N*S5 - S2*S1)/(N*S3 - S1↑2)
4180 LET C=(S2 - M*S1)/N
4190 LET R=(M *(S5 - S1*S2/N))/(S4 - S2↑2/N)
4192 PRINT
4193 PRINT"Y = M*X+C"
4194 PRINT"M =";M
4195 PRINT"C =";C
4196 PRINT
4197 PRINT"COEFF. OF CORRELATION =";SQR(R)
4199 RETURN
5000 END
```

Problem 4 – Young's modulus of the material of a bar

The bar was clamped horizontally at one end. A weight of mass M (kg) was attached to the other end, and was kept vibrating by an electro-magnet. The vibrating end of the bar was illuminated and was viewed through a slit in a rotating disc, using a telescope. The speed of the disc was gradually increased by adjusting the resistance, placed in series with the electric motor used to rotate the disc, until the bar appeared to be at rest when it was vibrating. A counting arrangement on the motor gave the number of rotations in a definite time.

It can be shown that the motion of the vibrating bar is simple harmonic with a period:

$$T = 2\pi \sqrt{\left(\frac{l^3(M + 33/140m)}{3Yi}\right)}$$

where

i is the moment of inertia of cross-section

Y is Young's modulus of the material of the bar

l is the length of the bar in metres

m is the mass in kg of the vibrating part of the bar

For a bar of rectangular cross-section (breadth b and depth d metres), $i = bd^3/12$.

Hence,
$$\frac{3Ybd^3T^2}{48\pi^2l^3} = M + 33/140m$$

T^2 (seconds) plotted for different values of M (kg) gives a straight line graph, and Y may be found using the slope of the graph as follows:

$$Y = \frac{1}{\text{slope of graph}} \quad \frac{16\pi^2l^3}{bd^3}$$

Write a program to output Young's modulus for a bar in Newtons/m^2. Use the linear regression routine given in Table 12.6, to find the slope of the graph for the values of T and M given in Table 12.7. The dimensions of the bar are: $b = 1.58$ cm, $d = 0.312$ cm, $l = 40$ cm. The results are given in Appendix B.

Note: Remember to caiculate T^2 for the linear regression 'Y' values; the 'X' values are those listed under M.

Table 12.7 Data for Problem 4

M *(kg)*	T *(seconds)*
.097	0.12
.147	0.139
.157	0.145
.177	0.15
.197	0.16

12.5 Simulation

12.5.1 Background

Simulation requires the writing of a program that models a situation. Changes are brought about in the model, either by the user or by inbuilt routines so that the behaviour of the model can be studied. From studying the behaviour of the model under varying circumstances it is hoped to gain a better understanding of the reality represented by the model.

Some models consist of specific relationships (e.g. a Balance Sheet). In such a case, if you make a change in one variable this leads to a specific revised Balance Sheet. You can, by this means, simulate the effect of changes in labour costs on the profits.

Many forms of simulation require the values of some of the variables to be sampled from a probable range of values. The probable range of values is usually expressed as a probability (or frequency) distribution. In these models, the outcomes and their interactions need to be studied over many simulations to obtain a representative picture of the model's behaviour.

A simple simulation model of this type is discussed below. As the basis of the variability is the sampling from a frequency distribution, the program has been written to make use of the two subroutines previously developed in Chapter 9, section 9.6. Note how the subroutine can be used several times by transferring values to and from the variables common to the subroutine.

12.5.2 Simulation of combined units

The problem is to simulate the breakdown pattern of a combined unit comprising a motor assembly and a gear assembly from the breakdown pattern of the individual assemblies.

The running time of a combined unit can be simulated by sampling in turn from the running time distributions of the motor unit and the gear unit. The shorter running time will be the running time of the combined unit. By simulating many such samples the MTBF (mean time between failure) for a combined unit can be obtained.

12.5.3 Output required

For a short simulation it is convenient to monitor the course of each pass through the program. Therefore, in this case, the output can be the sampled lives of the motor and gear assemblies, the life of the combined unit and the MTBF to date. For longer simulations this amount of detail would be time consuming to print. It could be incorporated for debugging purposes and then dropped, the final program only producing the ultimate MTBF.

However, a single final statement of the value of the MTBF is not as informative as a running output of the variable. The decision to terminate a simulation is often taken once the variable under inspection has settled down. These considerations, in this case, lead to the idea that there should be an option to continue the run if the fluctuation in the MTBF is not within the desired limits.

12.5.4 Description of the program

The BASIC listing of the main routine is shown in Table 12.8 and of the subroutines in Tables 9.12 and A13 (see Chapter 9).

The motor unit frequency distribution is input after control is transferred to the subroutine from line 40. The input is returned in array D and the contents copied to array M. This allows array D and hence the subroutine to be used again. This time the gear unit frequency distribution is input and on return to the main routine it is copied from array D to array G.

The next stage of the program initialises the variables T and K in readiness for the simulation. Variable T is the cumulative combined unit running time, and variable K is the starting (or continuation) value of the simulation count. K is initially set at one for the first run

Table 12.8a Main routine for simulation program

```
20 DIM M(10,2),G(10,2),D(10,2),X(10,2)
40 GOSUB 800
50 FOR I=1 TO N
52 LET M(I,1)=D(I,1)
54 LET M(I,2)=D(I,2)
56 NEXT I
60 GOSUB 800
70 FOR I=1 TO N
72 LET G(I,1)=D(I,1)
74 LET G(I,2)=D(I,2)
76 NEXT I
80 LET U$="------------------------------------"
90 LET T=0
100 LET K=1
115 PRINT
120 INPUT"LENGTH OF SIMULATION";L
140 PRINT
145 PRINT U$
150 PRINT"SIM";TAB(6);"MOTOR";TAB(14);"GEAR";
160 PRINTTAB(22);"COMB.";TAB(30);"MTBF"
170 PRINT U$
180 FOR S=K TO L
200 FOR I=1 TO 10
202 LET X(I,1)=M(I,1)
204 LET X(I,2)=M(I,2)
206 NEXT I
```

(line 100) and is revised in line 370 in case the FOR loop is to be continued.

The initial length of the simulation is input at line 120. Lines 150 and 160 print the required heading. PRINT U$ produces a line of dashes and is used to highlight the headings (lines 145 and 170). Each line of calculated output is produced within the FOR loop from lines 180 to 350.

To sample from the motor unit distribution (array M) it is copied to array X by lines 200 to 206. The subroutine starting at line 900 is entered and a sample from array X is returned as variable V. In line 220 this value is retained for future reference as variable U1. This procedure is then repeated for the gear unit, the sampled value being retained as U2. Lines 280 to 300 carry forward the lower of the two values as variable C (this is the running time of the combined unit). The cumulative running time is calculated in line 320 and the current average running time (the MTBF) is calculated in line 330 as A. Having completed a pass through the FOR loop, a line of output provides the current simulated values of U1, U2, C and A.

After simulating the stipulated number of times (i.e. L), the FOR loop is left. In anticipation of continuing, the value of K is reset in line 370. Lines 380 to 400 allow you to reset L, or, if you enter zero, the run stops.

Table 12.8b Main routine for simulation program

```
210 GOSUB 900
220 LET U1=V
240 FOR I=1 TO 10
242 LET X(I,1)=G(I,1)
244 LET X(I,2)=G(I,2)
246 NEXT I
250 GOSUB 900
260 LET U2=V
280 LET C=U1
290 IF U2>U1 THEN 320
300 LET C=U2
320 LET T=T+C
330 LET A=T/S
335 LET A=INT(10*A)/10
340 PRINTS;TAB(6);U1;TAB(14);U2;TAB(22);C;TAB(30);A
350 NEXT S
355 PRINT U$
370 LET K=L+1
380 PRINT"ENTER ADDITIONAL SIMULATIONS"
390 PRINT"REQUIRED, OR ZERO TO STOP";
400 INPUT L
405 IF L>0 THEN 420
410 END
420 LET L=K+L-1
430 GOTO 170
```

To separate this interactive part of the run from the previously calculated output, PRINT U$ is now used in line 355. If the run is to be continued, control is returned to line 170 to separate the subsequent output in a similar way. This means of trying to keep the output tidy is best appreciated by studying extracts from a run of this program as shown in Table 12.9.

Problem 5 – Combined units simulation

Use the simulation program (Table 12.8) to calculate the mean time between failure for a combined unit consisting of motor and gear units having the failure pattern shown in Table 12.10. Simulate 100 failures. The answer is given in Appendix B.

Table 12.9 Example of output from Table 12.8

INPUT LENGTH OF SIMULATION ? 10

SIM	MOTOR	GEAR	COMB.	MTBF
1	16	18	16	16
2	16	14	14	15
3	8	16	8	12.6
4	16	18	16	13.5
5	16	14	14	13.6
6	8	14	8	12.6
7	12	22	12	12.5
8	20	22	20	13.5
9	16	10	10	13.1
10	4	14	4	12.1

ENTER ADDITIONAL SIMULATIONS
REQUIRED, OR ZERO TO STOP ? 5

11	16	14	14	12.3
12	12	14	12	12.3
13	16	18	16	12.6
14	16	16	16	12.8
15	12	18	12	12.8

ENTER ADDITIONAL SIMULATIONS
REQUIRED, OR ZERO TO STOP ? 0

Table 12.10 Failure pattern of units

Motor Unit		*Gear Unit*	
Life (weeks)	*Cum % Freq*	*Life (weeks)*	*Cum % Freq*
4	20	10	10
8	40	12	15
12	50	14	40
16	90	16	60
20	100	18	75
		20	80
		22	100

12.6 Financial

Many financial calculations relate to the calculation of interest over a period of time. A common example involving repayment of interest (and capital) is a mortgage repayment. Once a mortgage has been obtained there is little you can do about the repayments required. A computer program, however, could be particularly useful in examining the effects of changing the variables to assist in choosing the most suitable mortgage.

12.6.1 Mortgage calculations

The repayments required on a mortgage can be calculated from the following formula:

$$R = \frac{Pi(1+i)^n}{(1+i)^n - 1}$$

where

P = Principal (the amount borrowed)
n = duration of mortgage
i = interest rate per annum
R = required annual repayment

Many organisations providing mortgages allow you to repay monthly. The monthly repayments are usually $\frac{1}{12}$ of the annual repayments because they are regarded as simply advance payments of the annual premium. These monthly advance payments do not themselves earn interest.

12.6.2 Requirements of the program

In examining alternative mortgage proposals you would want to change P, n and/or i as required. As successive changes were made it would be useful to be reminded as to the current values of these three variables.

This program is the type likely to be used by someone such as a broker in a working environment. As he is not likely to have any programming knowledge the PRINT messages need to be clear and the data entered in the most natural way. Thus the variables to be revised are indicated by entering i, P or n rather than entering a numeric alternative such as 1, 2 or 3. Although the program is

slightly more complex as a result, this is regarded as a secondary consideration.

The input to the program is straightforward; the interest rate is entered as a percentage (i.e. 12.5 not .125) as this is how it is commonly quoted.

12.6.3 Description of the program

A listing of the program is given in Table 12.11 and an example of the output in Table 12.12.

Table 12.11 Program for mortgage calculation

```
10 REM MORTGAGE REPAYMENT
30 DEF FNM(X)=INT(X*100 + .5)/100
40 INPUT"ENTER INTEREST RATE AS A %";I
60 LET I = I/100
70 INPUT"ENTER SIZE OF MORTGAGE";P
90 INPUT"ENTER PERIOD OF LOAN (YRS)";N
120 LET R=(P*I*(1+I)↑N)/(((1+I)↑N)-1)
130 PRINT"MONTHLY REPAYMENTS =";
135 PRINT FNM(R/12)
140 PRINT
160 PRINT"ENTER I,P OR N TO REVISE"
165 PRINT"INTEREST,PRINCIPAL OR YEARS"
170 PRINT:PRINT"EXISTING VALUES ARE"
175 PRINT I;P;N
180 PRINT:PRINT"OR ENTER S TO STOP"
190 INPUT A$
200 IF A$ = "S" THEN 350
210 INPUT"ENTER REVISED VALUE";X
230 IF A$ = "I" THEN 290
240 IF A$ = "P" THEN 310
250 IF A$ = "N" THEN 330
260 PRINT:PRINT"REVISION ERROR: ";
270 PRINT A$;" ENTERED":PRINT
280 GOTO 160
290 LET I = X/100
300 GOTO 120
310 LET P = X
320 GOTO 120
330 LET N = X
340 GOTO 120
350 END
```

Line 30 defines the function FNM which rounds to two decimal places thereby representing monetary amounts to the nearest pence. Lines 40 to 90 request the starting values of I, P and N. The

annual repayment is calculated in line 120 and printed as a monthly repayment in line 135. A blank line (line 140) is printed before looping and producing revised output.

Table 12.12 Example of output from Table 12.11

```
ENTER INTEREST RATE AS A % ? 12.5
ENTER SIZE OF MORTGAGE ? 10000
ENTER PERIOD OF LOAN (YRS) ? 25
MONTHLY REPAYMENTS = 109.95

ENTER I,P OR N TO REVISE
INTEREST, PRINCIPAL OR YEARS
EXISTING VALUES ARE
.125      10000      25
OR ENTER S TO STOP
? P
ENTER REVISED VALUE ? 8000
MONTHLY REPAYMENTS = 87.96

ENTER I,P OR N TO REVISE
INTEREST, PRINCIPAL OR YEARS
EXISTING VALUES ARE
.125      8000      25
OR ENTER S TO STOP
? S
```

Line 190 allows you to revise optionally the values of I, P or N (line 175 reminds you of the current values). The option you enter is identified by the program over lines 200 to 250. If an inappropriate character is entered, this character 'falls through' these lines and the error message (line 260) is printed. Otherwise the revised value entered in line 210 is assigned accordingly over lines 290 to 330. The program then loops back to line 120 to recalculate R.

Problem 6 – Monthly repayments

Run the program shown in Table 12.11 using the following data:

Interest rate, 11%; Loan, £15,000; Period of loan, 20 years

Then revise the loan to £20,000. The two monthly repayments are given in Appendix B.

12.7 Morse Trainer

This program makes use of the 64's sound facilities to 'playback' in Morse code any word entered at the keyboard. In addition, dots and dashes are displayed on the screen to match the Morse sounds.

The program is given in Tables 12.13a and 12.13b. Line 20 sets the basic unit of sustain used in subsequent delay loops. By making S smaller, the Morse will be played quicker but the *relative* timings of dots, dashes and pauses will be maintained. Line 70 transfers

Table 12.13a Morse trainer

```
10 DIM M$(26),L$(40)
20 S=15
30 PRINT"■■"
40 PRINT"            MORSE TRAINER"
50 PRINT"            -.-.-.-.-.-.-"
60 PRINT"■■"
70 GOSUB 400
75 GOSUB 1000
80 INPUT"ENTER MESSAGE ";S$
90 PRINT"■"
100 GOSUB 500
110 FOR K = 1 TO L
120 L$=L$(K)
130 GOSUB 3000
140 NEXT K
150 PRINT
160 END
400 FOR I = 1 TO 26
410 READ M$(I)
420 NEXT I
430 DATA .-,-...,-.-.,-..,.
440 DATA ..-.,--.,....,..,.---
450 DATA -.-,.-..,--,-.,---
460 DATA .--.,--.-,.-.,...,-
470 DATA ..-,...-,.--,-..-,-.--
480 DATA --..
490 RETURN
500 REM TRANSLATE
510 L=LEN(S$)
520 FOR I = 1 TO L
530 L$(I)=CHR$(32)
535 FOR J = 1 TO 26
540 IF MID$(S$,I,1)<>CHR$(64+J) THEN 550
545 L$(I)=M$(J)
550 NEXT J
560 NEXT I
570 RETURN
```

control to subroutine 400 which sets up an array M$(26) with strings of dots and dashes to represent the Morse code (in alphabetic sequence). This routine could be extended to include the full Morse code set by increasing the array size and adding suitable DATA statements.

Line 75 then calls the subroutine at line 1000 which clears and sets up the sound registers. The loop, lines 1020 to 1040, clears the registers. Line 1050 sets the volume and line 1060 POKEs attack/decay and sustain/release values that set up a 'pure' organ tone.

The message to be transmitted in Morse is input at line 80. This is examined character by character in the subroutine (lines 500–570) and as each character is identified in line 540, its Morse code equivalent (M$(J)) is copied to array L$. Each character occupies one cell in the array. The dimension statement (line 10) therefore limits the current program to messages of 40 characters.

Lines 110 to 140 in the main program then transfer the Morse code string from *array* L$ to the *variable* L$ within a loop. The current contents of the *variable* L$ then is used by the subroutine 3000 to 3100 to determine which sound subroutines should be called.

If L$ is a space, then the 'end word' subroutine (lines 7000–7030) is entered. Lines 4000 to 4070 or lines 5000 to 5070 are executed if L$ is a dot or dash, respectively. Having sounded appropriately, the 'end letter' subroutine (lines 6000–6030) is entered.

The relative timing is determined by the delay loops in the sound subroutines. Taking the dot character as a unit of duration, the time interval between characters is also equivalent to a dot. The duration of a dash is three dots. Between letters there is a pause equivalent to three dots, but as the last dot or dash has a pause of one unit within its own subroutine, the 'end letter' subroutine needs to add only two dot equivalents. The end of a letter is signified on the screen by the / symbol incorporated in the print statement at line 6010.

Similarly, as the interval between words is equivalent to *seven* dots, the subroutine (lines 7000–7300) adds a further pause equivalent to *six* dots. These timings are approximate because the time spent in the rest of the program will add a small fixed element.

The above program can be modified in several ways; for example, by making the INPUT from tape, practice tapes could be compiled by someone else so that the listener does not already know the message.

Table 12.13b Morse trainer

```
1000 REM CLEAR & SET UP SOUND REGISTERS
1010 SS=54272
1020 FOR I= SS TO SS+24
1030 POKE I,0
1040 NEXT I
1050 POKE SS+24,15
1060 POKE SS+5,0:POKE SS+6,240
1070 RETURN
3000 REM SOUND CHARACTERS
3010 IF L$<>CHR$(32) THEN 3030
3020 GOSUB 7000:GOTO 3100
3030 W=LEN(L$)
3040 FOR I = 1 TO W
3050 X$=MID$(L$,I,1)
3060 IF X$=CHR$(46) THEN GOSUB 4000
3070 IF X$=CHR$(45) THEN GOSUB 5000
3080 NEXT I
3090 GOSUB 6000
3100 RETURN
4000 REM DOT
4010 PRINT"●";
4020 POKE SS+0,75:POKE SS+1,34
4030 POKE SS+4,17
4040 FOR D = 1 TO S : NEXT D
4050 POKE SS+4,16
4060 FOR D = 1 TO S : NEXT D
4070 RETURN
5000 REM DASH
5010 PRINT"▬ ";
5020 POKE SS+0,75:POKE SS+1,34
5030 POKE SS+4,17
5040 FOR D = 1 TO 3*S : NEXT D
5050 POKE SS+4,16
5060 FOR D = 1 TO S : NEXT D
5070 RETURN
6000 REM END LETTER
6010 PRINT" / ";
6020 FOR D = 1 TO 2*S : NEXT D
6030 RETURN
7000 REM END WORD
7010 PRINT:PRINT
7020 FOR D = 1 TO 6*S : NEXT D
7030 RETURN
```

13

Using a Printer and Disk Unit

13.1 Disk units and printers

So far we have considered only a few of the possible peripheral devices that can be attached to the Commodore 64 microcomputer. Input of data has been from the keyboard with output to the screen (television set or video monitor).

In Chapter 2, we suggested that you should save your programs on cassette so that you can load them into the 64's memory at any time. You can save programs on disk in a similar way. However, it is much quicker to load a program from disk as the disk's directory allows direct access to the required program, which is then transferred to the 64's memory at high speed.

Cassettes and disks can also be used for holding data files which are separate from the programs which use them. The use of data files is explained in Chapter 14. A comprehensive file processing example is given in Chapter 15; this uses a sequential file held on disk.

Another important peripheral is a printer. This is used for both listing programs and providing printed output of results from programs.

Although attaching a single disk unit gives a useful improvement in performance, it is preferable to have two disk units for business use. A typical system for business applications will include two disk units and printer as shown in Figure 13.1.

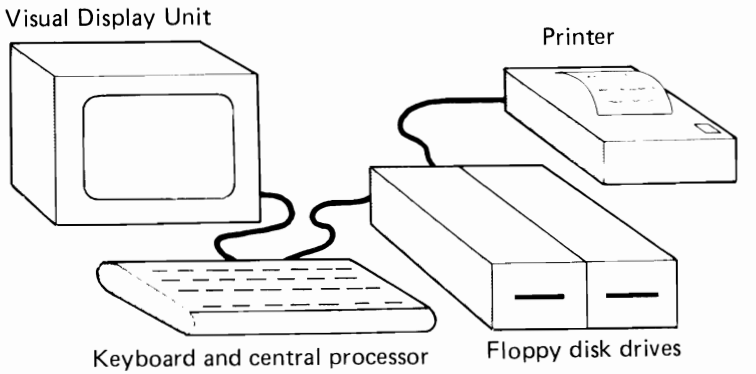


Figure 13.1 A typical business system

The disk units allow the information held on a disk to be copied easily to another disk for making back-up copies, using special disk commands. All important programs and data files should have a back-up copy, in case the original copy becomes corrupted. If this happens, the first thing you should do is to make *another* back-up copy.

Two main types of printer that are used for business applications are matrix printers and daisy-wheel printers. Matrix printers print each character as a grid of dots, while daisy-wheel printers have the character set embossed on the circumference of the print-wheel. The latter is interchangeable so that different character sets and type founts can be used. Daisy-wheel printers tend to be more expensive than matrix printers and are used for 'letter-quality' printing.

Matrix printers have a standard character set built into the printer, but special characters can be programmed such as the Σ sign shown in Figure 10.3 (page 97). However, note that printer characters are seven dots high and therefore the lowest line is omitted, and the denary values are calculated on a column by column basis. As the precise details vary from printer to printer, it is important to consult the appropriate printer manual.

Print statements can include control characters for the printer, for example to move the printer stationery to a certain position, but use of these is outside the scope of this book.

13.2 Using a printer for program listings

To obtain program listings on the printer, you need to use two commands. The first one is an OPEN command giving the file number of the program in the 64's memory which is to be listed, and the device number of the printer. This is followed by a CMD command.

In the following example, the program has been given a file number of 1, and the printer has been specified as device number 4. CMD 1 is used to leave the printer addressed and 'listening' for output from file 1, in this case:

```
OPEN 1,4:CMD 1
```

Next you type a LIST command, which can be any one of its available forms. For example, LIST will output the whole of the program, while LIST 10–200 will output lines 10 to 200, inclusively.

When you have finished listing the program, you need to stop the printer listening and close the program file by typing:

```
PRINT#1:CLOSE 1
```

13.3 Output to the printer from a program

A program which outputs to the printer is shown in Tables 13.1a and 13.1b. This is an enhanced version of the simple program given in Table 2.2 (page 12), and gives the user a choice of printing letter headings, notebook labels or envelope labels. The enhancements include:

A menu of options	(lines 60–130)
Variable parameters for each option	(lines 210–240, 410–420, 710)
Centering the name in a border of asterisks	(lines 430–620)

Cursor control characters have been used in INPUT and PRINT statements. For example, line 10 clears the screen and moves one line down the screen; line 20 moves one line down. A list of cursor control characters that you can use in INPUT messages and PRINT strings is given in Appendix E. The cursor down character, shown as a reverse video Q in the listing, is an alternative to using PRINT on

its own. For example, line 80 in Table 13.1a causes two blank lines to be left on the screen, and could be replaced by inserting two cursor down characters at the beginning of the PRINT string (that is, just before the 1) in line 90.

Table 13.1a Enhanced name and address program

```
10 INPUT"□TITLE";T$
20 INPUT"NAME";N$
30 INPUT"ADDRESS 1";A$
40 INPUT"ADDRESS 2";B$
50 INPUT"ADDRESS 3";C$
60 PRINT"□    SELECT OPTION"
70 PRINT"     -------------"
80 PRINT:PRINT
90 PRINT"1  LETTER HEADING"
100 PRINT"2  NOTEBOOK LABELS"
110 PRINT"3  ENVELOPE LABELS"
120 PRINT"4  END PROGRAM"
130 INPUT"OPTION";C
140 ON C GOSUB 200,400,700,800
150 GOTO 60
200 REM LETTER HEADING
210 INPUT"□NUMBER OF SHEETS";N
220 INPUT"PAGE LENGTH";P
230 INPUT"TOP MARGIN";T
240 INPUT"START POSITION";S
245 OPEN 1,4
250 FOR I=1 TO N
260 FOR J=1 TO T:PRINT#1:NEXT J
270 PRINT#1,TAB(S);A$
280 PRINT#1,TAB(S);B$
290 PRINT#1,TAB(S);C$
300 FOR J=1 TO P-T-3:PRINT#1:NEXT J
310 NEXT I
315 CLOSE 1
320 RETURN
```

The output to the printer is achieved by specifying PRINT#1 as the print command, having opened file 1 to the printer in lines 245, 425 and 715. At the end of each print option, file 1 is closed (lines 315, 625 and 785).

Note output to the screen is by means of ordinary PRINT statements (for example, lines 60–120). The INPUT messages also appear on the screen (for example, lines 10–50 and line 130).

When outputting to the printer, you should use a SPC (space) function instead of TAB, after the first TAB in a print statement.

Table 13.1b Enhanced name and address program

```
400 REM NOTEBOOK LABELS
410 INPUT"■■NUMBER OF LABELS";N
420 INPUT"■START POSITION";S
425 OPEN 1,4
430 L=LEN(N$)
440 FOR I=1 TO N
450 PRINT#1,TAB(S);"************************"
460 PRINT#1,TAB(S);"*                      *"
470 PRINT#1,TAB(S);"*                      *"
480 SP=INT((20-L)/2)
490 PRINT#1,TAB(S);"*";
500 FOR K=1 TO SP
510 PRINT#1," ";
520 NEXT K
530 PRINT#1,N$;
540 FOR K=1 TO 20-L-SP
550 PRINT#1," ";
560 NEXT K
570 PRINT#1,"*"
580 PRINT#1,TAB(S);"*                      *"
590 PRINT#1,TAB(S);"*                      *"
600 PRINT#1,TAB(S);"************************"
610 PRINT#1:PRINT#1:PRINT#1
620 NEXT I
625 CLOSE 1
630 RETURN
700 REM ENVELOPE LABELS
710 INPUT"■■TOTAL NUMBER OF LABELS(EVEN)";N
715 OPEN 1,4
720 FOR I=1 TO N/2
730 PRINT#1,TAB(10);T$;" ";N$;
735 PRINT#1,SPC(34-LEN(N$+T$));T$;" ";N$
740 PRINT#1,TAB(10);A$;SPC(35-LEN(A$));A$
750 PRINT#1,TAB(10);B$;SPC(35-LEN(B$));B$
760 PRINT#1,TAB(10);C$;SPC(35-LEN(C$));C$
770 PRINT#1:PRINT#1:PRINT#1
780 NEXT I
785 CLOSE 1
790 RETURN
800 END
```

When using SPC, it is important that the number of spaces moved takes into account the length of any previously printed output on that line. One way of achieving this is shown in line 740, in which SPC(35-LEN(A$)) 'spaces' from the end of A$ to the thirty-fifth print position from the start of A$.

A new statement has been introduced in line 140, the ON . . . GOSUB statement. This works in a similar way to the ON . . .

GOTO statement discussed in Chapter 5, but branches to *subroutines* according to the value of the expression or variable after the word ON (that is, C in the example in line 140).

Line 140 causes a branch to line 200, 400, 700 or 800 depending on whether 1, 2, 3 or 4 has been entered in response to line 130 for variable C. For example, if 1 is entered from the keyboard in response to OPTION? (line 130), then a branch will be made to line 200 and lines 210 to 315 will be executed. When line 320 (RETURN) is reached, the program will branch to the line following the ON . . . GOSUB statement, that is, line 150 which branches back to line 60 to select the next option. To end the program run, 4 is entered as the option and a branch is made to 800 (END), so no RETURN statement is required.

For some applications, you may want to give a choice of outputting to the screen *or* to the printer. The screen is device 3, so by using a variable for the device number in the OPEN statement, the output device can be selected at run time. For example, the following statements will cause output to go to the screen or to the printer, depending on whether 3 or 4 is entered in response to line 10:

```
10 INPUT "ENTER 3 FOR SCREEN OR 4 FOR
PRINTER";N
20 OPEN 2,N
30 PRINT#2, TAB(12);"REORDER LIST"
```

In this example, the output file has been designated as file 2, and needs to be closed at the end of the program by, say,

```
100 CLOSE 2
```

If you are writing a program for a business user, it may be preferable to use codes rather than numbers for the devices in the INPUT statement. For example:

```
10 INPUT "OUTPUT TO SCREEN OR PRINTER,S/P";A$
20 IF A$ = "S" THEN D = 3
30 IF A$ = "P" THEN D = 4
40 IF A$ <> "S" AND A$ <> "P" THEN 10
50 OPEN 2,D
```

In this case, if neither S nor P has been entered, as tested in line 40, the program branches back to line 10 and the user is prompted again.

13.4 Disk commands

A range of commands exist for communicating with the disk unit and processing disk-based programs. Some, such as LOAD, SAVE and VERIFY are directly equivalent to cassette commands; others relate to manipulating programs on the disk (for example, COPY, RENAME), while others control the disk system (for example, to obtain an error message).

LOADing, SAVEing and VERIFYing

When a disk is inserted into the drive, programs on the disk may be loaded into the 64 by using the LOAD command in the following format:

LOAD *"program name"*,8

where 8 is the device number of the disk drive (preset by CBM). For example:

```
LOAD "MORSE",8
```

will load a program named MORSE from the disk drive.

When the disk drive contains many programs, the precise program name may have been forgotten. It is possible to load the disk directory into the 64 by using $ in place of the program name. Typing LIST will then display the disk directory on the screen, for example:

```
LOAD "$",8
FOUND $
LOADING
READY
LIST
```

A program can be saved on a disk by a similar command, i.e.

SAVE *"program name"*,8

When a program has been saved, it is good practice to verify the saved version. This is done by:

VERIFY *"program name"*,8

OPEN and PRINT

The 64 can communicate with the disk drive via command channel 15. This channel first needs to be OPENed by:

OPEN *filenumber*,8,15

For example:

```
OPEN 15,8,15
```

The file number can be from 1 to 255 but it is convenient to use 15 when using command channel 15.

Once the channel has been OPENed commands can be sent to the disk drive via a PRINT instruction as follows,

PRINT#*filenumber*, "*command*"

The format of the commands that can be sent will be discussed in the next sections. In these sections it will be assumed that OPEN 15,8,15 has already been used.

NEW

This command erases an entire disk and sets it up to be used afresh. A brand new *blank* disk must be NEWed before it can be used by a 64. The format of the command is:

PRINT#15,"NEW *drive number:name, id*"

where

drive number	= 0 for a single disk drive
name	= the required name of the disk
id	= any two-character identifier

For example:

```
PRINT#15, "NEW 0:DEMO DISK, 01"
```

In all subsequent examples it will be assumed that the relevant disk drive is number 0.

COPY

This command allows a program to be copied under a different name. The format is:

PRINT#15, "COPY 0:*new file* = 0:*file1*"

For example:

```
PRINT#15, "COPY 0:MAR = 0:JAN"
```

RENAME

This command allows an existing disk file name to be changed. The format is:

PRINT#15,"RENAME 0:*new name = old name*"

For example:

```
PRINT#15, "RENAME 0:PHONE = TEL"
```

SCRATCH

This command allows you to erase unwanted files from the disk. The format is:

PRINT#15, "SCRATCH 0:*file name*"

For example:

```
PRINT#15, "SCRATCH 0:TEL"
```

INITIALISE

The use of this command restores the disk drive to its initial powered-up state. This is useful if an error condition has arisen, causing the current command to be inoperable. The format is:

PRINT#15, "INITIALISE"

Error conditions

If the error light on the disk drive comes on, the error condition can be cleared and an error message displayed by running the following routine:

```
10 OPEN 15,8,15
20 INPUT#15,A$,B$,C$,D$
30 PRINT A$,B$,C$,D$
```

To obtain a fuller explanation of the error message the disk user's manual should be consulted.

Abbreviating commands

All the commands within the PRINT#15 statement can be abbreviated to their initial letter, for example:

```
PRINT#15, "R0:PHONE = TEL"
```

Pattern matching and wild carding

When specifying file names, use can be made of two special symbols. These are the asterisk and the question mark. Incorporating a question mark in the file name of a disk command means that the character in that position is irrelevant. For example:

```
PRINT#15, "SCRATCH 0:C?M"
```

would scratch all three character file names that began with C and ended with M(i.e. CBM, CAM, COM, etc.).

When an asterisk is used as the last character of a file name it implies that the rest of the name is irrelevant. Thus

```
PRINT#15, "SCRATCH 0:C*"
```

would scratch all files whose name began with C.

DOS support program

A demonstration disk is usually supplied with the disk drive. On that disk there is likely to be a DOS (Disk Operating System) support program that simplifies the use of the disk unit. The program is usually named "C-64 WEDGE".

When this program is loaded and run, the divide key (/) replaces the LOAD command, for example:

```
/ MORSE
```

Note also that the device number (8) is not required.

Instead of PRINT#15, the 'greater than' key (>) is used. For example:

```
>S0:TEL
```

will scratch the file named TEL.

Keying >$ will display the disk directory on the screen without affecting the program in memory.

If the disk error light comes on, the error condition can be cleared and an error message displayed by simply keying >.

The DOS support program is so useful that it is a good idea to load it as a matter of routine whenever disks are used. However, it should be noted that this program may interfere with the correct reading and writing of data with cassette files.

14

Using Data Files

14.1 Data files

When a large amount of common data is required by a program it is inconvenient to enter this data each time via the keyboard. A preferable method is to store the data in DATA statements within the program, as described in Chapter 3. However, this is still restrictive as these DATA statements are not readily available to other programs. The most flexible approach is to store your data in separate files from your programs so that the data files may be used by more than one program. You can then also prepare standard programs to analyse and process different data set up in data files.

A data file is created by a BASIC program so that the contents and format are under your control. In practice this means you are likely to write several programs – for example, one to create the data file, one to update the data file, and some to process the data. This chapter shows how such data files may be created and read.

There are two ways of processing files: sequentially and randomly. Random access of files can only be carried out from disks, whereas sequential access can be carried out from disks or magnetic tape. This book deals only with sequential file systems.

14.2 File records

The contents of a data file may be regarded as the equivalent of a series of DATA statements within a program. Although the data consists of one long 'column' of values, it is useful for you to think

and design the logic of your program round the concept of records. For example, a stock record might consist of a stock number, item description, stock level, unit cost, reorder level and order quantity as shown in Table 14.1.

Table 14.1 Stock records

Stock No	*Description*	*Stock*	*Unit Cost*	*Reorder Level*	*Order Quantity*
1234	Pens	15	45	20	60
2340	Pencils	50	12	40	100
2679	Erasers	8	5	10	50
3456	Rulers	20	26	30	100
4567	Writing pads	40	35	50	200
4568	Notebooks	60	40	30	100
6770	Labels	70	15	25	75
6775	Pins	40	15	20	60
6979	Envelopes	40	20	60	200
7050	Cash books	30	22	40	100

The data contained in Table 14.1, recorded sequentially record by record, would give rise to a 'column' of values as shown below:

1234
PENS
15
45
20
60
2340
PENCILS
50
etc.

In transferring this data to and from memory it is more convenient to assign separate variable names to each part of a record and move one record at a time. This keeps the program logic simpler, although within a particular program a variable (unit cost, say) may not be manipulated or used.

As the program examples in this chapter use the data shown in Table 14.1, this is a convenient place to define the variable names to be used:

K = stock number
D$ = description
S = stock
C = unit cost
R = reorder level
Q = order quantity

14.3 OPEN and CLOSE statements for data files

All data files used by your program need to be declared before they are used. Prior to reading or writing to a data file, the file needs to be 'opened' by means of an OPEN statement. When all the information has been passed between the file and the 64, the file then needs to be 'closed' by means of a CLOSE statement.

A separate OPEN statement is required for each file to be used in a program, and each file must be identified by a different file number. The syntax of the OPEN statement varies slightly according to whether the file is on cassette or disk.

OPEN statement for cassette files

This has the general form:

line number OPEN *x, y, z, "filename"*

where *x, y, z* are file parameters as follows:

x = the file number, as chosen by the user for this particular program
y = the number of the device containing the file, which is 1 for the 64's cassette recorder
z = an indicator stipulating whether the file is to be opened to be read, or to be written to

For example:

```
10 OPEN 1,1,1,"STK-DATA"
```

indicates that a file named "STK-DATA" will be referred to subsequently as file 1 and is located on device number 1. If $z = 1$, this indicates the file is to be written to, while $z = 0$ is used for a file which is to be read.

The default option, OPEN *x*, means that the next file encountered will be opened to read only and will be assigned the number *x*.

OPEN statement for disk sequential files

This has the general form:

line number OPEN *x,y,c, "d:filename,type,direction"*

where

x = the file number, as chosen by the user
y = the number of the device containing the file (the disk unit is device number 8)
c = a data channel number (any number from 2 to 14)
d = the disk drive number, which is zero for single disks
type = S for sequential files
direction = R for reading a file or W for writing to a file

For example:

```
10 OPEN 2,8,2,"0:STOCK,S,R"
```

indicates that the file named "STOCK" will be referred to subsequently as file 2 and is located on a disk in drive 0 of device number 8. The file is a sequential file and is to be opened for reading.

The part of the OPEN statement within quotes can be manipulated within a program using string variables, for example:

```
10 INPUT "FILE NAME";F$
20 OPEN 2,8,2,"0:" +F$+",S,R"
```

CLOSE statement for cassette and disk files

Regardless of the device being used for file processing, the data file is closed as described for print files in Chapter 13, that is,

line number CLOSE *x*

where

x = the chosen file number

For example:

```
100 CLOSE 1
110 CLOSE 2
```

14.4 File input–output statements

The statement used to output a file has the general form:

PRINT#*filenumber, variable list*

For example:

```
PRINT#1,K,D$,S,C,R,Q
```

The comma as a delimiter is suppressed so that in the above example K,D$,S,C,R,Q, is written as one string. You can preserve the variables separately by using separate PRINT statements, that is:

```
PRINT#1,K
PRINT#1,D$
PRINT#1,S
PRINT#1,C
PRINT#1,R
PRINT#1,Q
```

or you can retain the commas by enclosing them in quotes:

```
PRINT#1,K;",";D$;",";S;",";C;",";R;",";Q
```

The corresponding read statement is:

INPUT#1, *filenumber, variable list*

Thus a corresponding read statement might be:

```
INPUT#1,K,D$,S,C,R,Q
```

14.5 End of file records

It is convenient if you have within your program your own means of detecting the end of the data. This can easily be done by terminating your data files with a dummy record. The contents of this dummy record are chosen to make it unique. For example, in the stock record file previously discussed the dummy stock number could be made larger than any likely to be encountered (i.e. 9999 if four-digit codes are used).

Since a complete record is transferred as a whole, the remaining fields of the dummy record need to be provided with values as shown below:

9999,X,0,0,0

The general flow of processing when a dummy record is used is shown in Figure 14.1.

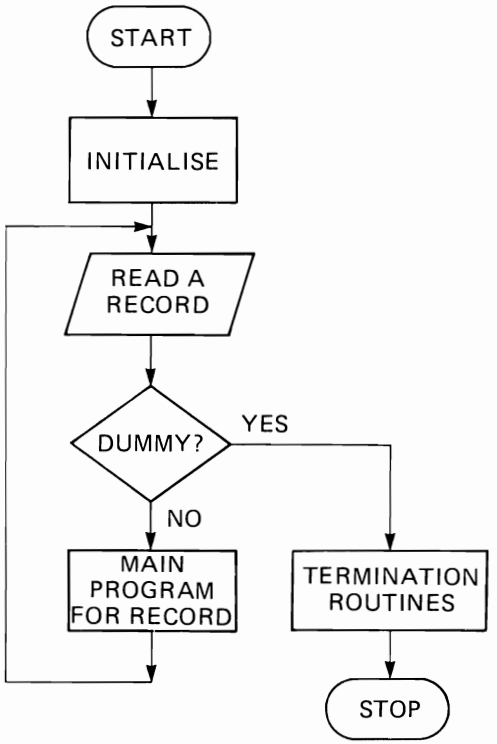


Figure 14.1 General flow with dummy record

14.6 Problems

Problem 1 – Stock data file
Write a program to create a stock data file on cassette incorporating a dummy end record for the data in Table 14.1.

A suitable program is listed in Table A19.

Problem 2 – Reorder list
Write a program to read the data file produced in Problem 1 and output a list of items to be reordered optionally to the screen or to a printer. A sample output is shown in Table 14.2.

Table 14.2 Output from reorder program

```
                REORDER LIST
                ------------

CODE       DESCRIPTION       ORDER QTY
----       -----------       ---------

1234        PENS                  60
2679        ERASERS               50
3456        RULERS               100
4567        WRITING PADS         200
6979        ENVELOPES            200
7050        CASH BOOKS           100
```

A suitable program is listed in Table A20.

Problem 3 – A data file search program

Write a program using string functions to search the stock data file produced in Problem 1 for any stock description containing a specified substring (e.g. PEN). An example of the output is shown in Table 14.3.

Table 14.3 Output from search program

```
STOCK FILE SEARCH
.....
ENTER SEARCH WORD ? PEN
PRESS PLAY ON TAPE
OK
.....................................................
  CODE        DETAILS        STOCK
.....................................................
   1234     PENS                 15
   2340     PENCILS              50
.....................................................
10 RECORDS READ
2 RECORDS LISTED
```

Allow for the stock details to be output either to the screen or to the printer. The program is listed in Table A21.

Notes

1 As mentioned in Chapter 2, the screen will go blank while the 64 is communicating with the cassette unit; that is, when a program writes to a tape or reads from it.

2 You must not have the disk operating system (DOS) loaded into the 64 when using data files on cassettes.

15

File Processing and Reporting Example

15.1 Introduction

This chapter describes a stock recording program which is a development of the problem and programs described in Chapter 14. This program also uses the same data. Note however that the data is entered in a different order from that shown in Table 14.1.

The program illustrates the application of many of the programming techniques discussed in this book, and the basic structure of the program is relevant to other applications. The program is menu driven, with all the options being written as subroutines. This will allow further options to be added easily at a later date. A discussion of ways in which the program might be developed is presented at the end of the chapter.

It should be emphasised that this program is given to illustrate programming techniques; it is not intended to be a commercially usable system. A sequential file is used in the example so that it can be used with a cassette unit as well as a disk unit.

15.2 The menu

An illustration of the menu is given in Figure 15.1.

The appropriate command (option) is selected by keying in A, C, D, E, R, S, U or V; that is, the initial letter of the required command. A brief description of each command is given below:

ADD This allows the user to add (that is, create) a further stock record.

CHANGE This displays an existing stock record and allows any part of its contents to be changed.

DELETE This allows a stock record to be deleted. It is not deleted from the computer's memory, but instead, the stock description is replaced by the word 'DELETED'. This ensures that it is not saved to the updated file.

EXIT This terminates the program. Before terminating, the user is given the option of saving the current set of records to tape or disk. Any deleted records will not be saved to the new file.

REORDER REPORT This produces a stock list of items currently below their reorder level. The list can be displayed on the screen or sent to the printer.

SEARCH This allows the user to search the stock file for a sub-string match in the stock description field. The output can be directed to the screen or to the printer.

UPDATE STOCK LEVELS This allows the stock quantity for a specified stock record to be updated.

VALUATION REPORT This produces a complete stock list, to the screen or printer, giving the value of each stock quantity and a cumulative total for the whole stock holding.

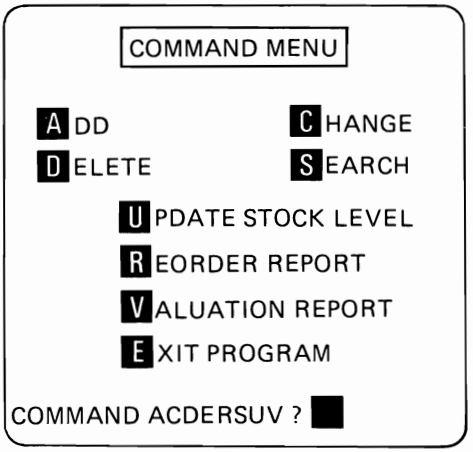


Figure 15.1 Command menu for stock recording program

15.3 Structure of the program

All the above menu options have been written as subroutines. In addition, a few routines that are common to these subroutines have been written as 'utility' subroutines. For example, at the end of each option the user is given the choice of repeating the current option or returning to the menu, that is:

CONTINUE OR MENU ?

Details of all the stock records are held in *arrays*. If a stock file already exists, it is read into the arrays during the 'start up' phase. Stock record changes are made to the array variables, the revised arrays being saved to a file when the EXIT option is chosen.

An outline flowchart is given in Figure 15.2.

A stock record consists of seven fields. They are all numeric fields, except the stock description, and none of them are of fixed length. The fields are as follows:

Stock number: A part or commodity number
Description: Any alphanumeric description
Unit cost: The cost of one unit of stock quantity. The monetary units should be consistent throughout the file.
Order quantity: The standard reorder quantity
Reorder level: Items below this level will appear in the reorder list
Stock quantity: The current stock level. This field can be updated by +/− transactions (i.e. by entering a number preceded by a + or − sign).
Stock value: The content of this field is calculated by the program and automatically updated whenever the unit cost or stock quantity is changed.

Although none of the data is held in fixed length field format, it is reformatted prior to some listings. These are the listings that might be sent to the printer, i.e. REORDER REPORT, SEARCH and VALUATION REPORT. The reformatting is done by a subroutine that converts all the numeric fields to eight character strings padded with leading spaces. A further subroutine is called to ensure that financial amounts are presented to two decimal places with the decimal point aligned. The stock description field is formatted to 15

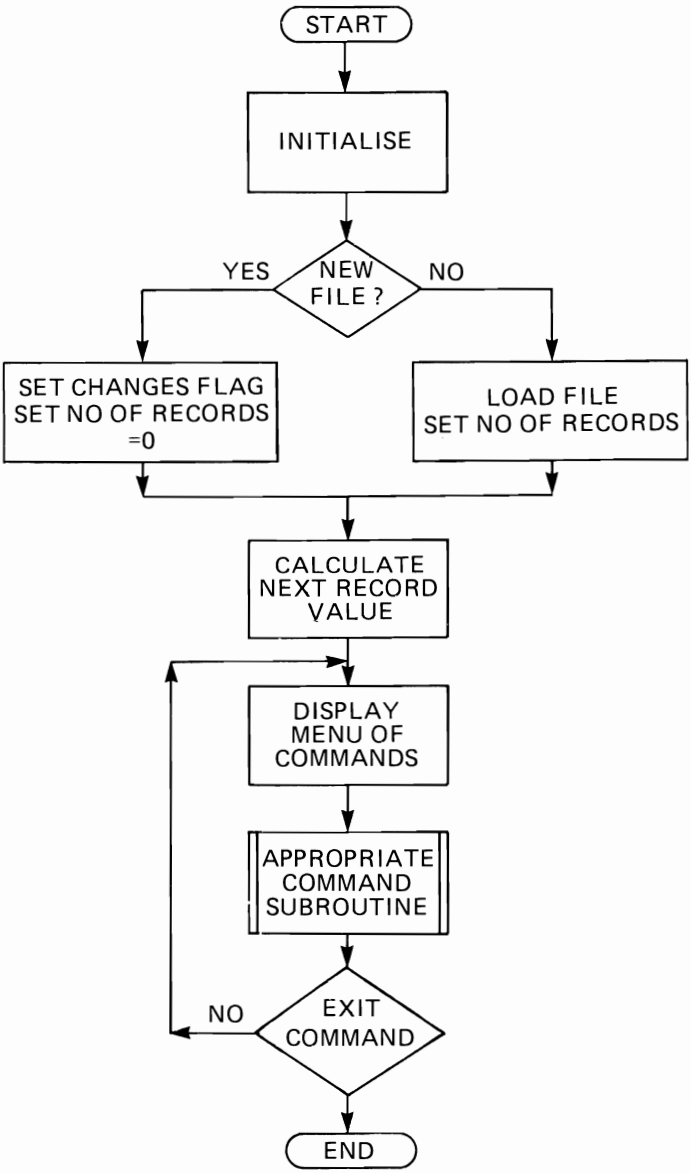


Figure 15.2 Outline flowchart

characters. This allows the description field plus any three other fields to be concatenated to produce a 39-character line of output. If 40 characters were used, the cursor would move to the next line on the screen, thereby producing a blank line between each line output.

The program automatically assigns a record number to each stock record, which is, in fact, the array subscript for that record. This record number is not used in any way by the user. The user always references a stock record by its stock number. If the stock number is not known, or has been forgotten, then the SEARCH option on the description field will allow the user to find the appropriate stock number.

15.4 Printed output

Printed output is available for three options: SEARCH, RE-ORDER REPORT and VALUATION REPORT.

The SEARCH option produces a list of all stock items whose description contains the string of characters specified by the user. For each stock item listed, the stock number, description and stock quantity is given. At the end of the listing there is a note of the total number of records searched and a count of those listed. An example of a printed SEARCH listing is given in Table 15.1.

Table 15.1 Example of output for search option

```
        SEARCH FOR BOOKS
------------------------------

  STK NO   DESCRIPTION    STK QTY
------------------------------
    4568   NOTE BOOKS          60
    7050   CASH BOOKS          30
------------------------------
 10 RECORDS SEARCHED
 2 RECORDS LISTED
```

This option is most likely to be used to find the stock number of an item from its description. The stock quantity is also listed because this is the next most likely reason for accessing a stock record. The number of records searched is given to 'reassure' the user that the whole file has been examined, particularly in the case where no string match has been found.

The REORDER REPORT comprises all those stock items whose current stock quantity is below the reorder level. For each item listed the specified order quantity is given. The program also calculates, from the unit cost and the order quantity, the order cost for each item and cumulates these costs to give a total cost for the complete reorder list. An example of a REORDER REPORT is shown in Table 15.2.

Table 15.2 Example of output for reorder report

REORDER REPORT

STK NO	DESCRIPTION	ORD QTY	COST
1234	PENS	60	2700.00
2679	ERASERS	50	250.00
3456	RULERS	100	2600.00
4567	WRITING PADS	200	7000.00
6979	ENVELOPES	200	4000.00
7050	CASH BOOKS	100	2200.00
	TOTAL		18750.00

The VALUATION REPORT lists all the items in the stock record and presents the current stock quantity together with the value of the stock. A cumulative total of the complete stock value is given at the end of the list as shown in Table 15.3.

Table 15.3 Example of output for valuation report

VALUATION REPORT

STK NO	DESCRIPTION	STK QTY	VALUE
1234	PENS	15	675.00
2340	PENCILS	50	600.00
2679	ERASERS	8	40.00
3456	RULERS	20	520.00
4567	WRITING PADS	40	1400.00
4568	NOTE BOOKS	60	2400.00
6770	LABELS	70	1050.00
6775	PINS	40	600.00
6979	ENVELOPES	40	800.00
7050	CASH BOOKS	30	660.00
	TOTAL		8745.00

15.5 Initialisation and start-up routine

The initialisation and start-up routine forms the start of the program and this part of the program is shown in Table 15.4. Lines 100 to 160 set up the headings for each field in the stock record. The headings are assigned to elements 1 to 7 in the array H$. To assist in neat screen presentations, all the fields are a fixed 18 characters in length. A further variable, B$, is set up as a string of spaces. This variable will be used to pad out strings in the 'format fields' subroutine (lines 8200–8290, Table 15.11).

Table 15.4 Initialisation and start-up routine

```
10 DIM SN(100),D$(100),UC(100),OQ(100)
20 DIM ROL(100),SL(100),V(100)
100 H$(1)="STOCK NUMBER      "
110 H$(2)="DESCRIPTION       "
120 H$(3)="UNIT COST         "
130 H$(4)="ORDER QUANTITY    "
140 H$(5)="REORDER LEVEL     "
150 H$(6)="STOCK LEVEL       "
160 H$(7)="STOCK VALUE       "
200 B$="                     "
300 PRINT"□":PRINT:PRINT
310 PRINT"    ■N■EW FILE":PRINT
320 PRINT"OR ■L■OAD EXISTING FILE":PRINT:PRINT
330 INPUT A$
340 IF A$<>"N" AND A$<>"L" THEN 300
350 IF A$="N" THEN N=0
360 IF A$="L" THEN GOSUB 5500:REM LOAD FILE
370 NR=N+1:DC=0
```

Lines 300 to 370 form a start up routine that initialises file record pointers. The user is asked whether the file to be processed is new or is to be loaded as an existing file (lines 310–330). If the file is new, then the number of records (N) is set to zero in line 350. If an existing file is to be loaded, line 360 goes to the file loading subroutine (GOSUB 5500). On returning from the subroutine, the number of records will have been assigned to N. Line 370 completes the initialisation by setting NR (next record variable) to N + 1 and setting DC to zero. DC is a variable that represents the number of records deleted during the course of the run (i.e. delete count).

15.6 File load routine

Table 15.5 File load routine

```
5500 PRINT"□":REM FILE LOAD ROUTINE
5510 PRINT
5520 INPUT"CASSETTE OR DISK FILE, C/D";F$
5530 IF F$="C" THEN D=1
5540 IF F$="D" THEN D=8
5550 IF F$<>"C" AND F$<>"D" THEN 5520
5560 PRINT
5570 INPUT"ENTER FILE NAME";FT$
5580 IF D=1 THEN OPEN 1,1,0,FT$
5590 IF D=8 THEN OPEN 1,8,2,"0:"+FT$+",S,R"
5592 PRINT
5595 PRINT"FOUND ";FT$
5610 INPUT#1,N
5612 PRINT
5614 PRINT"READING";N;"RECORDS"
5620 FOR I=1 TO N
5630 INPUT#1,SN(I),D$(I),UC(I),OQ(I),ROL(I),SL(I)
5635 V(I)=UC(I)*SL(I)
5640 NEXT I
5650 CLOSE 1
5670 RETURN
```

The file load routine is given in Table 15.5. On entering the file load routine the user is asked to input the storage device code (e.g. C for cassette or D for disk) and then the file name. Line 5580 opens the file for reading if it is on tape, while line 5590 opens a disk file for sequential reading. Once opened, the first data item read from the file is a value for N, the total number of records. A loop is then set up between lines 5620 and 5640 to read N sets of records into six arrays. The six variable names are:

SN(I)	Stock number
D$(I)	Description
UC(I)	Unit cost
OQ(I)	Order quantity
ROL(I)	Reorder level
SL(I)	Stock quantity (i.e. level)

A seventh variable V(I), stock value, is calculated within the loop, thereby ensuring that the stock value held in memory is based upon

the latest situation. After reading in all the records, the subroutine returns to the start-up routine at line 370 as previously described.

15.7 Main program and menu

The main program and menu subroutine are shown in Table 15.6.

Table 15.6 Main program and menu subroutine

```
400 GOSUB 1200:REM MENU
410 PRINT"□"
420 ON OP GOSUB 1500,2500,2000,6000,4000,3000,3500,4500
430 IF OP<>4 THEN 400
450 END

1200 REM MENU
1210 PRINT"□"
1220 PRINT"              ■COMMAND MENU■"
1230 PRINT
1240 PRINT"        ■A■DD              ■C■HANGE":PRINT
1250 PRINT"        ■D■ELETE           ■S■EARCH":PRINT
1260 PRINT"            ■U■PDATE STOCK LEVEL":PRINT
1270 PRINT"            ■R■EORDER REPORT":PRINT
1280 PRINT"            ■V■ALUATION REPORT":PRINT
1290 PRINT"            ■E■XIT PROGRAM":PRINT
1300 PRINT"COMMAND ACDERSUV";
1310 INPUT A$
1320 OP=0
1330 FOR I=1 TO 8
1340 IF A$=MID$("ACDERSUV",I,1) THEN OP=I
1350 NEXT I
1360 IF OP<1 OR OP>8 THEN 1200
1370 RETURN
```

The main program only occupies five lines, 400 to 450. The first line (400) calls the menu subroutine which return a value for OP depending upon the option selected. The main program in line 420 then branches to the appropriate option subroutine. Having completed the option the program returns to line 430. The exit option would have set OP to equal 4, therefore if OP does not equal 4 the program loops back to line 400 and presents the menu once again. If the exit option has been selected and therefore OP does equal 4, then the program goes instead to line 450 which halts execution.

The subroutine starting at line 1200 displays the menu and sets the value of OP according to the option selected. Lines 1220 to 1300 are

PRINT statements that set up the screen display of menu options. These options are referred to as 'commands' on the menu and the user makes his selection by entering the initial letter (i.e. A,C,D,E,R,S,U or V). Line 1340 compares the input from the user with successive letters in the string 'ACDERSUV' and sets the variable OP accordingly (e.g. OP = 1 for A, OP = 2 for C, etc.). Finally, line 1360 checks that the value allocated to OP lies between 1 and 8; if not, the program returns to the start of the subroutine at line 1200.

The subroutines associated with the menu options will be described in alphabetic order.

Add record routine

Table 15.7 Add record routine

```
1500 PRINT"■": REM ADD RECORD
1510 PRINT"■RECORD NO■";NR:PRINT:PRINT
1520 PRINTH$(1);:INPUT SN:PRINT
1530 PRINTH$(2);:INPUT D$:PRINT
1540 PRINTH$(3);:INPUT UC:PRINT
1550 PRINTH$(4);:INPUT OQ:PRINT
1560 PRINTH$(5);:INPUT ROL:PRINT
1570 PRINTH$(6);:INPUT SL:PRINT
1580 INPUT"ACCEPT DATA Y/N";A$
1600 IF A$<>"Y" THEN 1630
1605 XX=NR
1610 GOSUB 8000:REM SET UP RECORD
1620 NR=NR+1
1630 GOSUB 7200
1640 IF A$="C" THEN 1500
1650 RETURN

8000 REM SET UP RECORD
8010 SN(XX)=SN
8020 D$(XX)=D$
8030 UC(XX)=UC
8040 OQ(XX)=OQ
8050 ROL(XX)=ROL
8060 SL(XX)=SL
8070 V(XX)=SL(XX)*UC(XX)
8080 RETURN

7200 REM CONTINUE
7205 PRINT
7210 INPUT"■C■ONTINUE OR ■M■ENU";A$
7220 RETURN
```

The add record subroutine is shown in Table 15.7, together with two associated subroutines. The add record subroutine starts by displaying the record number. Although this is not directly used by the user, it is useful to know the size of the file when adding further records. Lines 1520 to 1570 display the stock record headings in turn and require the user to make an appropriate entry. When all the data for a new stock record has been input, line 1580 asks the user whether the data should be accepted. The data has been stored in temporary variables. If the data is accepted, then the record number (NR) is transferred to a temporary variable XX prior to the program going to subroutine 8000 that sets the record up in the required array elements. Having set up the record, the record counter is increased by 1 in line 1620. Another subroutine is then called that determines whether the user wishes to continue using this option (i.e. adding records) or to return to the menu. The subroutine is called in line 1630 and returns the variable A$ as a character C or M. If A$ = "C" then the subroutine is repeated from line 1500, otherwise control passes back to the main program (to line 430).

When the additional record data is not accepted in line 1580, then line 1600 causes the program to go to line 1630 without entering the temporary data into array elements. The user can then either repeat the option and enter acceptable data or return to the main menu.

It should be noted that the stock value is calculated within the 'set up record' subroutine in line 8070. The stock value is therefore only calculated and stored once the data has been accepted.

Change record routine

Table 15.8 Change record routine

```
2500 PRINT"□":REM CHANGE RECORD
2510 INPUT"STOCK NO TO CHANGE";SN
2540 FOR I=1 TO NR-1
2550 IF SN(I)<>SN THEN 2740
2560 PRINT:PRINT"RECORD NO";I:PRINT
2570 PRINTH$(1);SN(I);
2575 L=LEN(STR$(SN(I)))+2
2580 GOSUB 7600:INPUT SN
2590 PRINTH$(2);" "+D$(I);
2595 L=LEN(D$(I))+2
2600 GOSUB 7600:INPUT D$
2610 PRINTH$(3);UC(I);
2615 L=LEN(STR$(UC(I)))+2
```

```
2620 GOSUB 7600:INPUT UC
2630 PRINTH$(4);OQ(I);
2635 L=LEN(STR$(OQ(I)))+2
2640 GOSUB 7600:INPUT OQ
2650 PRINTH$(5);ROL(I);
2655 L=LEN(STR$(ROL(I)))+2
2660 GOSUB 7600:INPUT ROL
2670 PRINTH$(6);SL(I);
2675 L=LEN(STR$(SL(I)))+2
2680 GOSUB 7600:INPUT SL
2690 PRINT
2700 INPUT"ACCEPT DATA Y/N";A$
2720 IF A$<>"Y" THEN 2740
2725 IF D$(I)="**DELETED**"AND D$<>"**DELETED**"THEN DR=DR-1
2730 V(I)=SL(I)*UC(I)
2735 XX=I:GOSUB 8000
2740 NEXT I
2745 GOSUB 7200
2750 IF A$="C" THEN 2500
2760 RETURN

7600 REM CURSOR MOVE
7610 FOR Z=1TOL:PRINT"■";:NEXT:RETURN
```

The change record subroutine is given in Table 15.8, together with the cursor move subroutine. To change the details within an existing stock record the user has to specify the stock number (line 2510). A FOR . . . NEXT loop is then executed over lines 2540 to 2740 to search sequentially through the stock number array SN. If no match is found in line 2550 a jump is made to NEXT I to increase the array element, I, by one. When a match is found, the statements within the loop are executed. None of these statements cause a jump out of the loop, so that even when a match has been found and acted on, the search for similar records with the same stock number continues to the end of the file. Although there should not be two stock records with the same stock number, if such a situation arose, this subroutine would highlight the fact and the situation could be remedied by using either the change record option or the delete record option.

Within the loop, each field heading is displayed in turn together with the current contents for that record (e.g. line 2570). The length of the field is evaluated (e.g. line 2575) and subroutine 7600 is then used to reposition the cursor over the initial character. Pressing the return key re-enters the existing values into a temporary variable;

alternatively any changes made prior to pressing the return key will be input to the temporary variable (e.g. line 2580). Each field is treated in a similar manner until finally line 2700 is reached. If the 'data', as displayed and edited, is now accepted (i.e. A$ = "Y") then the stock value is re-calculated in line 2730 and line 2735 calls subroutine 8000 to transfer the values from the temporary variables to the appropriate array elements. If the data is not accepted, this transfer does not take place and a jump is made to the end of the loop (to line 2740).

Having processed one change, the final lines of the change record subroutine allow the user to continue making changes, or to return to the main menu.

Delete record routine

Table 15.9 Delete record routine

```
2000 PRINT"□":REM DELETE RECORD
2010 PRINT:PRINT
2020 INPUT"■STOCK NUMBER TO DELETE■";SND
2035 FOR I=1 TO NR-1
2036 IF SN(I)<>SND THEN 2108
2037 X=I
2040 PRINT"□"
2050 PRINT"          ■DELETE THIS RECORD■"
2060 GOSUB 7400:REM DISPLAY RECORD
2070 PRINT:INPUT"ARE YOU SURE Y/N";A$
2080 IF A$="N" THEN 2108
2090 IF A$<>"Y" THEN 2040
2100 D$(I)="**DELETED**"
2105 DR=DR+1
2108 NEXT I
2110 GOSUB 7200
2120 IF A$="C" THEN 2000
2130 RETURN


7400 REM DISPLAY RECORD
7410 PRINT"RECORD NO";X:PRINT
7420 PRINT H$(1);SN(X):PRINT
7430 PRINT H$(2);" "+D$(X):PRINT
7440 PRINT H$(3);UC(X):PRINT
7450 PRINT H$(4);OQ(X):PRINT
7460 PRINT H$(5);ROL(X):PRINT
7470 PRINT H$(6);SL(X):PRINT
7480 PRINT H$(7);V(X):PRINT:PRINT
7490 RETURN
```

The delete record and display record subroutines are shown in Table 15.9. The record to be deleted is identified by entering its stock number (line 2020). Each record within the file is then examined in turn for a match with the stock number within the loop 2035 to 2108. When a match is found, a 'DELETE THIS RECORD' message is displayed in reverse video. The relevant record details are displayed by calling the subroutine at 7400 from line 2060.

The subroutine at 7400 displays the record number, field headings and their contents. Having displayed the record the user is asked if he is sure that he wishes the record to be deleted (line 2070). If the response is N (No) then no action is taken and the program jumps to the end of the loop (i.e. NEXT I at 2108). On the other hand, if the response is Y (Yes), then line 2100 replaces the current stock description with the string '**DELETED**' and line 2105 increases the deleted records count by one.

The record is therefore not deleted immediately, but a flag is put into the stock description field which will allow the record to be identified during the file saving routine. The record will not be saved on the updated file. This approach allows the user to use the SEARCH option just prior to exiting the program and search for the string '**DELETED**'. A complete list of records about to be dropped from the updated file can then be obtained. If the user has any second thoughts about stock items appearing on the list he can then use the CHANGE RECORD option to replace the '**DELETED**' in the stock description by its proper description once again and the record will not then be dropped.

As with the other options, the lines 2110 and 2120 allow the user to continue with the DELETE option or to return to the menu.

Exit program routine

Table 15.10 Exit program routine

```
6000 PRINT"□":REM EXIT & SAVE ROUTINE
6010 INPUT"■SAVE CURRENT FILE, Y/N■";A$
6030 IF A$<>"N" AND A$<>"Y" THEN 6000
6040 IF A$="N" THEN 6300
6050 PRINT
6060 INPUT"CASSETTE OR DISK FILE, C/D";F$
```

```
6070 IF F$="C" THEN D=1
6080 IF F$="D" THEN D=8
6090 IF F$<>"C" AND F$<>"D" THEN 6060
6100 PRINT
6110 INPUT"ENTER FILE NAME";FT$
6120 IF D=1 THEN OPEN 1,1,1,FT$
6130 IF D=8 THEN OPEN 1,8,2,"0:"+FT$+",S,W"
6140 PRINT#1,NR-1-DR
6150 FOR I=1 TO NR-1
6155 IF D$(I)="**DELETED**" THEN 6220
6160 PRINT#1,SN(I)
6170 PRINT#1,D$(I)
6180 PRINT#1,UC(I)
6190 PRINT#1,OQ(I)
6200 PRINT#1,ROL(I)
6210 PRINT#1,SL(I)
6220 NEXT I
6230 CLOSE 1
6240 PRINT "FILE SAVED"
6300 RETURN
```

The exit program subroutine is shown in Table 15.10. When this option is chosen from the main menu the user is asked in line 6010 whether he wishes to save the current file. If the response is N (No) then the program jumps to the end of this subroutine (i.e. line 6300). If the response is Y (Yes) the main part of the subroutine is executed. If the response is neither N or Y, then line 6030 ensures that the question is repeated until a N or Y response is obtained.

The main part of the subroutine lies between lines 6060 and 6240. Line 6060 requests the appropriate storage device code to be entered (i.e. C for cassette tape and D for disk). Line 6110 requests the name of the proposed file. If the file is to be saved to tape, then line 6120 opens the file; if to disk, then line 6130 opens a disk file in sequential write mode.

The first data item to be recorded, in line 6140, is the total number of records to be saved, the total number being the current number of records, NR − 1, less the number of deleted records. Lines 6150 and 6220 form a loop that processes each record in turn. If the record is a 'deleted' record then line 6155 causes that record to be bypassed, otherwise each field of the record is written to the file over lines 6160 to 6210. Only the calculated 'value of stock' is not written to the file. After processing all the records, line 6230 closes the file and line 6240 displays a message confirming that the file has been saved.

Reorder report routine

Table 15.11 Recorder report routine

```
4000 PRINT"□":REM REORDER REPORT
4010 GOSUB 7800
4020 OPEN 1,D
4025 CT=0
4028 PRINT#1,"   "
4030 PRINT#1,"                REORDER REPORT"
4040 PRINT#1,"                --------------"
4050 PRINT#1,"   "
4060 PRINT#1," STK NO    DESCRIPTION    ORD QTY    COST "
4070 PRINT#1,"-----------------------------------------"
4080 FOR I=1 TO NR-1
4090 IF SL(I)>ROL(I) THEN 4155
4100 CST=OQ(I)*UC(I)
4105 AM=CST:GOSUB 7700:CST$=M$
4110 CST$=RIGHT$(B$+CST$,8)
4120 CT=CT+CST
4130 XX=I:GOSUB 8200
4140 L$=F1$+F2$+F4$+CST$
4150 PRINT#1,L$
4155 NEXT I
4160 PRINT#1,"-----------------------------------------"
4165 AM=CT:GOSUB 7700:CT$=M$
4170 PRINT#1,"                          TOTAL    ";
4175 PRINT#1,RIGHT$(B$+CT$,8)
4180 PRINT#1,"-----------------------------------------"
4190 CLOSE 1
4200 GOSUB 7200
4210 IF A$="C" THEN 4000
4225 RETURN



7700 REM TWO DECIMAL PLACE ROUTINE
7710 L$=STR$(INT(AM))
7720 R=INT(100*AM+.5)
7730 R$=RIGHT$(STR$(R),2)
7735 IF AM=0 THEN R$="00"
7740 M$=L$+"."+R$
7750 RETURN

7800 REM SET UP OUTPUT DEVICE
7810 PRINT
7820 INPUT"OUTPUT TO SCREEN OR PRINTER, S/P";A$
7830 IF A$="S" THEN D=3
```

```
7840 IF A$="P" THEN D=4
7850 IF A$<>"S" AND A$<>"P" THEN 7820
7860 RETURN


8200 REM FORMAT FIELDS
8210 F1$=RIGHT$(B$+STR$(SN(XX)),8)
8220 F2$=LEFT$(D$(XX)+B$+B$,13)
8230 F2$="  "+F2$
8240 F3$=RIGHT$(B$+STR$(UC(XX)),8)
8250 F4$=RIGHT$(B$+STR$(OQ(XX)),8)
8260 F5$=RIGHT$(B$+STR$(ROL(XX)),8)
8270 F6$=RIGHT$(B$+STR$(SL(XX)),8)
8275 AM=V(XX):GOSUB7700:V$=M$
8280 F7$=RIGHT$(B$+V$,8)
8290 RETURN
```

The reorder report is shown in Table 15.11 together with the three associated subroutines (starting at lines 7700, 7800, and 8200). The reorder report sub-routine starts by having line 4010 call a sub-routine at 7800. This allows the user to set the output device to be either the screen (device number 3) or the printer (device number 4). Line 4020 then opens the appropriate device. The cumulative monetary total (CT) is set to zero in line 4025 and then the report headings are printed (lines 4030–4070).

The individual records are processed, in turn, within the loop set up by lines 4080 and 4155. If the stock level is above the reorder level a jump is made to the end of the loop, from line 4090 to 4155. If a stock item is to be listed, then the cost of the stock order (CST) is calculated, in line 4100, by multiplying the order quantity by the unit cost. Line 4105 calls the subroutine at 7700 to set the cost to a two-decimal result in string format. Line 4110 adjusts the string to an eight-character string. The cumulative cost is updated in line 4120. The stock record fields are all formatted to strings in line 4130 by calling subroutine 8200. The results for one stock record are then concatenated into the string L$, in line 4140, and printed in line 4150.

Having processed all the records in the file, the loop, lines 4080 to 4155, is left and the final stages of the report produced. Line 4160 underlines the stock list. The cumulative cost is formatted as a string in line 4165 and printed out in line 4170. Finally line 4180 underlines the cumulative cost, and the output file is closed in line 4190. The

user can then repeat the report by 'continuing' or return to the main menu.

In practice, the user might send the report to the screen first, to check its contents, and then, if satisfied, 'continue' and send the report to the printer.

Search records routine

Table 15.12 Search records routine

```
3000 PRINT"□":REM SEARCH ROUTINE
3010 PRINT"■STOCK FILE SEARCH■"
3020 PRINT
3030 INPUT"ENTER SEARCH STRING";X$
3050 L=LEN(X$):E=0:F=0:XX=0
3060 PRINT
3070 GOSUB 7800
3075 OPEN 1,D
3078 PRINT
3080 PRINT#1,"     SEARCH FOR ";X$
3090 PRINT#1,"------------------------------"
3100 PRINT#1,"  "
3110 PRINT#1,"  STK NO  DESCRIPTION   STK QTY"
3120 PRINT#1,"------------------------------"
3130 FOR I=1 TO NR-1
3140 E=E+1
3150 W=LEN(D$(I))-L+1
3160 FOR P=1 TO W
3170 Z$=MID$(D$(I),P,L)
3180 IF Z$<>X$ THEN 3230
3190 F=F+1
3200 XX=I:GOSUB 8200
3210 L$=F1$+F2$+F6$
3220 PRINT#1,L$
3225 P=W
3230 NEXT P
3240 NEXT I
3250 PRINT#1,"------------------------------"
3260 PRINT#1,E;"RECORDS SEARCHED"
3270 PRINT#1,F;"RECORDS LISTED"
3280 CLOSE 1
3290 GOSUB 7200
3300 IF A$="C" THEN 3000
3310 RETURN
```

The search records subroutine is shown in Table 15.12. The user is requested to enter the required search string at line 3030. The length of the search string is calculated in line 3050 and two search

parameters, E and F, set to zero. Line 3070 allows the user to specify the output device and line 3075 opens the output file. The report headings are printed out over lines 3080 to 3120. Each record is processed within the loop at lines 3130 to 3240.

As each record is examined, the record count, E, is increased by one in line 3140. Line 3150 calculates W, the number of successive comparisons of the search string that can be made within the stock description string. The string comparisons for one record are then made within the loop over lines 3160 to 3230. A sub-string of the stock description extracted at line 3170 is compared with the search string at line 3180. If a match is found, then the 'number found' count, F, is increased by one at line 3190; the fields are formatted for output in line 3200 and concatenated into a single string variable, L$, in line 3210. Details of the record are printed at line 3220.

Having processed each record the output is concluded by reporting the number of records searched and the number of matches found (lines 3250–3270). The output file is then closed at line 3280 and the user continues with another search or returns to the main menu.

Update stock routine

Table 15.13 Update stock routine

```
3500 PRINT"□":REM STOCK UPDATE
3510 INPUT"STOCK NUMBER TO UPDATE";SN
3520 PRINT
3530 FOR I=1 TO NR-1
3540 IF SN(I)<>SN THEN 3660
3560 PRINT"RECORD NO";I:PRINT
3570 PRINTH$(1);SN(I):PRINT
3580 PRINTH$(2);D$(I):PRINT
3590 PRINT"CURRENT STOCK";SL(I):PRINT
3595 SA=0
3600 INPUT"STOCK AJUSTMENT +/-";SA:PRINT
3605 SL=SL(I)+SA
3610 PRINT"REVISED STOCK";SL:PRINT
3620 INPUT"ACCEPT DATA Y/N";A$
3630 IF A$<>"Y"THEN 3660
3640 SL(I)=SL
3650 V(I)=SL(I)*UC(I)
3660 NEXT I
3670 GOSUB 7200
3680 IF A$="C"THEN 3500
3690 RETURN
```

The update stock subroutine is shown in Table 15.13. Having specified the stock number in line 3510 the stock records are searched within the loop, line 3530 to line 3660. Line 3540 ensures that records not specified are by-passed. When an appropriate record is found, its record number, stock number and description are printed by lines 3560 to 3580. The current stock level is printed by line 3590, the stock adjustment (SA) set to zero in line 3595, and an adjustment requested from the user in line 3600. A temporary variable (SL) is then used to hold the revised stock level and its value is printed by line 3610. If the adjustment is not accepted by the user in line 3620, then line 3630 by-passes the updating and the processing continues within the loop to the next record.

Accepted revised records are updated by transferring the stock level in the temporary variable to the correct array element (line 3640) and by the re-calculation of the value of stock (line 3650). The routine ends with the standard 'continue' or return to main menu option.

Valuation report routine

The valuation report subroutine is shown in Table 15.14. The valuation report routine is similar to the reorder report routine. Lines 4510 and 4520 allow the user to send the report to the screen or the printer. The cumulative total is set to zero in line 4530 and then the report headings are printed out over lines 4540 to 4580. Each record is processed over the loop at lines 4590 to 4635. Line 4600 formats the stock record fields as strings and line 4610 concatenates the required fields into one string. This string, L$, is printed out in line 4620. The cumulative stock value is updated in line 4630.

Having printed out all the stock records, the cumulative stock value is formatted as a string in line 4638 and the bottom lines of the report are printed out over lines 4640 to 4660.

Table 15.14 Valuation report routine

```
4500 PRINT"□":REM VALUATION REPORT
4510 GOSUB 7800
4520 OPEN 1,D
4530 CT=0
4535 PRINT#1,"  "
4540 PRINT#1,"              VALUATION REPORT"
4550 PRINT#1,"              ----------------"
4560 PRINT#1,"  "
4570 PRINT#1," STK NO   DESCRIPTION   STK QTY   VALUE"
4580 PRINT#1,"---------------------------------------"
4590 FOR I=1 TO NR-1
4600 XX=I:GOSUB 8200
4610 L$=F1$+F2$+F6$+F7$
4620 PRINT#1,L$
4630 CT=CT+V(I)
4635 NEXT I
4638 AM=CT:GOSUB7700:CT$=M$
4640 PRINT#1,"---------------------------------------"
4650 PRINT#1,"                          TOTAL   ";
4655 PRINT#1,RIGHT$(B$+CT$,8)
4660 PRINT#1,"---------------------------------------"
4670 CLOSE 1
4680 GOSUB 7200
4690 IF A$="C" THEN 4500
4695 RETURN
```

15.8 Further program developments

The program can be developed in three areas.

Firstly, the INPUT statements, where appropriate, can be replaced by a protected input routine based on GET, as described in section 9.8 (Chapter 9), to make a null routine impossible.

Secondly, range checks could be introduced for some of the INPUT statements to trap unacceptable responses.

Thirdly, the range of options could include a sort, a complete stock listing and different types of report. With these additions, for example, a stock listing in decreasing value of stock could be produced.

Appendix A

Programs (Tables A1 to A21)

Table A1 Number of £s required

```
20 INPUT"A,E,P,N,R";A,E,P,N,R
25 PRINT
30 LET T=((A+E)*N+P)/R
40 PRINT"LENGTH OF STAY(NIGHTS)    :";N
50 PRINT"ACCOMODATION(PER NIGHT) $:";A
60 PRINT"EXPENSES(MEALS ETC.)     $:";E
70 PRINT"ALLOWANCE FOR PRESENTS   $:";P
80 PRINT"EXCHANGE RATE($ TO THE £):";R
85 PRINT
90 PRINT"POUNDS STERLING REQUIRED :";T
95 PRINT"**********************************"
100 END
```

Table A2 Cost of stationery

```
10 INPUT"NO OF DELEGATES";N
12 INPUT"COST OF FOLDERS AND PADS";F,P
14 INPUT"COST OF PENS AND DISCOUNT";S,D
20 PRINT
25 PRINT
30 LET C=N*(F+P+2*S*(100-D)/100)/100
40 PRINT"NO OF DELEGATES        :";N
50 PRINT"COST OF FOLDERS        :";F;"P EACH"
60 PRINT"COST OF PAPER          :",P;"P PER PAD"
70 PRINT"COST OF PENS LESS";D;"%:";S;"P EACH"
80 PRINT
85 PRINT"TOTAL COST OF STATIONERY = £";C
90 PRINT"**********************************"
95 END
```

Table A3 Using the ON . . . GOTO statement

```
20 PRINT"CALCULATIONS FOR DIFFERENT CODES"
25 PRINT"--------------------------------"
30 PRINT
50 INPUT "NUMBER OF SETS OF DATA";N
55 PRINT:PRINT
60 FOR I=1 TO N
70 INPUT "CODE,X,Y";C,X,Y
75 PRINT:PRINT
80 ON C GOTO 90,100,110,120,130
90 LET R=X+Y
95 GOTO 140
100 LET R=X-Y
105 GOTO 140
110 LET R=X*Y
115 GOTO 140
120 LET R=X/Y
125 GOTO 140
130 LET R=X↑Y
140 PRINT"CODE:";C";RESULT =";R
145 PRINT"**********************"
150 PRINT:PRINT
160 NEXT I
170 END
```

Table A4 Centering a rectangle

```
10 INPUT"ENTER WIDTH,DEPTH";W,D
15 LET S=20-INT(W/2+0.5)
20 PRINT"□":REM CLEAR
25 FOR I=1 TO 11-INT(D/2+0.5)
30 PRINT
35 NEXT I
40 PRINT TAB(S);"┌";
45 FOR I=1 TO W-2
50 PRINT"─";
55 NEXT I
60 PRINT"┐"
65 FOR I=1 TO D-2
70 PRINT TAB(S);"│ ";TAB(W-1+S);" │"
75 NEXT I
80 PRINT TAB(S);"└";
85 FOR I=1 TO W-2
90 PRINT"_";
95 NEXT I
100 PRINT"┘"
105 END
```

Table A5 Animated face

```
8 PRINT"□"
10 SM=1024:CM=55296
20 S=SM+40*8+17
30 C=CM+40*8+17
40 FOR I=0 TO 5
50 FOR J=0 TO 5
60 READ X
70 POKE S+40*I+J,X
80 POKE C+40*I+J,3
90 NEXT J
100 NEXT I
110 X=74:Y=75:CL=3
115 XA=85:YA=73
120 FOR I=1 TO 10
130 POKE S+40*4+2,X
140 POKE C+40*4+2,CL
150 POKE S+40*4+3,Y
160 POKE C+40*4+3,CL
170 FOR D=1 TO 1000:NEXT D
180 XB=X:X=XA:XA=XB
185 YB=Y:Y=YA:YA=YB
190 NEXT I
200 DATA 85,102,102,102,102,73
210 DATA 93,87,96,96,87,93
220 DATA 93,96,85,73,96,93
230 DATA 93,96,74,75,96,93
240 DATA 93,96,96,96,96,93
250 DATA 74,64,64,64,64,75
260 END
```

Table A6 Radius of circumcircle

```
20 INPUT "SIDES OF TRIANGLE";A,B,C
30 PRINT
40 LET X=(A*A+C*C-B*B)/(2*A*C)
50 LET R=B/(2*SIN(ATN(SQR(1-X*X)/X)))
60 PRINT
70 PRINT "RADIUS =";R;"M"
75 PRINT "**********************"
80 END
```

Table A7 Volumes of solids

```
10 DATA CUBOID,CYLINDER,"HEX BAR"
20 DEF FNR(A)=INT(A/F+0.5)*F
40 INPUT"ENTER CODE AND SCALE";C,F
60 IF C=0 THEN 250
70 INPUT"ENTER TWO DIMENSIONS";D1,D2
80 INPUT"ENTER HEIGHT";H
90 PRINT
100 ON C GOTO 110,130,150
110 LET A=D1*D2
120 GOTO 160
130 LET A=π*D1*D1
140 GOTO 160
150 LET A=SQR(27)/2*D1*D1
160 FOR I=1 TO C
170 READ N$
180 NEXT I
190 PRINT "VOL OF ";N$;" =";
195 PRINT FNR(A*H);" CUBIC CM"
200 PRINT"**********************************"
210 PRINT
220 PRINT
230 RESTORE
240 GOTO 40
250 END
```

Table A8 Copying an array

```
20 INPUT "NO OF ELEMENTS IN ARRAY";N
25 PRINT:PRINT
30 DIM A(20),B(20)
40 FOR I=1 TO N
50 READ A(I)
60 NEXT I
70 FOR I=1 TO N STEP 5
90 FOR J=I TO I+4
100 LET B(J)=A(N+1-J)
110 PRINT B(J);
130 NEXT J
140 PRINT
150 NEXT I
160 DATA 1,2,3,4,5,6,7,8,9,10,11,12,13
170 DATA 14,15,16,17,18,19,20
180 END
```

Table A9 Sum of elements

```
20 DIM A(5,5)
30 INPUT "ENTER M FOR M X M ARRAY";M
40 IF M=0 THEN 260
50 LET D=0
60 FOR I=1 TO M
70 FOR J=1 TO M
80 READ A(I,J)
90 PRINT A(I,J);
100 NEXT J
110 PRINT
120 D=D+A(I,I)+A(I,M+1-I)
130 NEXT I
140 PRINT
150 IF M/2=INT(M/2) THEN 180
160 LET N=INT(M/2)+1
170 LET D=D-A(N,N)
180 PRINT"SUM ON DIAGONALS =";D
185 PRINT"**********************"
190 PRINT
200 PRINT
210 RESTORE
220 GOTO 30
230 DATA 10,11,12,13,14,15,16,17,18,19
240 DATA 20,21,22,23,24,25,26,27,28,29
250 DATA 30,31,32,33,34
260 END
```

Table A10 Sorting a list of numbers

```
20 INPUT "NO OF NUMBERS";N
30 PRINT"□":REM CLEAR
40 DIM A(11)
50 FOR I=1 TO N
60 READ A(I):PRINT A(I);
70 NEXT I
80 PRINT
90 FOR I=1 TO N-1
100 LET E=0
110 FOR J=1 TO N-1
120 IF A(J)<=A(J+1) THEN 170
130 LET S=A(J)
140 LET A(J)=A(J+1)
150 LET A(J+1)=S
160 LET E=1
170 NEXT J
180 IF E=0 THEN 260
200 FOR K=1 TO N
210 PRINT A(K);
220 NEXT K
230 PRINT
240 NEXT I
250 DATA 15,3,20,22,22,9,4,23,2,0,-25
260 END
```

Table A11 Plot of percentage pastureland

```
20 DIM V(100)
30 INPUT"ENTER NO OF YEARS";N
50 PRINT
60 FOR I = 1 TO N
70 PRINT"% PASTURELAND, YR";I;
80 INPUT V(I)
90 NEXT I
100 PRINT
110 GOSUB 1010
120 END
```

Table A12 Pastureland histogram

```
20 DIM V(100),X(15),F(15)
30 INPUT"ENTER NO OF PARISHES";N
50 PRINT
60 FOR I = 1 TO N
70 PRINT"PARISH";I
80 INPUT V(I)
90 NEXT I
100 PRINT
110 GOSUB 2000
120 GOSUB 3000
130 GOSUB 4000
140 END
```

Table A13 Input subroutine

```
800 PRINT
810 INPUT"NO OF ROWS IN FREQ DIST.";N
830 IF N<11 THEN 850
840 PRINT"NOT MORE THAN 10, TRY AGAIN"
845 GOTO 800
850 PRINT
860 PRINT"INPUT X & CUM FREQ"
870 FOR I=1 TO N
880 INPUT D(I,1),D(I,2)
890 NEXT I
899 RETURN
```

Table A14 Multicoloured goldfish DATA and subroutine values

```
200 DATA 0,38,64,128,166,160
202 DATA 33,154,120,38,153,170
204 DATA 33,105,152,128,166,144
206 DATA 0,26,128,0,0,0,0,0,0
208 DATA 0,0,0,0,0,0,0,0,0
210 DATA 0,0,0,0,0,0,0,0,0
212 DATA 0,0,0,0,0,0,0,0,0
214 DATA 0,0,0,0,0,0,0,0,0
220 END
300 REM MULTICOLOUR MODE
310 POKE SM+28,3
320 POKE SM+37,7
330 POKE SM+38,0
340 POKE SM+39,8
350 POKE SM+40,8
360 RETURN
```

Table A15 Timer alarm

```
100 SS=54272
110 GOSUB 2000
120 GOSUB 3000
125 GOSUB 4000
130 REM TICK AWAY
140 POKE SS+0,75: POKE SS+1,34
150 POKE SS+4,17
160 FOR L=1 TO 500: NEXT L
170 POKE SS+4,16
180 REM ELAPSED TIME
190 D=TI-B
200 M=INT(D/3600)
210 S=INT(D/60)-60*M
220 T$=MID$(STR$(M),2)+":"+MID$(STR$(S),2)+" "
230 PRINT"SQQQ]]]]]]]ELAPSED TIME ";T$
240 IF TI < F THEN 130
250 REM TRILL
260 POKE SS+5,9: POKE SS+6,220
270 POKE SS+4,17
280 POKE SS+0,149: POKE SS+1,68
290 FOR L=1 TO 5: NEXT L
300 POKE SS+0,0: POKE SS+1,0
310 GET A$
320 IF A$=""THEN 270
330 END
2000 REM CLEAR SOUND REGISTERS
2010 FOR L=SS TO SS+24
2020 POKE L,0
2030 NEXT L
2040 RETURN
3000 REM SET UP TIMER DURATION
3010 PRINT"SET TIMING DURATION"
3020 PRINT
3030 INPUT"NUMBER OF MINUTES ";T
3040 IF T<0 THEN 3030
3050 P=T*60*60
3060 PRINT
3070 PRINT"PRESS SPACE BAR TO START"
3075 GET R$:IF R$="" THEN 3075
3080 B=TI:F=B+P
3090 RETURN
4000 REM SET UP SOUND ENVELOPE
4010 POKE SS+24,15
4020 POKE SS+5,1: POKE SS+6,0
4030 RETURN
```

Table A16 Cos X

```
20 INPUT"NO OF TERMS FOR COS X";N
40 INPUT"VALUE OF X (DEGREES)";X1
60 LET X=(X1*π/180)↑2
70 LET T=1
80 LET C=1
90 FOR I=2 TO N*2 STEP 2
100 LET T=(-1)*T*X/((I-1)*I)
105 LET C=C+T
110 NEXT I
115 PRINT
120 PRINT"COS";X1;"=";C
125 PRINT"**********************"
130 END
```

Table A17 Roots of quadratic equations

```
10 DEF FNR(A)=INT(A/0.01+0.5)*0.01
20 PRINT"ROOTS OF QUADRATIC EQUATIONS"
30 PRINT"----------------------------"
40 INPUT"ENTER A,B,C  (ZEROS TO STOP)";A,B,C
70 IF A=0 THEN 210
80 LET D=B*B-4*A*C
90 IF D<0 THEN 150
100 IF D=0 THEN 170
110 LET D=SQR(D)
120 PRINT"REAL ROOTS: ";
125 PRINT FNR((-B+D)/(2*A));
130 PRINT"AND ";FNR((-B-D)/(2*A))
140 GOTO 180
150 PRINT"COMPLEX ROOTS"
160 GOTO 180
170 PRINT"COINCIDENT ROOTS: ";
175 PRINT FNR(-B/(2*A))
180 PRINT"**************************"
190 PRINT
200 GOTO 40
210 END
```

Table A18 Width of a slit

```
10 PRINT "WIDTH OF A SLIT"
15 PRINT "---------------"
20 INPUT "ENTER WAVELENGTH";L
40 PRINT
50 INPUT "ENTER NO OF FRINGES";N
70 FOR I=1 TO N
80 PRINT "ENTER TWO PAIRS OF VERNIER READINGS"
90 PRINT "(DEG,MIN FOR EACH) FOR FRINGE NO";I
100 FOR J=1 TO 2
110 INPUT P(I,J),Q(I,J),R(I,J),S(I,J)
120 LET B(J)=P(I,J)*60+Q(I,J)-R(I,J)*60-S(I,J)
130 NEXT J
140 LET A(I)=(B(1)+B(2))/4
150 LET W(I)=I*L*60*180/(A(I)*π)
160 NEXT I
170 PRINT
180 PRINT "FRINGE  VERNIER READINGS   A   WIDTH OF"
190 PRINT "NUMBER  DEG MIN  DEG MIN  MIN  SLIT  CM"
200 PRINT "---------------------------------------"
210 PRINT
220 FOR I=1 TO N
230 LET W(I)=INT(W(I)/0.0001+0.5)*0.0001
240 PRINT TAB(8);P(I,1);TAB(12);Q(I,1);
250 PRINT TAB(17);R(I,1);TAB(21);S(I,1)
260 PRINT TAB(2);I,TAB(27);A(I);TAB(32);W(I)
270 PRINT TAB(8);P(I,2);TAB(12);Q(I,2);
280 PRINT TAB(17);R(I,2);TAB(21);S(I,2)
290 PRINT "---------------------------------------"
300 PRINT
310 LET W=W+W(I)
320 NEXT I
330 PRINT
340 LET W=INT(W/N/0.0001+0.5)*0.0001
350 PRINT "WIDTH OF SLIT =";W;"CM"
360 PRINT "**************************"
370 END
```

Table A19 Stock data file creation

```
5 REM "STK-MAKE"; CREATES "STK-DATA"
10 PRINT"■■STOCK FILE CREATION"
20 PRINT"--------------------"
25 PRINT
30 OPEN 1,1,1,"STK-DATA"
40 INPUT"CODE";K
45 IF K=9999 THEN 115
50 INPUT"DESCRIPTION";D$
60 INPUT"STOCK LEVEL";S
70 INPUT"UNIT COST";C
80 INPUT"RE-ORDER LEVEL";R
90 INPUT"ORDER QUANTITY";Q
100 PRINT#1,K;",";D$;",";S;",";C;",";R;",";Q
110 GOTO 40
115 PRINT#1,9999;",";"X";0;",";0;",";0;",";0
120 CLOSE 1
130 END
```

Table A20 Reorder list

```
5 REM "ORD-LIST" PRODUCES A REORDER LIST
10 INPUT"■■ENTER 3 FOR SCREEN OR 4 FOR PRINTER";N
20 OPEN 2,N
25 OPEN 1,1,0,"STK-DATA"
30 PRINT"■■"
40 PRINT#2,TAB(12);"REORDER LIST"
45 PRINT#2,TAB(12);"------------"
50 PRINT#2
60 PRINT#2,TAB(1);"CODE";SPC(5);"DESCRIPTION";
62 PRINT#2,SPC(5);"ORDER QTY"
65 PRINT#2,TAB(1);"----";SPC(5);"-----------";
67 PRINT#2,SPC(5);"---------"
70 PRINT#2
80 INPUT#1,K,D$,S,C,R,Q
90 IF K=9999 THEN 130
100 IF S>R THEN 80
110 PRINT#2,K;SPC(5);D$;SPC(18-LEN(D$));Q
120 GOTO 80
130 CLOSE 1
135 CLOSE 2
140 END
```

Table A21 Stock file search

```
5 REM INTERROGATE"STK-DATA" FILE
10 INPUT"■■ENTER 3 FOR SCREEN OR 4 FOR PRINTER";N
20 OPEN 2,N
25 PRINT"■■"
30 PRINT"STOCK FILE SEARCH"
35 PRINT"-----------------"
40 PRINT
45 INPUT"ENTER SEARCH WORD";X$
50 L=LEN(X$)
55 OPEN1,1,0,"STK-DATA"
60 LET E=0
65 LET F=0
70 LET U$="--------------------------"
75 PRINT#2,U$
80 PRINT#2,TAB(1);"CODE";SPC(3);"DETAILS";
82 PRINT#2,SPC(6);"STOCK"
85 PRINT#2,U$
90 INPUT#1,K,D$,S,C,R,Q
95 IF K=9999 THEN 150
100 LET E=E+1
105 LET W=LEN(D$)-L+1
110 FOR I=1 TO W
115 LET Z$=MID$(D$,I,L)
120 IF Z$<>X$ THEN 140
125 LET F=F+1
130 PRINT#2,K;SPC(2);D$;SPC(13-LEN(D$));S
135 LET I=W
140 NEXT I
145 GOTO 90
150 CLOSE 1
155 PRINT#2,U$
160 PRINT#2,E;"RECORDS READ"
165 PRINT#2,F;"RECORDS LISTED"
170 CLOSE 2:END
```

Appendix B

Answers to Problems

Chapter 5

1 CALCULATIONS FOR DIFFERENT CODES

CODE	X	Y	CALC. VALUE
3	51	4	204
1	25	13	38
2	8	34	−26
5	4	3	64
4	62	5	12.4

Chapter 7

1 RADIUS = 443.334 M

3 VOLUME OF CYLINDER = 111.33 CUBIC CM
VOLUME OF HEXAGONAL BAR = 103118 CUBIC CM
VOLUME OF CUBOID = 155.8 CUBIC CM

Chapter 12

1 Cos 30° = 0.866025 (to five terms)

2 REAL ROOTS : −.24 AND −2.76
COMPLEX ROOTS
COINCIDENT ROOTS : 4

REAL ROOTS :	.85	AND	−2.35
REAL ROOTS :	−1	AND	.33
COMPLEX ROOTS			
COINCIDENT ROOTS :	−.5		

3 WIDTH OF SLIT = .0675 CM

4 Slope = 0.1096
Coefficient of correlation = 0.9953
Young's modulus = 1.92×10^{11} Nm^{-2}

5 The answer will vary slightly depending upon the selection of random numbers but should be close to 11.1 weeks.

6 Monthly repayments = 156.97 and 209.29, respectively.

Appendix C

ASCII and CHR$ Codes

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0		17	"	34	3	51
	1		18	#	35	4	52
	2		19	$	36	5	53
	3		20	%	37	6	54
	4		21	&	38	7	55
	5		22	·	39	8	56
	6		23	(	40	9	57
	7		24	)	41	:	58
DISABLES 	8		25	*	42	;	59
ENABLES 	9		26	+	43		60
	10		27	,	44	=	61
	11		28	–	45		62
	12		29	.	46	?	63
RETURN	13		30	/	47	@	64
SWITCH TO LOWER CASE	14		31	0	48	A	65
	15	SPACE	32	1	49	B	66
	16	!	33	2	50	C	67

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
D	68	←	95		122	brown	149
E	69		96		123	l red	150
F	70		97		124	grey l	151
G	71		98		125	grey 2	152
H	72		99		126	l green	153
I	73		100		127	l blue	154
J	74		101		128	grey 3	155
K	75		102	orange	129		156
L	76		103		130		157
M	77		104		131		158
N	78		105		132		159
O	79		106	f1	133	SPACE	160
P	80		107	f3	134		161
Q	81		108	f5	135		162
R	82		109	f7	136		163
S	83		110	f2	137		164
T	84		111	f4	138		165
U	85		112	f6	139		166
V	86		113	f8	140		167
W	87		114	SHIFT RETURN	141		168
X	88		115	SWITCH TO UPPER CASE	142		169
Y	89		116		143		170
Z	90		117		144		171
[	91		118		145		172
£	92		119		146		173
]	93		120		147		174
↑	94		121		148		175

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	176		180		184		188
	177		181		185		189
	178		182		186		190
	179		183		187		191

CODES	192–223	SAME AS	96–127
CODES	224–254	SAME AS	160–190
CODE	255	SAME AS	126

Appendix D

Colour Codes

Colour	POKE *values for Border, Background and Sprite colour*	CHR$
Black	0	144
White	1	5
Red	2	28
Cyan	3	159
Purple	4	156
Green	5	30
Blue	6	31
Yellow	7	158
Orange	8	129
Brown	9	149
Light Red	10	150
Grey 1	11	151
Grey 2	12	152
Light Green	13	153
Light Blue	14	154
Grey 3	15	155

Note
To change Border POKE 53280,X
To change Background POKE 53281,X
where X is one of the POKE values from the above table

Appendix E

Symbols Displayed for Special Keys

	MEANING	CHR$
	CLR	147
	HOME	19
	RVS ON	18
	RVS OFF	146
	CURSOR UP	145
	CURSOR DOWN	17
	CURSOR LEFT	157
	CURSOR RIGHT	29
	BLACK	144
	WHITE	5
	RED	28
	CYAN	159
	PURPLE	156
	GREEN	30
	BLUE	31
	YELLOW	158
	ORANGE	129
	BROWN	149
	LIGHT RED	150
	GREY 1	151
	GREY 2	152
	LIGHT GREEN	153
	LIGHT BLUE	154
	GREY 3	155

Appendix F

Screen Display Codes

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	R	r	18	$		36
A	a	1	S	s	19	%		37
B	b	2	T	t	20	&		38
C	c	3	U	u	21	'		39
D	d	4	V	v	22	(		40
E	e	5	W	w	23	)		41
F	f	6	X	x	24	*		42
G	g	7	Y	y	25	+		43
H	h	8	Z	z	26	,		44
I	i	9	[		27	–		45
J	j	10	£		28	.		46
K	k	11	]		29	/		47
L	l	12	↑		30	0		48
M	m	13	←		31	1		49
N	n	14	SPACE		32	2		50
O	o	15	!		33	3		51
P	p	16	"		34	4		52
Q	q	17	#		35	5		53

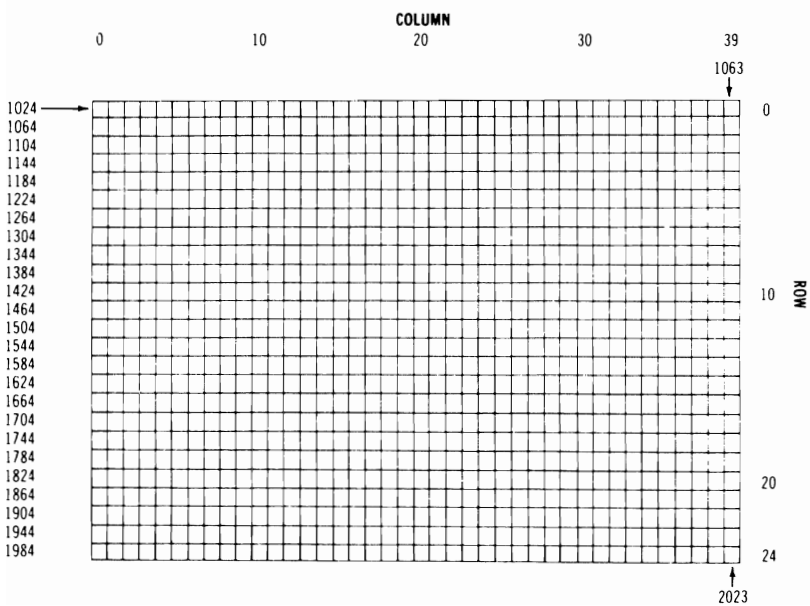
SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
6		54		O	79			104
7		55		P	80			105
8		56		Q	81			106
9		57		R	82			107
:		58		S	83			108
;		59		T	84			109
<		60		U	85			110
=		61		V	86			111
>		62		W	87			112
?		63		X	88			113
		64		Y	89			114
	A	65		Z	90			115
	B	66			91			116
	C	67			92			117
	D	68			93			118
	E	69			94			119
	F	70			95			120
	G	71	SPACE		96			121
	H	72			97			122
	I	73			98			123
	J	74			99			124
	K	75			100			125
	L	76			101			126
	M	77			102			127
	N	78			103			

Codes from 128–255 are reversed images of codes 0–127.

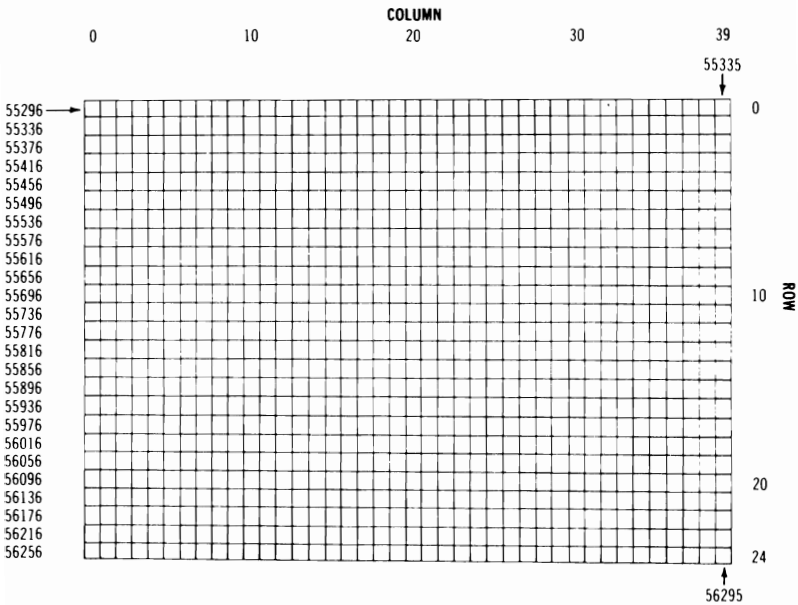
Appendix G

Screen and Colour Memory Maps

SCREEN MEMORY MAP



COLOUR MEMORY MAP



Appendix H

Sprite Registers

SPRITE NO	7	6	5	4	3	2	1	0
X component	14	12	10	8	6	4	2	0
Y component	15	13	11	9	7	5	3	1
msb X	16							
on/off	21							
expand Y	23							
background priority	27							
multicolour select	28							
expand X	29							
sprite-sprite collision	30							
sprite-bkgrd collision	31							
sprite multicolour 0				37				
sprite multicolour 1				38				
sprite colour	46	45	44	43	42	41	40	39
SPRITE NO	7	6	5	4	3	2	1	0
denary values used in common registers	128	64	32	16	8	4	2	1

Note: The above register numbers are offsets from memory location 53248

Appendix I

Sound Registers and Control Settings

The values in the tables are offsets from memory location 54272

	Voice 1	*Voice 2*	*Voice 3*
low frequency	0	7	14
high frequency	1	8	15
low pulse (LO)	2	9	16
high pulse (HI)	3	10	17
waveform	4	11	18
attack/decay	5	12	19
sustain/release	6	13	20
volume (common to all voices)	24		

Types of waveform	*Value to* POKE *to wave form offset*	
	to turn on	*to turn off*
Triangle	17	16
Sawtooth	33	32
Pulse	65	64
Noise	129	128

Appendix J

Envelope Rates and Levels

Value	*Attack rate*	*Decay and Release rate*	*% Sustain level*
0	2 ms	6 ms	0
1	8 ms	24 ms	7
2	16 ms	48 ms	13
3	24 ms	72 ms	20
4	38 ms	114 ms	27
5	56 ms	168 ms	33
6	68 ms	204 ms	40
7	80 ms	240 ms	47
8	100 ms	300 ms	53
9	250 ms	750 ms	60
10	500 ms	1.2 s	67
11	800 ms	2.4 s	73
12	1 s	3 s	80
13	3 s	9 s	87
14	5 s	15 s	93
15	8 s	24 s	100

The values to be POKEd can be calculated from:

attack/decay POKE value = attack value × 16 + decay value
sustain/release POKE value = sustain value × 16 + release value

Appendix K

Music Note Values

Note	Note–Octave	Hi Freq	Low Freq
0	C–0	1	18
1	C#–0	1	35
2	D–0	1	52
3	D#–0	1	70
4	E–0	1	90
5	F–0	1	110
6	F#–0	1	132
7	G–0	1	155
8	G#–0	1	179
9	A–0	1	205
10	A#–0	1	233
11	B–0	2	6
12	C–1	2	37
13	C#–1	2	69
14	D–1	2	104
15	D#–1	2	140
16	E–1	2	179
17	F–1	2	220
18	F#–1	3	8
19	G–1	3	54
20	G#–1	3	103
21	A–1	3	155
22	A#–1	3	210
23	B–1	4	12
24	C–2	4	73
25	C#–2	4	139
26	D–2	4	208
27	D#–2	5	25

Note	Note–Octave	Hi Freq	Low Freq
28	E–2	5	103
29	F–2	5	185
30	F#–2	6	16
31	G–2	6	108
32	G#–2	6	206
33	A–2	7	53
34	A#–2	7	163
35	B–2	8	23
36	C–3	8	147
37	C#–3	9	21
38	D–3	9	159
39	D#–3	10	60
40	E–3	10	205
41	F–3	11	114
42	F#–3	12	32
43	G–3	12	216
44	G#–3	13	156
45	A–3	14	107
46	A#–3	15	70
47	B–3	16	47
48	C–4	17	37
49	C#–4	18	42
50	D–4	19	63
51	D#–4	20	100
52	E–4	21	154
53	F–4	22	227
54	F#–4	24	63
55	G–4	25	177
56	G#–4	27	56
57	A–4	28	214
58	A#–4	30	141
59	B–4	32	94
60	C–5	34	75
61	C#–5	36	85
62	D–5	38	126
63	D#–5	40	200
64	E–5	43	52
65	F–5	45	198
66	F#–5	48	127
67	G–5	51	97
68	G#–5	54	111
69	A–5	57	172
70	A#–5	61	126
71	B–5	64	188

Note	Note–Octave	Hi Freq	Low Freq
72	C–6	68	149
73	C#–6	72	169
74	D–6	76	252
75	D#–6	81	161
76	E–6	86	105
77	F – 6	91	140
78	F#–6	96	254
79	G–6	102	194
80	G#–6	108	223
81	A–6	115	88
82	A#–6	122	52
83	B–6	129	120
84	C–7	137	43
85	C#–7	145	83
86	D–7	153	247
87	D#–7	163	31
88	E–7	172	210
89	F–7	183	25
90	F#–7	193	252
91	G–7	205	133
92	G#–7	217	189
93	A–7	230	176
94	A#–7	244	103

Index

COMPUTER PROGRAMMING IN COBOL

MELINDA FISHER

COBOL (COmmon Business Oriented Language) is the computer language which is most widely used in business and commerce, for invoicing, stock control, payroll and management information systems, on both microcomputer and mainframe installations.

This book looks first at the basic concepts of computers and computer programming before examining the language, logic design and special features of COBOL. Starting from first principles, the book systematically introduces and explains the main COBOL facilities, language statements and relevant syntax. Thus the reader progresses in easy stages from data description and manipulation, through sequence control and handling sequential files and tables, to producing printed reports and indexing. Sample programs illustrate the everyday application of COBOL facilities and a fully documented example shows how a tested, working program is developed from an initial specification. Exercises (with answers) are provided throughout.

TEACH YOURSELF BOOKS

COMPUTER PROGRAMMING IN FORTRAN

ARTHUR S. RADFORD

FORTRAN (FORmula TRANslator) is an established computer language, which has been widely adopted for scientific, technical and commercial applications.

This book provides a step-by-step introduction to FORTRAN 77 – the most up-to-date standard version of the language. Arthur Radford looks first at the basic concepts of computing and program design before explaining the syntax and structure of FORTRAN. He then describes the system commands for programming conditional statements, loops, arrays and subroutines, and the input and output requirements, including file processing. A wide range of illustrative examples and exercises (with answers) gives the beginner a practical grounding in FORTRAN programming while a comparison with the earlier standard FORTRAN 66 makes the book suitable also as a conversion course for more experienced programmers.

TEACH YOURSELF BOOKS

COMPUTER PROGRAMMING IN PASCAL

DAVID LIGHTFOOT

Pascal is a general-purpose computer language which has become popular for its compact yet powerful facilities on a wide variety of microcomputers (including personal computers) and mainframe systems alike.

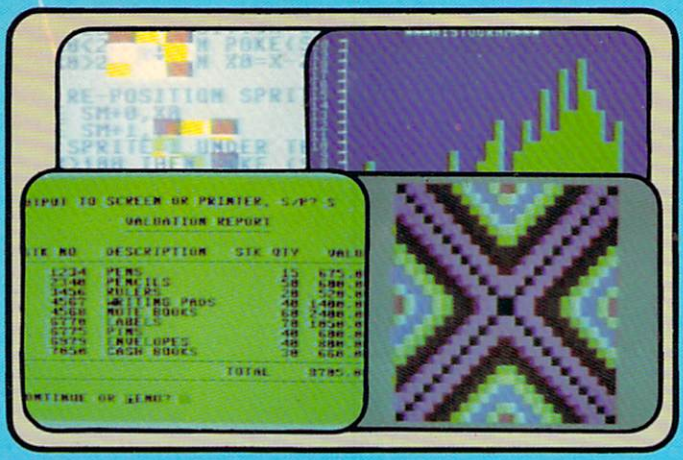
The simple nature of Pascal makes for clearly structured programming giving rise to programs which are easy to read, modify and show to be correct. As such, it is also one of the easier computer languages to learn.

This book provides a sound introduction to Pascal in its standard (BSI) and UCSD implementations. David Lightfoot first describes its general rules and standard types. Then, with the aid of many examples and illustrations, he gives a step-by-step explanation of Pascal statements and syntax – including branching, looping and dealing with sequential files – and practical advice on program design. Exercises (with answers) are included to give a thorough grounding in programming for a wide range of applications, while a graded selection of sample programs highlights the flexibility offered by Pascal.

TEACH YOURSELF BOOKS

This book provides a practical grounding in BASIC, the most widely used microcomputer programming language.

Using the Commodore 64, the authors explain how to write, develop and test BASIC programs for a wide range of applications at home and at work. They then focus on the special features of the Commodore 64 and show how to get the most out of its colour, sprite graphics and sound synthesis facilities.



Clear explanation and examples, and a comprehensive series of sample programs and practical exercises (with answers), make this an invaluable self-teaching introduction and user handbook for the Commodore 64.

TEACH YOURSELF BOOKS/
HODDER AND STOUGHTON
Published in USA by
David McKay Company, Inc.

ISBN 0-340-34638-8

90275

9 780340 346389

UNITED KINGDOM £2.75