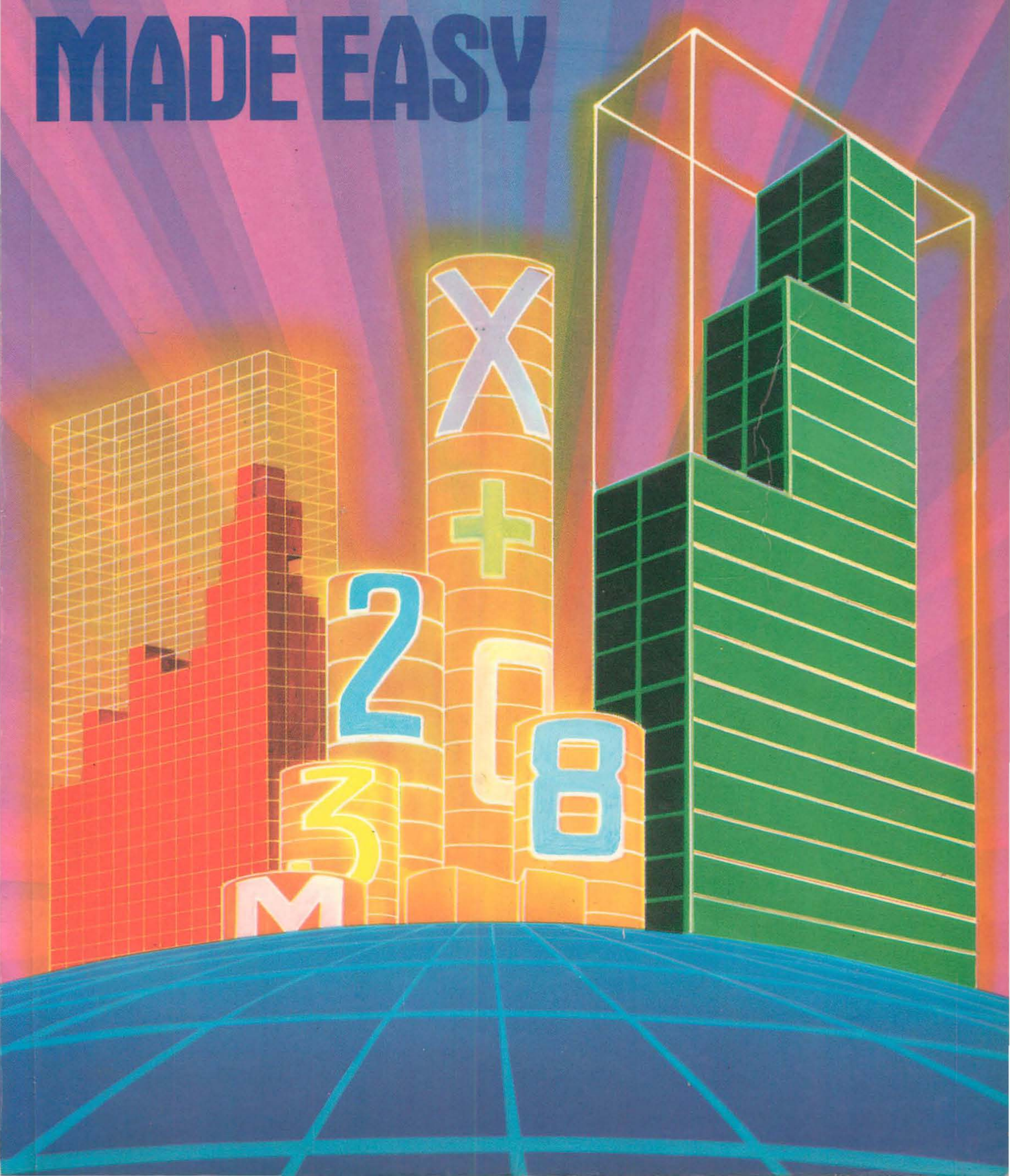
JAMES GATENBY

DATA HANDLING ON THE COMMODORE 64 MADE EASY



Data Handling on the Commodore 64 Made Easy

Useful Programming for the Home, Student and Small Business

James Gatenby

GRANADA
London Toronto Sydney New York

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

First published in Great Britain by
Granada Publishing 1984

Copyright © James Gatenby 1984

British Library Cataloguing in Publication Data
Gatenby, James
Data handling on the Commodore 64 made easy.
1. Commodore 64 (Computer)—Programming
2. Basic (Computer program language)
I. Title
001.64'24 QA76.8.C64

ISBN 0-246-12454-7

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

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system or
transmitted, in any form, or by any means, electronic,
mechanical, photocopying, recording or otherwise,
without the prior permission of the publishers.

Contents

Preface		iv
1	Introduction	1
2	Programming	6
3	Storing Data in the Memory	19
4	Developing the Program	28
5	Making Decisions	34
6	Searching	39
7	Sorting Data	53
8	Menus, Modules and Subroutines	59
9	Developing a General-Purpose Data Handling Program	64
10	Files	75
11	Extending the System	91
Appendix 1:	A Glossary of Terms used in Data Handling	108
Appendix 2:	Calculation Supplement	112
Index		121

Preface

You don't need to be a great mathematician or teenage electronics wizard to do useful and enjoyable work with the Commodore 64. All you really need is the patience to learn a simple new language, similar to English, and you can enter this fascinating new world. Don't worry if you never liked mathematics – most computer programming involves no more than simple counting! In fact, the mainstream of computing nowadays is in data processing, which is concerned with sorting raw facts (or data) to produce useful information. Typical data processing activities might involve large files of addresses, sporting records or data on hobbies such as photography, music or motoring, gardening, cooking or wine-making. In each case the data could be searched, printed or stored far more efficiently than by manual methods.

The main aim of this book is to introduce the Commodore 64 user to planned or 'structured' programming in BASIC, followed by detailed instruction in the writing of useful programs to handle information. Numerous sample programs are included to show you how to approach the work in a planned and logical manner. Another of my aims has been to avoid, wherever possible, the excessive use of technical jargon; I know from experience that this causes many people to switch off from computers altogether.

So, even if you are not turned on by ROMs, EPROMs or joysticks you may still use the phenomenal power of the Commodore 64 to do rewarding work. This book is particularly aimed at:

- Commodore 64 owners who wish to graduate from computer games to real work!
- Students doing project work for examinations in computer studies or data processing.
- Business users who want to write programs which are tailor-made for a particular purpose.

The Commodore 64 is the ideal machine for data processing work. After mastering the rudiments of BASIC programming, you can extend the basic computer into a complete business system. The additional devices, such as the disk drive and line printer, are available cheaply off the shelf and fit straight into the Commodore 64, without the expensive conversion work necessary on some other makes of machines. Such a system will enable you to develop large programs, to create files of data, and run small business programs, such as stock control, accounts or mailing lists.

Whatever your level of programming knowledge, I hope you will find this book interesting and enjoyable. Numerous complete and well-tested programs – which you may copy directly into your Commodore 64 – are provided. The vast majority are also suitable for the VIC 20 and Commodore 4000 series machines.

James Gatenby

Chapter One
Introduction

By the end of this book, you should be able to program the Commodore 64 to store large quantities of data, to display the data on the screen in an attractive and readable way, and to search the data to select particular items, printing out useful information in a special order. It will be possible to do this using numbers or letters, or a mixture of both.

Furthermore, you will be able to handle the data with little or no use of mathematics, although a short section is included as an appendix to show how the Commodore 64 does arithmetic and how it may be used as a calculator.

You should, for instance, be able to produce a file of names, addresses and telephone numbers, which can be amended to include alterations or additions. Similarly, a data file could be based on recipes, a catalogue of goods for sale, or an estate agent's register of houses, displaying, say, those in a particular price range or locality.

The great advantage of these methods is that really massive quantities of data can be stored neatly on tapes or floppy diskettes, and amended or updated, searched or sorted in a way which is just not possible using traditional document files on paper.

Throughout this book, jargon and technical details have been avoided where possible, but the following pages introduce a few specialist words necessary for the operation of the machine.

The Commodore 64

The Commodore 64 is a desk-top microcomputer which can store a set of instructions, known as a *program*. This enables it to perform various operations with far greater speed, accuracy and reliability than a human being. Today's micros are more powerful than the enormous computers of a few years ago, which filled a whole room

with their bulky valves and transistor circuits. This has been made possible by the development of the microprocessor, a minute 'chip' of silicon on which thousands of components are etched.

The processor, often known as the *central processing unit* (CPU) contains a *memory*, in which the current program is held, an *arithmetic and logic unit*, which does any sorting or calculations, and a *control unit* which transmits electronic signals around the machine to co-ordinate the various operations. We need not worry further about the technical details of these components.

The Commodore 64 also has a typewriter-style keyboard, with the keys arranged in the QWERTY configuration and a *visual display unit* (VDU) or screen, on which the results or *output* are displayed. You need have no fear of damaging the machine via the keyboard.

The Commodore 64 may be connected to add-on (peripheral) devices, such as printers, diskette drives, and cassette recorders. The diskettes and cassettes are used for saving programs and files of data since information held in the computer's memory is lost when the machine is switched off.

Tape cassettes are a cheap form of storage compared with disks, but are very much slower, since it is necessary to read the cassette from end to end to find a particular item of data. Recent developments should, however, bring the disk drive within the reach of many computer owners.

To give some idea of the efficiency of computer storage, a wafer-thin floppy diskette, 5¼ inches in diameter, can store 170000 characters.

Operating the computer

The cursor

When the machine is switched on, a flashing *cursor* appears on the screen. This corresponds to the nib of a pen, and is the point at which 'writing' or text is displayed. The cursor can be moved around the screen in the directions given on the cursor keys (CRSR) on the keyboard (the upper movements on the key being obtained by pressing SHIFT together with the required movement).

Before typing on the screen, it is a good idea to remove any unwanted text. This can be achieved by pressing SHIFT together with the key marked CLR/HOME. The Commodore 64 contains the 'auto-repeat' facility on the CRSR keys so that if you wish to move the cursor twenty spaces to the right, say, you simply keep the

required key held down, rather than making twenty individual keystrokes.

After typing a line of text, the cursor is advanced to the beginning of the next line by pressing the RETURN key. (The RETURN key must always be pressed whenever we need to enter an instruction to the computer. If we do not press RETURN, nothing will happen.) To move the cursor to the top left-hand corner of the screen, press the CLR/HOME key.

You can feel confident about typing anything you like on the screen. You cannot damage the Commodore 64 via the keyboard and the machine will give an 'error message' when you do something it does not like. If, for example, you type PLINT instead of the correct word PRINT the following error message will occur:

```
? SYNTAX ERROR
```

To overcome this, just retype the line correctly and press RETURN.

It is particularly easy to make corrections on the Commodore 64. To delete a character, just move the cursor to the character immediately to the *right* of the unwanted character and press INST/DEL.

To insert an extra character or characters, move to the *right* of the required position and press SHIFT together with INST/DEL the required number of times. The additional characters can now be typed in the space created.

The computer distinguishes between the letter O and the figure Ø. Care should be taken to ensure that these are not accidentally interchanged.

So, to start programming we just need to learn the language of most microcomputers, BASIC, or Beginners All-purpose Symbolic Instructional Code. Although slight variations or dialects exist for different models of computer, the main core is common to all and users have little difficulty in transferring between different types of machine.

Modes of operation

There are two alternative ways of operating the computer: immediate mode and program mode.

Immediate mode enables the machine to be used like a calculator. We simply give the computer a command, which it carries out without storing anything in its memory. For example:

```
PRINT "DAVID"
```

If we type this and press RETURN, the characters between inverted commas will be printed literally on the screen but not stored in the memory. Thus, the word DAVID is printed.

Immediate mode also enables the computer to be used as a calculator. For example:

```
PRINT 5+7
```

will cause the answer to the sum to be printed, and the answer appears on the screen. However, immediate mode is most used for typing in the commands which control the running of the computer.

Program mode enables a set of instructions to be stored in the memory. Each program consists of a set of lines or *statements*, which are numbered in ascending order so that the computer 'knows' the order in which to carry out these instructions.

The following single line is, in fact, a computer program, which may be stored, run, saved and loaded.

```
1ø PRINT "A PROGRAM"
```

To enter this program, simply type the line and press RETURN. This simple program will remain in the memory of the computer until one of the following events occurs:

- The machine is switched off.
- The command NEW is typed.
- Another program is LOADED from storage.

Of course, this simple example of a one-line program would be of little practical use on a powerful microcomputer, and in reality we will write programs which may be several pages long.

The size of the program which may be stored in the computer will be limited by the size of the memory of our particular machine. The jargon for memory is RAM (or Random Access Memory), and the unit of memory size is 1K, which means 1024 characters (i.e. numbers, letters, spaces, punctuation marks, etc.). So, a 64K machine will hold 64 × 1024 characters in its memory. (Here we are referring to 'user-available' memory; other areas of the memory contain permanently stored programs used in the operation of the computer. We cannot store in this part of the memory, only read from it, so it is called ROM – Read Only Memory.)

The sequence of events when developing a program is as follows:

- Write the program using pencil and paper.
- Perform a 'dry run', again using pencil and paper (testing the program with sample data).

- Type the program into the memory and test it by running using the RUN command.
- Modify and improve the program.
- SAVE the program on diskette or cassette.
- *Use* the program by loading from storage medium (diskette or cassette) whenever required, using the LOAD command.

Commands

The following list is by no means complete, but contains enough for the beginner to make a start. The commands should always be typed on a new line and followed by pressing the RETURN key.

NEW	Delete the current program from the computer's memory – always used before typing in a new program.
LOAD	Load the next program encountered on tape into the computer's memory.
LOAD "JILL"	Search for and load the program which has been saved with the name "JILL". No deviation from the original spelling of the name is possible; in this case, it would not load "JILLY".
LIST	Display on the screen all of the instructions or STATEMENTS of the program currently held in the computer's memory.
RUN	Execute or carry out the current program.
SAVE "JILLY"	Copy onto tape the program currently in the computer's memory, giving it the name "JILLY".
VERIFY	Check that the recording just made on tape is identical to the program in the memory.
More jargon:	
DATA	Raw facts (i.e. names, addresses, words, numbers, etc.).
HARDWARE	The pieces of equipment.
SOFTWARE	The programs, usually on cassette or diskette.

Chapter Two
Programming

Line numbers

We have seen that a program is a set of instructions stored in the computer, and that the order in which the instructions are carried out is determined by their *line numbers*. Special instructions also allow us to jump, out of sequence, forwards or backwards to a specified line within the program.

Line numbers usually are in the range ∅ to 63999 and it is common to ascend in steps of 1∅, so that intermediate lines may be added as an afterthought as your program evolves. For example:

```
1∅ PRINT "DAVID"
2∅ PRINT "RICHARD"
```

Inserting a line

We could, if we wish, insert another line as an afterthought, by typing LIST, pressing RETURN and appending our new line as follows:

```
1∅ PRINT "DAVID"
2∅ PRINT "RICHARD"
15 PRINT "JILL"
```

On pressing return at the end of line 15, this new line will be inserted in the program in the required position, between lines 1∅ and 2∅.

Deleting a line

To delete an unwanted line, simply type in the line number and press return. For example, 2∅ followed by RETURN will delete line 2∅ from the memory.

It is good practice to leave ample space between line numbers since we invariably decide to insert further statements later.

Most data processing programs involve the same fundamental operations, such as:

- Reading data into the computer
- Carrying out some sorting, searching (or calculating)
- Amending or updating the data already stored
- Outputting the results by displaying on the screen or printing on paper ('hard copy').

Later on we will see how it is possible to choose the particular part of the program we wish to use by selecting from a list of options or *menu*. The menu gives us access to the various blocks of the program, known as *modules*, and we find that the same modules can be frequently used in different programs. Writing programs, then, involves simply assembling the various modules we require rather than writing a complete program from scratch.

Figure 2.1 shows the plan for a typical data processing program.

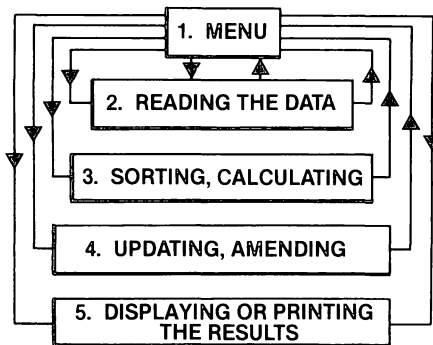


Fig. 2.1. A modular program.

From the menu it is possible to move to any of the program blocks or modules, before returning to the menu and choosing another module to access.

Little skill is required to write the program in this way, since special instructions are available and are discussed later. The reason for introducing the idea of modules now is that it affects the way we allocate line numbers to our program.

Instead of starting at line 1Ø and letting our program grow erratically to, say, line 479, it is often better to decide on the modules we need. We then allocate blocks of, say, 1ØØØ to each module and write and develop each module as a separate program, as shown in Fig. 2.2.

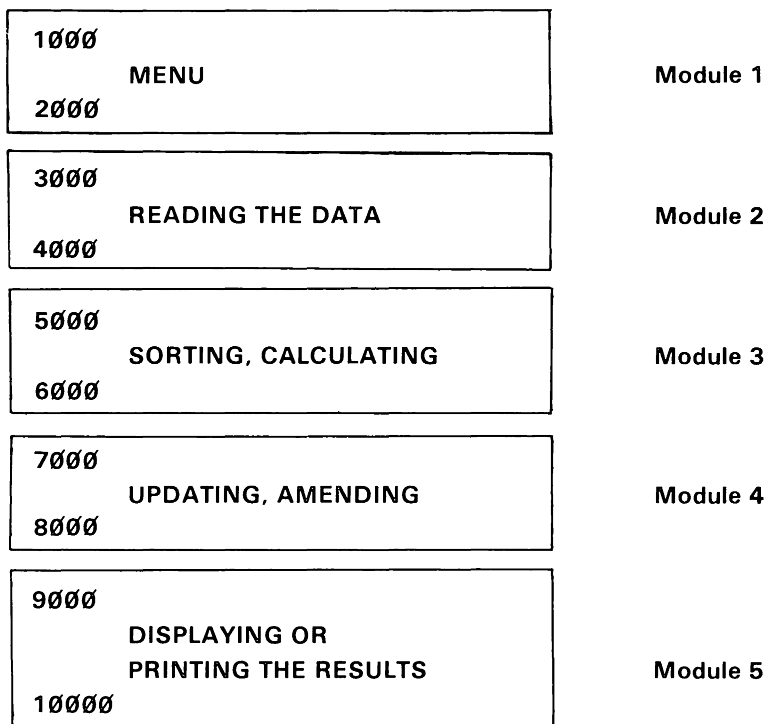


Fig. 2.2. Allocating blocks of line numbers.

Although you do not yet have the expertise in the language to write such a program, it is helpful to appreciate at this early stage the idea of 'blocking' line numbers with ample space between them.

Learning the language

BASIC consists of simple statements, i.e. words or groups of words, in English, which instruct the computer to do something. These are usually concerned with reading or printing data, or making decisions.

The PRINT statement is one of the most frequently used, since it enables the contents of the computer's memory to be communicated to the user. PRINT enables you to:

- Display any message you type in, on the VDU (screen).
- Display on the screen a copy of the contents of the memory and the results of our programs.

- Print on paper ('hard copy') the results or answers for our programs.

Try this:

```
1Ø PRINT "YOUR NAME"
2Ø GOTO 1Ø
```

(press RETURN after each line)

Type RUN and press RETURN to execute or run the program.
Press STOP to end the running of the program.
Now clear the screen and LIST followed by pressing RETURN.
Alter Line 1Ø, or retype, with a semi-colon at the end.

```
1Ø PRINT "YOUR NAME";
2Ø GOTO 1Ø
```

Type RUN, press RETURN and note the effect of the semi-colon.
Now repeat, but replace the semi-colon in line 1Ø with a comma.

The PRINT statement can also be used to print blank lines, i.e. spaces, between each line of output (Fig. 2.3):

```
10 PRINT"YOUR NAME"
20 PRINT
30 GOTO 10
```

Fig. 2.3. Printing blank lines.

After you have RUN this, experiment with line 2Ø as shown in Fig. 2.4 in which the colon has been used to allow multiple statements to appear on one line. This can be used for statements other than PRINT, and saves space; it also makes programs harder to understand!

```
10 PRINT"YOUR NAME"
20 PRINT:PRINT:PRINT
30 GOTO 10
```

Fig. 2.4. Multiple statements on one line.

To slow the computer down, we simply make it count up to a certain number, as shown in Fig. 2.5.

```
10 PRINT"YOUR NAME"
20 PRINT
25 FOR T= 1 TO 100:NEXT T
30 GOTO 10
```

Fig. 2.5. Inserting a time-delay loop.

In line 25, we have now introduced a FOR ... NEXT loop, which is discussed shortly, but simply makes the computer repeat a process a certain number of times – in this case 1ØØ. Try experimenting with different values of T. Try, for example:

```
25 FOR T=1 TO 5ØØ:NEXT T
```

or

```
25 FOR T=1 TO 1ØØØ:NEXT T
```

Note: Each time we type in a new line 25 and press RETURN, the old line 25 is deleted from the memory, being overwritten by the new version. Also note that the new line 25 is automatically slotted into the correct position in the program order – check this by typing LIST followed by pressing the RETURN key.

Remember the commands for operating a program:

- NEW followed by RETURN before starting a new program.
- RUN followed by RETURN to execute the program.
- LIST followed by RETURN to display the statements to allow alteration (editing) or appending further statements. (LIST 2ØØ say, will list the individual statement at line 2ØØ. We may also use LIST 1ØØØ-2ØØØ, say, or LIST -5ØØØØ or LIST 7ØØØ- to give partial listings).

Repeating an operation

We have seen that GOTO allows us to go back to an earlier line, and repeat a process indefinitely.

It is often useful to repeat a process a specified number of times. We could print a word 10 times as shown in Fig. 2.6, by enclosing the PRINT statement in a FOR ... NEXT loop.

```
10 FOR N=1 TO 10
20 PRINT "YOUR NAME"
30 PRINT
40 NEXT N
```

Experiment with different values for N, e.g. 1Ø FOR N = 1 TO 5Ø.

Fig. 2.6. Repeating a process – the FOR ... NEXT loop.

Every FOR statement must have a corresponding NEXT statement to complete the LOOP.

In the above example, N is a counter to tell the computer when the

required number of 'passes' through the loop has been achieved. We could also use N in the process carried out *within* the loop as shown in Fig. 2.7.

```
10 FOR N=1 TO 10
20 PRINT N
30 NEXT N
```

Type and run this and note what happens.

Fig. 2.7. Using the FOR ... NEXT loop to increment the contents of a store.

Here N is being used as a *store*, rather like a box containing any number we care to put in. The FOR ... NEXT loop first puts 1 in the store labelled N and at line 2ϕ prints on the screen a copy of the contents of this store. Line 3ϕ continues the loop, returning to line 1ϕ where the contents of store N are changed to the next value, i.e. 2, and these are printed at line 2ϕ. This process continues until the final figure in the FOR statement is stored. In this example, at the end of the loop, store N will contain 1ϕ, all previous values having been *overwritten*. The program will now proceed to the next line after the FOR ... NEXT statement, 4ϕ, and store N will continue to contain the value 1ϕ unless it is overwritten by a new value later in the program.

Our FOR statement can contain any range of numbers – for example,

```
1ϕ FOR T=5 TO 1ϕϕϕ
```

and we could specify an increment or *step* other than 1, such as

```
1ϕ FOR T=5 TO 1ϕϕϕ STEP 5
```

which means that the contents of T will successively be 5, 1ϕ, 15 ... 1ϕϕϕ.

Note that any letter may be used as the store in the FOR ... statement, but it must have an associated NEXT statement, e.g.

```
2ϕ FOR X=1 TO 1ϕϕϕ
-------------------
3ϕ NEXT X
```

To appreciate the power of your computer, run the little program in Fig. 2.8, which prints out the squares of the first 1000 numbers. Note the spaces which have been incorporated to improve readability.

```
20 FOR M=1 TO 1000
30 PRINT M" TIMES "M" IS "M*M
40 NEXT M
```

Fig. 2.8. The squares of the first 1000 numbers.

Screen layout

The most brilliant program is no good if the output from it is a jumbled mess which the user cannot easily understand. Obviously, the screen should display only information which is necessary at any particular stage of the program.

To clear the screen during a RUN of the program, the clearing operation is incorporated into a PRINT statement as follows:

```
90 PRINT"♥MICRO"
```

When the SHIFT/CLR keys are depressed inside inverted commas a graphics character appears - ♥ The effect in this example is to clear the screen before printing the word MICRO.

To improve the layout, we may wish to start printing near the centre of the screen. We have seen how to move down the screen using, for example, the following routine which would leave three blank lines:

```
7Ø PRINT:PRINT:PRINT
```

TAB

We can also move from left to right by leaving a specified number of blank spaces before printing begins.

```
8Ø PRINT TAB(15)"MICRO"
```

The line above leaves 15 spaces before printing starts. Now try to print your name in the centre of a clear screen.

SPC

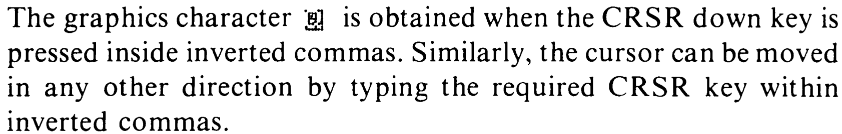
Any number of spaces can be printed in between two strings of characters by using the space function. For example,

```
PRINT"TOM"SPC(12)"BROWN"
```

would leave 12 spaces between TOM and BROWN.

Programmed cursor movements
A further feature of the Commodore 64 is the ability to program movement of the cursor within the inverted commas in a PRINT statement. For instance, to begin printing 5 rows down we would use:

```
100 PRINT "MICRO"
```



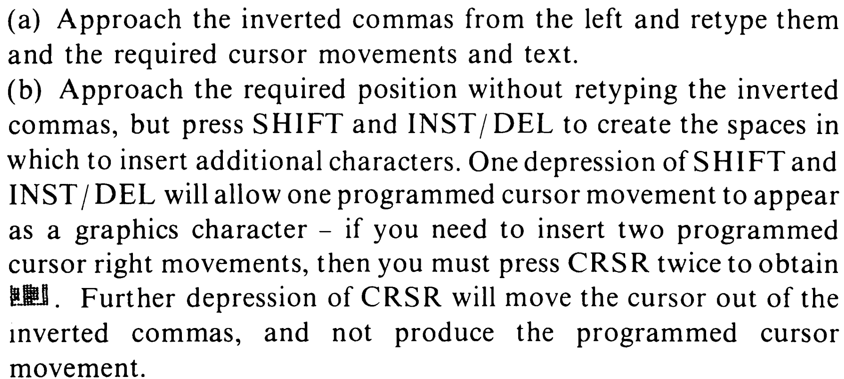
The graphics character is obtained when the CRSR down key is pressed inside inverted commas. Similarly, the cursor can be moved in any other direction by typing the required CRSR key within inverted commas.

Multiple cursor operations
Several operations may be programmed in one PRINT statement:

```
100 PRINT "MICRO"
```

The above programmed cursor operations would first clear the screen (SHIFT/CLR), then move the cursor down 4 rows (CRSR), and then 4 columns to the right (CRSR). (*Note:* It must be stressed that the above graphics characters are only produced after the inverted commas are typed and these must be closed after the required PRINT statement has been completed.)

If, subsequently, you want to amend a line containing these programmed cursor movements, you must do either of two things:



(a) Approach the inverted commas from the left and retype them and the required cursor movements and text.
(b) Approach the required position without retyping the inverted commas, but press SHIFT and INST/DEL to create the spaces in which to insert additional characters. One depression of SHIFT and INST/DEL will allow one programmed cursor movement to appear as a graphics character – if you need to insert two programmed cursor right movements, then you must press CRSR twice to obtain . Further depression of CRSR will move the cursor out of the inverted commas, and not produce the programmed cursor movement.

The advantage of these programmed cursor movements on the Commodore 64 is that we can achieve printing anywhere on the screen in a much more economical way than using several

PRINT:PRINT statements or PRINT TAB, etc. Try experimenting with the cursor movements to position a word anywhere on a clear screen.

Further operations are possible, such as printing titles on a white background to improve appearance. This requires the inclusion of reverse field (RVS ON) after the beginning of the inverted commas and RVS OFF before the close, as shown in Fig. 2.9.

```
100 PRINT"▒ WHITE BACKGROUND ▒"
```

Fig. 2.9. Switching reverse field on and off to obtain a white background to the enclosed characters.

Summary of uses of the PRINT statement

The PRINT statement is of paramount importance since it is the main source of output from the computer. The various forms in which it is used are summarised below.

PRINT "JMK:1Ø1"	Any combination of characters may be printed *literally* by enclosing between inverted commas.
PRINT A	Displays the number stored in address A.
PRINT"NUMBER IS ";A	Prints the message in inverted commas, followed by the contents of store A. e.g. NUMBER IS 17
PRINT A$	Displays the contents of store A$ – a string of characters.
PRINT A,B	The comma causes the screen to be divided into 4 blocks of 10 columns each, the data being printed at the beginning of each block.
PRINT"SMITH","JONES", "BROWN"	This produces: SMITH JONES BROWN
PRINT"SMITH";"JONES"	The semi-colon causes the next item to be printed *immediately following* the

	previously printed item. The example shown would produce: SMITHJONES
PRINT"SMITH" PRINT"JONES"	The semi-colon again causes the next item to be printed to follow immediately from the previous items, again producing SMITHJONES. (Separation could be achieved by pressing a space at the end of SMITH, i.e. "SMITH ".)
PRINT TAB(X);A$	X spaces are left blank before printing the contents of string store A$.
PRINT"SMITH"SPC(17)"JONES"	17 spaces are left between SMITH and JONES.

Abbreviations

Most of the Commodore 64 commands can be shortened. For PRINT, the abbreviation is ?, while many other statements require only the first two letters to be typed. While typing the second letter, SHIFT is depressed.

```
SAVE=S SHIFT A=S♠
LOAD=L SHIFT O=L┌
LIST=L SHIFT I=L╮
VERIFY=V SHIFT E =V▔
RUN=R SHIFT U=R╭
```

Fig. 2.10. Some abbreviations used on the Commodore 64.

As an exercise, try finding out for yourself the complete list of abbreviations on the Commodore 64.

Further uses of the PRINT statement – ASCII Codes, CHR$, ASC

So far we have seen that PRINT can be used to display, on the screen, the contents of a store:

```
10 PRINT A$
```

Or to display, literally, whatever appears between inverted commas:

```
10 PRINT "THE COMMODORE 64"
```

In addition, it is possible, using a system of code numbers, to use the PRINT statement to carry out certain *operations*, as well as printing text. For a particular make of machine, there is a set of codes, although most computers use a standardised set, known as the American Standardised Code for Information Interchange (ASCII, pronounced 'Askey').

Internally the computer works with strings of 0's and 1's, so that, for instance, the letter A is 0100 0001, which is the binary arithmetic equivalent of the decimal number 65. These decimal equivalents of the machine's binary codes form the set of ASCII codes.

CHR$

On the Commodore 64, the various characters have an ASCII code of between 32 and 127, with further graphics characters from 160 onwards. The full list of ASCII codes is given in the Commodore 64 handbook, but a few examples are given in Fig. 2.11.

		ASCII code	*Binary*
Digits:	0	48	0011 0000
	1	49	0011 0001
	2	50	0011 0010
	.	.	.
	.	.	.
	.	.	.
	9	57	0011 1001
Letters:	A	65	0100 0001
	B	66	0100 0010
	C	67	0100 0011
	.	.	.
	.	.	.
	.	.	.
	Z	90	0101 1010
Instructions:			
Carriage RETURN		13	0000 1101
Change to lower-case letters		14	0000 1110
Change to upper-case letters		142	1000 1110

Fig. 2.11. Examples of the ASCII codes and their binary equivalents.

The ASCII codes may be used in a PRINT statement to perform *operations*, as well as PRINT on the screen. For example:

PRINT CHR$(14)

This means 'carry out the operation represented by the character string with ASCII code 14'. In this case the screen display will all be changed to lower-case lettering. To revert to upper-case letters type:

PRINT CHR$(142)

(While operating with lower-case lettering, it is still possible to obtain capitals by depressing SHIFT with the required letter. Lower-case letters give a neater and more readable appearance on large sections of text.)

It is, if desired, possible to print the normal alphabetical, numerical and graphical characters using PRINT CHR$, if for any reason we want to make a program harder to understand!

```
10 FOR N=33 TO 127
20 PRINT CHR$(N)
30 NEXT N
```

Fig. 2.12. Experimenting with CHR$.

Run, for example, the program in Fig. 2.12 and you will see that it prints out a large part of the character set for the Commodore 64.

PRINT CHR$ is also useful for programming operations which do not result in any screen display. Later we will see how it is necessary to program a carriage RETURN character (which has an ASCII code representation in the same way as a letter, etc.). This is used in data files to separate items of data.

ASC

PRINT ASC is the reverse of the PRINT CHR$ statement. PRINT ASC produces the code number for a character string or operation. It is used in the following ways:

PRINT ASC("B")

would print the ASCII code for the letter B.

PRINT ASC(B$)

would print the ASCII code for the first letter of string B$. Now try the program in Fig. 2.13.

```
10 INPUT G$
20 PRINT ASC(G$)
30 GOTO 10
```

Fig. 2.13. A program to print the ASC code of any character which is input.

When you run this program, a question mark will appear. Type in any string you like and press return. The screen will display the ASCII code of the character you entered (or the first character of a longer string).

Chapter Three
Storing Data in the Memory

Numbers

In order to do useful processing, we need to store our words and numbers in the memory, so that we can retrieve them whenever they are needed for sorting, calculating or printing. The data will be stored in locations within the memory which can be thought of as boxes. We must label the boxes in some way so that we know where the data is stored. One of the most common ways is to store all numbers in locations labelled with the letters of the alphabet. For example,

2∅ LET A=7	Line 2∅ instructs the computer to store the
3∅ PRINT A	number 7 in the location labelled A.

If we run this program, line 3∅ causes the contents of store A to be printed, i.e. 7 is output on the screen.

Line 3∅ tells the machine to print out a *copy* of the contents of store location A, while leaving the store unchanged.

Overwriting the contents of a store

During the course of a program it is possible to assign a new value to a store, replacing the original contents. If we add the following lines to the above program:

```
4∅ LET A=42
5∅ PRINT A
```

and RUN

and then press the RETURN key, the output will be

```
7        followed by
42
```

(*Note:* Both values of A are printed, but after running, only 42 remains in the store.) Line 4ø *overwrites* the contents of store A with the number 42 and line 5ø displays the new contents.

We say that the number 42 has been assigned to the store A, and this value will remain in store A until it is overwritten by a new value. Of course, if the machine is switched off or a new program is loaded, the current value of store A will be lost.

We can use any letters for storing numbers, but it is usual to choose one which reminds us of the data it is holding. For instance, we might decide to store Tax in store T, or a mark in store M. Because the contents of the store may change, we say that store A (or B or C etc.) is a *variable.*

We can also change the contents of a store by adding, subtracting, etc., other numbers in a way which makes no sense mathematically. For example:

```
2ø LET C=13
3ø LET C=C+1
4ø PRINT C
```

Run this and note the effect. Line 3ø actually means: 'Let store C now contain the old value of C plus 1'.

Although we are avoiding mathematics as far as possible, to reinforce the idea of a store's contents being *overwritten* by a new value, try these:

```
1ø LET Z=7
2ø LET Z=15*Z         (* means multiply)
3ø PRINT Z

1ø LET T=18
2ø LET T=T/2          (/ means divide)
3ø PRINT T
```

We can also assign the sum of the contents of other stores to a new store as shown in Fig. 3.1.

```
10 LET A=7
20 LET B=15
30 LET C= A+B
40 PRINT C
```

Fig. 3.1. The use of numeric stores to hold two numbers and their sum.

It is often useful to give meaning to the variables by including a message in our print statement, before printing out the contents of the store (see Fig. 3.2). Remember, whatever is included between the inverted commas in a print statement will be output *literally*, and extra spaces may be inserted to improve legibility.

```
10 LET Q=15
20 PRINT "MY NUMBER IS "Q
```

Fig. 3.2. Including a message before a printed answer.

The output of the program in Fig. 3.2 will be:

MY NUMBER IS 15

We have touched on some elementary arithmetic in order to demonstrate, as simply a possible, the idea of stored data. In most of our data processing, although we may store both numbers and words, we may not do much calculating on the data. However, for those who wish to do some more advanced calculating with the computer, a calculating section is included in Appendix 2.

Character strings

We can now consider the most important aspect of storing in data processing, i.e. names and *strings* of characters (numbers, letters, etc.) such as car registration numbers, names and addresses, telephone numbers, employee records etc.

To distinguish string data from pure numbers, we assign the strings to stores whose letters are followed by a $ sign, e.g.

```
1ø A$="JACK"
2ø PRINT A$
```

Note: The inverted commas must always be present in assignment statements such as line 1ø. We can still hold *numbers* in the stores for strings, but these will then be treated literally as a succession of individual figures which cannot be used in calculations. Like the figures in a car registration number or a telephone number, they have no mathematical significance.

VAL

Sometimes it is more convenient to enter numerical data in string form, but because calculations subsequently need to be performed,

this data must be converted into 'mathematically workable' numbers – hundreds, tens, units, etc.

This is achieved by the VAL function, which works only on those digits at the *beginning* of a string, e.g.

```
1∅ LET A$ = "2∅∅∅ ABC"
2∅ LET A = VAL(A$)
3∅ PRINT A
```

In line 1∅, 2∅∅∅ is simply a figure 2 followed by three individual ∅'s. VAL(A$) converts this string of separate characters into the number two thousand. The contents of store A can now be processed mathematically.

STR$

To reverse this process, i.e. to convert numbers into strings of individual characters, the STR$ function is used, e.g.

```
LET A$ = STR$(A)
```

We can also combine strings by "adding" them as in Fig. 3.3.

```
10 F$="TOTTENHAM "
20 S$="HOTSPUR"
30 PRINT F$+S$
```

Fig. 3.3. Combining two strings – 'concatenation'.

The + sign, of course, has no mathematical meaning when used with string data, and simply causes the two words to be printed next to one another. This is called *concatenation.*

READ and DATA

To save typing in our data in LET statements (which incidentally will work without the word LET e.g. A$="JONES"), we can include them in combined READ and DATA statements (see Fig. 3.4).

```
10 READ A,B,C
20 DATA 15,27,32
30 PRINT A,B,C
```

Fig. 3.4. A program to READ 3 numbers from DATA.

The READ and DATA statements are mutually dependent, every READ demanding a corresponding DATA statement, which must

match in every way. Line 1∅ assigns the first number to store A, the second to store B and so on.

The DATA statements may appear anywhere in the program, but it is common practice to include them at the end, so that additions may be made with the minimum disturbance to line numbering in the program.

Now we may write a program to store 4 names and print them on the screen (see Fig. 3.5).

```
10 READ W$,X$,Y$,Z$
20 PRINT"FIRST NAME IS "W$
30 PRINT"THIRD NAME IS "Y$
40 PRINT"SECOND NAME IS "X$
50 PRINT"FOURTH NAME IS "Z$
60 DATA BROWN,JONES,WALKER,DOBSON
```

Fig. 3.5. A program to READ and PRINT names from DATA.

Remember, unless we give the computer new information, store W$ will hold the name BROWN while this program is in the memory.

We could have printed the names out by simply typing

2∅ PRINT W$,Y$,X$,Z$

Try this and note the effects of the comma on the screen layout.

Entering large quantities of data - subscripts

We can enter very large quantities of data into the memory by using the idea of a *subscript*. Suppose we are using the store name V$. We can use this name to store a hundred different names by calling them V$(1), V$(2), V$(3), V$(4) ... V$(1∅∅). For these large quantities of data we would have run out of names using A$, B$, C$, etc.

In addition, the programming becomes much simpler, since we can READ and PRINT our hundred items using single instructions for each operation.

We will demonstrate the principle using only four names, but we can easily extend the method to cover a hundred items.

First we must enclose our READ statement in a FOR ... NEXT loop (Fig. 3.6).

The FOR ... NEXT loop is simply a counter which instructs the computer to carry out the enclosed instruction the specified number of times (in this case, four).

```
10 FOR I=1 TO 4
20 READ A$(I)        Read module
30 NEXT I
40 FOR P=1 TO 4
50 PRINT A$(P)       Print module
60 NEXT P

70 DATA JILL,JIM,DAVID,RICHARD   Data module
```

Fig. 3.6. Using subscripted variables to READ and PRINT several names.

On the first 'pass' through the loop, I has the value 1, so the first name in the DATA, i.e. JILL, goes into store A$(1). Similarly, A$(3) will contain DAVID and A$(4) will contain RICHARD.

We check this in the printing loop in lines 4∅–6∅. Note, also, that the letter used in the subscript need not be I; any letter may be used, but it must be used throughout the loop, i.e. in the FOR statement, the NEXT statement and as the subscript.

Using immediate mode to 'peep' at the contents of a store

It is sometimes useful to check the contents of a store by typing in immediate mode as follows:

```
PRINT A$(2)
```

If the above is typed, without a statement number, on pressing RETURN, we should see the contents of store A$(2) copied onto the screen (in this program JIM should appear).

Warning the computer to expect large quantities of data

The computer needs to allocate space in its memory for large quantities of data and needs notification at the beginning of the program. The DIM (dimension) is used to specify the maximum number of items of data being stored in a particular subscripted store (known as an *array*).

Supposing names are stored in an array known as N$(I) and although at present we have seventy names to store, we may need to extend the data later to hold a hundred names. We would prepare the computer by the dimension statement as follows:

```
1Ø DIM N$(1ØØ)
```

More than one variable name may be dimensioned in one statement:

```
1Ø DIM N$(1ØØ),T$(1ØØ)
```

Error Messages

Now that we have covered several BASIC programming operations, it is perhaps worth summarising some of the most common errors which occur.

Syntax error
This is frequently a spelling mistake in the BASIC language, such as PLINT instead of PRINT or LIFT instead of LIST. A syntax error may also be caused by a missing comma or a missing bracket.

Out of data
Every time a READ statement is executed, there must be corresponding DATA, which matches the expectations of the READ statement exactly. Commonly this occurs when not enough data has been included. Also, when adding several lines of DATA, if the user forgets to press the RETURN key at the end of a line, two or more lines of data will be 'concatenated' and in effect the number of separate items of data will be reduced. (The RETURN key must be pressed even if the cursor automatically returns to the start of a new line during the normal typing of data.)

Bad subscript
If you are using subscripted variables and you have tried to store more than eleven pieces of data, then there must be a dimension statement (DIM), specifying the maximum number of elements in the array. For example, the program in Fig. 3.7

```
100 DIM A$(20)
110 FOR I=1 TO 30
120 READ A$(I)
130 NEXT I
140 DATA ........
```

Fig. 3.7. An inadequate number in the DIM statement.

The program in Fig. 3.7 gives an inadequate number in the DIM statement and will produce the error message:

BAD SUBSCRIPT ERROR IN 120

This is because the program is only prepared to expect a maximum of twenty elements for store A$(I) by the DIM statement.

This can be corrected by altering line 100 to:

100 DIM A$(30)

Of course, if we have not supplied thirty items in the DATA statement, we will now produce:

OUT OF DATA IN 120

Type mismatch error

This occurs when we mix numeric data with string variables, or vice versa as in Fig. 3.8:

```
10 READ A$
20 PRINT 9*A$
30 DATA 7
```

Fig. 3.8. A type mismatch error: attempting to multiply a string.

This results in the error message:

TYPE MISMATCH ERROR IN 20

Line 10 reads the first item of data into string store A$. This is permissible, but the 7 is only stored as the character string 7, not as the mathematical number 7. Since the character has no numerical value it cannot be multiplied, as instructed in line 20 and the error is reported.

Of course, numeric data such as telephone numbers can be stored quite legitimately in string stores (A$,B$,etc.), as long as we do not attempt to perform calculations on the numbers.

Redim'd array

This occurs when you accidentally repeat the DIMension statement for a particular variable – only one is permissible. For example:

10 DIM A$(100)
20 DIM A$(200)

This will produce

REDIM'D ARRAY ERROR IN 2∅

NEXT without FOR

Every FOR ... statement requires a corresponding NEXT statement to complete the loop and vice versa. This error is particularly likely to occur if several loops are 'nested', as shown in Fig. 3.9.

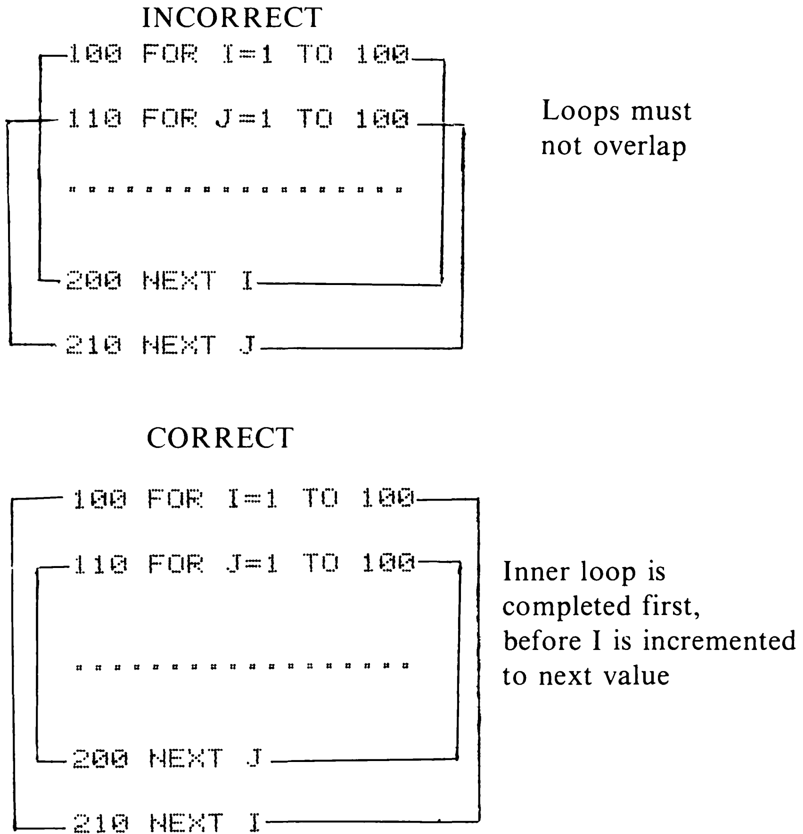


Fig. 3.9. Loops may be 'nested' but must not overlap.

The incorrect version above will cause the message

NEXT WITHOUT FOR ERROR IN 21∅

Chapter Four

Developing the Program

Now that we have covered many of the basic program elements, we can start to assemble a complete program. As our example, we shall try developing, step by step, a personal telephone directory. To demonstrate a complete program as simply as possible, we will include only five names and telephone numbers in our data statements, but dimension to twenty to allow for additions (Fig. 4.1). The method for dealing with, say, a hundred names would be basically the same, except that we would need to alter the numbers in the DIM and FOR statements.

```
10 DIM N$(20),T$(20)
20 FOR I=1 TO 5
30 READ N$(I),T$(I)
40 NEXT I
50 FOR I=1 TO 5
60 PRINT N$(I),T$(I)
70 NEXT I

500 DATA DR. FOSTER,GLOUCESTER 3694
510 DATA ACME MOTORS,SUTTON 319
520 DATA RED LION,LONGFORD 2483
530 DATA CHEMIST,ELY 472
540 DATA JILL,ETWALL 572
```

Fig. 4.1. A program to READ and PRINT five names and telephone numbers.

We have put the data statements at the end of the program to allow additions to be made as our directory is increased in size. The line numbers between 7∅ and 5∅∅ are available to include additional program lines to do extra operations like *sorting* in alphabetical order or *searching* for a particular name, and these are covered later in the chapter on searching and sorting.

Holding a screen display
It is often useful to allow the user to study some text displayed on the screen, and then to allow the user to proceed when ready by pressing the space bar. For instance, we may have a catalogue of house particulars stored in the memory, and the user may wish to study each house for a while before displaying the next one.

Fortunately, there is a statement which expects something to be typed into the keyboard and assigns it to a store location. This is the GET statement, and is similar to the INPUT statement to be discussed later. GET is faster than INPUT; it does not display a question mark and does not wait for the RETURN key to be depressed, as the INPUT statement does.

GET

This statement may be used as follows:

```
62 PRINT "PRESS ANY KEY TO CONTINUE"
65 GET G$:IF G$=""THEN 65
```

If inserted in our telephone directory program with these line numbers, it would have the following effect:

During the first pass through the FOR ... NEXT loop between lines 5ø and 7ø, the computer would print out at line 6ø N$(1) and T$(1), the first pieces of data.
At line 62 the message "PRESS ANY KEY TO CONTINUE" is displayed on the screen, and this is 'frozen' by the GET statement at line 65.
Line 65 actually means 'get something which will be typed at the keyboard, and assign it to store G$'. The colon then signifies a new statement.

```
65 GET G$ : IF G$ ="" THEN 65
```

The IF ... THEN statement says 'if nothing is typed at the keyboard, go back to the beginning of line 65'. (Two pairs of inverted commas " " with nothing in between are interpreted as nothing.) So, if *anything* at all is pressed, the IF ... THEN statement will not be true, the computer will not 'hang' at line 65 but instead will continue onto the next line of the program.

This is not always ideal, since if we are studying an interesting display on the screen, and we accidentally press *any* key, the computer will move on to the next part of the program.

We can overcome this by only allowing the computer to proceed if one *particular* key is pressed, commonly the space bar, e.g.

```
62 PRINT "PRESS SPACE BAR TO CONTINUE"
65 GET G$:IF G$<> " " THEN 65
```

We have introduced the symbols < (less than) and > (greater than) which when combined mean 'not equal to'. So, if anything *other than* the space bar is pressed, the computer hangs in an endless loop at 65. However, if we do press the space bar, so that G$=" ", and therefore the IF ... THEN statement is not true, we do not return to the beginning of line 65 but continue with the next line in the program.

To show how the 'press space bar' routine may be used, consider a file of one hundred used cars. The details of the first car would be displayed on the screen using PRINT statements in the following output.

```
MAKE:         FORD
MODEL:        ESCORT
YEAR:         1984
CONDITION:    IMMACULATE
PRICE:        £4,500

PRESS THE SPACE BAR TO CONTINUE
```

On pressing the space bar, the second car will be displayed, and so on, working through the whole catalogue of cars in the data file. (*Note:* " " is programmed by a single depression of the space bar within inverted commas.)

This small *routine* can be inserted anywhere in the program, whenever we want to stop execution for a while and then continue when we are ready. It would be much easier if, however, we only typed the routine once, yet could refer to it whenever necessary.

Subroutines

To save typing in the PRINT, GET and IF ... THEN statements every time, we can use what is known as a *subroutine* (discussed later). This idea is used whenever we have a set of program lines which we may wish to use repeatedly.

The frequently used subroutine is included once at the end of the program. The advantage is that we can jump to the subroutine and RETURN from it whenever needed.

Our "PRESS SPACE .... " routine could become, say, a subroutine starting at line 8∅∅∅.

```
8∅∅∅ PRINT "PRESS SPACE BAR TO CONTINUE"
8∅1∅ GET G$:IF G$<>" "THEN 8∅1∅
8∅2∅ RETURN
```

To call up the subroutine, we simply write:

```
62 GOSUB 8∅∅∅
```

At line 62 in the program, the computer jumps to line 8∅∅∅, performs the statements 8∅∅∅ and 8∅1∅, and after pressing the space bar, line 8∅2∅ instructs the computer to RETURN to the line *immediately* following our GOSUB 8∅∅∅ at line 62. We can, therefore, use this subroutine as often as we require by simply inserting GOSUB 8∅∅∅. Later we will see how our entire program can consist of subroutines, to which we gain access by a menu or choice of options.

We will now revise the techniques we have covered so far by incorporating them into our telephone directory program (see Fig. 4.2).

```
10 DIM N$(20),T$(20)
20 FOR I= 1 TO 5
30 READ N$(I),T$(I)
40 NEXT I
50 FOR I=1 TO 5
60 PRINT N$(I),T$(I)
62 GOSUB 8000
70 NEXT I
500 DATA DR. FOSTER,........
510 DATA...........
1000 END
8000 PRINT "PRESS SPACE BAR TO CONTINUE"
8010 GET G$:IF G$<>" " THEN 8010
8020 RETURN
```

Fig. 4.2. A short program to display records one at a time. The user must insert additional DATA at lines 51∅, 52∅, etc.

At line 1∅∅∅, END has been inserted to stop the computer when it has finished the program and to prevent it from entering the subroutine unintentionally, which would produce an error message.

Using READ and DATA, if we save our program on tape, we have a permanent record of our data which can be updated by typing

in a new line number or amended by typing in a replacement for an existing line. In this way it would be possible to change a person's telephone number or add a new person to our directory. This method has been used successfully to keep personnel records of hundreds of people on tape, although more sophisticated methods of *file handling* are discussed later. We will also see later how it is possible to sort the records into alphabetical order – a relatively simple task for the computer, but a daunting one by manual methods.

INPUT

Another method of supplying data to the computer is the INPUT statement, in which the computer displays a question mark and waits for the user to INPUT the data. One disadvantage with this method is that no permanent record of the data is stored when the program is saved. To demonstrate the use of INPUT, try running the following short program:

```
10 INPUT N$
20 PRINT N$
```

After RETURN is pressed, the computer will display a question mark. If the user responds by typing in a name, the name will be stored in store N$ for the duration of the RUN of the program, unless subsequently overwritten. On a new run of the program, store N$ will be set to zero, before a new INPUT statement assigns another value to store N$.

INPUT is usually used with in-built prompts to the user, as shown in Fig. 4.3.

```
20 INPUT "WHAT IS YOUR NAME";A$
30 PRINT "WELL "A$" I AM VERY PLEASED"
40 PRINT "TO MEET YOU"
```

Fig. 4.8. The use of a PROMPT during INPUT.

We have now seen how to enter data, such as names or numbers, into the computer, and to display them on the screen by copying from the memory using the PRINT statement. We have also seen how to halt the program while we study the display, and then continue the program, stopping and restarting by pressing the space bar.

Unfortunately, our simple example of five telephone numbers does not justify the use of a powerful computer and the next section gives methods of automatically counting large quantities of data as it is entered. We will also see how it is possible to check when the end of the data is reached.

Chapter Five

Making Decisions

Although early computers were greatly used for mathematical and scientific calculations, it is now the case that *processing data* is a major computing activity. In particular, the computer may be used to read into its memory the details of hundreds of cars, and pick out the owners, say, of red Ford Escorts, far more quickly than is possible by human beings using manual methods. Similarly, hundreds of names and associated details could be sorted into alphabetical order, or a group of students' names and examination marks could be sorted into the rank order. This is the type of work for which computers are especially suited to improve efficiency and reduce boring repetitive work.

The searching can be achieved very simply by the IF ... THEN ... statement already discussed. We need first to learn a few simple symbols used when comparing data, such as < (less than), which can be used with both numbers and letters.

If A < B THEN ... (If A is numerically less than B)
If A$<B$ THEN ... (If the word in store A$ comes *before* the word in B$ in the alphabet)

Similarly > means 'greater than' with numbers and 'later in the alphabet than' with words.

We will also frequently use =, meaning 'numerically equal to' or, in the case of character strings, 'identical to'. Similarly <> , is used to mean 'not equal to', and less frequently we may need:

<= less than or equal to
>= greater than or equal to

For example:

```
1Ø INPUT "TYPE IN MAKE OF CAR";C$
2Ø IF C$ = "PORSCHE" THEN .........
```

If the equality at line 2∅ is satisfied, i.e. the name Porsche is typed in response to the INPUT question mark, then we branch to the line number following THEN.

We could also print something after THEN:

2∅ IF C$ = "PORSCHE" THEN PRINT N$

where N$ might be the name of the owner.

The idea of searching for a particular item of data can be illustrated by a modification to our telephone program. In our first attempt we have specified the number of sets of data to be read, in this case 5. However, if we had huge quantities of data, it is more efficient to 'ask' the computer to count it automatically (by simply inserting a counter C = C+1 after it has read each item) and to search the data for a deliberate *end of data marker*. This piece of 'dummy data' will be tested for each time the data is read. When it is eventually found, the computer will leave the reading loop and move on to the next part of the program (see Fig. 5.1):

```
10 DIM N$(20),T$(20)
20 LET C=0
25 LET I=1
30 READ N$(I),T$(I)
35 IF N$(I)="FINISH" THEN 50
40 C=C+1:I=I+1
45 GOTO 30
50 FOR I=1 TO ..........
```

Fig. 5.1. The use of the counter C=C+1. Testing for an end of data marker – the word "FINISH".

The statements in Fig. 5.1 set store C to ∅ and store I to 1. Since I = 1, the first name is read into store N$(1) and the first number into T$(1). At line 35 we test for the 'dummy' end of data marker, the name "FINISH" which appears in the data lines at the end of the program, as shown in Fig. 5.2.

```
590 DATA ............

600 DATA.............

700 DATA"FINISH",***
```

Fig. 5.2. Marking the end of the data with a complete set of 'dummy' data.

Since the name will not be "FINISH" after the first read statement, we move to line 4∅ where stores C and I are increased by

1. Line 45 returns us to read the next items of data, which are stored in N$(2) and T$(2). When the computer ultimately reads "FINISH" into store N$(I), the IF ... THEN ... at line 35 recognises it and the program branches to line 5ø.

Two important points should be understood at this stage.

- Store C now contains the total number of names and telephone numbers we have read.
- We now have in the memory two arrays of data in numbered stores:

N$(1) contains the first name we read in.
N$(2) contains the second name we read in.
N$(3) contains the third name we read in.
N$(C) contains the last name we read in.
(The computer will have a value for C.)

Similarly the telephone numbers will be stored in T$(1), T$(2), etc.

So, in order to use any name or telephone number, we simply refer to their variable name (such as T$(2)). We will not have to read them in again in order, say, to print them or sort them in alphabetical order.

To display the contents of a particular store, we could type in *immediate* mode:

```
PRINT N$(3)
```

When we press RETURN the computer will display whatever it has stored in N$(3), i.e. the third name in our telephone directory.

Using the counter

We have seen how the statement

```
C = C + 1
```

enabled us to count the number of items read into the memory. If we had read 99 values into the memory, then C would contain 99. This is useful, because we can now specify C as the final number in the FOR ... NEXT loop for the PRINT module as shown in Fig. 5.3

Let us now put the whole program together and follow it through (Fig. 5.4).

Your data should be entered in line 5øø onwards, remembering to end with your 'dummy' data, which could, for simplicity, be "XYZ" or something equally short and unlikely to be a genuine item of data.

```
45...........

50 FOR I= 1 TO C

60 PRINT N$(I),T$(I)
```

Fig. 5.3. Using the counter C in the FOR statement.

```
10 DIM N$(110),T$(110)
20 LET C=0
25 LET I=1
30 READ N$(I),T$(I)
35 IF N$(I)="FINISH"THEN 50
40 C=C+1:I=I+1
45 GOTO 30
```

Read module

```
50 FOR I=1 TO C
60 PRINT N$(I),T$(I)
65 GOSUB 8000
70 NEXT I
```

Print module

```
500 DATA DR. FOSTER,GLOUCESTER 3694
510 DATA.............................
520 DATA.............................
799 DATA FINISH,000
```

Data module

```
1000 END
8000 PRINT"PRESS SPACE TO CONTINUE"
8010 GET G$:IF G$<>" " THEN 8010
8020 RETURN
```

Sub-routine

Fig. 5.4. A program to READ and PRINT up to 110 names and telephone numbers.

Your dummy must, however, be identical to the name tested for in line 35, e.g.

35 IF N$(I) = "XYZ" THEN 5Ø

The basic format of this program could be used for any data processing subject – not just names and telephone numbers. We would simply extend the number of variable names, to 4, for instance:

3Ø READ A$(I),B$(I),C$(I),D$(I)

and simply alter the DIM and PRINT statements. It is vital, however, to check that the variable names in the READ statements are compatible with the type of data in the DATA statements.

Using a lot of variables, we may wish to print each on its own separate line, as follows:

```
6Ø PRINT A$(I)
62 PRINT B$(I)
63 PRINT C$(I)
```

Why bother with a computer?

Taking our simple example of our personal telephone directory, we could keep these on a piece of paper. As new names are added, we extend the list, perhaps onto a new sheet. When people change numbers, we cross the old ones out and write in the new. We usually finish up with a complete mess which we never quite get around to rewriting. With our telephone program stored on tape, however, we simply load in the program, and amend as follows:

- For alterations we either alter or re-type the appropriate DATA line. This will automatically replace the old line when RETURN is pressed.
- To add a new item of data we simply insert it at the next available DATA line with a suitable number.

You can check your amendments by *listing* the program. Of course, to have a permanent record you must now *resave* the program on tape and *verify*. If you are fortunate enough to have a printer you can now obtain a printout of the latest version.

Remember that the numbers in the dimension statement must be at least equal to the number of items of data, preferably larger to allow for future updating. Also, make sure that your 'dummy' data is the very *last* DATA statement.

We have already seen, then, that computer storage is neater and more convenient to amend and keep up-to-date than the equivalent paper documents. However, we still have not considered the main advantage of the computer – its ability to search and sort data at high speed. This is covered in the next two chapters.

Chapter Six
Searching

Searching for a particular piece of data

As already stated, once the data has been read into the memory during the READ module, it is available to us in stores whose addresses we know, such as N$(1),N$(2), etc. This means we can pass through the complete set of data by simply setting up a FOR...NEXT loop.

Now, let's suppose that we wish to search through our 100 names and pick out the telephone number for a particular person.

We set up the loop to pass through all of the data to be sure of finding the one we need: if the computer finds the name we want it leaves the loop and prints out this name and telephone number.

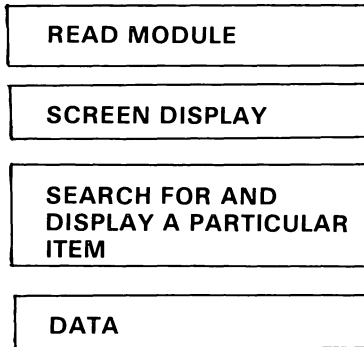
Suppose we are looking for "CHEMIST". Our search module will therefore be as shown in Fig. 6.1.

```
100 FOR I= 1 TO C
110 IF N$(I)="CHEMIST" THEN 130
120 NEXT I                              } Search
125 GOTO 140                            } module
130 PRINT N$(I),T$(I)
```

Fig. 6.1. Searching for a name.

(*Reminder:* C is the number of sets of data, which the computer counts during the reading module – one set being made up of one name and its associated number. One complete set is known as a *record*.)

This small search routine can be slotted into the program, so that the program now consists of the following modules:



In the next chapter we will see how these modules can be reached from a *menu*, so that we only run those parts of the program we particularly want.

We can improve the flexibility of the search module by adding the facility to type in the name to be searched for, in response to the INPUT prompt, as shown in Fig. 6.2.

```
90 INPUT "TYPE IN THE NAME";Z$
105 FOR I=1 TO C
110 IF N$(I)=Z$ THEN 130
120 NEXT I
125 GOTO 140
130 PRINT N$(I),T$(I)
```

Fig. 6.2. Searching for a name in the DATA which corresponds to a name which has been INPUT.

One fault in this routine may be apparent; if the name is not spelt exactly right, the computer will complete the loop without finding anything. This will also happen if the name does not exist. In this case we should send the user to check the spelling, or alternatively check through the entire screen display of the file. We could do this more effectively with the *menu* (to be discussed later), but an improvement would be to put a message at line 135 to give instructions to the user, as follows:

```
134 PRINT "NAME NOT FOUND"
135 PRINT "CHECK YOUR SPELLING"
14Ø GOTO 9Ø
```

Line 14Ø returns the execution of the program to line 9Ø, which instructs the user to type the name, this time correctly.

Searching for values which fall between a particular range

We might wish to search our data and print out all items which fall within a given range, such as cars with an engine size between 1500 cc and 2000 cc inclusive. If E(I) is our engine size in ccs the required cars would be identified by line 11∅ in Fig. 6.3.

```
100 FOR I=1 TO C
110 IF E(I)<1500 OR E(I)>2000 THEN 130
120 PRINT "MAKE";M$(I)
125 PRINT "MODEL";Q$(I)
130 NEXT I
```

Fig. 6.3. Bypassing the PRINT statements when search criteria are not fulfilled (the required cars are printed at lines 12∅ and 125).

Written in this way, those cars *outside* of the range are ignored when the program steps ahead to line 13∅. Those falling *within* the range are printed at line 12∅.

A routine for houses with prices between £20000 and £40000 inclusive is shown in Fig. 6.4.

```
100 FOR I=1 TO C
110 IF P(I)<20000 OR P(I)>40000 THEN 130
120 PRINT "AREA";A$(I)
125 PRINT "DETAILS";D$(I)
130 NEXT I
```

Fig. 6.4. Rejecting those houses outside of the required price range (those within the range are printed at lines 12∅ and 125).

In this routine, houses which are less than or greater than the required range are not printed, since the program jumps ahead from line 11∅ to line 13∅, and on to the next set of data.

We have now seen how to search our data for particular names, or to pick out all sets of data where a number fell within a certain range. The basic method for reading, printing and sorting data is the same,

whatever the type of data to be handled. So, we could modify the program to build data files on any subject we like, such as:

Ornithology:	Name, marking, size, habitat, migration, nesting.
Car records:	Performance details, prices, searching for fuel consumption, price range, maximum speed.
Estate agents:	Locality, number of bedrooms, detached/semi, central heating, price range, etc.
Wine-making:	Name, type, year, ingredients, quantities.
Recipes:	Dish, ingredients, quantities, temperatures, etc.

The most common mistakes can be avoided if the reader bears the following points in mind:

- The DIMension statement must be adequate for the number of items of data.
- For every READ statement there must be corresponding DATA statements supplying exactly the right number of pieces of data; i.e. for every string variable in READ, there must be a corresponding string constant in DATA and similarly for numeric variables and constants.

Adapting the format for other subjects

Using this basic layout we can store data for any subject we choose, such as the stock in a small business or 200 favourite recipes.

We will consider the records of 300 items in the catalogue of a small business. Each item or product is known as a *record* in computing jargon, each detail within the record is a *field* and the whole collection of 300 items is known as a *file* (see Fig. 6.5.).

We now have to decide on the names to give the various pieces of data or fields. Since in our previous program we numbered (or subscripted) the variable names, our record number will already be provided by store I. Therefore we only need to read in part no., description, number in stock and unit price.

BASIC has a way of including messages into the program – it is the REM or reminder statement. We will use it to allocate variable names to our data.

```
11 REM P$(I) = Part number
12 REM D$(I) = Description
13 REM N(I) = No. in stock
14 REM U(I) = Unit cost
```

Stock File				
	Fields			
Record no.	Part no.	Description	No. in stock	Unit price
1	CU1032	Widget	300	0.59 *
2	EF2436	Flange	250	1.50 *
. . . .	. . . .	. . . .	. . . . . .	. . . . * . . . . *
300	RB6124	Socket	100	0.89 *

* One record is one horizontal row.

Fig. 6.5. Extracts from a data file consisting of 300 records.

Note that the REM statement does not play any part in the running or execution of the program, it simply allows us to include notes to add meaning. Also, note that the number in stock and the unit cost are *numbers*, which we might later wish to use in calculations, so they are stored in the *numeric* variables N(I) and U(I) respectively.

The DIM statement must be modified to reserve space for at least 300 items of DATA. Otherwise, the program is basically the same as for the telephone directory. To allow for additions we will actually dimension for 310 records.

The complete stock records program, to READ and PRINT up to 310 stock records is shown in Fig. 6.6.

We could have search modules in this program to print out the details of a particular record, given the part number, description, number in stock or price. This would be particularly useful if we wished to reorder when stocks dropped below a certain level, say 100 items.

We simply pass through the data, comparing the number in stock with our reorder level by adding the routine in Fig. 6.7 which shows how to search for items which need to be reordered.

```
10 DIM P$(310),D$(310),N(310),U(310)
11 REM P$(I)=PART NUMBER
12 REM D$(I)=DESCRIPTION
13 REM N(I) =NO. IN STOCK
14 REM U(I) =UNIT COST
20 LET C=0
25 LET I=1
30 READ P$(I),D$(I),N(I),U(I)
35 IF P$(I)= "XYZ" THEN 50
40 C=C+1:I=I+1
45 GOTO 30
50 FOR I=1 TO C
60 PRINT "RECORD NO ";I
61 PRINT P$(I)
62 PRINT D$(I)
63 PRINT N(I)
64 PRINT U(I)
65 GOSUB 8000
70 NEXT I
500 DATA CU1032,WIDGET,300,0.59
501 DATA EF2436,FLANGE,250,1.50
502 DATA ......................
799 DATA RB1624,SOCKET,100,0.89
800 DATA XYZ,***,0,0
1000 END
8000 PRINT "PRESS SPACE BAR TO CONTINUE"
8010 GET G$:IF G$<>" "THEN 8010
8020 RETURN
```

Fig. 6.6. A program to READ and PRINT up to 310 stock records.

```
90 INPUT "TYPE IN THE RE-ORDER LEVEL";R
100 FOR I=1 TO C
110 IF N(I)>=100 THEN 120
112 PRINT "RE-ORDER"
113 PRINT P$(I),D$(I),N(I),U(I)
120 NEXT I
```

Fig. 6.7. Searching for items which need to be reordered.

At line 11∅,>=means 'greater than or equal to', so if the stock N(I) is less than 100, the program continues to line 112.

Note that there could be several items below the reorder level of 100 items, so we must remain inside the loop after we have found one

– unlike the telephone directory, where we left the loop on finding the name we wanted. We could modify this by inserting a counter at line 115, so that if we had no items below the reorder level, a message would be printed, but this is left as an exercise for you to try out.

Searching a file for records fulfilling more than one criterion

In a file of cars, for instance, we may not just want to find and print out, say, all Ford cars. We may wish to refine the search to extract only Ford Escorts, for example. This implies searching for the two separate fields of make and model which would be stored in individual array variables, such as MA$(I) and MO$(I) respectively.

After reading the data, the program would use a FOR ... NEXT loop to pass through all of the records, comparing the makes and models stored with those requested by the user in response to INPUT prompts. Since the car must satisfy *both* make and model, any car which fails to satisfy *either* is rejected (by-passing the print statements during the pass through the loop). A suitable routine is given in Fig. 6.8.

```
310 FOR I= 1 TO 20
320 IF MA$(I)<>A$ OR MO$(I)<>B$ THEN 400
330 PRINT "MAKE",MA$(I)
340 PRINT "MODEL",MO$(I)
350 PRINT "PRICE",P(I)
400 NEXT I
```

Fig. 6.8. By-passing the PRINT statements if either (or both) of two criteria are not satisfied.

Note the use of OR in line 32∅. If AND were used, a car would only be excluded if it were neither say, Ford, nor Escort. *All* models of Ford would pass through line 32∅ and be printed.

A sample program is shown in Fig. 6.9, and the reader may wish to modify it for different search criteria, such as make and colour of car, or an entirely different subject altogether. Twenty complete records, or sets of data, must be added.

Line 1∅∅∅ contains 1 record, consisting of three fields, make, model and price. The program must be extended to include a further 19 records to agree with the 20 in the FOR ... NEXT loops in the program.

Note that the price of car was assigned to the numeric variable

```
90 DIM MA$(30),MO$(30),P(30)
100 REM:MA$=MAKE:MO$=MODEL:P=PRICE
110 REM:READING THE DATA
120 FOR I=1 TO 20
130 READ MA$(I),MO$(I),P(I)
140 NEXT I
150 REM:INPUTTING SEARCH FIELDS
160 PRINT"ENTER THE NAME"
170 INPUT A$
180 PRINT:PRINT
190 PRINT"ENTER THE MODEL"
200 INPUT B$
300 REM:SEARCHING THROUGH THE DATA"
310 FOR I= 1 TO 20
320 IF MA$(I)<>A$ OR MO$(I)<>B$ THEN 400
330 PRINT "MAKE",MA$(I)
340 PRINT "MODEL",MO$(I)
350 PRINT "PRICE",P(I)
400 NEXT I
1000 DATA AUSTIN,MINI,2500
1010 ....................
```

Fig. 6.9. A program to search the records of twenty cars.

P(I) rather than a string variable such as P$(I). This would allow calculations to be performed on the price, such as PV(I) = P(I)*1.15 to include VAT, etc. This would not be possible using P$(I), which stores numbers only as a mathematically meaningless string of characters, like the figures in a telephone number or car registration number.

The search routines in the previous examples work well enough, but depend on the user knowing the exact spelling of the required name. In practice, we may only remember part of a name, such as the first letter or a combination of characters contained somewhere within the required string.

The next section gives more refined searching methods which allow us to search for right-hand, left-hand and middle sections of character strings appearing within our strings of data.

Manipulating parts of strings using LEN, LEFT$, RIGHT$ and MID$

It is often useful to be able to examine not only whole strings of characters, but also parts of strings. For example, if a witness noted that a car registration number contained the letters AL, then all cars with these letters somewhere in their registration could be identified.

Similarly, it might be necessary to pick out all names beginning with a particular letter or perhaps those containing either MR, MRS, etc. If a code, such as M and F for male and female, were appended to a name, then the computer could identify these if necessary.

Four of the main functions used in this work are as follows:

(1) LEN

This is the number of characters in the string. For example:

```
1∅ A$ = "COMMODORE 64"
2∅ PRINT LEN(A$)
```

RUN this and 12 should be printed. Note that the space in "COMMODORE 64" counts as a character. (To the computer it is just another string of ∅'s and 1's entered when the space bar is pressed.)

(2) LEFT$

This enables us to manipulate one or more characters on the left of a string. For example:

```
1∅ A$ = "COMPUTER"
2∅ PRINT LEFT$(A$,4)
```

When the above is run, line 2∅ prints the 4 left-most characters of string A$, in this case "COMP".

Try the program in Fig. 6.10, which experiments with LEFT$.

```
10 A$="ANYTHING"
20 FOR N=1 TO LEN(A$)
30 PRINT LEFT$(A$,N)
40 FOR T=1 TO 1000:NEXT T
50 NEXT N
```

Fig. 6.10. Experimenting with LEFT$.

Repeat this with various strings instead of "ANYTHING".

(3) RIGHT$

This is a similar function to LEFT$, but operating from the right-hand side of the string. Repeat the previous program in Fig. 6.10, but replace LEFT$ in line 3∅ with RIGHT$.

Note that when N = LEN(A$), (i.e. the number of characters in string A$), we are asking the computer to print out the whole string. Line 4∅ is just a time delay which may be adjusted as required by altering the number 1∅∅∅.

If we needed to search a file of data to find all names beginning with certain letters, we could program it by inserting some lines into the program as shown in Fig. 6.11.

```
5000 INPUT S$
5020 FOR N=1 TO ......
5030 IF LEFT$(A$(N),LEN(S$))<>S$THEN6000
5050 PRINT A$(N)
.................
```

Fig. 6.11. Part of a program to search for names which begin with a certain letter or group of letters.

In this routine, we enter the letter(s) we wish to search for at line 5∅∅∅. Since this could be the first letter of a name, or the first few letters, or even the whole name, we cannot just compare the input name with the names stored. For instance, if we are only inputting the first letter, then we must only look at the first letters of those stored. If we input 3 letters, we must only examine the first 3 of those stored. So we examine LEFT$(A$(N),LEN(S$)) where LEN(S$) is the number of characters we are searching for and A$(N) is the array of names in our DATA.

Now we can write a program to read thirty names, say, and search for and print out all those beginning with certain letters or combinations of letters (see Fig. 6.12). This is purely a demonstration program searching thirty names – in practice, hundreds or thousands of names might be interrogated.

(continue adding data until 30 names are included).

In the following program we can input either the first letter or several letters on the left of the required name, or the whole name itself. If we only enter the letter K in response to the input prompt (INPUT?) then the computer will pick out all names beginning with the letter K. In this case KEITH,KIM, etc. would be printed.

```
100 DIM A$(30)
110 FOR N=1 TO 30
120 READ A$(N)
130 NEXT N
135 PRINT "[CLR]"
140 PRINT "ENTER THE CHARACTER(S)"
150 INPUT S$
160 FOR N= 1 TO 30
170 IFLEFT$(A$(N),LEN(S$))<>S$ THEN 190
180 PRINT A$(N)
190 NEXT N
200 GOTO 135
210 DATA PETER,JOHN,MIKE,KEITH,TOM
220 DATA KIM,JILL,DAVID,RICHARD...
230 DATA.........
```

Fig. 6.12. A program to identify names beginning with certain letters.

(4) MID$

This function enables us to identify groups of characters contained anywhere within a string. For example:

```
20 A$ = "COMMODORE"
30 PRINT MID$(A$,3,5)
```

In the above example, the PRINT statement displays 5 characters from "COMMODORE", starting with the third. The third character is M and so MMODO should be displayed when this program is run.

Repeat this with different strings for A$ and vary the number of characters to be printed and also the position of the first character to be printed.

Checking whether a string contains another string

We can use MID$ to check whether a string contains another string. One method is to work through the main string from left to right, comparing the required character(s) with those in the main string. If, for example, the main string were "COMPUTER" and we were checking for all words which contained, say, "MP", then we would compare MP with first CO, then OM, then MP, when a match would be found. In this way, we could pick out all strings from a large list, which contained a certain group somewhere within them.

(We can search for a 'mid-string' of *any* length, not just the two-character example shown on the previous page.)

Example:
This is not a practical search program, but just a demonstration of the principle. In practice, you would search hundreds of strings for those containing a required combination of characters.

```
10 A$="ANYTHING YOU LIKE"
20 INPUT S$
30 FOR N=1 TO LEN(A$)
40 IF MID$(A$,N,LEN(S$))=S$ THEN 70
50 NEXT N
60 PRINT A$"DOES NOT CONTAIN"S$:GOTO 20
70 PRINT A$"CONTAINS"S$:GOTO 20
```

Here, S$ is the small string we wish to search for, and A$ is the complete string which is tested throughout its length to see if it contains S$.

We should now be able to include LEFT$, RIGHT$, MID$, in search modules in larger programs, to improve the flexibility of search routines.

You should note that the program logic is different when we leave the loop as soon as we have found a *unique* name, compared with the case when we remain within the loop to print out, say, all words beginning with a certain letter.

A learning program

The program in Fig. 6.13 allows the user to be tested on topics such as the capitals of countries, language translation, etc., where single word questions require single word answers. The question (e.g. country) and the answer (its capital) are entered in adjacent pairs in DATA statements. The user types in his/her answer in response to an INPUT prompt and the program compares the user's answer with that in the DATA. Suitable instructions are embodied in the program to respond to either a correct or incorrect reply from the user.

As we do not need to search or sort through the DATA (since we are only interested in individual pairs taken one at a time) we do not need to store all of them in an array of subscripted variables, so a DIM statement is not necessary.

```
10 REM C$=COUNTRY:CA$=CAPITAL
20 REM A$=USER'S ATTEMPTED ANSWER
100 READ C$,CA$
110 IF C$="XXX" THEN 500
115 PRINT"♥":PRINT:PRINT:PRINT
120 PRINT"WHAT IS THE CAPITAL OF ";C$
125 PRINT:PRINT
130 PRINT"TYPE IN YOUR ANSWER":PRINT:PRINT
135 INPUT A$:PRINT:PRINT
150 IF A$=CA$ THEN 200
160 PRINT"SORRY-TRY AGAIN"
170 FOR T= 1 TO 2000:NEXT T
180 GOTO 115
200 PRINT"CORRECT"
210 FOR T=1 TO 2000:NEXT T
220 GOTO 100
500 PRINT:PRINT:PRINT"♥"
505 PRINT"ENTER F TO FINISH "
506 PRINT"OR R TO REPEAT THE TEST"
510 GET G$:IF G$<>"F" AND G$<>"R"THEN510
520 IF G$="R" THEN 540
530 END
540 RESTORE :GOTO 100
550 DATAAMERICA,WASHINGTON,RUSSIA,MOSCOW
560 DATA FRANCE,PARIS,ITALY,ROME
570 DATAENGLAND,LONDON,SCOTLAND,EDINBURGH
580 DATA HUNGARY,BUDAPEST,XXX,ZZZ
```

Fig. 6.13. Testing countries and capitals.

You can extend this data to include as many questions and answers as you like, provided that the 'dummy' variables XXX,ZZZ are included to mark the end of the data.

The same program might easily be modified to test, say, translation between two languages, writing the equivalent words side by side in the DATA statements. For example:

550 DATA A HOUSE,UNE MAISON,A FISH,UN POISSON

This program could be enhanced by displaying the correct answer if the user is unable to proceed. One method might be to include a counter and increase it by one every time an incorrect answer is input, displaying the correct answer after a certain number of failures. Fig. 6.14 shows how this may be done.

```
160 C=C+1
170 IF C<5 THEN 180
175 PRINT:PRINT"THE CAPITAL IS";CA$
176 FOR T=1 TO 2000:NEXT T:PRINT"♥"
177 GOTO 100
180 PRINT"SORRY-TRY AGAIN":GOTO115
```

Fig. 6.14. Displaying the correct answer after a certain number of attempts.

Time delay loop

In order to delay the messages "CORRECT" and "SORRY – TRY AGAIN" we have inserted counting loops at lines 170 and 200 in Fig. 6.13. which simply make the computer count up to a certain number before moving on to the next statement. The delay can be adjusted to suit the user by replacing the number 2ØØØ in the FOR ...NEXT loops, with larger or smaller values.

By entering the data for question and solution in adjacent pairs in DATA statements in this way, this program may be adapted for testing in any subject where a single correct solution exists, such as spelling, geography, languages or history.

Chapter Seven
Sorting Data

It is often useful to have our data arranged in a particular order, usually alphabetical or numerical. Obviously it is more convenient to scan a list for a particular item if it is ordered in some way.

The alphabetical sorting routine compares the first two names (using >=) and, if they are already in order, moves on to compare the second and third. If the first and second were not in order, they are swapped. The swap routine is shown in Fig. 7.1.

```
220 LET K$=A$(I)
230 LET A$(I)=A$(I+1)
240 LET A$(I+1)=K$
```

Fig. 7.1. The swap routine. K$ preserves a copy of store A$(I) before it is overwritten.

K$ is a spare store used to keep a copy of what was initially in store I. Line 23∅ says 'let store A$(I) now contain a copy of the contents of store A$(I+1)'. Line 24∅ has the following effect 'let store A$(I+1) now contain a copy of store K$, which contains the original contents of store A$(I)'. (*Note*: It was necessary to introduce the 'keeper' store K$, since the original contents of store A$(I) are overwritten at 23∅ and would otherwise be lost forever.)

Initially I will have the value 1, so we start by comparing the values A$(1) and A$(2). We will now enclose the swap routine in a FOR ... NEXT loop and include the IF ... THEN statement which tests for alphabetical precedence. Figure 7.2 shows how to sort six names into alphabetical order.

In the following routine, if the two names were already in alphabetical order, the program leaps to line 4∅∅ and no swap takes place. If the two names are not in order at line 22∅ the program continues to line 26∅ where the swapping begins.

```
210 FOR I=1 TO 5
220 IF A$(I)<=A$(I+1)THEN400
260 LET K$=A$(I)
270 LET A$(I)=A$(I+1)
280 LET A$(I+1)=K$
400 NEXT I
```

Fig. 7.2. Part of a program to sort six names into alphabetical order.

The program continues through the data until consecutive pieces of data have been sorted in pairs. At this stage, however, the whole list is not yet in alphabetical order and we will illustrate this with six names as follows:

SMITH	BROWN	BROWN	BROWN	BROWN
BROWN	SMITH	JONES	JONES	JONES
JONES	JONES	SMITH	SMITH	SMITH
WALKER	WALKER	WALKER	WALKER	ABLE
ABLE	ABLE	ABLE	ABLE	WALKER
BONNER	BONNER	BONNER	BONNER	BONNER

[dashed box] compared

[solid box] compared and swapped

Number of swaps = 4

It can be seen from the example that although we have made some progress, the list is still not in alphabetical order. So, the process is repeated by passing through the FOR ... NEXT loop again.

BROWN	BROWN	BROWN	BROWN	BROWN
JONES	JONES	JONES	JONES	JONES
SMITH	SMITH	SMITH	ABLE	ABLE
ABLE	ABLE	ABLE	SMITH	BONNER
BONNER	BONNER	BONNER	BONNER	SMITH
WALKER	WALKER	WALKER	WALKER	WALKER

Number of swaps = 2

During subsequent passes, the words are 'bubbled' up into the correct position, so that this is known as a *bubble sort*.

Although the names are still not in alphabetical order, we are getting nearer. Eventually a pass through the data will result in no swaps taking place, and the list will then be in alphabetical order. We

can easily test for this situation by setting a counter in store S to zero, and incrementing by 1 every time a swap takes place (see Fig. 7.3). If, at the end of the pass, S = zero, no swaps have taken place, so the list must be in alphabetical order.

In our swap routine, S must be set to zero *before* we enter the loop, and increased by 1 after each swap has taken place. S therefore registers the number of swaps on a particular *pass* through all of the data.

```
200 LET S=0
210 FOR I=1 TO 5
220 IF N$(I)<=N$(I+1) THEN 400
260 LET K$=N$(I)
270 LET N$(I)=N$(I+1)
280 LET N$(I+1)=K$
290 S=S+1
400 NEXT I
440 IF S<>0 THEN 200
```

Fig. 7.3. Testing for alphabetical order, i.e. no swaps having taken place.

At line 44ϕ we test to see if any swaps have been made during the pass through the data, i.e. if S is not equal to ϕ. If swaps have been made we must return to line 4ϕϕ and re-pass until no swaps are made, i.e. the list is in alphabetical order.

We will now consider the remainder of the sorting process for the six names. Three more passes will be required before the data is known to be in alphabetical order.

Third pass				
BROWN	BROWN	BROWN	BROWN	BROWN
JONES	JONES	ABLE	ABLE	ABLE
ABLE	ABLE	JONES	BONNER	BONNER
BONNER	BONNER	BONNER	JONES	JONES
SMITH	SMITH	SMITH	SMITH	SMITH
WALKER	WALKER	WALKER	WALKER	WALKER

Number of swaps = 2

Fourth pass				
BROWN	ABLE	ABLE	ABLE	ABLE
ABLE	BROWN	BONNER	BONNER	BONNER
BONNER	BONNER	BROWN	BROWN	BROWN
JONES	JONES	JONES	JONES	JONES
SMITH	SMITH	SMITH	SMITH	SMITH
WALKER	WALKER	WALKER	WALKER	WALKER

Number of swaps = 2

After the fourth pass, the names are in alphabetical order: the computer will pass through them again and on counting no swaps (i.e. S = ∅) will leave the sort routine to print the final sorted list.

We have now rearranged the data so that, although the content of the data is unchanged, some or all of it may have been swapped into new stores with different subscripts. In our example, N$(5) originally stored ABLE but this was 'bubbled' up in the sorting process until it finally resided in store N$(1).

So now that the data is stored in alphabetical order, it is simply a matter of printing out the sorted list in a FOR...NEXT loop (Fig. 7.4).

```
460 FOR I=1 TO 6
480 PRINT N$(I)
490 NEXT I
```

Fig. 7.4. Printing the sorted list of names.

We can now fit this alphabetical sort routine into the telephone directory program discussed earlier. We will encounter one problem, however, unless we also swap the telephone numbers along with their names – i.e. swap T$(I) as well as N$(I). We can achieve this by swapping the numbers at the same time as the names, in multiple statement lines. This is shown in Fig. 7.5.

```
260 LET K$=N$(I):LET Q$=T$(I)
270 LET N$(I)=N$(I+1):LET T$(I)=T$(I+1)
280 LET N$(I+1)=K$:LET T$(I+1)=Q$
```

Fig. 7.5. Swapping both name and number.

Sorting numbers

The sort routine for numbers is identical to that for words except that we will, of course, use numeric variable names, instead of string names. Take, for example,

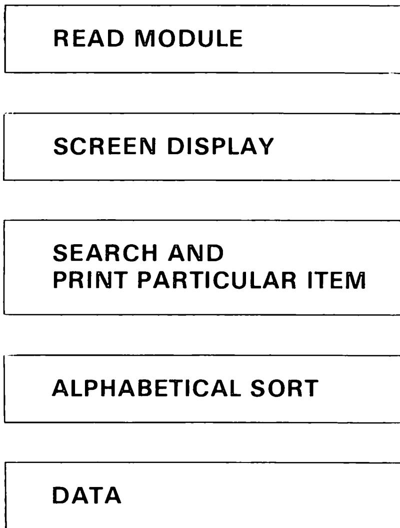
IF A(I) >= A(I+1)THEN ...

This could mean, say: 'If the number in store A(2) is less than or equal to the number stored in A(3) then ...'

Remember – sorting takes time! When running the sort routine with, say 300 names, there may be a delay while the machine does the sorting. Do not worry, it has not 'crashed'.

Telephone directory - including alphabetical sort and search routines

Now let us rewrite the telephone directory program to include the alphabetical sort and the search routines. First, set out the modules of the program:



The telephone directory program is shown in full in Fig. 7.6.

```
10 DIM N$(110),T$(110)
20 LET C=0:LET I=1

30 READ N$(I),T$(I)
35 IF N$(I)="FINISH" THEN 50
40 C=C+1:I=I+1
45 GOTO 30

50 FOR I= 1 TO C
55 PRINT"□"
60 PRINT N$(I),T$(I)
65 GOSUB 8000
70 NEXT I
75 PRINT "□"
```

Lines 30–45: Read module

Lines 50–75: Printing entire file

```
90 INPUT "TYPE IN THE NAME";Z$
105 FOR I= 1 TO C
110 IF N$(I)=Z$ THEN 130
120 NEXT I
130 PRINT N$(I),T$(I)
131 GOSUB 8000
```

Searching for, and printing a particular name and number

```
200 LET S=0
210 FOR I= 1 TO C-1
220 IF N$(I)<=N$(I+1)THEN400
260 LET K$=N$(I):LET Q$=T$(I)
270 LET N$(I)=N$(I+1):LET T$(I)=
    T$(I+1)
280 LET N$(I+1)=K$:LET T$(I+1)=Q$
290 S=S+1
400 NEXT I
440 IF S<>0 THEN 200
```

Alphabetic sort module

```
450 PRINT"[CLR]"
460 FOR I=1 TO C
470 PRINT N$(I),T$(I)
475 NEXT I
```

Printing the sorted directory

```
500 DATA DR.FOSTER,GLOUCESTER 7694
510 DATA..........................
520 DATA..........................
799 DATA FINISH,000

1000 END

8000 PRINT"PRESS SPACE BAR TO CONTINUE"
8010 GET G$:IF G$<>" "THEN 8010
8020 RETURN
```

Fig. 7.6. Telephone Directory: a program to 'handle' up to 110 names and numbers. Later chapters show how this program may be improved.

Note: In the sort process we only need to go to C−1 in the FOR...NEXT loop, since in our sorting we compare N$(I) with N$(I+1).

So, we compare N$(C−1) with N$(C−1+1), i.e. N$(C), and swap if necessary.

Chapter Eight

Menus, Modules and Subroutines

The telephone directory program in the previous chapter will definitely work, but it can be improved. At present the program has the following limitations:

(1) As it has evolved, the line numbers have increased erratically and we could be short of numbers if we develop the program further.
(2) The program consists of standard routines which are common to most programs, so we could have allocated *blocks* of line numbers from the start.
(3) The way the program is written we will have to execute each module whether we want to or not. Yet, at a particular time, we may not want an alphabetic sort or a display of the entire file.

We can overcome these difficulties, and tidy up the program generally in two ways:

(1) Divide the various phases of the program into independent *modules*, allocating blocks of line numbers.
(2) Create a *menu* or choice of options, enabling us to *access independently* only those modules which we wish to use.

Some parts of the program are not optional – we must, for instance, always READ the DATA at the start of the program in order to allocate the store locations such as A$(I).

The menu

Here we give the user a choice of options such as those shown in Fig. 8.1 which also incorporates an 'error trap' at line 1Ø9Ø.

Line 1Ø9Ø checks to see that a number between 1–3 inclusive is input at line 1Ø8Ø, otherwise the program returns to ask for the option to be typed in again. (We could use INPUT A, but if a letter is

```
1000 PRINT:PRINT"♥HERE ARE THE OPTIONS"
1010 PRINT:PRINT"1. DISPLAY THE FILE"
1020 PRINT:PRINT"2.SEARCH FOR A NAME"
1030 PRINT:PRINT"3.SORT ALPHABETICALLY"
1040 PRINT:PRINT"TYPE IN YOUR OPTION"
1050 PRINT:PRINT"IT MUST BE IN THE RANGE"
1060 PRINT:PRINT"1-3"
1080 INPUT A$:LET A=VAL(A$)
1090 IF A<>1 AND A<>2 AND A<>3THEN 1080
1100 ON A GOSUB 2000,4000,6000
1110 GOTO 1000
```

Fig. 8.1. A choice of options with an error trap at line 1Ø9Ø.

accidentally pressed, an error would occur. The use of A = VAL(A$) prevents this.)

At line 11ØØ we have introduced a very important idea – ON ... GOSUB ... We have previously mentioned the idea of a *subroutine* – a frequently used module which can be accessed and returned from at will. Now we can branch to the required subroutine by typing a single number. ON A GOSUB 2ØØØ,4ØØØ,6ØØØ means 'if 1 is input for A, go to the subroutine starting at line 2ØØØ, if 2 is input for A, go to the subroutine starting at 4ØØØ, etc.' (Similarly ON ... GOTO ... has the same effect without the RETURN capability of ON ... GOSUB ...)

So, each of our *optional* modules becomes a *subroutine*, accessed by inputting an option number A, into ON A GOSUB 2ØØØ,4ØØØ,6ØØØ. The optional subroutines will be:

(a)

```
2ØØØ   REM: DISPLAY ENTIRE FILE
3ØØØ   RETURN
```

(b)

```
4ØØØ   REM: SEARCH FOR A NAME
5ØØØ   RETURN
```

(c)

```
6ØØØ   REM: ALPHASORT AND PRINT
7ØØØ   RETURN
```

Just as we can branch to any one of the optional subroutines, we also return from them, inserting RETURN at the end of the subroutine.

This will return us to the line immediately after the ON ... GOSUB ... statement.

However, we will normally wish to return to the *beginning* of the menu, before making another choice. This can be achieved by inserting a GOTO at line 111∅ to transfer all returns from the subroutine to the beginning of the menu.

Immediately before printing the menu, it would be desirable to program the clearing of the screen. It is also worth taking trouble with the layout of the menu to make it 'user-friendly'. Taking an overall view of the program, we have an initial reading process, in which data is assigned to store locations, followed by a menu, from which we travel to, and return from, various parts of the program as often as we like. The menu gives complete control of the routines which are accessed – so the program is described as *menu-driven.*

Telephone directory program – using a menu to access subroutines

Our telephone directory can now be rewritten in modular form, complete with menu. This improved version is shown in Fig. 8.2.

```
100 DIM N$(110),T$(110)
110 LET C=0:LET I=1
119 REM: READ MODULE
120 READ N$(I),T$(I)
130 IF N$(I)="FINISH"THEN 1000
140 C=C+1:I=I+1
150 GOTO 120

999 REM: THE MENU
1000 PRINT:PRINT"JHERE ARE THE OPTIONS"
1010 PRINT:PRINT"1. DISPLAY THE FILE"
1020 PRINT:PRINT"2.SEARCH FOR A NAME"
1030 PRINT:PRINT"3.SORT ALPHABETICALLY"
1040 PRINT:PRINT"TYPE IN YOUR OPTION"
1050 PRINT:PRINT"IT MUST BE IN THE RANGE"
1060 PRINT:PRINT"1-3"
1080 INPUT A$:LET A=VAL(A$)
1090 IF A<>1 AND A<>2 AND A<>3THEN 1080
1100 ON A GOSUB 2000,4000,6000
1110 GOTO 1000
```

```
1999 REM: PRINTING THE WHOLE FILE
2000 FOR I=1 TO C
2010 PRINT"[CLR]":PRINT:PRINT
2020 PRINT N$(I),T$(I)
2030 PRINT:PRINT:PRINT
2040 GOSUB 8000
2050 NEXT I
2060 RETURN

3999 REM:SEARCHING FOR A NAME
4000 PRINT"[CLR]":PRINT:PRINT
4010 PRINT"TYPE IN THE NAME"
4015 INPUT Z$
4020 FOR I=1 TO C
4030 IF N$(I)=Z$ THEN 4050
4040 NEXT I
4050 PRINT:PRINT:PRINT
4060 PRINT"[CLR]":PRINT N$(I),T$(I)
4070 GOSUB 8000
4080 RETURN

5999 REM:SORTING INTO ALPHABETICAL ORDER
6000 LET S=0
6010 FOR I=1 TO C-1
6020 IF N$(I)<=N$(I+1) THEN 6070
6030 LET K$=N$(I):LET Q$=T$(I)
6040 LET N$(I)=N$(I+1):LET T$(I)=T$(I+1)
6050 LET N$(I+1)=K$: LET T$(I+1)=Q$
6060 S=S+1
6070 NEXT I
6080 IF S<>0 THEN 6000
6090 PRINT"[CLR]"
6100 FOR I=1 TO C
6110 PRINT N$(I),T$(I)
6120 NEXT I
6125 GOSUB 8000
6130 RETURN

8000 PRINT"PRESS SPACE BAR TO CONTINUE"
8010 GET G$: IF G$<>" " THEN 8010
8020 RETURN

9000 DATA DR.FOSTER,GLOUCESTER 7694
9001 DATA...........................
9500 DATA FINISH,000
```

Fig. 8.2. A menu-driven program with options to DISPLAY, SEARCH, or SORT a personal telephone directory.

So, this program allows us to read in a large amount of data and allows access to any one of three subroutines. After each subroutine has been executed, the user is returned to the menu to make a further choice of option.

Supposing we have completed our use of the program and wish to finish. All of the subroutines return the program execution to line 111∅, which uses a GOTO to branch back to the beginning of the memory. This would be a convenient place to give the user an opportunity to end the program.

Asking a question – to be answered 'yes' or 'no'

We can ask a question and get an answer using GET and the options Y for 'Yes' and N for 'No'. A module setting out these options is shown in Fig. 8.3. This module can be introduced into the telephone directory program.

```
1110 PRINT"♥END THIS RUN OF THE PROGRAM"
1120 PRINT"ENTER Y FOR YES,N FOR NO"
1130 GET G$
1135 IF G$<>"Y" AND G$<>"N" THEN1130
1140 IF G$="N" THEN 1000
1200 END
```

Fig. 8.3. A module which gives the option to end the current run of the program.

Line 113∅ ensures that the program will only continue if either Y or N is typed at the keyboard. At line 114∅, if we say 'No' to ending the program we return to the menu at line 1∅∅∅ (see Fig. 8.2). If we type Y the program continues through line 114∅ to end at line 12∅∅.

Note that although this END statement is not literally at the end of the program, it is in practice and it can only be approached by typing Y on return from any one of the subroutines.

Chapter Nine
Developing a General-Purpose Data Handling Program

We can now use the telephone numbers program as a model to make a DATA file on any subject we like, but the following points must be borne in mind:

- The DIMension statement must contain adequate numbers.
- The READ and DATA statements must be *exactly compatible* in the number and type of variables (numeric or string), and constants.

You are not limited to the two variable names N$(I) and T$(I), of course. You must, however, ensure that if you are reading in, say, five pieces of data, the five variable names must be consistent with the DATA statements and the PRINT statements. Wherever N$(I) and T$(I) appeared in the telephone directory program, they must be replaced by five variable names such as A$(I), B$(I), C$(I), D$(I) and E$(I).

This sample program includes REM statements or REMinders, which are simply notes which are ignored by the computer but make the program listing easier for humans to understand.

You can improve the program layout by inserting PRINT:PRINT etc. to give vertical spacing and TAB(X) to give horizontal movement, clearing the screen where necessary. The program can be extended to include as many items as you like, subject only to the memory size of your machine. You may do this by adding DATA statements from 9ØØØ onwards.

Note that the inverted commas are not always necessary in the data except when the data is to include leading or trailing spaces, commas, or colons.

Long strings of data

Items of data need not be confined to single words or strings of characters. For instance, string variable A$, say, could contain a sentence such as:

```
1∅ A$ = "THE BOY SAT ON THE BURNING DECK"
2∅ PRINT A$
```

This would be particularly useful for an address, which if it contained commas, must be enclosed in inverted commas. For example:

```
1∅ A$ = "26,LONDON ROAD,OXFORD"
2∅ PRINT A$
```

For simplicity, our previous work was based on the personal telephone directory. Now we will consider extending the basic program for more lengthy items of data. We could, for instance, have included both name *and* telephone number in one long string:

```
N$ = "DR. FOSTER:GLOUCESTER 7694"
```

However, for the purposes of printing, searching and sorting it is sometimes convenient to have different names for different parts of the data. For instance, if storing names and addresses, we might use variables as follows:

```
N$(I)  = name
A1$(I) = 1st line of address   e.g. street
A2$(I) = 2nd line of address   e.g. area
A3$(I) = 3rd line of address   e.g. town
A4$(I) = 4th line of address   e.g. postcode
```

While it might be possible to store the whole address in one variable name, it would be more difficult to search for a particular postcode or town, etc., than with the separate variables shown above.

We will now extend the basic program to allow, say, five separate lines of data to be processed for each record. This might be suitable for the five lines of an address or for recipes, etc. The choice of five lines is purely arbitrary, and you may extend or reduce this as necessary. Similarly, you may decide to put two data 'fields' into one variable, such as storing town and postcode together and printing side by side – although, as mentioned previously, this limits the searching/sorting processes.

Our data will be stored in variables:

A1$(I),A2$(I),A3$(I),A4$(I),A5$(I)

We now need to consider the telephone program for changes. Wherever we had N$(I), we must now have A1$(I) and similarly T$(I) is replaced by A2$(I).

In addition, all READ and PRINT statements should now contain the new list of five variables. Also, instead of printing name and number side by side, these new records, being much longer, need to be printed on separate lines. So, instead of

2020 PRINT N$(I),T$(I)

we shall need the routine showed in Fig. 9.1, which enables us to print 5 strings on separate lines.

```
2010 PRINT A1$(I)
2020 PRINT A2$(I)
2030 PRINT A3$(I)
2035 PRINT A4$(I)
2038 PRINT A5$(I)
2039 PRINT:PRINT:PRINT
```

Fig. 9.1. Printing five strings on separate lines.

The same effect could be achieved by *multiple program statements*, separated by colons, which will still print A1$(I),A2$(I), etc. on separate lines (Fig. 9.2).

```
2020 PRINTA1$(I):PRINTA2$(I):PRINTA3$(I)
2025 PRINT A4$(I):PRINT A5$(I)
2030 PRINT:PRINT
```

Fig. 9.2. Multiple statement lines.

The alphabetical sort module will also need to be modified to swap the five variables instead of the two in the telephone program. The new swap routine is shown in Fig. 9.3, assuming we sort on A1$(I), the first field in the data.

```
6005 LET S=0
6010 FOR I=1 TO C-1
6020 IF A1$(I)<=A1$(I+1) THEN 6070
6030 K$=A1$(I):L$=A2$(I):M$=A3$(I)
6035 N$=A4$(I):P$=A5$(I)
6040 A1$(I)=A1$(I+1):A2$(I)=A2$(I+1)
6041 A3$(I)=A3$(I+1)
```

```
6045 A4$(I)=A4$(I+1):A5$(I)=A5$(I+1)
6050 A1$(I+1)=K$:A2$(I+1)=L$
6051 A3$(I+1)=M$
6055 A4$(I+1)=N$:A5$(I+1)=P$
6060 S=S+1
6070 NEXT I
6080 IF S<>0 THEN 6005
6090 PRINT"□"
6100 FOR I=1 TO C
6110 PRINTA1$(I):PRINTA2$(I)
6111 PRINT A3$(I)
6115 PRINT A4$(I):PRINT A5$(I)
6120 NEXT I
6130 RETURN
```

Fig. 9.3. Alphabetical sort module for records containing five variables.

Here K$,L$,M$,N$ and P$ are the holding stores which preserve a copy of the original contents of a store after it has been overwritten in the swap process.

In lines 6∅3∅ to 6∅55 we have omitted LET from the assignment statements, for brevity although, for instance, A1$(I)=A1$(I+1) still means 'Let store A1$(I) now contain a copy of what is in store A1$(I+1)', etc.

In the above list, a *record* refers to one set of data, such as the name and address of one person, the details of one car or the recipe for one dish. A *field* is a separate piece of data within the record, having its own store location, such as postcode, telephone number, etc.

Identifiers

In our telephone program, we searched for a particular name and found the name and hence the number. Whenever we store data we are likely to interrogate it to find a particular record, so we need to have a variable in each record which uniquely identifies that particular record in some way. This could be done by including a record number with each record in the data statements, or alternatively the names might be used if they were all different.

The general data file program is shown on the following pages. If you wish to store more or less than five lines of data you must modify the variable list A1$(I),A2$(I),A3$(I), etc. accordingly. Alternatively, if you have more than five lines per record, perhaps two or more lines could be compressed into one variable. For example, instead of

A5$(I) containing LONDON and A6$(I) containing ENGLAND, we could simply put "LONDON,ENGLAND" in store A5$(I).

Searching for a category (containing more than one record)

The example shown is to store 300 recipes, searching for a particular dish by its name. This is not ideal, since we may not remember how to spell 'Creme Patissiere' or 'Gateau St. Honore' and, as mentioned earlier, any deviation from the spelling used in the storage of the data will result in a search which is unsuccessful. In the case of recipes, this problem could be overcome by inserting an extra module and searching for a *category* such as cakes or soups rather than a particular name. Our search would then produce a set of records, such as all cake recipes. As our previous program was written to assume that only one record would fit the search criteria (i.e. the name and telephone number of a unique individual), we left the loop knowing that the search was complete. If searching for a category, rather than an individual name, we must remain within the searching loop in order to identify further records in that category, such as all cake recipes, say. Figure 9.4 shows what our category search module then becomes:

```
7000 REM:SEARCHING FOR A CATEGORY
7005 PRINT"[CLR]":PRINT:PRINT:PRINT
7010 PRINT"TYPE IN THE CATEGORY"
7015 INPUTQ$
7020 FORI=1TOC
7030 IF A2$(I)<>Q$THEN 7080
7040 PRINT:PRINT:PRINT
7050 PRINT"[CLR]":PRINTA1$(I):PRINTA2$(I)
7060 PRINTA3$(I):PRINTA4$(I):PRINTA5$(I)
7070 GOSUB8000
7080 NEXTI
7090 RETURN
```

Fig. 9.4. A program module to search for a category, in this case the second variable in a record consisting of five variables.

In the previous module, we have stored the category, such as cakes in store A2$(I), while still keeping A1$(I) for our unique identifier such as the name. Thus,

A1$(I) = name (or some other unique word or code)

A2$(I) = category (group such as cakes, soups, starters)
A3$(I) = 1st line of ingredients
A4$(I) = 2nd line of ingredients
etc.

Since we have introduced a further module to the program (inserting at lines 7ØØØ onwards), we must also modify our menu to include the extra option to search for a category and modify the corresponding ON ... GOSUB in order to gain access to it. The complete general data program is shown on the following pages. Since we have included rather a lot of changes in a piecemeal fashion it may be worth giving an overall description of the program.

A general-purpose data handling program

Purposes:

(1) To enable, say, 300 records of data, such as recipes, to be stored in the computer or permanently on tape or diskette.
(2) To allow all of the data, if desired, to be displayed individually on the screen, under the control of the user (via the space bar).
(3) To allow the user to produce a list of all records, *sorted* alphabetically on the names of the records.
(4) To enable the user to search for an individual record (such as 'Maids of Honour'), provided a *unique name* or code is known.
(5) To enable all items in a particular *category* to be printed out.

This program is suitable for up to 300 records, each record containing up to five lines of data.

```
100 DIM A1$(300),A2$(300),A3$(300)
105 DIM A4$(300),A5$(300)
110 LETC=0:LETI=1
115 REM:THE READ MODULE
120 READA1$(I),A2$(I),A3$(I)
125 READ A4$(I),A5$(I)
130 IFA1$(I)="FINISH"THEN1000
140 C=C+1:I=I+1
150 GOTO120
```

```
1000 REM:THEMENU
1005 PRINT"□HERE ARE THE OPTIONS"
1010 PRINT:PRINT"1. DISPLAY THE FILE"
1020 PRINT:PRINT"2. SEARCH FOR A NAME"
1030 PRINT:PRINT"3.ALPHABETICAL SORT"
1035 PRINT
1040 PRINT"4.SEARCH FOR A CATEGORY"
1050 PRINT:PRINT"TYPE IN YOUR OPTION"
1055 PRINT
1060 PRINT"IT MUST BE IN THE RANGE 1-4"
1080 INPUTA$:LET A=VAL(A$)
1090 IFA<>1ANDA<>2ANDA<>3ANDA<>4THEN1080
1100 ON A GOSUB2000,4000,6000,7000
1110 PRINT"□END THIS PROGRAM RUN ?"
1115 PRINT
1120 PRINT"ENTER Y FOR YES, N FOR NO"
1130 GET G$:IFG$<>"Y"ANDG$<>"N"THEN1130
1140 IF G$="N"THEN1000
1200 END

2000 REM:PRINTING THE WHOLE FILE
2005 FORI=1TOC
2010 PRINT"□":PRINT:PRINT
2020 PRINTA1$(I):PRINTA2$(I):PRINTA3$(I)
2025 PRINTA4$(I):PRINTA5$(I):PRINT:PRINT
2030 GOSUB8000
2040 NEXTI
2050 RETURN

4000 REM:SEARCHING FOR A NAME
4005 PRINT"□":PRINT:PRINT:PRINT
4010 PRINT"TYPE IN THE NAME"
4015 INPUTZ$
4020 FORI=1TOC
4030 IFA1$(I)=Z$THEN4060
4040 NEXTI
4060 PRINT"□":PRINTA1$(I):PRINTA2$(I)
4065 PRINTA3$(I):PRINTA4$(I):PRINTA5$(I)
4070 PRINT:PRINT
4075 GOSUB8000
4080 RETURN
```

```
6000 REM:SORTING INTO ALPHABETICAL ORDER
6005 LETS=0
6010 FORI=1TOC-1
6020 IFA1$(I)<=A1$(I+1)THEN6070
6030 K$=A1$(I):L$=A2$(I):M$=A3$(I)
6035 N$=A4$(I):P$=A5$(I)
6040 A1$(I)=A1$(I+1):A2$(I)=A2$(I+1)
6042 A3$(I)=A3$(I+1)
6045 A4$(I)=A4$(I+1):A5$(I)=A5$(I+1)
6050 A1$(I+1)=K$:A2$(I+1)=L$:A3$(I+1)=M$
6055 A4$(I+1)=N$:A5$(I+1)=P$
6060 S=S+1
6070 NEXT I
6080 IFS<>0THEN6005
6090 PRINT"□"
6100 FORI=1TOC
6110 PRINTA1$(I):PRINTA2$(I):PRINTA3$(I)
6115 PRINTA4$(I):PRINTA5$(I):PRINT
6116 GOSUB8000:PRINT
6120 NEXTI
6130 RETURN

7000 REM:SEARCHING FOR A CATEGORY
7005 PRINT"□":PRINT:PRINT:PRINT
7010 PRINT"TYPE IN THE CATEGORY"
7015 INPUTQ$
7020 FORI=1TOC
7030 IF A2$(I)<>Q$THEN 7080
7040 PRINT:PRINT:PRINT
7050 PRINT"□":PRINTA1$(I):PRINTA2$(I)
7060 PRINTA3$(I):PRINTA4$(I):PRINTA5$(I)
7070 GOSUB8000
7080 NEXTI
7090 RETURN

8000 REM:HALTING THE SCREEN DISPLAY
8005 REM:UNTIL WE ARE READY TO PROCEED
8010 PRINT"PRESS SPACE BAR TO CONTINUE"
8020 GETG$:IFG$<>" "THEN8020
8030 RETURN
9000 DATA .............
9001 DATA .............
10000 DATA FINISH,***,***,***,***
```

The DATA can now be added from line 9ØØØ onwards – we can insert as much as the computer's memory will hold, but if there are

more than 300 records, the DIM statement must be altered. (In theory we can use line numbers up to 63999.)

Some reminders

- Insert PRINT:PRINT:PRINT etc. to produce spacing between lines (one PRINT gives one blank line).
- " " means the space bar.

Note: Your last statement must include the 'dummy' data marking the end of the data, made up to a complete set of five 'dummy' variables. For example,

```
10000 DATA FINISH,***,***,***,***
```

The DATA

The program can be used for any type of data and we will now use it to enter some recipes. The name of each recipe will be stored in A1$(I) and the category in store A2$(I). The remaining data can be stored in A3$(I), A4$(I) and A5$(I).

The program may be tested using only one or two recipes; to make a worthwhile file it is simply a case of typing in more DATA statements. Considering one recipe initially:

```
9000 DATA HONEYED BANANA SCONES,CAKE
9001 DATA"1 CUP SELF-RAISING FLOUR:½ TSP. SALT"
9002 DATA"1 CUP ALL-PURPOSE FLOUR:2 BANANAS"
9003 DATA"2 TBSPS. BUTTER: 2 TBSPS. HONEY"
```

Notice that the first item of data, the name, will be stored in A1$(I) and the second item, the category, will be placed in A2$(I). Inverted commas are necessary in the remaining data items since we have included colons. These last three lines of data are to be entered into stores A3$(I), A4$(I) and A5$(I).

Line 9001, for instance, provides the third item of data, so that 1 CUP SELF-RAISING FLOUR:½ TSP SALT will be stored in A3$(I). This will cause no technical problems but if the string exceeds 40 characters, it will spill over onto a second line when printed, so adjustment to spacing may be needed to give a screen display which is aesthetically pleasing. For this purpose it may be worth obtaining or drawing up some 40-column coding sheets (as shown in Fig. 9.5) so that our strings, when printed will not split words between two lines.

We must also check to ensure that since our variable list contains five variables, then for each record there should be five distinct fields

```
9ØØØ DATA HONEYED BANANA SCONES,CAKE
9ØØ1 DATA"1CUP SELFRAISING FLOUR:1/2 TSP
SALT"
9ØØ2 DATA"1CUP ALL-PURPOSE FLOUR:2 BANAN
AS"
9ØØ3 DATA"2TABLESPOONS BUTTER:2TABLESPOO
NS HONEY"

9ØØ4 DATA PAVLOVA,GATEAU,"4EGG WHITES :
1CUP OF SUGAR",1CUP HEAVY CREME
9ØØ5 DATA FRESH OR CANNED FRUIT

1ØØØØDATA FINISH,***,***,***,***
```

(lines 9ØØ1–9ØØ3: 1 RECORD; lines 9ØØ4–9ØØ5: 1 RECORD)

Fig. 9.5. Planning the data on a 40-column coding sheet.

Notes:

(1) In each record above there are 5 'fields' or pieces of data. The fields must be separated by either commas or depressing the return key to start a new data statement.

(2) If a field contains either a comma, colon or leading or trailing spaces it must be enclosed in inverted commas.

(3) The first field in each record is the unique name, the second field is the category. Both are used in search options.

Note: Your program will not work unless you remember the dummy data including a word like FINISH, ZZZ, etc. which is tested for during the READ module.

or pieces of data. The first two, such as PAVLOVA and GATEAU in record 2 are separated by a comma. The fourth and fifth fields are separated by starting new line numbers, with RETURN depressed between each.

Notice that in line 9ØØ2 our item of data has extended to two lines and this is acceptable provided we do not press RETURN until we reach the end of that particular string of data.

Try running this program, either with the limited data supplied in this book, or by extending it to include your own. Possible uses include:

- Estate agents' house file
- Car showroom stock file
- Medical records
- Product range in a small business
- Photographer's records
- Name and address file
- Gardener's seed/plant records
- Motoring records and expenses

Limitations

As written, the program provides for data to be stored in AI$(I),A2$(I),A3$(I),A4$(I),A5$(I). This set of variables would need to be extended if each record contained, say, ten lines of data which could not be compressed.

We might also fill the memory of the computer. In order to check this, type PRINT FRE(Ø). On pressing RETURN a number such as 5928 is displayed – the number of unused 'bytes' or characters of memory. In the recipe program, we have compressed several lines of data by including them in one string between inverted commas.

Supposing we do not have enough data to fill all five variable stores. This can be overcome by typing in dummy data, such as "***" or similar, so that if the computer expects five data items, then that is what it actually reads. If, for instance, we are printing out names and addresses, and only need four lines, the output including the dummy data would be as follows:

```
JOHN BROWN
129 LONDON ROAD,
OXFORD,
ENGLAND.
*****
```

Chapter Ten
Files

1. Program files

Saving permanently on cassette tape

Having entered a large quantity of data into the computer's memory via the keyboard, it is necessary to preserve a permanent copy so that the program and its data may be used in the future without retyping. The program will normally be saved on 'floppy' diskette or cassette tape by using the SAVE command and a suitably short name, such as:

SAVE "MY FILE"

If we have invested a substantial amount of time in entering the program and its data, and bearing in mind the relatively low cost of a tape or a diskette, it is usual to make at least two copies of a program for security purposes. Commercial organisations such as banks keep duplicate tapes in different places, as a precaution against theft, fire, or accidental damage. In the case of a cassette tape it is usual to snap off small tabs which make over-writing impossible and on diskettes a small tab is stuck on to *'write protect'* the stored data.

When saving an important file on cassette tape it is worth dedicating one tape to that file alone, so that there is no confusion about finding and loading. For this purpose, the special 12- or 15-minute computer tapes are ideal and generally far more efficient than 90-minute music cassettes containing numerous programs.

Apart from being much faster than tape, the disk drive enables the user to display *rapidly* a catalogue of all files (programs and data). This allows the user to check the exact spelling of a filename, before trying to load from diskette into the memory. The Commodore 64 disk drive provides the user with 170K of backing storage to supplement the computer's memory.

Maintaining the file

Let's assume that we had used the General-Purpose Data Handling Program to record, say, 100 names and addresses. The data, saved as an integral part of the program in lines 9ØØØ onwards, will not be of maximum use to us unless it is always up-to-date. Some names and addresses may need to be removed from the file entirely, details may change, or new records may need to be added. This process is called *updating* the file and the three operations are known as:

- *Amending* Modifying one or more fields of an existing record within the file, such as a change of telephone number only.
- *Deleting* Removing a record no longer needed such as the name and address of a person leaving an area.
- *Appending* Adding a new record to the file.

With our file stored in DATA statements, the procedure would be as follows:

(1) LOAD the program into the memory from cassette or diskette.
(2) *Amend* any records by retyping or 'screen editing' the required DATA statement.
(3) *Delete* a record by typing the appropriate line number(s) and pressing RETURN.
(4) *Append* a new record by typing a suitable line number and DATA statement, checking for consistency with previous records for type and number of variables. (The new record could either have a line number at the end of the data, or be inserted by allocating an intermediate line number.)
(5) LIST the program and check that the required changes have in fact been accepted into the memory.
(6) SAVE the updated file by recording on tape or diskette with a suitable new name.

Out-of-date files

It may not be necessary to keep every old file after it has been updated, perhaps several times. We may need the space the file occupies on a floppy diskette, or we may not want numerous cassettes creating clutter. However, with important files, it may be necessary to have the ability to retrieve information should our latest copies of the file be lost or damaged. The 'grandfather-father-son' system requires the 3 latest versions of a file to be kept, so that as an updated file is made (becoming the 'son') the previous

'grandfather' may be destroyed and replaced as 'grandfather' by the previous 'father'.

So far we have only considered program files, in which our program contained the data in an integral unit. It has been shown that this system works and may be used to handle the records of, say, 250 people, sorting and updating as necessary. However, the program file has the following disadvantages:

(1) Every statement must be preceded by characters such as 9Ø1Ø DATA ..., which uses up memory space and takes typing time.
(2) The updating requires some knowledge of programming in order to alter the DATA statements. In other words, it is not 'user-friendly' to everyone who may wish to use the program.
(3) We may wish to use our General Purpose Data Handling Program to handle files on several different subjects or several separate files on the same subject. It is inefficient to have to save the same program, involving several thousand characters, along with the data on every file. It is rather like having an expensive tin-opener attached to every tin in the larder. One good separate device would give us access to all of the various contents of the tins and we could, if necessary, improvise with another tool.

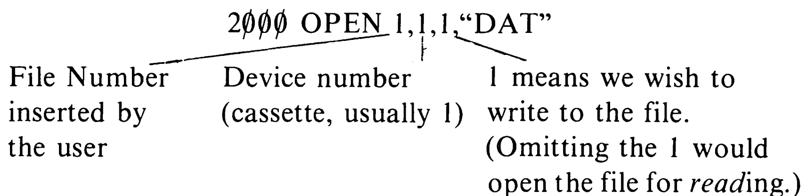
In the same way, it is possible to store data, separate from the program, but accessible to the 'purpose-built' program and also to other programs which we could write to examine the file. We will now consider special data files in which data alone is stored, separate from the program, but accessible to it.

2. Data files

The data file is simply a very long string of all of the 'fields' of data in our records. Our program will take care of presenting the data in rows or columns, etc. by PRINT statements, when it is transferred to the screen or paper.

Each field will have a variable name, as in the General Purpose Data Handling Program, such as A1$(I), etc. and these fields are stored on tape as separate pieces of data. We can either *write to* a data file, or *read from* it and separate programs, or program modules can be written simply to do this.

First it is necessary to OPEN the file, with a statement like:



"DAT" is any name invented by the user for their particular file, preferably short and unique.

After the file has been used it must be closed as follows:

```
3ØØØ CLOSE 1
```

Writing to a file

Having opened the file to write, the PRINT# statement copies the contents of the variable name stores onto the tape (Fig. 10.1).

```
2000 OPEN 1,1,1,"DAT"
2005 FOR I=1 TO C
2010 PRINT#1,A1$(I)
2020 PRINT#1,A2$(I)
" " " " " " " " " " " " " " "
```

Fig. 10.1. Opening a file to write data into it.

Note that in order to record A1$(I) and A2$(I) as distinctly separate pieces of data, they have been included in separate PRINT# statements, so that a carriage RETURN separates each variable name. If we want to include the variables in a single PRINT# statement, then separation must be achieved by programming a carriage return using the CHR$ function:

```
2006 C$=CHR$(13)
2010 PRINT#1,A1$(I),C$,A2$(I),C$,A3$(I)....
```

where CHR$(13) means character string 13, the coded representation of a carriage return. (All of the keys have a CHR$ number – experiment with PRINT CHR$(#) inserting numbers from 33 onwards in the brackets.)

After writing DATA to the file, the computer writes an *end-of-file-marker* at the end of the data.

Reading from a file

The file must first be opened to read, then the variables are transferred from file to computer by the INPUT# statement as shown in Fig. 10.2.

```
2990 OPEN 1,1,"DAT"
3000 FOR I=1 TO C
3010 INPUT#1,A1$(I),A2$(I)
3050 NEXT I
4000 CLOSE 1
```

Fig. 10.2. Opening a file to read data from it, back to the computer.

Unlike the PRINT# statement, several variables may appear on one INPUT# statement separated by commas.

The reading of the file is complete when the end-of-file marker is detected, and individual machines will have different tests for this. So, the file of data will exist independently on its own tape or diskette.

We can either write a small program to read the file or a more complex package of modules to read and write to the file, including options to update the file. Usually we deal with *sequential* files on home computers, where the file is read from end to end, although on diskette systems, in addition to sequential files, we can access data *directly*.

User-friendliness

It is now possible to write a program, using a sequential file, so that instead of using data statements, the user is asked to type in the data in response to INPUT statements. These include a *prompt* to help the user, such as:

```
2ØØØ PRINT "ENTER THE DATE"
2Ø1Ø INPUT D$
```

or

```
2ØØØ INPUT "ENTER THE DATE";D$
```

The complete record is then displayed on the screen by PRINT statements, and if the user is satisfied with the display, it is *written to* a file, before the next record is entered. There is, therefore, no separate operation to save data on tape, since the recording is accomplished by the PRINT# statements built into the program.

With further options, chosen from a menu, to read and update the file, it is possible for an operator to *use* the program without becoming involved in the programming language data statements. At the end of the process, the user will have a program which still contains no data whatsoever, and a tape or diskette file, which contains a long string of data but no program instructions.

Creating a sequential data file

The following program shows how to create a sequential data file on cassette tape. The procedure for a diskette sequential file would be the same, although there are slight differences in the opening and closing statements. These are discussed in more detail in the section on disk drives in the next chapter.

The sample program shown uses a set of cars for sale, with fields for make, model, year of manufacture and price. These are written to the tape, quite separate from the program, and for an important file it would be worth devoting a new tape exclusively to this file. The program would be loaded into the memory from tape and then the program cassette would be removed and replaced by a new cassette on which the sequential file would be written.

The user enters the data in response to INPUT prompts, such as:

```
ENTER THE MAKE
?
```

When a complete record for one car has been entered during four INPUT operations, the user is given the opportunity to retype the record or save it on tape. After an affirmative reply, the record is saved using PRINT# statements. Further records may be entered until the user chooses to cease, in reply to:

"ENTER ANOTHER ITEM - Y OR N?"

Using a simple menu, the program allows the user to open the file to read, and search for all cars of a particular make. This make is entered by the user after an INPUT prompt.

Reading of the program continues until an end of file marker is detected. During this process, any cars in the file which match the make entered by the user are printed out.

The user may notice the cassette drive stopping and starting without the keys being depressed. Under the control of the computer, data is temporarily stored in a *buffer*, before it is displayed on the screen. When the buffer is 'full', the cassette will stop and the data will be displayed. The cassette drive will then restart and refill the buffer, the process continuing until the whole file has been read.

A sequential file for car records

Figure 10.3 sets out the program for a sequential file of car records. Note that this program contains no data, either before or after

```
1000 REM:MENU OF OPTIONS
1010 PRINT"ENTER 1 TO CREATE A FILE"
1020 PRINT:PRINT
1030 PRINT"ENTER 2 TO READ FROM A FILE"
1040 PRINT:PRINT
1050 PRINT"ENTER 3 TO FINISH"
1060 PRINT:PRINT
1070 PRINT"TYPE IN YOUR OPTION:1,2 OR 3"
1080 GET A$:LET A=VAL(A$)
1090 IF A<>1 AND A<>2 AND A<>3THEN1080
1100 ON A GOSUB 2000,3000,4000
1110 GOTO 1010
2000 REM:WRITING THE FILE
2010 REM:WITH AN END OF FILE MARKER
2020 OPEN 1,1,1,"CARS"
2030 INPUT"ENTER MAKE";N$
2040 INPUT"ENTER MODEL";M$
2050 INPUT"ENTER YEAR";Y$
2060 INPUT"ENTER PRICE";P
2070 PRINT "ENTER Y TO SAVE-N TO RETYPE"
2080 GET G$
2090 IF G$<>"Y"ANDG$<>"N" THEN2080
2100 IF G$="N" THEN 2030
2110 PRINT#1,N$
2120 PRINT#1,M$
2130 PRINT#1,Y$
2140 PRINT#1,P
2150 PRINT"ENTER ANOTHER RECORD-Y OR N?"
2160 GET Y$
2170 IF Y$<>"Y" AND Y$<>"N" THEN 2160
2180 IF Y$="Y" THEN 2030
2200 CLOSE 1:RETURN
3000 REM:READING THE FILE
3010 OPEN1,1,0,"CARS"
3020 INPUT"ENTER MAKE OF CAR";T$
3040 INPUT#1,N$,M$,Y$,P
3060 IF T$<>N$ THEN 3180
3080 PRINT "MAKE",N$
3100 PRINT"MODEL",M$
3120 PRINT"YEAR",Y$
3140 PRINT "PRICE",P
3160 REM:TESTING FOR END OF FILE
3180 IF ST<>64 THEN 3040
3200 CLOSE 1:RETURN
4000 END
```

Fig. 10.3. Creating a sequential file of car records.

running the program – the only data is stored on the separate data file. Thus we could use one program to access several files of the same type – in this case several different files of cars – each on an individual, labelled tape.

When the file is written to the tape, it is recorded as a continuous string of characters, including the carriage return characters separating the strings of car details.

FORD <CR> CORTINA <CR> 1982 <CR> 25ØØ ...

As stated previously, the carriage return may either be achieved by separate PRINT# statements for each variable name, as in the previous program:

```
2110 PRINT#1,N$
2120 PRINT#1,M$
```

or by using a single PRINT# statement, including the carriage return between the variable names, e.g.

```
2105 C$=CHR$(13)
2110 PRINT#1,N$,C$,M$,C$,Y$,C$,P
```

CHR$(13) is the code for a carriage return, and this is assigned to store C$ for economy in the PRINT# statement. The file consists only of our strings of data and carriage return strings. The data on the tape is not assigned variable names as in a file in the computer's memory, so when the file is read back from tape we can asign it to any variables we choose, provided they are of the right type, either string ($) or numeric.

Examining a previously recorded file

Let's suppose that we want to have a look at an existing file. Provided we know the file number given to it on recording (usually 1 if you only use files one at a time), we can open it to read without a file name:

1Ø OPEN 1,1 (open file number 1 on the cassette unit)

GET#
Using the GET# statement we can read the file a character at a time, using the program in Fig. 10.4.

```
10 OPEN 1,1
20 GET#1,A$
30 PRINT A$
40 IF ST<>64 THEN 20
50 CLOSE 1
```

Fig. 10.4. Reading a file one character at a time.

The result of this program would be a display of the sequential file as a succession of individual characters as follows:

F
O
R
D

C
O
R
T
.
.

Note:
The gap between D and C is the carriage return written to the tape to separate the string constants.

By inserting a semi-colon at line 3∅,

3∅ PRINT A$;

the display would become

FORD CORTINA ... etc.

The fact that the sequential file is stored merely as a continuous single line of data does not affect the final display of data on the screen or printout on paper. This is controlled by the format used in the PRINT statements after the data has been read from the tape and assigned to the variable names which we have chosen to use in the INPUT# statements. These may be different from the variable names used during the saving of the data in the PRINT# statements.

Sequential file maintenance

A file of records may not be accurate for long – it will usually need

updating from time to time. For example, a person may change his address or telephone number. Prices of goods may need to be increased in line with inflation. It is necessary, therefore, to have the ability to modify parts of a file without retyping the entire set of records.

Unfortunately, with cassette files, it is not easy to overwrite fields in the old data with the modifications, since the characters to be replaced cannot be located with sufficient precision. The method used is to read the complete file into the computer's memory, carry out all necessary modifications, then re-save the new file in its entirety, preferably on a new and separate tape.

Provided records were kept of the modifications, the old tapes could be kept as security copies. If the latest version of the file were lost or damaged, it could be reconstructed using the old file and the modifications. It is normal to keep only the last three in the sequence of up-dated files, known as the grandfather, father and son. When a new file is created, it becomes the son, and the previous grandfather may be scrapped. For a very important file it may be worth making duplicate copies to store in a different location, in case of fire, or theft, etc.

The next program shows how a sequential file may be written, and a module is included to perform the various maintenance operations. The program has been written to allow five lines of data to be entered via INPUT statements; the program could easily be modified to allow more lines of data by extending the list of variables.

Any number of records, such as names and addresses may be entered. No data appears in the program itself – it is written directly to the tape file after INPUT by the user. The same program may, therefore, be used to create files on any subject. The amount of data stored in the file is governed by the available space on the tape or diskette, not by the space in the memory.

Updating the file

The operations which may be needed either singly or in combination are:

- Modify an existing record, by altering one or more fields.
- Add a new record to the end of the file (appending).
- Delete a complete record which is no longer required.

Before any alterations can be made to the file, it must be read from tape into the memory. The strings are read into subscripted variable

names A$(I),B$(I), etc. and the subscript I is provided by a counter as shown in Fig. 10.5.

```
2000 OPEN 1,1
2010 I=1
2020 INPUT#1,A$(I),B$(I),C$(I)
2025 INPUT#1,D$(I),E$(I)
2030 I=I+1
2040 IF ST<>64 THEN 2020
2050 CLOSE1:RETURN
```

Fig. 10.5. Counting the number of strings read from a file.

When all of the file has been read, the final value of I is assigned to N, for use in the FOR I=1 TO N ... NEXT loops when displaying or rewriting the file.

Modifying a record

When the file has been read into the memory using subscripted variables, it may be displayed on the screen using the subscript as the record number (Fig. 10.6).

```
3010 FOR I =1 TO N
3020 PRINT"RECORD NO.":I
3030 PRINT A$(I):PRINT B$(I)
```

Fig. 10.6. Displaying the file after reading.

etc.

To modify, say, record number 17, we are asked to INPUT the record number, then retype the amended record. Thus the strings A$(17),B$(17) etc. will be *overwritten* by the new version, and the process may be repeated with different records as required. At this stage the new file exists only in the computer's memory, and if no further changes are needed the complete file must be written to tape for permanent storage.

Appending a record

The appended record must be given the next subscript to that of the last record in the old file. This can be obtained by reading and displaying the file. The user is asked to INPUT the required record number and the program modifies N, the number of records, to allow for the appended record.

```
4020 INPUT A:IF A>N THEN N=A
```

The record to be appended is now entered in response to the INPUT prompts. Again, the complete file must be recorded on tape for permanent storage.

Deleting a record

The procedure is the same as for modifying a record, except that we retype the first line with a 'dummy' string, such as "DELETED", filling in the other lines with more dummy characters, such as **,**.

When the file is written to tape we test for the word "DELETED" and if found that particular record will not be written to the file. Figure 10.7 shows how this is done.

```
4320 FOR I=1 TO N
4325 IF A$(I)="DELETED" THEN 4350
4330 C$=CHR$(13)
4340 PRINT#1,A$(I),C$,B$(I),C$,C$(I)
4345 PRINT#1,D$(I),C$,E$(I)
4350 NEXT I
```

Fig. 10.7. Deleting a record as a file is rewritten.

Alternatively, if we also need to add a new record, this could be *inserted*, by overwriting the unwanted record, typing the new record with the same record number.

Creating and maintaining a file of data such as names and addresses

A diagram of the modules for a program to create and maintain a file of data is shown in Fig. 10.8. If it becomes necessary to perform additional operations, such as searching or sorting, then these could be inserted as additional modules. It would, of course, be necessary to modify the menu and the ON ... GOSUB statements. A program for creating and maintaining a file of data such as names and addresses is shown in Fig. 10.9.

You may wish to enter the program in Fig. 10.9 and create a file (on a separate tape from the program). You can improve the presentation by screen clearing and using PRINT:PRINT etc., TAB or programmed cursor movements.

Additional instructions to the user may be inserted such as:

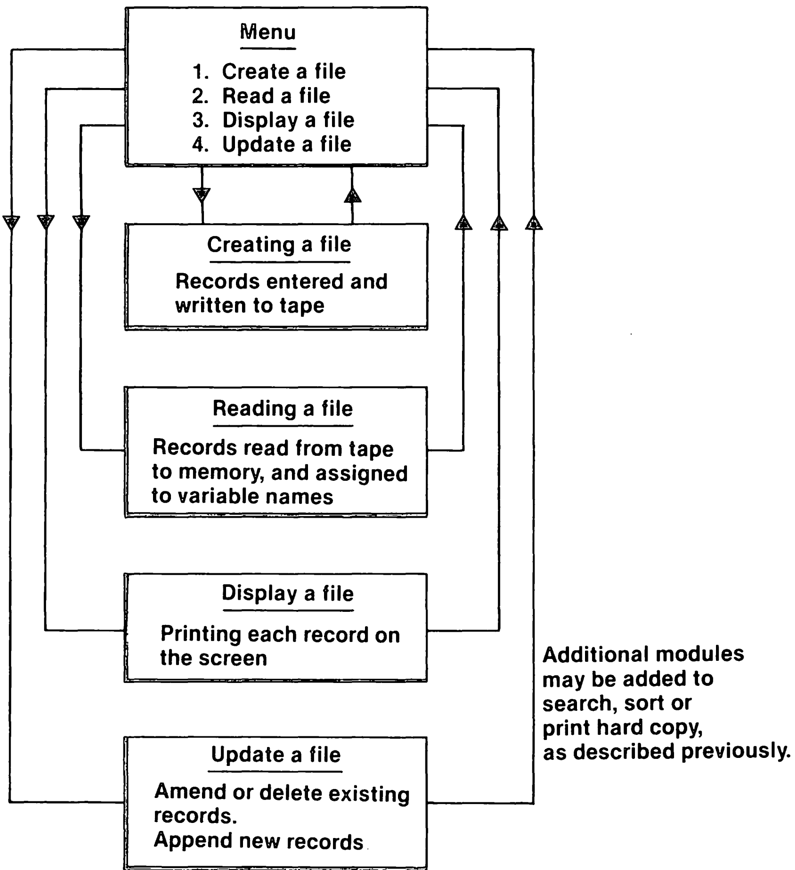


Fig. 10.8. The modules of a general program to create and maintain a file of data.

```
10 DIM A$(100),B$(100),C$(100)
20 DIM D$(100),E$(100)
100 REM:N$=NAME:L1$ TO L4$=DATA
```

```
199 REM:THE MENU
200 PRINT"□ENTER 1 TO CREATE A NEW FILE"
210 PRINT:PRINT"ENTER 2 TO READ A FILE"
215 PRINT
220 PRINT"ENTER 3 TO DISPLAY A FILE"
225 PRINT
230 PRINT"ENTER 4 TO UPDATE A FILE"
240 GET A$:LET A=VAL(A$)
245 IF A<>1ANDA<>2ANDA<>3ANDA<>4THEN240
250 ON A GOSUB 1000,2000,3000,4000
260 PRINT"END THIS RUN OF THE PROGRAM?"
265 PRINT:PRINT"PLEASE TYPE Y OR N"
270 GET G$
275 IF G$<>"Y" AND G$<>"N" THEN270
280 IF G$ ="N" THEN 200
290 END

999 REM:CREATING A NEW FILE
1000 OPEN1,1,1,"DATA"
1010 INPUT"NAME";N$:INPUT"LINE 1";L1$
1020 INPUT"LINE 2";L2$:INPUT"LINE 3";L3$
1030 INPUT"LINE 4";L4$
1040 PRINT"ENTER Y TO RECORD-N TO RETYPE"
1050 GET G$:IFG$<>"Y"ANDG$<>"N"THEN1050
1060 IF G$="N" THEN 1010
1070 C$=CHR$(13)
1080 PRINT#1,N$,C$,L1$,C$,L2$
1085 PRINT#1,L3$,C$,L4$
1090 PRINT"ENTER ANOTHER RECORD-Y OR N?"
1100 GET G$:IFG$<>"Y"ANDG$<>"N"THEN1100
1110 IF G$="Y" THEN 1010
1120 CLOSE 1:RETURN

1999 REM:READING A FILE
2000 OPEN 1,1
2010 I=1
2020 INPUT#1,A$(I),B$(I),C$(I)
2025 INPUT#1,D$(I),E$(I)
2030 I=I+1
2040 IF ST<>64 THEN 2020
2050 CLOSE 1:RETURN
```

```
2999 REM:DISPLAYING THE FILE
3000 LET N=I-1
3010 FOR I=1 TO N
3020 PRINT "▮RECORD NO";I
3030 PRINT A$(I):PRINT B$(I)
3040 PRINTC$(I):PRINTD$(I):PRINTE$(I)
3050 PRINT"PRESS SPACE BAR TO CONTINUE"
3060 GET G$:IF G$<>" "THEN 3060
3070 NEXT:RETURN

3999 REM:UPDATING A FILE
4000 PRINT"ENTER THE NO. OF THE RECORD"
4010 PRINT"TO BE CHANGED OR APPENDED"
4020 INPUT A:IF A>N THEN N=A
4030 INPUT"NAME";A$(A)
4035 INPUT"LINE 1";B$(A)
4040 INPUT"LINE 2";C$(A)
4045 INPUT"LINE 3";D$(A)
4050 INPUT"LINE 4";E$(A)
4060 PRINT"MODIFY ANOTHER RECORD ?"
4090 PRINT"ENTER Y OR N"
4100 GET G$:IF G$<>"Y"ANDG$<>"N"THEN4100
4110 IF G$="Y" THEN 4000
4120 REM:WRITING THE UPDATED FILE
4130 OPEN 1,1,1,"NEW"
4140 FOR I=1 TO N
4150 IF A$(I)="DELETED" THEN 4180
4160 C$=CHR$(13)
4170 PRINT#1,A$(I),C$,B$(I),C$,C$(I)
4175 PRINT#1,D$(I),C$,E$(I)
4180 NEXT I
4190 CLOSE 1:RETURN
```

Fig. 10.9. A program to create and maintain a file of data such as names and addresses.

"PLACE A NEW TAPE IN THE CASSETTE UNIT"

Frequently appearing messages, such as

"PRESS SPACE BAR TO CONTINUE"

may be displayed in reverse field to distinguish them from the data itself.

Once this program has been entered and recorded it may be used to save data of any type. After a file has been created for, say, names, addresses and telephone numbers, the same program could be used

to set up a file of photographic records, a personal library, a music library or a recipe file.

Although this program has been written for records each consisting of five pieces of data, it could easily be extended or reduced by modification of the list of variables:

Chapter Eleven
Extending the System

The previous chapters described work which can be carried out using the Commodore 64 and cassette recorder. Additional equipment is available to extend the computer into a powerful business system which is still sufficiently compact to fit unobtrusively into the average home.

The complete system consists of the Commodore 64, disk drive and printer, and although eminently suitable for the small business, the home user will also find immense advantages in obtaining the extra equipment (Fig. 11.1). These include the greater speed of the disk drive compared with cassette tape, the ability to produce permanent output as hard copy on paper and the development of larger programs with the aid of printed program listings.

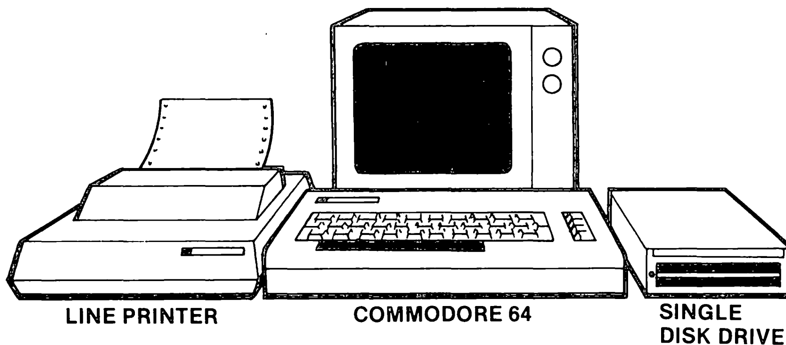


Fig. 11.1. The Commodore 64 expands easily into a complete system for the small business or experienced programmer.

All this additional equipment is readily available from Commodore dealers, without the need for special conversion work which involves considerable expense on some other brands of computer.

Disk files

The advantages of diskette files

The new owner of a Commodore 64 may not initially be able to justify the additional expense of the floppy disk unit. However, as you become more experienced and begin to write lots of substantial programs and data files, the benefit to be gained is enormous. As an indication of the relative speeds, a large program which takes 3 minutes to load from cassette is in the computer's memory after only 22 seconds from diskette! Similar savings in time occur with data files, so a disk unit is essential for the serious data processor. Once you have experienced the pleasure of using a disk unit you will be reluctant to return to cassette.

Apart from much greater speed in loading and saving programs and searching data files, the diskette has several other advantages:

- You can display, in seconds, a directory or catalogue of all programs and data files stored on a particular disk. This is not possible on cassette.
- One diskette can store 170K of programs or data files, or both. Banks of these can be stored neatly in special lockable cabinets containing masses of data in a very small space; the 'paperless office' becomes a possibility.
- Many files can be stored on the same disk and accessed during the same program. I have used such a system to store student bank accounts: each student's account represents an individual file and any one of these files may be accessed during a run of the program. Each disk can hold 144 different files on the Commodore 64.

There are also several 'housekeeping' operations which are just not possible using cassette tapes:

- On diskette it is possible, using a simple command, to resave a new version of a program or data file, with the same name as used previously. This is very useful, for instance, in updating a file and resaving, and avoids the need for 'MARK 1', 'MARK 2', etc. (Of course, earlier versions may still be retained if desired.)
- It is also possible to rename an existing file, to duplicate a file with a different name or to erase an unwanted data file or program from the disk.

Such precise operations are not possible with cassette tapes because it is not possible to locate the files with the same degree of accuracy.

This precision is made possible because the Commodore 1541 disk drive has its own microprocessor which monitors where everything is stored at known locations on the disk.

The floppy disk

The Commodore 64 uses 5¼-inch, single-sided, single-density diskettes. These are inserted into the drive with the label uppermost and to the right, and with the 'write protect' notch to the left. (A tab stuck over this notch prevents valuable programs or files from being overwritten.)

Warning! The disk is encased in a 5¼-inch square cardboard case and this case must not be removed.

Using the 1541 floppy disk unit

The diskette should be entered into the drive with the 'write-protect' notch to the left and the labels uppermost and to the rear (see Fig. 11.2).

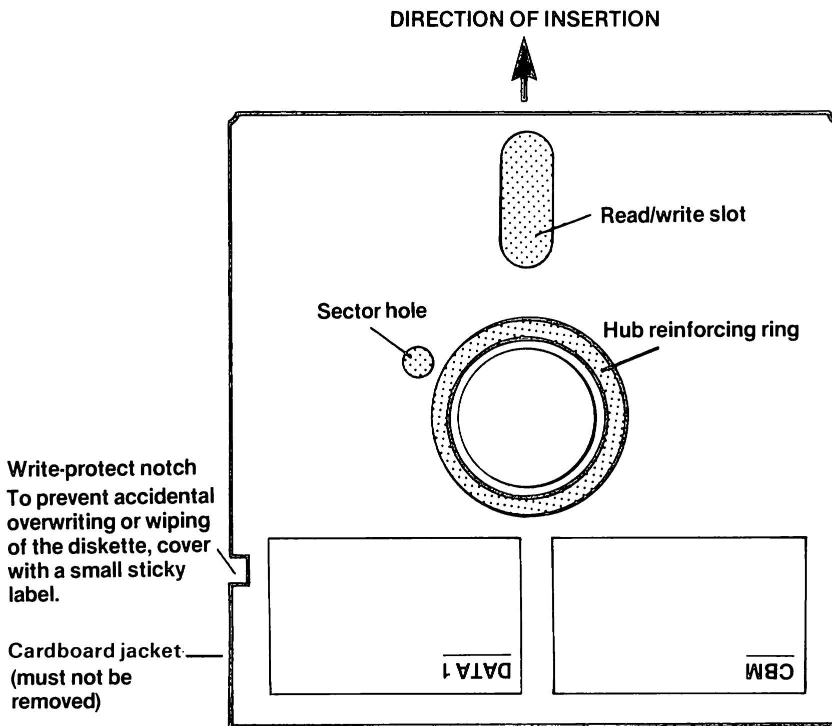


Fig. 11.2. The floppy disk in its cardboard jacket.

Under no circumstances should the diskette be removed from its cardboard jacket and care should be taken to avoid damage from dust, magnetic fields, bending, coffee or extremes of temperature or humidity! In practice I have found floppy disks to be extremely reliable and durable, provided they are handled sensibly. The lockable cabinets which contain 50 diskettes are ideal and as this represents 8500K of backup storage for the Commodore 64, they provide a very compact filing system for the home or small business.

While a diskette is being processed, the red warning light will appear. Should this not be extinguished on completion of the operation an error has occurred and reference should be made to the User's Manual.

The Commodore 1541 disk unit uses 35 track disks. Each concentric track is divided into a number of blocks, as shown in Fig. 11.3. The outer tracks, being longer, accommodate more blocks than the inner. One block of the disk can hold 256 bytes or characters. As there are 664 available blocks, the capacity of the diskette is 664 × 256, i.e. 169984 in addition to the directory track, number 18.

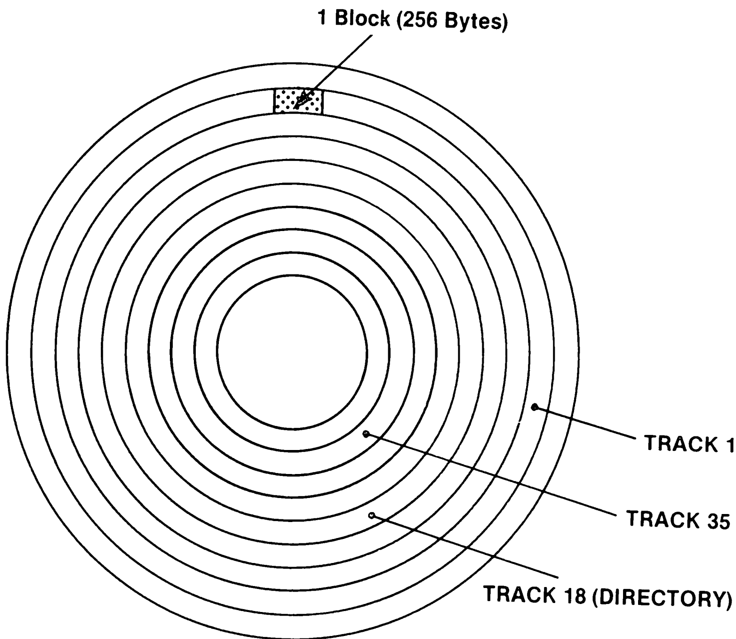


Fig. 11.3. The 5¼-inch floppy disk as used on the Commodore 64. (Specification: 5¼-inch, single-sided, single density, soft sectored.)

Preparing a new disk
In order to prepare a new diskette or to clear an old one, the *formatting* process must be carried out. This erases the entire diskette and divides the tracks into sectors or blocks. As already explained, the Commodore 1541 disk unit uses 35 track disks. When the diskette is formatted the concentric tracks, numbered from 1 to 35 inwards are divided into varying numbers of blocks, depending on the distance of the track from the centre (and therefore the length of the concentric track) - see Fig. 11.4.

Track number	Blocks per track	Total blocks
1-17	21	17 × 21 = 357
18-24	19	7 × 19 = 133
25-30	18	6 × 18 = 108
31-35	17	5 × 17 = 85
		TOTAL 683

Fig. 11.4. The organisation of blocks on a floppy disk (diskette).

Therefore the diskette is divided into 683 blocks or sectors during the formatting process. At the same time the directory is written on track 18 (using 19 blocks), leaving 664 blocks available to the user.

The format statement
To format a new disk, we must give the entire disk a name, such as FLOPPY, and a short 2-character identification code, such as Ø1. The format statement (typed in immediate mode) would then be:

```
OPEN 15,8,15,"NØ:FLOPPY,Ø1"
```

This will take about 85 seconds since the whole disk has to be processed. The same statement will also wipe clean a previously used diskette.

The disk is now ready for use in saving program or data files, the formatting process having written on the disk the title and the identification (Ø1 in this case).

The directory
To check the directory, type LOAD"$",8 then, after loading, type

LIST, press RETURN. Figure 11.5 shows what should appear on the screen.

∅ "FLOPPY "∅1 2A

664 BLOCKS FREE

Fig. 11.5. The directory of a newly-formatted diskette.

Saving a program

To save a program which is in the memory, give the program a short name, such as "TEST" and type:

SAVE "TEST",8

We will use as an example a simple one-line program such as:

1∅ PRINT"COMMODORE 64"

When we type SAVE"TEST",8 the screen will display SAVING TEST then READY. While saving, the *red* light will appear on the disk unit and then go out when saving is complete. To verify that an exact copy of the memory has been made on the diskette, type:

VERIFY "TEST",8

The screen will display:

SEARCHING FOR TEST
VERIFYING
O.K.

Again, the red light will come on during the verifying process, then go out when the operation has been successfully completed. We can now examine the directory of the disk to check that it contains the new program:

Type LOAD"$",8

The screen should now display:

SEARCHING FOR $
LOADING
READY

Since the directory is now stored in the memory, it can be listed like a program, so type LIST followed by RETURN. The directory has now been updated to include our one line program "TEST", and is displayed as shown in Fig. 11.6.

```
ø "FLOPPY         "ø1 2A
1 "TEST"             PRG
663 BLOCKS FREE
```

Fig. 11.6. The directory of a diskette containing one program file.

Comparing this with the initial directory obtained after the diskette was formatted, shows that there are now 683 blocks free. Our one-line program has used 1 block (shown on the left) and is saved as a *program* file (PRG), to distinguish it from a sequential file (SEQ) or a relative file (REL) which may be saved and listed in the same directory.

Saving a new program with an existing name

Sometimes we may wish to replace an earlier version of a program (or a data file), by a new version, using the same name. For instance, our simple program "TEST" could be altered, then resaved under the name TEST. This avoids using TEST 1, TEST 2, etc. Suppose the new one-line program was:

```
1ø PRINT"SAVING ON DISK"
```

To save this new program under the old name "TEST", enter:

```
SAVE"@ø:TEST",8
```

After verifying, if we now type LOAD"TEST",8 and LIST, the latest version of the program should appear, having replaced the previous version of the same name.
Note: The directory is loaded and listed as if it were a normal program file. Therefore, before typing in a new program it is necessary to type NEW and press RETURN – otherwise the directory will become the first few lines of your program!

Removing a program – the SCRATCH command
This enables you to erase a program (or data) file using the SCRATCH command (abbreviated to S) as follows:

```
OPEN 15,8,15,"Sø:TEST"
```

If we type this and press RETURN, the program TEST will be removed from the disk. This can be checked by displaying the directory.

Most of the commands for cassette file handling may be used with

diskette files, provided they are suffixed by ,8 to specify device 8, the disk unit. For example:

```
SAVE"TEST",8
VERIFY"TEST",8
LOAD"TEST",8
```

In addition, these can be abbreviated by the normal method available on the Commodore 64, i.e. the first letter of the command, followed by the second letter pressed together with SHIFT. Some of the abbreviations which can be used are shown in Fig. 11.7.

```
LOAD=L SHIFT O = L┌

SAVE=S SHIFT A = S♠

VERIFY =V SHIFT E = V─
```

File 11.7. Abbreviations used on the Commodore 64.

Data files

Sequential files

These were covered in the section on cassette files and most of the information given there applies equally to sequential files on diskette. The essential differences are:

- Although the file is still read in sequence, from end to end, it is much faster on the disk unit.
- A directory can be displayed showing all of the files stored on the disk, and the amount of space they occupy. A maximum of 144 different files can be stored on one disk – this is obviously a totally different concept from cassette files. Each of these files can be accessed very quickly by simply typing in its unique name which will be used in an OPEN statement.

The main difference between programming sequential files on disk and those on cassette is in the OPEN statements, which must specify device 8. All other instructions, such as PRINT# to write to the disk, and INPUT#, to read from it, are identical.

Writing a diskette file

A short program to write names to a file (until a 'dummy' word 'end' is typed in) is shown in Fig. 11.8.

```
10 OPEN 4,8,4,"0:DAT,S,W"
20 INPUT A$
30 PRINT#4,A$
40 IF A$<>"END"THEN 20
50 CLOSE 4
```

Fig. 11.8. Writing to a diskette sequential file.

Only line 1ϕ contains any new ideas compared with the earlier chapter on cassette files:

1ϕ OPEN 4,8,4,"ϕ:DAT,S,W"

In the above statement:

- The first 4 is the file number, which can be any number from 1 to 255.
- The 8 specifies the disk unit, device 8.
- The second 4 is the channel number which can be between 2 to 14, inclusive.
- The ϕ specifies disk drive ϕ. (Some devices have two drive slots, ϕ and 1.)
- DAT is any short name which we care to give this particular file.
- S is the abbreviation for sequential file.
- W specifies that we are opening to *write* to the file.

When we run this program, we can keep entering names until we type END to finish. There will be some initial activity at the disk unit while the name is written on the file. While we are inputting the names the drive may be silent, until we type END, when the file will be closed and all data in the buffer will be written onto the diskette.

If we now load the directory (LOAD"$",8) and LIST, we should see what is shown in Fig. 11.9 on the screen.

```
ϕ "FLOPPY            "ϕ1  2A
1 "TEST"                  PRG
1 "DAT"                   SEQ
662 BLOCKS FREE
```

Fig. 11.9. Diskette directory containing one program and one sequential file.

Our sequential file "DAT" appears next to our program file "TEST" (see Fig. 11.9), classified by the letters SEQ and shown to occupy 1 block of data. (Again, remember to type NEW after using the directory, to avoid mixing the directory with future programs.)

Reading a diskette file

To read from the diskette file, the procedure is the same as for cassette files, the only difference being the OPEN statement (Fig. 11.10).

```
10 OPEN4,8,4,"0:DAT,S,R"
20 INPUT#4,A$
40 PRINT A$
50 IF ST<>64 THEN 20
60 CLOSE 4
```

Fig. 11.10. Opening a diskette file to read.

Line 1Ø OPENS the sequential (S) file called DAT for a read (R) operation. Data is INPUT from the file until the end of the file is detected by a status equal to 64.

Apart from the OPENing statements just described, the programs and information given in the previous chapters may be applied to diskette files. When writing to a diskette file with the PRINT# statement, strings must again be separated either by individual PRINT statements for each variable or by programmed carriage return characters (CHR$(13)) – see Fig. 11.11.

```
1000 C$=CHR$(13)
1010 PRINT#4,A$C$B$
```

Fig. 11.11. Using the carriage return (CHR$(13)) to separate strings in a sequential diskette file.

Similarly, the GET# statement may be used as previously described, to retrieve single characters from a diskette-file.

Replacing a file by a new file of the same name

Frequently a file may need to be updated then resaved. In the case of a sequential file, it is necessary to resave the whole file. The replace option (available only on disk) allows the new file to replace the old file with the same name. The statement to OPEN to write a replacement file is as follows:

OPEN 4,8,4,"@Ø:DAT,S,W"

The reader may test this by rewriting a second file of names under the file name DAT as before, then reading them back from disk. A further check could be made by listing the directory.

Relative files
The reader will find that sequential files on diskette are a great improvement on the equivalent files on cassette. However, relative files enable the user to access a particular piece of data without reading through the whole file. This is obviously quicker when large quantities of data are to be processed. The programming required for relative files is rather more complicated than for sequential files and it is not considered within the scope of this book.

Line printers

It is often useful (if not essential) to produce a permanent record of the computer's memory on paper – 'hard copy' in computing jargon. There are two main uses of the printer: (a) for program listings (the actual BASIC statements which make your program) and (b) for program output (the results or answers which your program generates when it RUNS).

Program listings
When you start to develop large programs, it is inconvenient and inefficient to try to modify statements using the screen. It is not possible to see the whole program at once and therefore it is difficult to see how various parts or 'modules' interrelate.

When the current version of a program needs to be modified, it is easier to make amendments and corrections on paper first, before entering them into the computer. If more than one person needs to use the system, this approach maximises the utilisation of the equipment.

It is also useful to keep blocks or modules from successful programs to be used in other programs at a later date. If these listings are kept on printout, writing of programs then involves the assembly of standard modules rather than starting from first principles every time. If, at various stages in the evolution of a program, a program listing on paper is made and labelled, it will also be easier to make comparisons should problems arise in later versions.

Printed output
Hard copy is simply the diversion of the screen display to the paper in the line printer. The results of the running of the program may need to be communicated to another person needing a permanent

record, such as a payslip, gas bill, or a bank statement. Also, hard copy is needed if the computer is used for word processing or printing address labels.

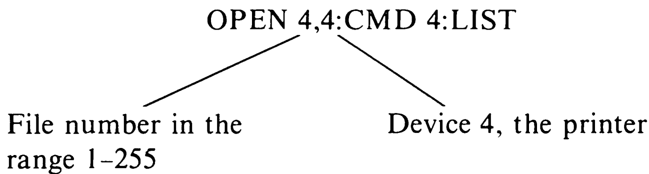
Most printers accommodate rolls of sticky labels (in addition to normal 'listing' paper) so that addresses can be stored on disk, then printed out whenever necessary. In addition, it is possible to obtain more expensive 'letter quality' printers if presentation is very important.

To operate the Commodore 64 printer, it is necessary to OPEN the computer to the external device (the printer, device No. 4), specifying a file number (in the range 1–255). The OPEN statement is typed in immediate or program mode.

Obtaining a printed listing

The computer should be opened to the printer as follows, to obtain a program listing.

In immediate mode:



When RETURN is pressed, the program listing will be printed on paper. To return all output to the screen it is necessary to close the channel to the printer as follows:

```
PRINT#4:CLOSE 4
```

Diverting program output to the printer

To print the *results* of the program on paper instead of on the screen, it is necessary to insert – at an appropriate line number within the program – the statement to open up the channel to the printer. For example:

```
5ØØØ OPEN 4,4:CMD 4
```

The above line must appear before the statements which we wish to cause printing on paper, but *after* any statement which we wish to cause printing on the screen. For instance, a program may need to print instructions on the screen, and perhaps display menus and input prompts.

The statement to OPEN the printer must come after these screen

displays, but before the required paper-printed output.

As soon as the print statements for the output on paper are complete, we must close the printer to return the display to the screen. The following statement might be used on the Commodore 64:

```
55ØØ PRINT#4:CLOSE 4
```

Note that you may insert the printer opening and closing statements anywhere in the program, using any line numbers.

A very simple example is shown in Fig. 11.12.

```
100 INPUT"ENTER A WORD";A$
110 OPEN4,4:CMD4
120 PRINT"MY WORD IS";A$
130 PRINT#4:CLOSE4
140 GOTO 100
```

Fig. 11.12. Inserting the printer OPEN and CLOSE commands around the PRINT statements.

This program displays the INPUT prompt on the screen after line 1ØØ:

ENTER A WORD?

When a name is entered and RETURN is pressed, line 11Ø causes the channel to the printer to be opened. The statement CMD4 causes all subsequent output to be diverted to paper on the line printer, not onto the screen. So the word we input will be printed on paper by the PRINT statement at line 12Ø. At 13Ø, future output is diverted back to the screen by the PRINT#4 statement, and CLOSE 4 closes the 'File' which was opened up for communication with the printer.

Note: The normal Commodore 64 abbreviation for PRINT (i.e. ?) cannot be used in a PRINT# statement, which should also be typed without spaces.

By inserting the OPENING and CLOSING lines in the program, printed output can be obtained from any part of the program, (as often as desired). For example, in a large program, we may wish to give the choice of displaying on the screen or printing on paper. From the previous chapters, we can see that this can be achieved by two similar subroutines, one enclosed by printer OPEN and CLOSE statements and the other not. Each subroutine would be accessed from a menu, as discussed earlier. Figure 11.13 shows the modules of

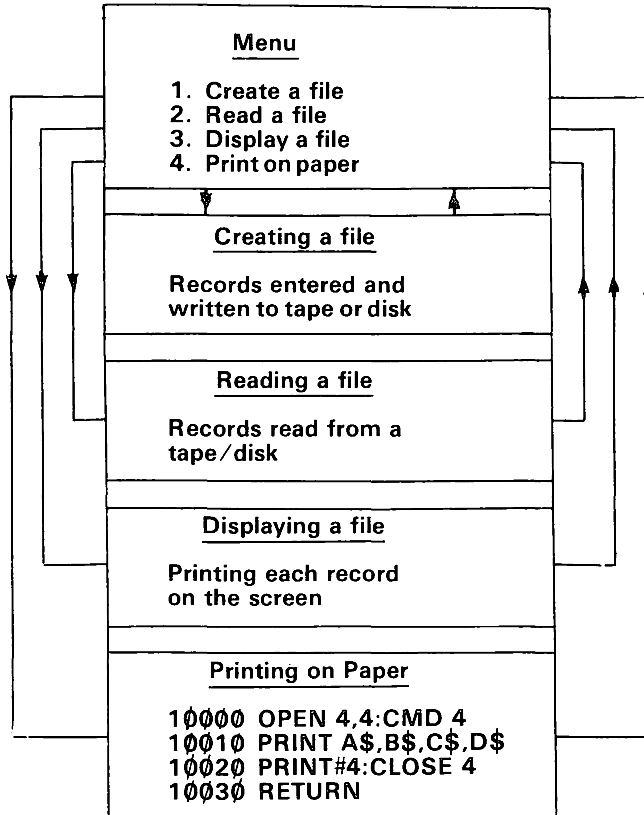


Fig. 11.13. Inserting an option to PRINT output on paper. The OPEN and CLOSE of the line printer may be programmed as an independent module accessed as a subroutine from the main menu with subsequent RETURN.

a simple program which includes an option to print the output on paper.

Print layout

The line printer uses a moving electromechanical print head, quite different from printing text on the screen. In addition, printers normally print 80 characters per line, instead of the 40 columns displayed on the screen. In general, it is not possible to improve our layout by the method of programmed cursor movements described earlier in this book.

One of the PRINT formatting techniques which does work on the line printer is as follows:

; The semi-colon will cause succeeding PRINT statements to be printed next to each other (concatenation) as shown in Fig. 11.14.

```
10 OPEN 4,4:CMD 4
20 PRINT"COMMODORE
30 PRINT"64"
40 PRINT#4:CLOSE4
```

Fig. 11.14. Using the semi-colon to cause 'concatenation' on the line printer.

Run this and COMMODORE 64 should be printed on paper.

Vertical spacing

This can be achieved by printing blank lines in between the appropriate lines as in Fig. 11.15.

```
10 OPEN 4,4:CMD 4
20 PRINT"SMITH"
30 PRINT:PRINT
40 PRINT"JONES"
50 PRINT#4:CLOSE4
```

Fig. 11.15. Using PRINT:PRINT to obtain two blank lines on the line printer.

If we run this, the two PRINT statements at line 3Ø cause a space on the paper of two blank lines as follows:

SMITH
JONES

This is useful when regular spacing needs to be adjusted, say, when printing address labels which must have a set distance between corresponding lines on adjacent labels. Any number of PRINT statements may be inserted wherever necessary, such as

PRINT:PRINT:PRINT

which gives a space of three blank lines.

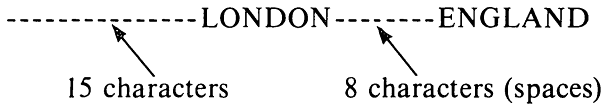
Horizontal spacing – SPC

The SPC statement allows a specified number of spaces to be left between printed strings, as shown in Fig. 11.16, for example,

```
10 OPEN4,4:CMD 4
20 PRINT SPC(15)"LONDON"SPC(8)"ENGLAND"
30 PRINT#4:CLOSE4
```

Fig. 11.16. The use of SPC to obtain spacing on the line printer.

Line 2Ø leaves 15 spaces before printing "LONDON", then 8 spaces before printing "ENGLAND":



This method works for single lines of printing but does not allow several rows of names to be printed in vertical columns underneath one another.

The following program will demonstrate the problem. We need to print 5 pairs of names in pairs, with the beginnings of the names in a vertical line. Our first attempt might be to leave 3 spaces between each pair of names using SPC(3) as in Fig. 11.17.

```
10 OPEN 4,4:CMD 4
20 FOR I= 1 TO 5
30 READ A$,B$
40 PRINT A$SPC(3)B$
50 NEXT I
60 PRINT#4:CLOSE4
70 DATA SMITH,JONES,BROWN,MITCHELL
80 DATA WILLIAMS,PETERSON,SCOT,MCNAB

90 DATA ARCHIBALD,LEE
```

Fig. 11.17. A first attempt at printing two vertical columns of names on the line printer.

When we run this program, A$ and B$ are printed on paper as follows:

```
SMITH   JONES
BROWN   MITCHELL
WILLIAMS   PETERSON
SCOT   MCNAB
ARCHIBALD   LEE
```

The problem in the above erratic layout is that SPC(3) leaves 3 spaces after the end of each name printed. Obviously, the position of the start of printing the second name depends on the *length* of the first word.

Now, if we were to work from the *beginning* of the first word, and wish all second words to start at, say, column 21, we have:

(Length of first name) + (number of spaces) = 2Ø
Therefore, number of spaces = 2Ø − length of first name

We have seen earlier that the length of a string can be counted using the LEN function. So, to start printing in column 21 we may use:

SPC(2∅−LEN(A$)).

The full program is shown in Fig. 11.18.

```
10 OPEN 4,4:CMD4
20 FOR I=1 TO 5
30 READ A$,B$
40 PRINT A$SPC(20-LEN(A$))B$
50 NEXT I
60 PRINT#4:CLOSE4
70 DATA SMITH,JONES,BROWN,MITCHELL
80 DATA WILLIAMS,PETERSON,SCOT,MCNAB
90 DATA ARCHIBALD,LEE
```

Fig. 11.18. Using LEN to obtain vertical columns on the line printer.

When the program in Fig. 11.18 is RUN, the output should now appear in vertical columns:

```
SMITH               JONES
BROWN               MITCHELL
WILLIAMS            PETERSON
SCOT                MCNAB
ARCHIBALD           LEE
```

This technique can be used to obtain a second vertical column of names anywhere on the paper; you simply alter the number 2∅ to whatever number of spacings you require. Similarly, the idea can be extended if you were printing more than two strings on a line.

And finally

I hope you have found this book helpful and that my aim to provide a text which is easy to read and understand has been achieved. The examples included in the text will, I hope, provide a skeleton on which you can build larger and more 'friendly' programs.

Don't be afraid to explore new ideas; you cannot damage the machine via the keyboard, although you may waste a lot of time!

The satisfaction which you will feel on successfully using a program which you have developed will more than compensate for the frustrations you have suffered in writing it!

Appendix 1
A Glossary of Terms used in Data Handling

BASIC: Beginners All-purpose Symbolic Instructional Code. A language used on most personal computers, it must be translated or interpreted into the machine code language of the computer.

Byte: A set of eight binary digits (∅'s and 1's) representing a character (e.g. P,7, etc.) in the internal code of the machine.

Cassette: Small magnetic tape used for recording computer programs – special C15 computer cassettes are available.

Character: A figure, letter, punctuation mark, graphics symbol and also a space.

Constant: A number or a string of characters to be stored in a named memory location, e.g. in LET A=17 and LET A$= "JONES", the number 17 and the string "JONES" are constants (unlike the *variables* A and A$ whose contents may be overwritten or changed).

Data: Facts, figures, words, characters, input in a 'raw' or unprocessed form, to the computer.

Data bank: A large collection of files or databases. One program might gain access to several of the files in a data bank.

Diskette: See *floppy disk*.

Documentation: A written report on a program, including its purpose, limitations, listing, sample output and operating instructions.

Execution: Carrying out the instructions during a RUN of the program.

Field: An individual piece of data within a record, such as a postcode within a name and address.

File: A set of records, such as the names and addresses of all of the members of a club.

Floppy disk: Magnetic disk, often 5¼ inches in diameter, on which data can be stored and retrieved at high speed, using concentric tracks. Single-sided/single-density disks may hold 100K

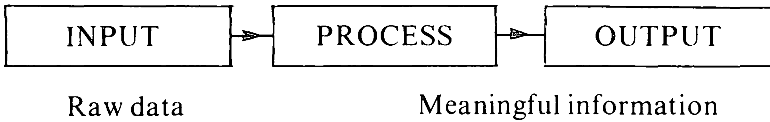
characters while double-sided/double-density have the capacity for 400K per disk. Dual disk drive units give the facility for copying. Disk is much faster, but more expensive than cassette tape.

Grandfather-father-son system: When files are frequently updated with new records, etc. it is normal to keep only the last three in the line of development – the latest version being the son.

Hard copy: The name given to the print-out from a computer on paper, rather than as a screen display.

Hardware: The metal, plastic, silicon components of the computer.

Information: The final output from the computer after processing, which may have included sorting or calculating. Information implies facts which are meaningful to the reader.



K (kilobyte): The unit of memory capacity – 1K is 1024 (or 2^{10}) characters.

Listing: A display on the VDU or printer of all of the lines of the program currently in the memory.

Loop: A section of a program whose instructions are repeated until some condition is satisfied, usually achieved by FOR ... NEXT and on some machines REPEAT ... UNTIL.

Machine code: The internal language of the computer, consisting of combinations of Ø's and 1's, enabling instructions, calculations, sorting and storing of data to be performed, e.g. the letter C is often 1ØØØØ11; the figure 7 is Ø11Ø111.

Megabyte: A million bytes.

Menu: A list of options displayed on the screen from which the user can opt to run any one of several independent routines, before returning to the menu to make a further choice, e.g. if we INPUT A as either 1,2, or 3 respectively, we gain access to one of three subroutines starting at lines 5ØØØ,7ØØØ,9ØØØ by the statement:

```
ON A GOSUB 5ØØØ,7ØØØ,9ØØØ
     1
     2
     3
```

Microprocessor: A chip, often silicon, containing an arithmetic and logic unit, a memory and a control unit. Microprocessors are used

in washing machines, watches, calculators, etc. as well as microcomputers.

Microcomputer: In addition to a microprocessor, the microcomputer contains additional storage, a typewriter-style keyboard, and facilities to display output on a screen (VDU) or print on paper.

Peripherals: External devices separate from, but connected to, the computer, e.g. printer, disk drive, cassette tape recorder.

Program: A set of instructions known as statements, which can be stored in the computer, enabling a particular task to be repeated as required.

QWERTY: The standard layout for the letters on the keyboard of many computers (and typewriters).

RAM: Random Access Memory. That part of the computer's memory which is available to the user to 'write' to and 'read' from.

Record: One set of data, within a file, such as the name and address of one person, or the recipe for one dish.

ROM: Read Only Memory. Part of the memory which contains *permanently* stored programs used in the operation of the machine and not available to the user for storing programs. (*EPROM* – Erasable, Programmable Read-Only Memory – may be reprogrammed by the user with specialist knowledge and equipment).

Searching: Scanning the data to identify those items which fulfil particular requirements. For example

IF A$(I) = "PORSCHE" THEN ...

Software: Programs and files, usually stored on diskette or tape. Commercial programs for a particular application are known as 'software packages'.

Sorting: Arranging data in a particular order, such as alphabetical or numerical.

String: A string of characters, e.g. "CECIL" or "A549LAP".

Subroutine: A section of a program which is included only once, but which may be called up and 'executed' frequently. Usually called up by GOSUB ... and left by RETURN.

Subscript: A number appended in brackets to a variable so that one letter may uniquely identify a large *array* of stores. For example:

A$(1),A$(2),A$(3)..........A$(3ØØ)

The following routine shows how to read numbers into an array.

```
1000 FOR I=1 TO 300
1010 READ A$(I)
1020 NEXT I
```

Syntax: The rules for the programming language. Thus, PLINT instead of PRINT would produce a SYNTAX error.

Variable: The label given to a memory store, whose contents can be changed and therefore *vary.* Numeric data is stored in A,B,...Z and A1,A2, etc. String data is stored in A$,B$,...Z$ and A1$,A2$, etc.

VDU: Visual Display Unit – the television or monitor screen on which programs and output are displayed.

Word processing: Typewriting using a microcomputer. Text may be checked and corrected on the screen before printing on paper. Standard letters and files may be permanently stored on diskette for subsequent recall, *infilling* from a file of names and addresses and other individual details. Multiple copies may be made.

Appendix 2
Calculation Supplement

This book was specifically written to show that computing had extended its scope well beyond the limited field of mathematical calculation, for which the early computers were predominantly used.

However, the owner of a personal computer may well wish to do some arithmetic in such tasks as accounts, VAT calculations, etc. This section gives the necessary additional statements which may be inserted as calculation modules into the programs given in the previous chapters.

We have already seen that the computer may be used as a calculator in *immediate* mode. For instance, type in (without a line number)

```
PRINT 7*9
```

Then press RETURN (* means 'multiply' in BASIC). As a result, 63 is output on the screen.

This method may be used for all of the mathematical operations:

```
PRINT 7/3   divides 7 by 3
PRINT 2↑3   raises 2 to the power of 3 (i.e. 2³ or 2×2×2)
```

When several operations are combined in the same line, they are done in a definite order or hierarchy. For example:

```
PRINT 3*4↑2-57*(8+7)
```

The order of operations here is:

First : Evaluating brackets
Second : Raising to a power (↑)
Third : Multiplication (*) and Division (/)
Fourth : Addition and subtraction

In the example above, the bracket would be evaluated to give 15,

then the power 4^2 to give 16, followed by the multiplication. The remaining operations of addition and subtraction would, being of equal priority, be performed from left to right. So, the answer to this problem would be worked out as follows:

Normal mathematical format: $3 \times 4^2 - 57(8+7)$

BASIC format: 3*4↑2−57*(8+7)

Sequence of operations:
3*4↑2−57*15
3*16−57*15
48−855
−8ø7

Note that while in mathematics 57(8+7) is assumed to mean 57×(8+7), in BASIC the multiplication sign must be present, i.e. 57*(8+7).

Our arithmetic can, of course, be written into BASIC statements. Figure A.1 shows how to add two numbers and print their total:

```
10 LET A=7
20 LET B=6
30 LET C=A+B
40 PRINT "TOTAL IS ";C
```

Fig. A.1. Adding two numbers and printing their total.

Figure A.2 shows how to find the mean of 5 numbers:

```
10 READ A,B,C,D,E
20 LET S=A+B+C+D+E
30 LET M=S/5
40 PRINT "MEAN IS"M
50 DATA 57,319,25,62,84
```

Fig. A.2. A program to find the mean of 5 numbers.

A program to print the squares of the first 200 numbers is given in Fig. A.3:

```
10 PRINT "NUMBER","SQUARE"
20 FOR N=1 TO 200
30 PRINT N,N↑2
40 NEXT N
```

} similarly for cubes (↑3) etc.

Fig. A.3. A program to print the squares of the first 200 numbers.

(*Note:* Square roots may be obtained by using N ↑ .5 or SQR(N) .)

The sort of problems we are likely to tackle on a personal computer are calculations such as VAT, where a price must be increased by 15% i.e. a factor of 1 $^{15}/_{100}$ or 1.15. A suitable program to do this is shown in Fig. A.4.

```
10 REM VE=VAT EXCLUSIVE PRICE
20 REM VI=VAT INCLUSIVE PRICE
30 INPUT"ENTER THE PRICE";VE
40 VI=VE*1.15
50 PRINT"PRICE WITH VAT"VI
60 GOTO 30
```

Fig. A.4. A program to calculate VAT on prices INPUT by the user.

In the same way, National Insurance at, say, 6% would be calculated as $^{6}/_{100}$ or a factor of 0.06 of gross salary:

```
3Ø LET NI = GS*Ø.Ø6
```

Compound interest

Assuming a constant interest rate of say, 8% or $^{8}/_{100}$, we multiply the principal or original amount by 1.08 to obtain the amount after one year. The new amount is again multiplied by 1.08 to obtain the amount after 2 years and similarly for succeeding years. This leads to a general formula:

$$A = P\left(1 + \frac{r}{100}\right)^n$$

where A is the total amount, P is the principal or original amount, r is the interest rate as %, and n is the number of years. A suitable program is set out in Fig. A.5.

```
10 INPUT "ENTER THE PRINCIPAL";P
20 INPUT "ENTER THE RATE %";R
30 INPUT "ENTER THE TIME IN YEARS";N
40 LET A=P*(1+R/100)↑N
50 LET I=A-P
60 PRINT"INTEREST AFTER "N" YEARS ON "
70 PRINT P"POUNDS IS "I" POUNDS"
80 END
```

Fig. A.5. A compound interest program.

Try running this program with a range of values for P,R and N and improve the screen layout by clearing the screen, and PRINT: PRINT to give spacing between lines (see the section on screen layout in Chapter 2).

Random numbers - the RND function

This function allows you to generate 'random' numbers within any desired range. The quotation marks are included because the numbers are not truly random, but part of a predetermined sequence of random numbers permanently stored in the computer. If we type in immediate (or program) mode

PRINT RND(1)

a random number will appear when this statement is executed. If you now switch off your machine and then switch on again, retyping

PRINT RND(1)

will again produce the same random number, in this case .185564016. (*Note:* To save switching off and on again it is possible to achieve the same effect by typing the machine code instruction SYS 64738 and pressing RETURN.)

The above test shows that when using the RND function, we are using a fixed sequence of random numbers each time the machine is switched on. To investigate this further, type SYS 64738, press RETURN, then RUN the program in Fig. A.6 which prints ten random numbers in the range 0–5.

```
10 FOR N= 1 TO 10
20 PRINT INT(6*RND(1));
30 NEXT N
```

Fig. A.6. Printing 10 random numbers in the range 0–5 inclusive.

Line 2ø may be explained as follows. RND(1) generates random numbers to nine places of decimals in the range from 0.000000000 to 1.000000000 'inclusive':

.1855640̸16
.0̸46898635
.82774380̸1
etc.

The probability, when working to 9 places, of achieving either exactly 0.000000000 or exactly 1.000000000 is so small as to be practically impossible. So RND(1) generates random numbers from a fixed sequence, in a range greater than 0 and less than 1.

6*RND(1) produces 'random' numbers in the range 0–6, although again this is not, in practice, inclusive. For example:

1.1133841
.281391809
4.9664628
etc.

Rounding down using INT

In line 20 in Fig. A.6, INT(6*RND(1)); rounds the numbers *down* to the nearest integer (i.e. whole number), so the only possible results must be taken from the set of integers:

{ 0,1,2,3,4,5 }

Running the program of Fig. 13.6 produces the following output:

1 0 4 3 5 3 5 5 1 5

If we run the program again, we produce

4 2 4 4 5 2 2 4 5 3

then

4 1 3 2 0 1 4 3 4 4

Clearly, although each run of the program is producing a different set of 10 random numbers, the same sequence of sets will be repeated every time the machine is switched on and this program is run.

To start off with a different set of random numbers every time the machine is switched on, type:

RND(-TI)

We have seen that:

INT(6*RND(1));

produces random numbers in the range 0–5 inclusive. We could, for example, simulate dice throwing by using

INT(6*RND(1))+1   giving a range 1–6 inclusive.

Similarly, a program to test multiplication tables at random might use:

```
INT(12*RND(1))+1
```

to generate numbers in the range 1–12 inclusive. Finally, to generate random numbers in a given range A to B inclusive, where A is the larger number, we can use:

```
INT ((A–B)*RND(1))+B
```

Obviously, the RND function will probably find most use in learning, games or simulation programs. It could, though, be useful in choosing names at random from a large data file for market research or advertising purposes or to simulate some process in which events occur randomly.

INT

As mentioned earlier, INT rounds numbers down to the nearest lower whole number or integer. For example,

```
1Ø PRINT INT(39.765)
```

produces 39 when RUN.

INT can be useful to test whether a number is divisible by another number. For example,

35÷8 = 4.375
INT(35÷8) = INT(4.375) = 4
∴35÷8 does not equal INT (35÷8)
However, 4Ø÷8 = 5
INT(4Ø÷8) = INT(5) = 5
∴ 4Ø÷8 = INT(4Ø÷8)

The test, therefore, to see if a number A is divisible by a number B is that:

A/B = INT(A/B)

A practical application of this might occur when we were printing out records in a FOR ... NEXT loop, and wished to halt the screen display after every 3 records. The program is shown in Fig. A.7.

```
5000 FOR N=1 TO 200
5010 PRINT A$(N):PRINT B$(N):PRINT C$(N)
5020 IF N/3=INT(N/3) THEN GOSUB 8000
5030 NEXT N
```

Fig. A.7. Halting the screen after every three records have been printed (using N/3=INT(N/3)).

Note: GOSUB 8ØØØ branches to the subroutine "PRESS SPACE BAR TO CONTINUE". This is not a complete program – to apply it to your programs, insert a line like 5Ø3Ø in your print module. If necessary, replace 3 by the number of records to be displayed on the screen at any one time. For example, for 4 records use:

..... IF N/4 = INT(N/4) THEN GOSUB 8ØØØ.

Obviously, you must have the subroutine at line 8ØØØ, as used frequently earlier in this book. This halts the screen while "PRESS SPACE BAR TO CONTINUE" is displayed on the screen.

Correcting to the nearest whole number

As we have already seen, INT rounds numbers *down* to the nearest lower whole number. Clearly this is not always the most accurate approximation, since, for example, PRINT INT(14.9999999) would yield 14 as the nearest integer.

Obviously, this rounding down using INT is not always satisfactory and in some cases could lead to serious errors. A common mathematical practice is to round *down* all integers when the first decimal place is less than 5, and to round *up* when the decimal is 5 or more. A comparison of this 'true' method with the results produced using INT (using numbers between 7 and 8 as an arbitrary example) is set out in Fig. A.8.

It can be seen from this table that rounding 7.9 down to 7.0 obviously causes a large percentage error. A neat device to overcome

X	INT(X)	'True' approximation for X
7.Ø	7.Ø	7.Ø
7.3	7.Ø	7.Ø
7.5	7.Ø	8.Ø
7.7	7.Ø	8.Ø
7.9	7.Ø	8.Ø

Fig. A.8. Comparison of INT(X) with normal mathematical approximation.

this problem is to add 0.5 to all values of X. The effect of this is shown in Fig. A.9.

X	(X+∅.5)	INT(X+∅.5)	'True' approximation for X
7.∅	7.5	7.∅	7.∅
7.3	7.8	7.∅	7.∅
7.5	8.∅	8.∅	8.∅
7.7	8.2	8.∅	8.∅
7.9	8.4	8.∅	8.∅

Fig. A.9. Use of INT(X+∅.5) to give an accurate approximation.

It can be seen that by adding 0.5 to all of the values for X, the INT function now produces the same results as the normal mathematical method.

A practical application of this might be the correction of money to the nearest penny. Suppose that an item costing £3.97 is marked up by 12%, i.e. multiplied by a scale factor of $^{112}/_{100}$ or 1.12. Further multiplying by $^{115}/_{100}$ or 1.15 to include VAT at 15% would result in the following calculation:

3.97 × 1.12 × 1.15

Running this in a program would produce £5.11336 but such a degree of accuracy would be unnecessary and unsuitable for output on a printed statement. To correct £5.11336 to the nearest penny, we must convert to pence multiplying by 100, then use our INT(X+∅.5) function:

1∅∅ × 5.11336→511.336
INT(511.336 + ∅.5) = INT(511.836) = 511 pence.

We must now divide by 100 to convert 511 pence to £5.11.

This method can be summarised as follows:

INT(1∅∅*P + ∅.5)/1∅∅

where P is the amount in £.

Obviously a calculation like this is quite complicated to program every time we want to correct money to the nearest penny.

Such a routine, often referred to as an *algorithm*, is worth keeping in a handy place for future use, or to save remembering or working out from first principles.

The Commodore 64 has a method which permits repeated use of such a frequently used function, without the need to continually retype the whole function. It is simply necessary to define the function once, then the function may be referred to whenever needed.

DEF FN

The function is first defined by the DEF FN statement as follows:

DEF FNA(P) = INT(1ØØ*P+Ø.5)/1ØØ

We have given this particular function the letter A. Within a program there may be other functions, B, C, etc. defined in statements like:

DEF FNB( ) = ..., DEF FNC( ) = ...

In the statement DEF FNA(P) = INT(1ØØ*P+Ø.5)/1ØØ. P is a 'dummy' variable, which will be replaced by a variable name every time the function is used on that particular variable.

As a simple example we will consider the calculation of selling prices, after marking up the cost price by 12% and adding VAT at 15% (Fig. A.10).

```
10 REM CP=COST PRICE:VT=VAT
20 REM SP=SELLING PRICE:VP=TOTAL
30 DEF FNA(P)=INT(100*P+0.5)/100
40 INPUT"[CLR]COST PRICE";CP
50 SP=CP*1.12
60 VT=SP*0.15
70 VP=SP+VT
80 PRINT"[CLR]COST PRICE ";CP
90 PRINT"SELLING PRICE ";FNA(SP)
100 PRINT"VAT ";FNA(VT)
110 PRINT"TOTAL PRICE ";FNA(VP)
```

Fig. A.10. Using DEF FN... and FN... to correct money to the nearest penny.

Having set up the function A in the DEF FN ... statement at line 3Ø, we can apply it to any variable which we wish to correct, anywhere in the program. The 'dummy' variable P in

DEF FNA(P) = INT(1ØØ*P+Ø.5)/1ØØ

is replaced successively by variables SP, VT and VP later in the program. Further functions may be defined for other algorithms, replacing DEF FNA by statements such as

DEF FNB ... DEF FNC..., DEF FNX ...

This defined function is similar to a subroutine, in that a relatively complicated expression or routine is programmed only once, but called up, or referred to, frequently.

Index

abbreviations, 15, 98
algorithm, 120
alphabetical sort, 53
amending, 76
appending, 76, 85
approximation, 118
array, 24, 36
ASC, 17
ASCII code, 16
assignment, 19

BASIC, 108
binary, 16
block, 95
bubble sort, 53
buffer, 80
byte, 108

calculation, 20, 112
carriage return, 78, 100
cassette, 77
cassette file, 77
category search, 68
character strings, 21
CHR$, 16, 78
clearing the screen, 12
CLOSE, 78, 103
CMD, 103
coding sheet (40 column), 73
colon, 14
commands, 5
Commodore 1541 disk unit, 93
compound interest, 114
concatenation, 22
constant, 108
counter, 20, 35
CRSR keys, 13
cubes of numbers, 113
cursor, 13

DATA, 5, 22, 72
data bank, 108
data files (cassette), 77
data files (disk), 92, 98
DEF FN, 120
deleting a file, 97
deleting a line, 6
deleting a record, 86
dimensioning, 24
directory, 92, 95, 99
disk drive unit, 93
disk files, 92
documentation, 108
dummy data, 35, 73

END, 31
end of data marker, 35, 73
end of file marker, 78, 100
error message, 25
error tapping, 59, 63

field, 108
file, 75, 108
floppy disk, 93, 108
floppy disk unit, 93
formatting (disk), 95
FOR ... NEXT, 10
FRE(Ø), 74

GET, 29
GET#, 83
GOSUB, 30
GOTO, 9
grandfather,father,son, 84, 109
greater than, 34

halting the screen, 29
hard copy, 109
hardware, 5, 109

identifier, 67
IF ... THEN, 34
immediate mode, 3, 24
information, 109
INPUT, 32
INPUT#, 78
inserting, 6, 86
INT, 116, 117
inverted commas, 14, 73

keeper, 53
kilobyte, 109

learning program, 50
LEFT$, 47
LEN, 47, 106
less than, 34
LET, 19
line numbers, 6
line printer, 101
LIST, 5, 10
listing on paper, 101
LOAD, 5, 98
loop, 10, 52, 109

maintenance of files, 83
megabyte, 109
menu, 7, 59, 109
microcomputer, 2, 110
microprocessor, 109
MID$, 49
module, 7
multiple cursor operations, 13
multiple statement lines, 66

nested loops, 27
NEW, 5
numbers, 19
ON ... GOSUB, 60
ON ... GOTO, 60
OPEN, 78, 99
output on paper, 101

peripherals, 91, 110
press space, 30
PRINT, 4, 6, 9, 14
printer formatting, 104
PRINT#, 78, 98
program, 4, 110
programmed cursor movement, 13
program mode, 4
prompts, 79

QWERTY, 2, 110

RAM, 4, 110
random numbers, 115
READ, 22
reading from a file, 78
record, 39, 110
REM, 42
repetition, 10
RETURN, 60
reverse field, 14
RIGHT$, 48
RND, 115

SAVE, 5, 96
SCRATCH, 97
screen layout, 12
searching, 34
sector, 95
semi-colon, 14
sequential files, 80
SHIFT/CLR, 12
software, 110
sorting, 53, 110
SPC, 12, 106
squares of numbers, 12, 113
STEP, 11
store, 11, 19
string, 65, 110
STR$, 22
subroutine, 30, 60
subscripted variable, 23, 110
swap routine, 53
syntax, 25, 111

TAB, 12
telephone directory, 61
time delay, 10, 52

unique name, 50
updating a file, 84
user-friendliness, 79

VAL, 22, 60
VAT, 114
variable name, 19, 111
VDU, 8, 111
VERIFY, 5, 38, 96

word processing, 111
write protect notch, 93
writing to a file, 78, 98

APPLE II

APPLE II PROGRAMMER'S HANDBOOK
0 246 12027 4 £10.95

AQUARIUS

THE AQUARIUS AND HOW TO GET THE MOST FROM IT
0 246 12295 1 £5.95

ATARI

GET MORE FROM THE ATARI
0 246 12149 1 £5.95

THE ATARI BOOK OF GAMES
0 246 12277 3 £5.95

BBC MICRO

ADVANCED MACHINE CODE TECHNIQUES FOR THE BBC MICRO
0 246 12227 7 £6.95

ADVANCED PROGRAMMING FOR THE BBC MICRO
0 246 12158 0 £5.95

THE BBC MICRO: AN EXPERT GUIDE
0 246 12014 2 £6.95

BBC MICRO GRAPHICS AND SOUND
0 246 12156 4 £6.95

DISCOVERING BBC MICRO MACHINE CODE
0 246 12160 2 £6.95

DISK SYSTEMS FOR THE BBC MICRO
0 246 12325 7 £7.95

HANDBOOK OF PROCEDURES AND FUNCTIONS FOR THE BBC MICRO
0 246 12415 6 £6.95

INTRODUCING THE BBC MICRO
0 246 12146 7 £5.95

LEARNING IS FUN: 40 EDUCATIONAL GAMES FOR THE BBC MICRO
0 246 12317 6 £5.95

TAKE OFF WITH THE ELECTRON AND BBC MICRO
0 246 12356 7 £5.95

21 GAMES FOR THE BBC MICRO
0 246 12103 3 £5.95

PRACTICAL PROGRAMS FOR THE BBC MICRO
0 246 12405 9 £6.95

THE COLOUR GENIE

MASTERING THE COLOUR GENIE
0 246 12190 4 £5.95

COMMODORE 64

BUSINESS SYSTEMS ON THE COMMODORE 64
0 246 12422 9 £6.95

ADVENTURE GAMES FOR THE COMMODORE 64
0 246 12412 1 £6.95

COMMODORE 64 COMPUTING
0 246 12030 4 £5.95

COMMODORE 64 DISK SYSTEMS AND PRINTERS
0 246 12409 1 £6.95

THE COMMODORE 64 GAMES BOOK
0 246 12258 7 £5.95

COMMODORE 64 GRAPHICS AND SOUND
0 246 12342 7 £6.95

COMMODORE 64 WARGAMING
0 246 12410 5 £6.95

SOFTWARE 64: PRACTICAL PROGRAMS FOR THE COMMODORE 64
0 246 12266 8 £5.95

INTRODUCING COMMODORE 64 MACHINE CODE
0 246 12338 9 £7.95

40 EDUCATIONAL GAMES FOR THE COMMODORE 64
0 246 12318 4 £5.95

DRAGON

THE DRAGON 32 AND HOW TO MAKE THE MOST OF IT
0 246 12114 9 £5.95

THE DRAGON 32 BOOK OF GAMES
0 246 12102 5 £5.95

THE DRAGON PROGRAMMER
0 246 12133 5 £5.95

DRAGON GRAPHICS AND SOUND
0 246 12147 5 £6.95

INTRODUCING DRAGON MACHINE CODE
0 246 12324 9 £7.95

ELECTRON

ADVANCED ELECTRON MACHINE CODE TECHNIQUES
0 246 12403 2 £6.95

ADVANCED PROGRAMMING FOR THE ELECTRON
0 246 12402 4 £5.95

ADVENTURE GAMES FOR THE ELECTRON
0 246 12417 2 £6.95

ELECTRON GRAPHICS AND SOUND
0 246 12411 3 £6.95

ELECTRON MACHINE CODE FOR BEGINNERS
0 246 12152 1 £7.95

THE ELECTRON PROGRAMMER
0 246 12340 0 £5.95

HANDBOOK OF PROCEDURES AND FUNCTIONS FOR THE ELECTRON
0 246 12416 4 £6.95

PRACTICAL PROGRAMS FOR THE ELECTRON
0 246 12362 1 £7.95

21 GAMES FOR THE ELECTRON
0 246 12344 3 £5.95

40 EDUCATIONAL GAMES FOR THE ELECTRON
0 246 12404 0 £5.95

TAKE OFF WITH THE ELECTRON AND BBC MICRO
0 246 12356 7 £5.95

IBM

THE IBM PERSONAL COMPUTER
0 246 12151 3 £6.95

LYNX

LYNX COMPUTING
0 246 12131 9 £6.95

MEMOTECH

MEMOTECH COMPUTING
0 246 12408 3 £5.95

THE MEMOTECH GAMES BOOK
0 246 12407 5 £5.95

ORIC-1

THE ORIC-1 AND HOW TO GET THE MOST FROM IT
0 246 12130 0 £5.95

THE ORIC PROGRAMMER
0 246 12157 2 £6.95

THE ORIC BOOK OF GAMES
0 246 12155 6 £5.95

TI99/4A

GET MORE FROM THE TI99/4A
0 246 12281 1 £5.95

VIC-20

GET MORE FROM THE VIC-20
0 246 12148 3 £5.95

THE VIC-20 GAMES BOOK
0 246 12187 4 £5.95

ZX SPECTRUM

AN EXPERT GUIDE TO THE SPECTRUM
0 246 12278 1 £6.95

INTRODUCING SPECTRUM MACHINE CODE
0 246 12082 7 £7.95

LEARNING IS FUN: 40 EDUCATIONAL GAMES FOR THE SPECTRUM
0 246 12233 1 £5.95

MAKE THE MOST OF YOUR ZX MICRODRIVE
0 246 12406 7 £5.95

THE SPECTRUM BOOK OF GAMES
0 246 12047 9 £5.95

SPECTRUM GRAPHICS AND SOUND
0 246 12192 0 £6.95

THE SPECTRUM PROGRAMMER
0 246 12025 8 £5.95

THE ZX SPECTRUM AND HOW TO GET THE MOST FROM IT
0 246 12018 5 £5.95

WHICH COMPUTER?

CHOOSING A MICROCOMPUTER
0 246 12029 0 £4.95

LANGUAGES

COMPUTER LANGUAGES AND THEIR USES
0 246 12022 3 £5.95

EXPLORING FORTH
0 246 12188 2 £5.95

INTRODUCING LOGO
0 246 12323 0 £5.95

INTRODUCING PASCAL
0 246 12322 2 £5.95

MACHINE CODE

Z80 MACHINE CODE FOR HUMANS
0 246 12031 2 £7.95

6502 MACHINE CODE FOR HUMANS
0 246 12076 2 £7.95

SOFTWARE GUIDES

WORKING WITH dBASE II
0 246 12376 1 £7.95

USING YOUR MICRO

COMPUTING FOR THE HOBBYIST AND SMALL BUSINESS
0 246 12023 1 £6.95

DATABASES FOR FUN AND PROFIT
0 246 12032 0 £5.95

FIGURING OUT FACTS WITH A MICRO
0 246 12221 8 £5.95

INSIDE YOUR COMPUTER
0 246 12235 8 £4.95

SIMPLE INTERFACING PROJECTS
0 246 12026 6 £6.95

PROGRAMMING

COMPLETE GRAPHICS PROGRAMMER
0 246 12280 3 £6.95

THE COMPLETE PROGRAMMER
0 246 12015 0 £5.95

PROGRAMMING WITH GRAPHICS
0 246 12021 5 £5.95

WORD PROCESSING

CHOOSING A WORD PROCESSOR
0 246 12347 8 £7.95

WORD PROCESSING FOR BEGINNERS
0 246 12353 2 £5.95

FOR YOUNGER READERS

BEGINNERS' MICRO GUIDES: ZX SPECTRUM
0 246 12259 5 £2.95

BEGINNERS' MICRO GUIDES: BBC MICRO
0 246 12260 9 £2.95

BEGINNERS' MICRO GUIDES: ACORN ELECTRON
0 246 12381 8 £2.95

MICROMATES

SIMPLE ANIMATION
0 246 12273 0 £1.95

SIMPLE PICTURES
0 246 12269 2 £1.95

SIMPLE SHAPES
0 246 12271 4 £1.95

SIMPLE SOUNDS
0 246 12270 6 £1.95

SIMPLE SPELLING
0 246 12272 2 £1.95

SIMPLE SUMS
0 246 12268 4 £1.95

GRANADA GUIDES: COMPUTERS
0 246 11895 4 £1.95

Data processing – that is, sorting raw facts to produce useful information – can be just as satisfying as playing games, and more rewarding. This book explains how to use the Commodore 64 to process information for the home and small business.

Straightforward examples show you how to program your micro to store large quantities of data, to display data on the screen in an attractive and readable way, and to search the data to select particular items and print out useful information in a special order.

The programs in this book have been written to appeal to micro users of all ages:

- **Home computer owners wishing to graduate from computer games**
- **Business users wishing to write programs tailor-made to their particular needs**
- **Students preparing for examinations in computer studies and data processing**

The Author

James Gatenby is a chartered engineer and Head of Computer Studies at a school and community centre.

Other Commodore 64 books from Granada

COMMODORE 64 COMPUTING
Ian Sinclair
0 246 12030 4

COMMODORE 64 DISK SYSTEMS AND PRINTERS
Ian Sinclair
0 246 12409 1

BUSINESS SYSTEMS ON THE COMMODORE 64
Susan Curran and Margaret Norman
0 246 12422 9

INTRODUCING COMMODORE 64 MACHINE CODE
Ian Sinclair
0 246 12338 9

Front cover illustration by Angus McKie

GRANADA PUBLISHING
Printed in Great Britain 0 246 12454 7

£5.95 net