CENTURY COMMUNICATIONS

THE COMMODORE 64 HANDBOOK



Lupton & Robinson

THE COMMODORE 64 HANDBOOK

CENTURY SOFTWARE VOUCHER

The major programs in this book are available on cassette from the publishers.

Please cut out this voucher (or copy the voucher) and send it together with a cheque / postal order for £5 to:

Software Department,
Century Communications Ltd,
Portland House,
12-13 Greek Street,
London W1.

VOUCHER

Please send me one cassette of the major programs in the COMMODORE 64 HANDBOOK at the special price of £5 including postage and packing.

NAME .....

ADDRESS .....

Please allow 28 days for delivery.

THE COMMODORE 64 HANDBOOK

PETER LUPTON & FRAZER ROBINSON



CENTURY COMMUNICATIONS

LONDON

Copyright © Peter Lupton and Frazer Robinson 1984

All rights reserved

First published in Great Britain in 1984
by Century Communications Ltd,
12–13 Greek Street,
London W1V 5LE

ISBN 0 7126 04197

Typeset by Spokesman, Bracknell
Reproduced, printed and bound in Great Britain by
Hazell Watson & Viney Limited,
Member of the BPCC Group,
Aylesbury, Bucks

NOTE: The programs in this book have been typeset directly from the Commodore 64.

CONTENTS

Acknowledgements
Introduction

1	Computers and Programs	1
2	Setting up your 64	3
3	First Steps	8
4	Programming	18
5	Program Control	34
6	Data and Programs	52
7	Pieces of Strings	59
8	Functions	73
9	Logical Thinking	82
10	Memory Management	89
11	Sound and Music	93
12	Character Graphics	119
13	High Resolution Graphics	146
14	Sprites	158
15	Permanent Storage	184
16	Advanced Techniques	204

APPENDICES

1	Abbreviations	209
2	BASIC Commands	210

3	BASIC Error Messages	221
4	Speeding Up Programs	227
5	Colour Codes	229
6	Numbering Systems	230
7	SID Registers	235
8	VIC II Registers	237
9	Screen Memory Maps	239
10	Character Codes	241
11	System Memory Map	246

Index

INTRODUCTION

This book is an introduction to the COMMODORE 64 computer and its facilities. It takes you from the first steps in BASIC programming, pointing out the pitfalls and explaining some 'tricks of the trade', through to a detailed understanding of how to write good structured programs with many examples along the way.

Separate chapters explain how to make the most of the 64's remarkable sound and graphics facilities - many of which are not described in the Users Guide. Each chapter contains many illustrative programs, and there are several longer programs which demonstrate the sound and graphics and will help you to master the more difficult aspects.

A set of useful appendices ensures that the book will remain a valuable reference work long after you have grasped the principles of the machine.

We hope that you find the book both enjoyable and instructive, as you discover the full potential of the COMMODORE 64.

Peter Lupton and Frazer Robinson 1984

ACKNOWLEDGEMENTS

Once again we must thank the people who contributed in some way to this book, and especially those colleagues whose efforts smoothed the path to completion.

Thanks also to our families and friends, who have continued to support and encourage us, despite having been deprived of our company for so long. We look forward to seeing them all again now the book is finished.

CHAPTER 1

COMPUTERS AND PROGRAMS

Despite everything you may have heard about computers being hyperintelligent machines which are on the point of taking over the world, a computer is not really intelligent at all. A computer is at heart little more than a high-speed adding machine, similar to an electronic calculator, but with more sophisticated display and memory facilities.

The feature that gives computers their immense flexibility and potential is this: they can store lists of instructions which they can read through and obey at very high speed. These lists of instructions are called programs, and most of this book is concerned with how these programs are written.

The computer instructions which form programs must follow certain rules, or the computer will not be able to understand them. The rules for writing programs resemble the rules of a spoken language, and so the set of instructions is often said to form a programming language. There are many different computer programming languages; the one that the Commodore 64 understands (in common with most other personal computers) is called BASIC. (The name BASIC is an acronym for Beginners' All-purpose Symbolic Instruction Code.)

A programming language is much simpler than a human language because computers, for all their power, are not as good at understanding languages as people are. The BASIC language used by the 64

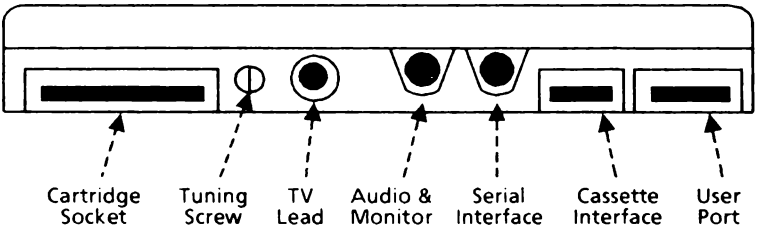
has less than 80 words. The rules for combining the words - the 'grammar' of the language - are much more strict than for a language like English, again because it is difficult to make computers that can use languages in the relaxed sort of way in which we speak English. These may seem like limitations, but in fact as you will discover BASIC is still a powerful language, and it is possible to write programs to perform very complex tasks.

Finally, remember this. Your computer will not do anything unless you tell it to, so whatever happens, you're the boss. The 64 won't take over the world unless you make it!

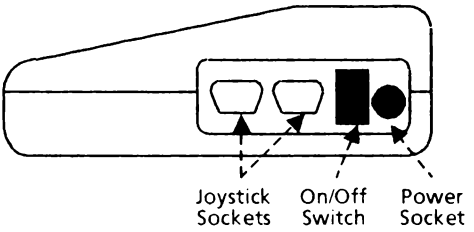
CHAPTER 2

SETTING UP YOUR 64

Before you can use your Commodore 64 you must connect it to a power supply and to a television. To



The Commodore 64 Connectors - Rear View



The Commodore 64 Connectors - Side View

load and save programs you will also need to connect a cassette unit to the 64. Before connecting anything, make sure you know what should plug in where. The diagrams above show all the connector sockets of the 64.

There are several sockets through which the 64 passes and receives information, and one through which it gets the electrical power it needs to operate. The sockets are not all labelled, so be sure to refer to the diagram before plugging anything in.

POWER

The 64 needs a low voltage DC supply, and this is obtained from the power supply unit supplied with your computer. Plug this power supply into the mains, and plug the output lead into the computer. The socket for the power supply is at the right-hand side of the machine. Do not switch on yet!

DISPLAY

The Commodore 64 uses a standard domestic TV to communicate with you, and for this almost any TV will do. To get the best results, use a modern, good quality colour TV. If you use a black and white set, the colour displays produced by the computer will appear as shades of grey.

To connect the 64 to the TV, plug the supplied aerial lead into the aerial socket of the TV, and plug the other end into the socket at the back of the computer (check the diagram). The lead has a different type of plug at each end, so take care that you don't try to force in the wrong one.

CASSETTE RECORDER

The Commodore 64, like most other small computers, uses cassette tapes to save programs or information, so that you don't have to type them in every time you need them.

You cannot use an ordinary cassette recorder with the 64: you must use the special Commodore

cassette unit, which is available at extra cost. The cassette unit plugs into a socket at the rear of the computer. No separate power supply is needed for the cassette unit as this is provided by the computer.

DISK DRIVE

Programs can also be stored on floppy disks. A floppy disk drive is faster than a cassette unit and is more flexible in use. (It is also much more expensive). The disk drive connects to the round serial interface socket at the rear of the computer. The disk unit requires its own power supply and so must also be plugged into the mains.

PRINTER

If you have a printer, its lead plugs into the serial interface socket at the rear of the computer. If you are also using a disk drive you should plug the printer lead into the spare socket at the rear of the disk drive. The printer also has a separate mains lead.

SWITCHING ON

When you have connected everything together, you are ready to switch on. The equipment must be switched on in the right order, or there is a risk of damaging the computer.

First switch on the TV.

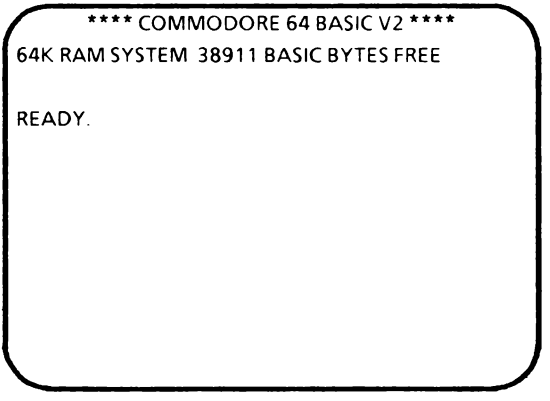
Second, switch on the disk drive and then the printer. (Remember that you should never switch a disk unit on or off with disks inside it.)

Last, switch on the computer itself.

Switching off should be carried out in the reverse order: first the computer, then the disk drive and printer, and last of all the TV.

TUNING

To get a display to appear on the TV screen, tune to channel 36, or, on a pushbutton set, use a spare channel and keep tuning until you see this appear on the screen:



If you are unable to tune the television, perform the following checks:

1 Check that the aerial lead is connected.

2 Make sure the computer is connected to the mains and switched on. The red power indicator should be on.

3 Try tuning the TV again, and - if possible - try a different TV.

With a little time and careful tuning, it is possible to get a clear and stable display on nearly all types of TV. If you are unsuccessful, consult your dealer.

If you are thinking of buying a TV especially for use with your 64, it's worthwhile taking the computer to the shop, as certain types of TV seem to give better results than others.

For the best quality display, you could buy a monitor. This is a display specifically designed for use with a computer, and contains no circuitry for TV reception. However, a good colour monitor can cost twice the price of the computer!

The 64 provides a standard output for a monitor, through the socket next to the cassette recorder socket. Your dealer can advise you on the connecting leads required.

CHAPTER 3

FIRST STEPS

Before you can make use of your Commodore 64, you must find out how to communicate with it.

Communication is a two-way process: you must give the 64 instructions, and you must be able to find out how it responds to them. You give instructions using the keyboard, and the 64 displays its response on the TV screen.

Type the following phrase on the keyboard:

```
PRINT "CBM 64"
```

You will see the letters appear on the screen as you type them. The flashing square - called the cursor - will move to indicate where the next letter you type will appear. Nothing else happens though - the computer has not yet obeyed your instruction.

Now press the RETURN key. The words 'CBM 64' appear on the screen, and the word 'Ready' is printed to tell you that the 64 is waiting for your next command. (If instead of printing 'CBM 64' the computer prints '?SYNTAX ERROR' or 0, it means you have made a typing mistake. Try again!).

So, to give the 64 a direct command you type it at the keyboard and press RETURN. Try another command:

```
PRINT "HELLO!"
```

Again, the letters between the quotation marks are printed. You can tell the 64 to print any sequence of letters, but you must remember to put them between quotation marks. Try making it print your name.

You don't have to type **PRINT** in full every time - the question mark means the same to the computer. Try :

```
? "HELLO!"
```

(remembering to press RETURN, of course).

MANAGING THE DISPLAY

As well as typing on to the screen, there are a number of different ways to alter the display.

The Cursor

The cursor can be moved around the screen using the two keys labelled 'CRSR' at the bottom right of the keyboard. Keep one or other of the keys pressed to move the cursor in the direction of the arrows at the front of the key. To reverse the direction, hold down the SHIFT key while pressing the cursor key.

This method of assigning two functions to a key is exactly the same as on a typewriter, except that on the Commodore 64, it is taken a stage further, with one key meaning three or four different things depending upon which other key you press as well.

Try moving the cursor around, (it will continue to move for as long as you press the key) and watch what happens when it gets to the edge of the screen. When it reaches one side it re-appears at the other side. But when the cursor reaches the bottom of the screen, the display begins to move off

the top of the screen. This vertical movement is called **scrolling**. Notice that anything that disappears when the picture scrolls cannot be recovered.

To return the cursor to the top left hand corner of the screen (known as the HOME position), press the CLR/HOME key.

Corrections

If you notice - before you press RETURN - that you have made a typing mistake , you can correct the error by using the cursor keys. For example, if you type:

```
PRONT "CBM 64"
```

and then realise your mistake, position the cursor over the offending O using the cursor keys, and type an I. You can now press RETURN - there's no need to move the cursor to the end of the line. This is a general rule for the 64; pressing RETURN will tell the computer to consider the line upon which the cursor rests as a command. This means you must be careful to delete any rubbish from the line before pressing RETURN.

To remove large amounts of text, use the INST/DEL key which will delete characters to the left of the cursor for as long as it is held down. To insert text in a line, hold down the SHIFT key and press the INST/DEL key - this will create a space for your additions.

Clearing the screen

This can be done in two ways. You can hold down the SHIFT key and press the CLR/HOME key.

Alternatively, hold down the RUN/STOP key and press the RESTORE key to clear the screen, return the screen/border colours to dark blue/light blue and display the READY message.

NUMBERS

As well as printing words, the Commodore 64 can also handle numbers. Try:

```
PRINT 5          (RETURN)
```

You can do arithmetic.

Addition:

```
PRINT 5+3
```

Subtraction:

```
PRINT 7-4
```

Multiplication:

```
PRINT 3*5
```

Division:

```
PRINT 15/6
```

Powers (exponentiation):

```
PRINT 3↑2
```

You can ask the 64 to calculate longer expressions, such as:

```
PRINT 3*5 + (5-3)/(2+1) * 7/8
```

In working out expressions like this, the computer follows strict rules about the order in which the various arithmetical operations are performed.

First The expressions inside brackets are evaluated.

Second Any powers (squares, cubes, etc.) indicated by ↑ are worked out.

Third The 64 performs the multiplications and divisions.

Fourth The additions and subtractions are performed.

If you are unsure, put brackets round the bit you want calculated first. Try the following examples, and see if you can work out the answers yourself to check that you know what the computer is doing.

```
PRINT 2 * 2 + 2 * 2
PRINT 2 + 2 * 2 + 2
PRINT 6 + 3 / 5 - 4
PRINT 3 ↑ 2 + 4 ↑ 3 / 2
PRINT (3 + 4) * 2
```

COLOUR

The colour of the screen display and its border can be changed to be any one of 16 colours, as can the colour of the characters displayed on the screen.

To change the colour of the characters, hold down the CTRL key and press one of the numeric keys - the cursor will change to the colour written on the front of the key. Now type some characters - these will appear in the same colour as the cursor. You can change the colour of the characters at any time

by holding down the CTRL key and pressing one of the number keys again.

The keys 1 to 8, in conjunction with the CTRL key will give eight colours. To obtain a further eight, hold down the CBM logo key (at the bottom left hand corner of the keyboard) and press one of the number keys. The colours obtained with the various keys are shown in Appendix 5.

The colour of the screen and border can be made to take on any one of these colours, by using a BASIC command. To see this happen, clear the screen (press SHIFT and CLR/HOME) and type:

```
POKE 53280,2
```
(and press RETURN)

If you typed correctly, the border will change from light blue to red. The **POKE** command will be explained more fully later on; for now it is enough to know that it puts a number (in this case 2) into a special area of memory which the 64 uses to determine the border colour. 2 is the code for red, so the border turns red.

You can also change the colour of the screen:

```
POKE 53281,7
```

This turns the screen yellow because 7, the code for yellow, has been placed in the area of memory which the 64 uses to determine screen colour. Don't worry if you can't follow what is happening - all will be made clear later, since we will make great use of the **POKE** command throughout this book.

RETURNING TO NORMAL

If at any time you want to return to the dark blue/light blue screen/border combination, simply hold down the RUN/STOP key and press the RESTORE key.

TEXT

There are two other things you can do which alter the appearance of the display.

When you first switch on and type, all the letters on the screen are displayed as capitals, or upper case. The 64 can also display text in lower case.

```
PRINT "THIS IS A TEST"
```

will print the message 'THIS IS A TEST'. Now, hold down the CBM logo key and press the SHIFT key - all the letters on the screen turn into lower case, as do any subsequently typed letters. Thus:

```
print "this is a test"
```

still works because the 64 understands both types of text. However,

```
PRInt "STILL TESTING"
```

will not work - instructions to the 64 must all be in the same case.

REVERSE MODE

Up to now, all our characters have been in light blue on a dark blue background. The 64 allows you to display reversed text. To try this hold down the CTRL key and press the 9 key (note that the words RVS ON are written on the front of the key). From now on, anything you type on that line appears reversed, with dark characters on a light background. To turn this off, hold down CTRL and press the 0 key (RVS OFF) or press RETURN, since reverse video is active only for one line at a time.

GRAPHICS CHARACTERS

You will notice that displayed on the front of many of the keys are two symbols or graphic characters. These can also be displayed on the screen by holding down the CBM key or the SHIFT key together with the appropriate key. Pressing a key in conjunction with the CBM key will display the left hand graphics character; using the SHIFT key instead selects the right hand character.

MULTIPLE COMMANDS

You can put more than one instruction in one command if you separate the instructions with colons (:). Try this: (see also the note which follows the examples)

```
PRINT "{CLS}":PRINT "3+4="3+4
:PRINT
```

Or this:

```
PRINT "{CLS}":POKE 53280,8:POKE
53281,5 :PRINT "{GRN}LURID COLOUR
SCHEME!!"
```

It doesn't matter if you run over the end of the line on the screen, as long as there are no more than two screen lines of characters (including the spaces) in the command.

Note

In the two examples above, the characters within the 'curly' brackets { } indicate which key to press when typing in the listing - they should **not** be typed, for example:

```
PRINT "{CLS}"
```

means type PRINT , followed by a quotation mark, then hold down the SHIFT key and press CLR/HOME, followed by a further quotation mark. The screen will display a reverse heart character within the quotation marks, which tells the 64 to clear the screen. Similarly, "{GRN}" means hold down the SHIFT key and press 6, to set Green.

This method of indicating certain key presses is used throughout this book and a full list of the abbreviations is given in Appendix 1, though most should be obvious.

Entering a number of instructions together like this can get very cumbersome. In the next chapter we will discover a way of giving the computer a very large number of instructions all at once - the **PROGRAM**.

SUMMARY

PRINT instructs the 64 to print something on the screen.

```
PRINT "ABCDEFG"
```

prints the characters between the quotation marks.

```
PRINT 3+4
```

prints the result of the sum.

These can be combined:

```
PRINT "3+4*5+7 = " 3+4*5+7
```

COLOURS The colour of the screen, its border and the characters on the screen can be set to any one of 16 colours.

MULTIPLE COMMANDS A number of separate instructions may be included on one line but must be separated by colons (:).

CHAPTER 4

PROGRAMMING

At the end of Chapter 3, we saw that it is possible to give the computer a number of instructions at one time by writing them one after another on one line. We will now look at a much more powerful way of doing this - using a **program.**

A computer program is a numbered list of computer instructions which are entered together and stored by the computer to be obeyed later. Let's look at the last example of Chapter 3 and see how we can turn it into a program. In Chapter 3 we had:

```
PRINT"{CLS}":POKE 53280,8:POKE
53281,5:PRINT"{GRN}LURID COLOUR
SCHEME"
```

Rewritten as a program it looks like this:

```
10    PRINT "{CLS}"
20    POKE 53280,8
30    POKE 53281,5
40    PRINT"{GRN}LURID COLOUR
      SCHEME!"
```

To enter the program into the computer, type each line in turn (remembering to press RETURN after each). You will see that the 64 does not obey the instructions as they are entered, and that the 'READY' message does not appear after you press RETURN. The computer recognises that each line

is part of a program, because each begins with a number, and the program is stored for future use.

When you have typed in all the lines, you can inspect the stored program by typing **LIST** (and pressing RETURN, of course). The instructions are listed in numerical order on the screen. You can list parts of the program by specifying line numbers. For example:

`LIST 30-50`	lists all lines from 30 to 50
`LIST 20`	lists only line 20
`LIST-30`	lists all lines up to and including line 30
`LIST 30-`	lists all lines from line 30 to the end of the program

If a program is very long, it will occupy more lines than there is space for on one screen. So the screen scrolls to display the program and this can be slowed by holding down the CTRL key during **LIST**ing.

To order the computer to act on the program you use another BASIC command: **RUN**. Type this in (followed by RETURN). This tells the computer to read through the stored program and obey each instruction in order. You will see that the program gives exactly the same results as the multiple instructions in Chapter 3. (If it doesn't, and the 64 prints a message such as 'SYNTAX ERROR IN LINE 20', there is a typing mistake in that line and the 64 cannot understand the instruction. **LIST** the line, and correct the mistake.)

A program may be **RUN** as many times as you like. It will be stored in the computer's memory until you switch it off.

It is possible to begin execution of a program from a line other than the first by specifying the line number. For example **RUN** 20 begins executing the program at line 20, and line 10, the instruction which clears the screen, is ignored.

Programs can be altered or added to at will. To alter an instruction, retype the line. If you enter:

```
30    POKE 53281,2
```

the original line 30 is replaced by the new one, and the next time you **RUN** the program the screen will turn red instead of green. (To return to the original screen colours, hold down the RUN/STOP key and press RESTORE.)

Extra lines are added by typing them as before:

```
35    PRINT "AN EXTRA LINE"
```

Notice that the computer inserts the extra line in the appropriate place: it is **not** put at the end.

NOTE It is a good idea always to number program lines in steps of 10 to allow extra or forgotten lines to be inserted without having to renumber the whole program.

To delete a line, type the line number and press RETURN. The line is deleted from the list in the computer's memory.

Now see if you can alter the example program to **PRINT** different words on the screen and in different colours.

When you have exhausted the possibilities of this program, you can use the command **NEW** to delete the whole program. You will then be ready for the next program.

CLOSE ENCOUNTERS

Here is a more interesting program for you to try - it may also be useful if you're in the habit of attempting to communicate with aliens! Don't worry if you don't understand all the instructions - you will do when you've read the rest of the book- just type it in, **RUN** it and wait for a 'Close Encounter'!

```
10    W=850:DIM NO(4,1)
20    DEF FNA(X)=INT(RND(1)*16)
30    FOR Z=0 TO 4
40    FOR J=0 TO 1
50    READ NO(Z,J)
60    NEXT J,Z
70    PRINT CHR$(147)
100   POKE 54296,15
110   POKE 54278,83
120   POKE 54277,245
130   POKE 54276,33
200   FOR L=0 TO 15
210   GOSUB 500
220   FOR P=0 TO 200:NEXT P
230   POKE 54276,32
240   FOR P=0 TO 400:NEXT P
250   POKE 54276,33
260   W=W-50
270   NEXT L
300   GOSUB 500:GOTO 300
500   FOR Z=0 TO 4
510   POKE 54273,NO(Z,0)
```

```
520   POKE 54272,NO(Z,1)
530   B=FNA(X)
540   S=FNA(X)
550   POKE 53280,B
560   POKE 53281,S
570   FOR P=0 TO W:NEXT P
580   NEXT Z
590   RETURN
600   DATA 18,209,21,31,16
610   DATA195,8,97,12,143
```

Check the program carefully for typing errors, and then **RUN** it. If the program stops and the message '? SYNTAX ERROR IN LINE xxx' is printed, it means you have made a typing mistake in that line. If any other error message is printed, check the whole of the program for mistakes.

If you do not succeed in making contact with any aliens, you may stop the program by holding down the RESTORE key and pressing the RUN/STOP key.

ARITHMETIC IN PROGRAMS - VARIABLES

We discovered in Chapter 3 that the 64 will print the answers to arithmetical problems in response to commands such as:

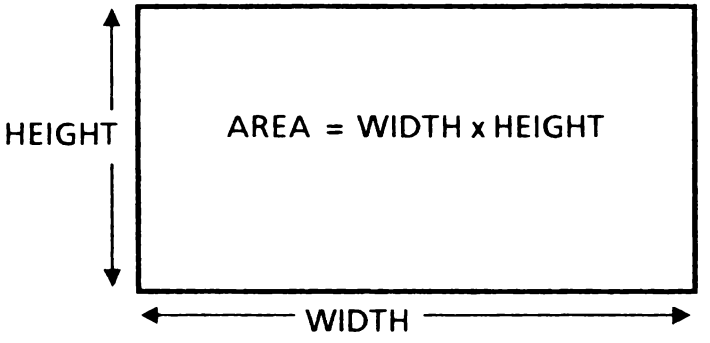
```
PRINT 3+5
```

This sort of instruction can be included in a program like this:

```
10   PRINT"{CLS}"
20   PRINT "3+5 = " 3+5
```

but it would be difficult to make much use of this for working out your personal finances.

What gives a computer the power to perform complex data processing (or 'number crunching') tasks is its ability to do algebra: to use names as symbols for numbers. This means that programs can be written with names to symbolise numbers, and then **RUN** with different numbers attached to the names. These number-names are called VARIABLES because they represent varying numbers. Let's try an example.



The area of a rectangle is equal to the width of the rectangle multiplied by the height. By using names instead of numbers we can write a program to work out the area of any rectangle. For example:

```
10  WIDTH = 8
20  HEIGHT = 12
30  AREA = WIDTH * HEIGHT
40  PRINT "AREA = " AREA
```

This program, when **RUN**, will print the area of a rectangle 12 inches by 8 inches. By changing the numbers in lines 10 and 20 we can obtain the area of a rectangle of any size.

VARIABLE NAMES

Any number of letters can be used in a variable name, but only the first two characters of the name are considered by the 64; the rest are ignored. This means, for example, that if you use these names:

FR

FRUIT

FRIEDEGG

the computer will treat them as the same variable, FR.

All variable names must begin with a letter, but the other characters may be numbers, for example A1, E7, L1. There must not be any spaces in a variables name: this will confuse the computer.

NOTE You cannot use as a variable name any of the BASIC command words, or any name which includes a command word. If you do, your program will not run, but will give SYNTAX ERROR messages whenever the name is used. For example, LENGTH contains **LEN**, and BREADTH contains **READ**, both of which are BASIC words. Similarly, you cannot use the BASIC reserved variables TI, TI$ and ST, whch are reserved for use by the 64's internal operations. (See Appendix 2 for a full list of all the BASIC reserved words.)

TYPES OF VARIABLE

There are three different types of variable, two of which represent numbers, the third representing words. The types which represent numbers are Real and Integer variables; the type representing words are called String variables because they contain sequences or 'strings' of characters. The variables in the previous program are real variables.

REAL VARIABLES

These are used to represent numbers and can have any value, whole numbers or fractions. Real variables should be used in all arithmetical programs. Real variables can have any value. For example:

```
REAL = 3.72
SIZE = 87.3 * 2.5
```

INTEGER VARIABLES

These can only represent whole numbers (or integers), not fractions. They may be used when counting events or objects, or to store constant integer values. For example, a program which kept a record of the number of chocolate bars in stock at a shop could use integer variables to count them, but the takings of the shop would have to be stored in real variables, or fractions of pounds (i.e. pennies) would be ignored.

Integer variables must be identified by the % sign after the variable name, as in CHOCBARS%. Here are some examples:

```
NUMBER%    = 3+6
COUNT%     = COUNT% + 1
CHOCBARS%  = 1000000
```

If you try to give an integer variable a fractional value only the whole number part is stored. So

```
10  NUMBER% = 7.5
20  PRINT NUMBER%
```

will print the value 7.

STRING VARIABLES

A string variable is used to store words or letters or numbers. The variable name for a string variable must be followed by a dollar sign ($) to distinguish it from the other types of variable. Examples are:

```
NAME$ = "WINSTON CHURCHILL"

STRING$ = "ABCDEFG"
```

Note that the letters defining the variable contents are enclosed by quotation marks.

The maximum number of characters a string variable can hold is 255. If you try to make a string longer than that the 64 will display the message 'STRING TOO LONG ERROR'.

NOTE It is possible to have variables of different types with the same name, such as:

```
NAME   = 87.76
NAME%  = 3
NAME$  = "GEORGE WASHINGTON"
```

The computer will not be confused, but you might, so be careful!

ARRAYS

Each of the three types of BASIC variable - real, integer and string - may be used in the form of what is called an Array . In normal use a variable name represents one stored number or string. In an array, the variable name represents a collection of stored information, with one or more reference numbers identifying the individual items. An array can be pictured as a collection of boxes. Here is a diagram of an array which uses one reference number.

REFERENCE NUMBER	CONTENTS
0	LUCY
1	JULIE
2	SHEILA
3	NICK
4	CAROL
5	SUE

Array NAME$

The array is a string array and has 6 elements numbered from 0 to 5. The array would be used like this:

```
10    NAME$(0) = "LUCY"
20    NAME$(1) = "JULIE"
30    NAME$(2) = "SHEILA"
```

```
40    NAME$(3) = "NICK"
50    NAME$(4) = "CAROL"
60    NAME$(5) = "SUE"
100   FOR N = 0 TO 5
110   PRINT N, NAME$(N)
120   NEXT N
```

As mentioned before, an array may have more than one reference number. The number of reference numbers per item is called the number of **dimensions** of the array. Here is a program using an array with two reference numbers per item:

```
10    DIM P$(2,2)
20    P$(0,0) = "FLORENCE"
30    P$(0,1) = "NIGHTINGALE"
40    P$(1,0) = "GEORGE"
50    P$(1,1) = "WASHINGTON"
60    P$(2,0) = "ISAAC"
70    P$(2,1) = "NEWTON"
100   FOR A = 0 TO 2
110   FOR B = 0 TO 1
120   PRINT P$(A,B); "   ";
130   NEXT B
140   PRINT
150   NEXT A
```

Here again the array can be thought of as a collection of storage boxes, this time laid out in a grid as shown opposite.

You can use arrays with more than two dimensions, but you will find the 64's memory will not hold arrays of more than about four dimensions (the exact number depends on the number of elements in each dimension). The maximum allowed value of each reference number may be anything up to several thousand, but this depends

	B = 0	B = 1
A = 0	FLORENCE	NIGHTINGALE
A = 1	GEORGE	WASHINGTON
A = 2	ISAAC	NEWTON

Array P$

on how much memory is occupied by the program itself, and by the other variables of the program.

A **DIM** command must be included at the beginning of any program which will use arrays of more than one dimension, or with more than eleven elements (0-10). The command tells the computer to reserve memory space for the array. You do not need a **DIM** command if you are using one-dimensional arrays with no more than eleven elements numbered from 0 to 10, as the 64 is able to handle these automatically.

GETTING VALUES INTO PROGRAMS

It would be very inconvenient to have to alter a program to make it handle different numbers, so there are a number of instructions which allow numbers or letters to be given to a program while it is running. The first of these is **INPUT**.

When the computer finds an **INPUT** statement in a program, a question mark is displayed on the screen. The computer waits for you to type in a number or letter string, which it will then store as a variable before continuing with the rest of the program. Type **NEW** (RETURN) to delete any

program currently in the computer's memory and then try this example:

```
10  INPUT NUMBER
20  PRINT NUMBER
```

When you **RUN** this program you will see a question mark on the screen. Type a number and press RETURN: the number is printed.

NOTE If you typed anything other than a number the computer would print the message 'REDO FROM START'. It would then display another question mark and wait for you to type the number in again. This is because the variable NUMBER is a real variable and can only store numbers. If you press RETURN without typing anything, the variable is unchanged, so in this case a zero is displayed.

The **INPUT** instruction can handle strings too, if you specify a string variable in the **INPUT** command:

```
10  INPUT NAME$
20  PRINT NAME$
```

This program will accept both numbers and letters and store them as a string variable. Pressing RETURN enters a null string but this time nothing is displayed.

Messages can be printed before the question mark to tell the person using the program what to type in. Change line 10 to:

```
10  INPUT "TYPE A WORD"; NAME$
```

and **RUN** the program again. You can use any message you like, but it must be contained within quotation marks, and there must always be a semicolon between the message and the variable which will store the number.

It is possible to use one **INPUT** command to input two or more numbers or strings.

```
10  INPUT "NAME, AGE";NAME$, AGE
20  PRINT NAME$
30  PRINT AGE
```

The variables must have commas separating them in the **INPUT** command; and when you type in the information you must use a comma to separate the items. If you press RETURN before entering all the items the 64 will print two question marks and wait for the remaining items. If you enter too many items the surplus ones will be ignored and the 64 will print '?EXTRA IGNORED'.

Let's use what we know about variables and **INPUT** to improve the area program. Clear the computer's memory by typing **NEW** (RETURN), and then type in this program.

```
10    REM  IMPROVED AREA PROGRAM
20    PRINT "{CLS}"
30    INPUT "ENTER WIDTH"; WIDTH
40    INPUT "ENTER HEIGHT"; HEIGHT
50    AREA = WIDTH * HEIGHT
60    PRINT: PRINT "AREA = " AREA
```

This program will read the two numbers you type in for width and height and print the area of the rectangle. Using **INPUT** has made the program much more flexible: we don't have to alter the program to use different numbers.

REMARKS

Line 10 of the last program is a **remark** or **comment** statement. These are used to hold comments, notes and titles to make the purpose of a program and the way in which it works clear to someone reading the listing. Remarks are identified by the word **REM** before the remark and have no effect when the program is **RUN**.

You should always put plenty of remarks in programs, because although you may understand how a program works when you write it, three months later you may have forgotten. If you need to modify a program at some time after writing it, remarks will make it much easier to remember how the program works.

SUMMARY

PROGRAMS

A **program** is a numbered sequence of computer instructions.

LIST displays the lines of a program.

RUN starts the execution of a program.

NEW deletes a program from the computer's memory.

VARIABLES

Variables represent numbers and words in programs.

There are three types of variable:

Real representing any numbers.

Integer representing whole numbers, and distinguished by % at the end of the variable name.

String representing words or numbers, and distinguished by $ after the name.

INPUT is used to enter numbers or words into variables while a program is running.

Variables may be collected in arrays, with the same variable name, but with distinguishing reference numbers.

REMARKS

REM is used to put remarks into programs for the programmer's benefit.

CHAPTER 5

PROGRAM CONTROL

In the last chapter we defined a program as a numbered list of instructions to the computer, which are obeyed in order from beginning to end. In this chapter we will find out how to write programs in which some instructions are executed more than once. This makes programs more efficient, as instructions which must be repeated need to be written only once. We will also discover that the computer can make decisions.

REPETITION

A section of program can be repeated many times using the instructions **FOR** and **NEXT**. Try this example:

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

which prints the numbers from 1 to 10 on the screen. The **FOR ... NEXT** instructions work like this:

FOR ... NEXT LOOPS

The section of program between the **FOR** and the **NEXT** commands is repeated for each value of COUNT, and COUNT is automatically increased by one every time the **NEXT** command is met.

The variable in a **FOR ... NEXT** loop need not be increased by one; any increase or decrease can be specified using the command **STEP**. Try changing line 10 to:

```
10    FOR COUNT = 0 TO 30 STEP 3
```

Then **RUN** the program again. COUNT is now increased by 3 each time the loop is repeated. The **STEP** can also be negative, so that the numbers get smaller:

```
10    FOR COUNT = 50 TO 0 STEP -2.5
```

It is not necessary to specify the variable after **NEXT**;

```
30    NEXT
```

would work just as well. However when you write longer programs you will find that they are clearer if the variables are always specified. The 64 will not be confused, but you will be as you write the program!

To recap, the instructions are used like this:

```
FOR V = X TO Y STEP Z
```

is used to begin a loop (V is a variable and X, Y and Z may be variables or numbers), and

```
NEXT V
```

marks the end of the loop.

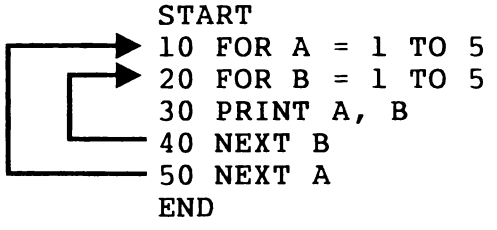
The variables used in **FOR ... NEXT** loops may be real or integer variables, but **not** string variables, and **not** elements of arrays.

NESTED LOOPS

FOR ... NEXT loops can be **nested** - that is, one loop can be contain one or more other loops. Here is an example :

```
10    FOR A = 1 TO 3
20    FOR B = 1 TO 4
30    PRINT A, B
40    NEXT B
50    NEXT A
```

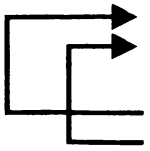
For any nested loops, it is **very important** that the enclosed loop is **completely within** the outer loop, otherwise the program **will not work.** To clarify this, let's examine the order in which instructions are obeyed in the program.



The two loops are arranged so that one is completely within the other. If lines 40 and 50 were swapped over, the result would be as shown opposite.

You can see that the two loops overlap and the order in which the instructions should be obeyed is not clear.

This rule always applies to **FOR ... NEXT** loops - the loops must **always** be one within the next.



```
START
10 FOR A = 1 TO 5
20 FOR B = 1 TO 5
30 PRINT A, B
40 NEXT A
50 NEXT B
END
```

Nine is the maximum possible number of **FOR ... NEXT** loops which can be nested one within another. If you write a program with too many loops nested within each other, when it is **RUN** the 64 will print the message 'OUT OF MEMORY ERROR'.

NOTE Where two or more nested **FOR ... NEXT** loops have their **NEXT** commands one after the other they may be combined by specifying both or all the variables after one **NEXT**. The previous program would therefore end:

```
40  NEXT B, A
```

Again the order of the variables in line 40 is the reverse of the order in which the corresponding **FOR** commands occur.

JUMPS

The computer can be instructed to jump from one program line to another using the command **GOTO**. Enter this program

```
10    GOTO 30
20    PRINT "LINE 20"
```

```
30    PRINT "LINE 30"
```

When it is **RUN** the message 'LINE 20' is not seen because the **GOTO** instruction in line 10 sends the 64 straight to line 30. (Note that **GOTO** can be typed with or without a space - **GO TO** works equally well.)

The jump can be to a lower numbered line. The following program will go on for ever, unless you stop it by pressing the RUN/STOP key:

```
10    PRINT "ON AND ";
20    GOTO 10
```

This is called an endless loop, and is to be avoided!

PAUSES

There are two ways to instruct the computer to pause during the **RUN**ning of a program.

For pauses, the duration of which is not critical, we can use a **FOR ... NEXT** loop which does nothing at all except count its way through the steps. The following program is an example :

```
10    FOR I= 0 TO 5000
20    NEXT I
30    PRINT "ABOUT 5 SECONDS"
```

This technique is fine for crude pauses, but for accurate timing, it is better to employ a second technique.

THE SYSTEM CLOCK

Whenever you switch on your 64, a built in clock starts running, and will continue to run until you switch off the machine. The clock continually

updates the variable TI every 60th of a second, as the following program demonstrates.

```
10    PRINT "{CLS}"
20    PRINT "{HOM}" TI
30    GOTO 20
```

Since TI is updated every 60th of a second, one minute has elapsed when the value of TI has increased by 3600. We can use this as a timer for precise pauses within a program, for example :

```
10    INPUT "DELAY IN SECONDS";D
20    T = TI
30    IF TI>= T+(D*60) THEN GOTO 50
40    GOTO 30
50    PRINT D " SEC DELAY!!!"
```

An easier way of using the system clock is to use another system variable TI$. TI$ is a six character string which indicates time since the computer was switched on in hours, minutes and seconds. The first two digits are hours, the middle two minutes, and the right hand pair seconds.

TI$ can also be used to reset the system clock, with a statement like :

```
TI$ = "000000"
```

We can see TI$ in operation in this short program:

```
10    PRINT "{CLS}"
20    TI$ = "000000"
30    PRINT "{HOM}" TI$
40    GOTO 30
```

Here is a longer program which makes use of the system clock to simulate a digital alarm clock:

```
10    PRINT "{RED}":POKE 53281,0
```

```
15    GOSUB 700
20    PRINT"{CLS}"
30    PRINT"{HOM}"
40    PRINT TAB(16)LEFT$(TI$,2)"."
      MID$(TI$,3,2)"."RIGHT$(TI$,2)
50    IF AS=0 THEN 100
60    IF TI$=AT$ THEN POKE 53280,2:
      POKE 54296,15
100   GET K$:IF K$="" THEN 30
110   IF K$="{F1}" THEN 200
120   IF K$="{F3}" THEN 300
130   IF K$="{F5}" THEN 400
140   IF K$="{F7}" THEN 500
150   GOTO 30
193   REM
194   REM ************
195   REM *          *
196   REM * SET TIME *
197   REM *          *
198   REM ************
199   REM
200   INPUT "{CLS}{CD}{CD}{CD}{CD}
      {CD}  CURRENT TIME";CT$
210   IF CT$="" THEN 20
220   A$=CT$:GOSUB 1000
230   IF OK=1 THEN TI$=CT$:GOTO 20
240   GOTO 200
293   REM
294   REM *************
295   REM *           *
296   REM * SET ALARM *
297   REM *           *
298   REM *************
299   REM
300   INPUT "{CLS}{CD}{CD}{CD}{CD}
      {CD}  ALARM TIME";AT$
310   IF AT$="" THEN 20
320   A$=AT$:GOSUB 1000
330   IF OK=1 THEN AS=1:GOTO 20
340   GOTO 300
393   REM
```

```
394   REM **************
395   REM *              *
396   REM * ALARM TIME *
397   REM *              *
398   REM **************
399   REM
400   IF AS=0 THEN 30
410   IF AF=1 THEN 430
420   PRINT"{HOM}{CD}{CD}"TAB(28)
      LEFT$(AT$,2)"."MID$(AT$,3,2)
      "."RIGHT$(AT$,2):GOTO 440
430   PRINT "{HOM}{CD}{CD}"TAB(28)
      "           ":REM 10 SPACES
440   AF=1-AF
450   GOTO 30
493   REM
494   REM *************
495   REM *             *
496   REM * ALARM OFF *
497   REM *             *
498   REM *************
499   REM
500   POKE 53280,14:POKE 54296,0
510   AS=0
520   GOTO 430
693   REM
694   REM ***************
695   REM *               *
696   REM * ALARM NOISE *
697   REM *               *
698   REM ***************
699   REM
700   POKE 54278,83
710   POKE 54277,245
720   POKE 54276,33
730   POKE 54273,18
740   POKE 54272,209
750   RETURN
993   REM
994   REM ***************
995   REM *               *
```

```
996   REM * CHECK ENTRY *
997   REM *             *
998   REM ***************
999   REM
1000  OK=0
1010  IF VAL(LEFT$(A$,2))<24 THEN
      1030
1020  GOTO 1060
1030  IF VAL(MID$(A$,3,2))<60 THEN
      1050
1040  GOTO 1060
1050  IF VAL(RIGHT$(A$,2))<60 THEN
      OK=1
1060  RETURN
```

Program Description

Line 10 sets the foreground colour to red and the screen colour to black.

Line 40 prints the current value of TI$ as hours, minutes and seconds, separated by a full stop.

Lines 100 to 150 control the mode of the clock:

F1 allows you to set the current time using lines 200 to 240.

F3 allows you to set the alarm time using lines 300 to 340.

F5 displays the alarm time if it is set, or, if it is already displayed, cancels the display.

F7 turns the alarm off in lines 500 to 530.

The program loops continuosly through the first section checking for key presses and comparing the current time with the set alarm time. If a key is pressed, the program jumps to the appropriate section. If the alarm is set and the current time

matches the alarm time, the screen border turns red and a noise (set up in lines 700 to 750) is switched on.

When an entry is made to set the time, it is checked for validity by the subroutine at line 1000, which indicates to the calling program that the entered time conforms to the 24 hour clock format, by setting the variable OK to 1.

DECISIONS

The commands **IF** and **THEN** are used in programs to make decisions. A variable is tested, and one of two alternative actions is taken, depending on the result of the test.

```
10    FOR X = 1 TO 10
20    PRINT X
30    IF X = 5 THEN PRINT "X = 5"
40    FOR P = 0 TO 2000 : NEXT P
50    NEXT X
```

The format of **IF ... THEN** is:

IF (condition) THEN (instructions)

The condition after **IF** may be one of many possible alternatives. Example are :

```
IF COUNT = 10
```

Continue when variable (COUNT) equals a number, in this case 10.

```
IF COUNT < 100
```

Continue when variable less than a number.

```
IF COUNT > NUMBER
```

Continue when variable greater than a number.

```
IF NUMBER < > VALUE
```

One variable not equal to another (greater than or less than).

```
IF X  >= Y
```

Greater than or equal to.

```
IF X  <= Y
```

Less than or equal to.

Two conditions may be combined, as in:

```
IF X = 1 OR X = 2

IF A = 3 AND NAME$ = "CBM 64"
```

In all of these, the items being compared may both be variables, or one may be a variable and the other a number. (There's not much point in comparing two numbers!) For further details see Chapter 9, Logical Thinking. The instructions after **THEN** are carried out if the condition is met, otherwise the program continues at the next line.

IF ... THEN can also be used to control jumps:

```
10    INPUT "WHAT NUMBER AM I
      THINKING OF"; N
20    IF N < > 3 THEN PRINT
      "WRONG":GOTO 10
30    PRINT "CORRECT!"
```

If the jump is the only instruction after **THEN**, the word **GOTO** can be omitted:

```
10   INPUT "WHAT AM I"; A$
```

```
20    IF A$ = "CBM 64" THEN 60
30    PRINT: PRINT A$"?  NO, TRY
      AGAIN"
40    PRINT
50    GOTO 10
60    PRINT: PRINT"GOOD GUESS!"
```

EXAMPLE PROGRAM - SORTING NUMBERS

As an example of what can be done using loops and decisions, here is a program which sorts ten numbers into ascending order.

```
10    REM   SORTING PROGRAM
20    DIM NUM(10),S(10)
30    REM INPUT 10 NUMBERS
40    PRINT "{CLS}"
50    FOR N=1 TO 10
60    INPUT"ENTER A NUMBER";NUM(N)
70    NEXT N
100   REM COPY NUMBERS TO ARRAY S
110   FOR C=1 TO 10
120   S(C) = NUM(C)
130   NEXT C
200   REM SORT NUMBERS
210   COUNT = 0
220   FOR N=1 TO 9
230   IF S(N+1) >= S(N) THEN 280
240   TEMP = S(N+1)
250   S(N+1) = S(N)
260   S(N) = TEMP
270   COUNT = COUNT + 1
280   NEXT N
300   IF COUNT <> 0 THEN 210
310   FOR Z=1 TO 10
320   PRINT NUM(Z),S(Z)
330   NEXT
```

The **REM**arks are used to indicate the functions of the different sections of the program, which works like this:

Line 20 dimensions the two arrays used in the program. (This is not strictly necessary for ten numbers, but it would be if any more were to be sorted.)

Lines 40 to 70 input ten numbers from the keyboard and store them in the array NUM( ), using a **FOR ... NEXT** loop.

Lines 110 to 130 copy the numbers from NUM( ) to a second array S( ) which will be sorted, while NUM retains the numbers in the order in which they were typed in. Again, a **FOR ... NEXT** loop is used.

The lines from 210 to 280 sort the numbers in the array S(10) into ascending order. A **FOR ... NEXT** loop compares each element of the array S with the next, and swaps them over if they are in the wrong order. A variable, COUNT, keeps track of the number of these swaps and, if at the end of the loop this isn't zero, the loop is repeated.

Lines 310 to 330 complete the program by **PRINT**ing the two sets of numbers.

SUBROUTINES

A subroutine is a section of program which is executed at a number of different times in the course of a program, but is written only once. The computer is diverted from the main program to the subroutine, and returns to the main program at the point from which it left.

The command **GOSUB** followed by a line number or a variable diverts the computer to the

subroutine in much the same way as the **GOTO** command. The difference is that the end of a subroutine is marked by a **RETURN** command which sends the computer back to the instruction after the **GOSUB** command. As an example of two very simple subroutines, type in the following program:

```
10   GOSUB 110
20   C = A+B
30   GOSUB 210
40   END
100  REM SUBROUTINE TO INPUT A & B
110  INPUT "A, B"; A, B
120  RETURN
200  REM SUBROUTINE TO DISPLAY
     RESULT
210  PRINT "{CLS}"
220  PRINT
230  PRINT A "+" B " = " C
240  RETURN
```

Lines 10 to 40 are the main program. Line 10 calls the subroutine at line 110, line 20 adds A and B and stores the result as C, and line 30 calls the subroutine at line 210. Line 40 marks the end of the program - as the last line to be executed is not the last line of the program we have to tell the 64 not to go on to the next lines, which contain the subroutines.

The subroutine at line 110 inputs two numbers and stores them as A and B. The subroutine at line 210 clears the screen and prints the two numbers and their sum.

When the program is **RUN**, the computer begins, as always, at the lowest numbered line, line 10. This line calls the first subroutine, and the

computer is diverted to line 110. Line 120 returns the computer to the instruction after the **GOSUB**, which is in line 20. The program proceeds as normal to line 30, which causes another diversion to line 210. Line 240 returns the computer to line 40, and the program ends.

So, a **GOSUB** command causes a diversion from the sequence of a program to a subroutine, and the **RETURN** command ends the diversion.

The use of subroutines saves a lot of effort in writing programs, as the program lines in the subroutine need to be written only once, instead of being retyped at every point in the program where they are needed. There is no limit to the number of times a subroutine may be called.

Another advantage of subroutines is that they can make the design of a program simpler. If you use subroutines for the repetitive and less important parts of a program the main program becomes much easier for you, the programmer, to follow, and is therefore much easier to write.

Subroutines, like loops, may be nested - that is, the subroutines may call other subroutines, which may in turn call others. A maximum of twenty-four subroutines may be nested like this. However a subroutine may not be called by another subroutine which has been called by the first one, or an endless loop occurs, and the program will crash when the 64 runs out of the memory it uses to store all the line numbers for the **RETURN**s.

STOPPING AND STARTING

In the last example program the new command **END** was used to indicate the last line of the program to be executed. The command tells the computer to stop running the program. There is a

second command, **STOP**, which also halts programs, but the two have slightly different effects. **END** stops the program running, and the 'Ready' message is displayed, but when **STOP** is used the program halts and the message 'BREAK IN LINE xxx' is printed (xxx is the line number of the **STOP** instruction). Unless you need to know where the program halted, **END** is the one to use.

If a program has been stopped by one of these instructions, or by holding down CTRL and pressing C, it may be restarted with the instruction **CONT**. The program will continue from the instruction after the last one to have been executed. **CONT** may only be used immediately after the program has been stopped: if you have altered the program the message '?CAN'T CONTINUE ERROR' will be displayed. You may not use **CONT** within programs, as this will give a '?SYNTAX ERROR'.

ON ... GOSUB (GOTO)

Both **GOSUB** and **GOTO** may be used in a second type of decision command which selects one of a number of destinations depending on the value of a chosen variable.

```
ON N GOTO 100, 200, 300
```

results in

```
GOTO 100    if N = 1

GOTO 200    if N = 2

GOTO 300    if N = 3
```

Similarly:

```
ON P GOSUB 500,200,500,400,100
```

selects the P'th destination in the list and calls it as a subroutine. If the value of the variable is greater than the number of destinations in the list, or if it is zero or less, no destination is selected and the program continues at the next instruction.

SUMMARY

LOOPS

The loop structure has the form:

> **FOR V = A TO B STEP C ... NEXT** in which the instructions between the **FOR** and **NEXT** commands are repeated once for each value of V indicated by the **FOR ... STEP** command. A, B and C may be variables or numbers; V must be a variable. Loops may be nested one within another.

DECISIONS

IF (condition) THEN (instruction) decides between two actions. The condition after **IF** is usually a test of a variable. There may be more than one instruction after **THEN**, in which case colons (:) must be used to separate them. The complete **IF ... THEN** command must be in one line.

GOTO need not be included if it would be the only command after **THEN**, but it must be if there is another command before it:

```
        100 IF ACE = 1 THEN 20

but     100 IF ACE = 2 THEN KING =
            7:GOTO20
```

SUBROUTINES

Diversions from a program, using **GOSUB** and **RETURN**. **GOSUB** calls the subroutine, **RETURN** at the end of the subroutine returns the computer to the instruction after the **GOSUB**.

ON ... GOSUB ON ... GOTO

Selects a destination from a list:

```
ON N GOTO  A, B, C
```

This causes a jump to the N'th destination - if there is an N'th item in the list.

```
ON P GOSUB X, Y, Z
```

causes a **GOSUB** to the P'th item in the list.

END halts the running of a program.

STOP halts a program and prints 'BREAK IN LINE xxx'.

CONT will restart a program if the program has not been altered.

CHAPTER 6

DATA AND PROGRAMS

So far we have not examined all of the ways in which information can be given to programs, nor by which programs can print out information on to the screen. There are several ways of getting information into programs which we will examine in this chapter, and we will also examine in more detail the use of **PRINT** and other commands to display data.

There is a BASIC command, **GET** which reads a single character from the keyboard.

GET

The **GET** command can be used in two ways:

Firstly, it can be used to check which key, if any, is being pressed as the program runs through, as in this example:

```
10    GET A$
20    PRINT A$
30    GOTO 10
```

Note that if no key is pressed, a blank line is printed and the program loops round to check again for a key press. The program will continue to loop in this way until a key is being pressed at the same time as line 10 is being executed.

The **GET** command can also be used to halt a program in order to wait for a key to be pressed. The character of the key which is then pressed is stored as a specified variable. We can demonstrate this with a modified version of the last program:

```
10    GET A$
15    IF A$ = "" THEN 10
20    PRINT A$
30    GOTO 10
```

The extra instruction in line 15 causes the program to loop between lines 10 and 15 until a key is pressed. When a key is pressed the character of that key is assigned to the variable A$.

If you change the variable in line 10 from a string variable to a numeric variable, then only the number keys, and the keys used in relation to numbers (. + and -) may be pressed. Pressing any other key will cause the computer to halt the program and print a 'SYNTAX ERROR' message.

The **GET** command is useful in programs which ask the operator to select an option, or answer Yes or No:

```
10    PRINT"PRESS C TO CLEAR SCREEN"
20    GET YN$ : IF YN$ = "" THEN 20
30    IF YN$ = "C" THEN PRINT
      "{CLS}":END
40    GOTO 10
```

READ AND DATA

To enter large amounts of data which will be the same each time the program is **RUN** we use the final type of data entry command, **READ**. Look at this program:

```
10   DIM MTH$(12):  DIM DAYS(12)
20   FOR M = 1 TO 12
30   READ MTH$(M)
40   READ DAYS(M)
50   PRINT MTH$(M)" HAS "DAYS(M)"
     DAYS"
60   NEXT M
1000 DATA  JANUARY,31, FEBRUARY,28
1010 DATA  MARCH,31, APRIL,30
1020 DATA  MAY,31, JUNE,30
1030 DATA  JULY,31, AUGUST,31
1040 DATA  SEPTEMBER,30 ,OCTOBER,31
1050 DATA  NOVEMBER,30, DECEMBER,31
```

The information for this program is written into the program in lines 1000 to 1050. Each item of **DATA** is separated from the next by a comma (or the end of a line). The information is copied into the arrays MTH$ and DAYS by the **READ** commands in line 30 and 40.

This method of entering information is much easier to use than simply writing it into the program by setting variables. Imagine the typing involved in entering lots of lines like

```
MTH$(2)="FEBRUARY"
```

With **DATA** lines you can lay out the information in a clear tabular form and check it much more easily. You can also alter it if you need to without touching the main program.

When a program is running, the computer uses a 'pointer' stored in its memory to keep track of how many **DATA** items it has read. Every time a **READ** command is met, the **DATA** item indicated by the pointer is copied into the variable, and the pointer moves on to the next item. This means you

must have a **DATA** entry corresponding to each occurrence of **READ**: the program will stop if there are too many **READ** commands, and the 64 will print 'OUT OF DATA ERROR'. The command **RESTORE** can be used to reset the data pointer to the first **DATA** item in the program.

The variables used in the **READ** command must match the type of data in the corresponding **DATA** entry. A **READ** with a string variable will read anything as a string, but **READ** with a number variable must find a number in the **DATA** entry or a 'SYNTAX ERROR' will result. Integer variables must find integral numbers; real variables may read decimal fractions as well.

MORE ABOUT PRINTING

So far, we have described the use of **PRINT** (or **?**) simply to display single items of data on the screen. The 64 has a number of extra facilities which allow you to control the screen layout and produce clear and orderly output from programs.

PUNCTUATING PRINT STATEMENTS

You can **PRINT** more than one item on a single line by including 'punctuation' in the **PRINT** command. You can use commas to separate the items, in a **PRINT** statement, and the effect is rather like the 'tab' function of a typewriter. Output on the screen is formatted at every tenth column. For example,

```
PRINT "A", "B", "C", "D"
```

will give a display:

```
A           B           C           D
```

The characters can be used within strings, for example:

```
10    PRINT "{CLS}"
20    PRINT "TOP LINE"
30    CD$ = "{10 * CD}"
40    CU$ = "{12 * CU}"
50    FOR DELAY = 0 TO 3000:NEXT
60    PRINT CD$"MIDDLE LINE"
70    FOR DELAY = 0 TO 3000:NEXT
80    PRINT CU$"BACK TO THE TOP!"
```

prints the three messages in various parts of the screen using the strings CD$ and CU$ to move the cursor.

The way in which data is presented on the screen is an important part of the program and can make the difference between a program being easy to use, or frustrating and confusing.

SUMMARY

GET is used to assign a keystroke value from the keyboard to a variable. **GET** does not normally wait for a key to be pressed before continuing, and it is usual to include programming to ensure that it does.

READ and **DATA** are used to copy large amounts of information into variables without writing lots of program lines.

PRINTING

There are a number of special commands which allow control of the format of data displayed on the screen.

CHAPTER 7

PIECES OF STRINGS

String variables are used to store sequences - 'strings' - of characters. There are a number of BASIC commands which are used to manipulate strings, and these are described in this chapter.

We saw in Chapter 4 that a string variable can store a sequence of characters. For example:

```
NAME$    = "WILLIAM SHAKESPEARE"
GAME$    = "SPACE INVADERS"
REFERENCE$    = "ABC123D"
```

A string can hold up to 255 characters. These may be letters, figures, punctuation marks, spaces; in fact any of the characters the 64 can print. The number of characters in a string can be counted using the function **LEN**. Try this:

```
10    NAME$ = "JOHN SMITH"
20    PRINT LEN(NAME$)
```

This program prints the number 10, which is the number of characters in NAME$ (including the space).

TYING STRINGS TOGETHER

Strings can be added together.

```
10    A$ = "FLORENCE"
20    B$ = "NIGHTINGALE"
30    SPACE$ = " "
```

```
40    NAME$ = A$ + SPACE$ + B$
50    PRINT NAME$
```

Notice that strings don't add like numbers. B$ + A$ does not give the same result as A$ + B$. Adding strings is called concatenation and this is how really long strings are constructed - but don't forget that you can't have a string longer than 255 characters. If a program tries to add too many characters together, the 64 will print a 'STRING TOO LONG ERROR' message on the screen and stop the program.

CUTTING STRINGS

There are three BASIC functions which are used to extract information from strings. The functions are:

LEFT$** **RIGHT$ and **MID$**

LEFT$** and **RIGHT$ are both very similar in operation. **LEFT$** is used to read characters from the beginning of a string; **RIGHT$** reads from the end of the string. The functions are used like this:

LEFT$ (string, number)

The string in the brackets may be a variable or an actual sequence of characters between quotation marks ("), and the number may be a number or a variable.

For example:

```
PRINT LEFT$ ("ABCDEFG",4)
```

displays 'ABCD', and

```
10    S$ = "UVWXYZ"
20    N=3
```

```
30    R$ = RIGHT$ (S$,N)
40    PRINT R$
```

displays 'XYZ'.

The third function, **MID$**, is a little more complex, as we have to specify not only the string and the number of characters to be read, but also where in the string the characters are to be found.

```
PRINT MID$ ("12345678",3,4)
```

prints four characters, beginning at the third, of '12345678': so it prints '3456'. If you do not specify the number of characters to be read, all characters to the right of and including the specified character are included. Therefore:

```
PRINT MID$ ("ABCDEFGHIJK", 4)
```

displays 'DEFGHIJK'.

NUMBERS AND LETTERS

Computers cannot handle characters (numbers, signs and letters) directly - they use numbers to symbolise the characters. Each letter, and each of the other characters the 64 can print, is represented by a different code number. There are several systems used in different computers for deciding which code number represents which letter. The 64 uses a system called **ASCII** - the American Standard Code for Information Interchange - which is the most common system for micros. The table in Appendix 4 shows how the ASCII code relates to letters and numbers. In fact the 64 also uses a variant of ASCII in certain circumstances, but more of that later.

From this table we see that the 64 uses a code number 65 to represent the letter A, code number 66 for B and so on. Even numerals have codes. For example the character '9' has the code number 57.

Well, what use is this knowledge? There are two functions in BASIC which allow us to convert characters to numbers and numbers to characters.

The function **CHR$** converts a number to a string containing the corresponding character:

```
10    C = 65
20    A$ = CHR$(C)
30    PRINT A$
```

This displays the letter 'A'. You can use a variable with **CHR$**, as in the example, or a number:

```
PRINT CHR$(42)
```

This displays an asterisk (*).

CHR$ can only give one character at a time, and the number or the variable inside the brackets must be less than 255, or the computer will complain of an 'ILLEGAL QUANTITY ERROR'.

The function **ASC** complements **CHR$**. **ASC** finds the ASCII codes of characters.

```
10    B = ASC("B")
20    PRINT B
```

displays '66', which is the ASCII code for the letter B. You can also use string variables with **ASC**:

```
10    B$ = "B"
20    PRINT ASC(B$)
```

also displays '66'. If the variable contains more than one character, **ASC** gives the code for the first character only:

```
PRINT ASC("ABCDE")
```

displays '65'.

FIGURES IN STRINGS

A string can contain any characters - including numerals. There are two functions which can be used to make strings of number characters from actual numerals and to find the numeric value of the number characters in a string.

The function **STR$** creates from a number a string containing the characters of that number:

```
10   A$ = STR$(12345)
20   PRINT A$
```

This converts the single number 12345 into a string by assigning the characters 12345 to A$ and displays the string.

The complementary function to **STR$** is **VAL**. **VAL** evaluates the numerical characters in a string:

```
10   A$ = "123456"
20   A = VAL(A$)
30   PRINT A
```

This displays the number '123456'.

If the string evaluated contains letters as well as numbers then only the numbers which appear to the left of all the letters are converted by **VAL**. This means that

```
PRINT VAL("123ABC45")
```

displays '123'.

The signs + and - may appear before the number characters. They will be treated by **VAL** as the sign of the number.

TESTING STRINGS

Strings can be compared with each other in the same way as numbers, using **IF ... THEN**. A routine like this one can be used to check that an entry is suitable for the program:

```
10   INPUT "WHAT'S THE PASSWORD"; P$
20   IF P$ <> "BEANS" THEN PRINT
     "WRONG !!!":GOTO 10
30   PRINT "{CLS}O.K."
```

Another use for comparing strings is when the **GET** command is used in conjunction with a 'menu'. (In computing, a menu is a list of options displayed on the screen to the program operator). The next program displays a simple menu of display options, **GET**s the selection from the keyboard and then calls the selected routine. Memory location 650 controls the repeat key function on the 64, and normally contains a 0, allowing only the cursor, space and INST/DEL keys to repeat. Placing (**POKE**ing) a value of 128 in this location causes all keys to repeat.

```
10   PRINT"{CLS}"
20   PRINT:PRINT"SELECT OPTION"
30   PRINT:PRINT"S..SCREEN COLOUR"
40   PRINT:PRINT"B..BORDER COLOUR"
50   PRINT:PRINT"K..KEY REPEAT"
60   GET K$:IF K$="" THEN 60
```

```
70    IF K$="S" THEN REG=53281:GOTO
      1000
80    IF K$="B" THEN REG=53280:GOTO
      1000
90    IF K$="K" THEN 2000
100   GOTO 60
1000  D=PEEK(REG)
1010  IF D=255 THEN D=239
1020  POKE REG,D+1
1030  GOTO 60
2000  IF PEEK(650)=0 THEN POKE
      650,128:GOTO 60
2010  POKE 650,0:GOTO 60
```

Warning

Be careful when using **STR$** and making comparisons. If you compare "123" with STR$(123) you will find that the 64 thinks they are not the same. This is because any number which is shown as a result of using **STR$** is preceded by a space (or a minus sign if negative). This seems to be a feature shared by all Commodore machines. However, you can get round the problem by using **MID$** to remove the first character of the string:

```
A$ = MID$(STR$(123),2)
```

As well as testing to see if two strings are the same, you can use the 'greater than' and 'less than' (> and <) tests to compare strings.

```
10    IF "B" > "A" THEN PRINT "OK"
```

This works because the letters are stored as numbers. The 64 is in fact comparing the ASCII codes of the letters in the strings. This is very

useful because it allows strings to be sorted into alphabetical order:

```
10   PRINT "{CLS}"
20   DIM W$(20)
30   C = 0:L = 0
40   INPUT "ENTER A WORD";W$(C)
50   IF W$(C) = "ZZZ" OR C = 20 THEN
     C = C-1:PRINT"{CLS}": GOSUB
     1000:GOTO 100
60   C = C+1:GOTO 40
99   REM SORT STRINGS
100  S = 0:L = 20
110  FOR Z=0 TO C
120  IF W$(Z+1)>W$(Z) THEN 170
130  TEMP$ = W$(Z+1)
140  W$(Z+1) = W$(Z)
150  W$(Z) = TEMP$
160  S = S+1
170  NEXT Z
180  IF S <> 0 THEN 100
190  PRINT "{CLS}"
200  GOSUB 1000:END
999  REM DISPLAY NUMBERS
1000 PRINT"{CLS}"
1010 FOR P = 0 TO C
1020 PRINT TAB(L)W$(P)
1030 NEXT P
1040 RETURN
```

This program works in the same way as the sorting program at the end of Chapter 5. You can input up to twenty strings, which may contain more than one word. If you wish to sort fewer than 20 strings, then enter ZZZ as the last string; this will tell the program that there are no more strings to come.

If you run this program a few times you will discover how effective string sorting can be. Notice that any characters will be sorted into ASCII order, so strings like */@£?% and /@$&)# will be sorted.

You will find that lower case letters are treated as completely different, because of course they are represented by different codes, and the 64 is interested only in the codes. The lower case letters have higher codes than the capitals, so they appear at the end of the sorted list. (You can switch in and out of lower case by pressing the CBM key and the SHIFT key simultaneously).

Finally, here is a more light hearted demonstration of string handling, combining most of the ideas in this chapter to give you a game of Hangman. You can of course change the 'dictionary' to suit yourself.

```
5     REM ***********
6     REM *           *
7     REM * HANGMAN *
8     REM *           *
9     REM ***********
10    REM
20    POKE 53280,5:POKE53281,13:
      PRINT"{GR1}"
30    PRINT"{CLS}{CD}" TAB(15)"{RVS}
      HANGMAN {ROF}"
40    DIM D$(30)
50    G=1:W=0:CF=0:W$=""
60    FOR Z=0TO30
70    READ D$(Z):NEXT Z
80    FOR Z=0 TO 9
90    READ D(Z),A(Z):NEXT Z
93    REM
94    REM ***************
95    REM *             *
96    REM * DRAW GALLOWS *
97    REM *             *
98    REM ***************
99    REM
100   G1$=CHR$(176)+CHR$(195)+CHR$
      (195)+CHR$(174)
110   G2$=CHR$(221)
```

```
120  G3$=CHR$(183)+CHR$(183)+CHR$
     (183)+CHR$(183)+CHR$(183)+"{CU}
     "
130  PRINT"{CD}{CD}{CD}{CD}"
140  PRINT TAB(6)G1$
150  FOR Z=0 TO5:PRINTTAB(6)G2$:NEXT
160  PRINT TAB(4)G3$
183  REM
184  REM ********************
185  REM *                  *
186  REM * RANDOM SELECTION *
187  REM *                  *
188  REM ********************
189  REM
190  N = INT(RND(1)*31)
200  C$ = D$(N)
210  L = LEN(C$)
220  FOR Z=1 TO L
230  PRINT TAB(14)"- ";:NEXT
240  PRINT"{HOM}{CD}"TAB(30)"GUESS
     ";G
243  REM
244  REM ********************
245  REM *                  *
246  REM * INPUT YOUR GUESS *
247  REM *                  *
248  REM ********************
249  REM
250  GET G$:IF G$=""THEN 250
255  IF G$<"A" OR G$>"Z" THEN 250
260  FOR Z=1 TO LEN(U$)
265  IF G$=MID$(U$,Z,1) THEN 250
270  NEXT Z
275  G=G+1:U$=U$+G$:GOSUB 400
280  IF W=10 THEN M$="{RED} YOU
     LOSE!!":GOSUB 360:GOTO 310
290  IF C=L THEN M$="{RED} YOU
     WIN!!":GOTO 310
300  GOTO 240
310  PRINT"{HOM}{CD}"M$
```

```
320  PRINT"{HOM}{18 * CD}"TAB(11)"
     ANOTHER GO (Y/N)?"
330  GET A$:IF A$=""THEN 330
340  IF A$="Y" THEN RUN
350  PRINT "{CLS}":END
360  PRINT"{HOM}{14 * CD}"TAB(14);
370  FOR Z=1 TO L
380  PRINT MID$(C$,Z,1)" ";:NEXT:
     RETURN
393  REM
394  REM ***************
395  REM *               *
396  REM * CHECK GUESS *
397  REM *               *
398  REM ***************
399  REM
400  FOR J=1 TO L
410  IF MID$(C$,J,1)=G$THEN CF=1:
     C=C+1:PRINT"{HOM}{14 * CD}"
     TAB(12+2*J)G$
420  NEXT J
430  IF CF=1 THEN 480
440  W$=W$+G$
450  PRINT"{HOM}{22 * CD}"TAB(5)W$
460  POKE A(W),D(W):POKEA(W)+54272,6
470  W = W+1
480  CF = 0:RETURN
993  REM
994  REM ************************
995  REM *                        *
996  REM * DATA FOR DICTIONARY *
997  REM *                        *
998  REM ************************
999  REM
1000 DATA "BYTE","BASIC","BINARY",
     "MEMORY","POKE","PEEK","DATA","
     PRINT","LIST"
1010 DATA "BUFFER","SEQUENTIAL",
     "REGISTER","CURSOR","SERIAL","P
     ARALLEL"
```

```
1020 DATA "RAM","ROM","SPRITE",
     "ASCII","ARRAY","STRING","SYNTA
     X","INTEGER"
1030 DATA"SCROLL","GOSUB","GRAPHIC",
     "RESTORE","LOOP","VARIABLE","VE
     RIFY","BEANS"
1993 REM
1994 REM **********************
1995 REM *                    *
1996 REM * DATA FOR CHARACTER *
1997 REM *                    *
1998 REM **********************
1999 REM
2000 DATA 81,1353,91,1393,112,1392
2010 DATA 110,1394,93,1433,99,1473
2020 DATA 103,1472,101,1474,103,1512
2030 DATA 101,1514
```

Program Description

20 - 90 Set up display, read the data for the dictionary into array D$(Z), read the data for the characters comprising the 'victim' into array (DZ) and their corresponding screen memory locations into array A(Z).

100 - 160 Draw the gallows using keyboard graphics characters (see Chapter 11).

190 - 240 Select a word at random from the dictionary and display a series of hyphens, one for each letter of the word.

250 - 380 Input a guess and check it for validity. Each time a letter is used it is placed in string U$, which is then used to prevent a letter being picked more than once (lines 260-270). Also controls the game and finishes it after

10 attempts have been made. If you lose, the correct answer is displayed by lines 360 - 380.

400 - 480 Check your guess against the chosen word and if correct, the appropriate letters are revealed in the answer. If not, one section of the victim is drawn by line 460.

SUMMARY

A string is a sequence of characters. These can be stored in string variables, which are distinguished by $ after the variable name.

LEN(string) gives the number of characters in a string.

Strings can be added together:

```
PRINT "ABC" + "DEF" + "GHI"
```

displays 'ABCDEFGHI'.

LEFT$, RIGHT$ and MID$ are used to separate parts of strings.

LEFT$(string,N) takes the first N characters of the string

RIGHT$(string,M) reads the last M characters

MID$(string,P,Q) reads Q characters, starting at character P

ASCII code is used by the 64 to symbolise letters. There is a table of the ASCII codes for the characters in Appendix 10.

CHR$ converts a number to the equivalent character:

```
PRINT CHR$(65)
```

displays 'A'.

ASC gives the ASCII code for a character:

```
PRINT ASC("A")
```

displays '65'.

STR$ turns a number into a string of number characters:

```
10   A$ = STR$(123)
20   PRINT A$
```

VAL evaluates the numerical characters in a string:

```
PRINT VAL("123ABC789")
```

displays '123'.

COMPARISONS

Strings can be compared using **IF ... THEN** and the relational operators =, <, and >. This is useful for testing inputs and sorting strings.

CHAPTER 8

FUNCTIONS

A 'function' in computing is an instruction which performs a calculation on a number. There are a number of functions available on the Commodore 64. For example, in Chapter 7 several functions were described which operate on numbers to give strings, on strings to give other strings, or on strings to give numbers. In this chapter we will look at some more of the functions available on the Commodore 64.

SQUARE ROOTS

The square root of a number is calculated by the function **SQR**(N). (The square root of a number or variable N is the number which when multiplied by itself, or *squared*, gives N.) Try:

```
PRINT SQR(4)
```

The 64 displays 2, because 2 squared (2*2) is 4.

```
10 FOR N = 1 TO 10
20 PRINT N, SQR(N)
30 NEXT
```

prints the numbers 1 to 10 and their square roots.

ABSOLUTE VALUES

The function **ABS** finds the **absolute value** of a number: the value of the numerical part, disregarding the sign. The function changes negative numbers to positive numbers, but has no effect on positive numbers.

```
PRINT ABS( 123.456)
```

displays '123.456' - no change. But

```
PRINT ABS( -543.345)
```

displays ' 543.345 ' - the minus sign has been removed. **ABS** rounds up to six decimal places.

INTEGER CONVERSION

The function **INT** removes the fractional part of a number and returns the next lower whole number. Try:

```
PRINT INT(123.4567)
```

Only the whole number part - 123 - is displayed.

Be careful with numbers less than zero. The **INT** function finds the first whole number lower then the number you give it. This means that for negative numbers the answer is not the whole part of the initial number, but one less (or minus one more!). Therefore:

```
PRINT INT(-2.875)
```

displays '-3'.

SIGNS

SGN(N) returns a value which indicates the sign of a number or variable N. If the number is positive, the result is 1; if the number is zero, the result is zero; and if the number is less than zero, **SGN** returns -1.

```
PRINT SGN (5)      displays '1'
PRINT SGN (0)      displays '0', and
PRINT SGN (-7)     displays '-1'.
```

TRIGONOMETRY

The trigonometrical functions of sine, cosine, tangent and arctangent are available in Commodore BASIC. They are written:

SIN(N)

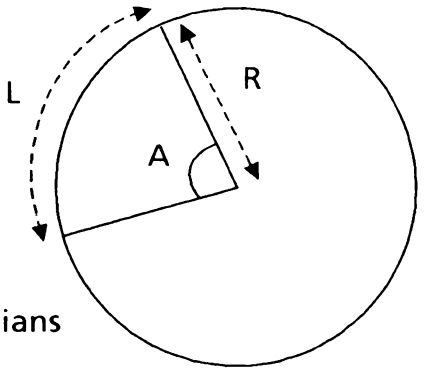
COS(N)

TAN(N)

ATN(N) where N is a number or a variable.

Note that the angles on which these functions operate must be given in **radians**, not in degrees. A radian is the ratio of the length of an arc of a circle to the radius. In the diagram overleaf, the angle A in radians is L/R. The circumference of a circle is 2π times its radius ($2\pi R$), so it follows that 360 degrees is equivalent to 2π radians.

From this we can work out that 1 degree is $2\pi/360$ radians (about 0.175), and 1 radian is $360/2\pi$ degrees (about 57.3°). To convert from degrees to



The angle A is L/R radians

radians multiply by 2π/360. The value of π can be obtained on the CBM 64 by using the π symbol (press SHIFT and ↑). To convert from radians to degrees multiply by 360/2π.

So, the sine of 45 degrees (0.7071) is given by:

```
10 RAD = 2*π/360
20 PRINT SIN(45*RAD)
```

The arctangent of 1 (45 degrees) is given by:

```
10 RAD = 2*π/360
20 ANGLE = ATN(1)/RAD
30 PRINT ANGLE
```

LOGARITHMS

Natural logarithms are provided by the function **LOG( )**.

```
PRINT LOG(10)      displays '2.30258509'

PRINT LOG(5000)    displays '8.5171932'
```

The antilogs of natural logarithms are calculated by **EXP**(N). Try:

```
PRINT EXP(LOG(100))
```

The answer is 100.

Logs can be used to calculate roots. This program finds the cube root of 8.

```
10 A = 8
20 R = LOG(A)/3
30 PRINT EXP(R)
```

The answer is 2, because 2 cubed (2*2*2) is 8.

This program calculates fifth roots:

```
10 N = 243
20 R = LOG(N)/5
30 PRINT EXP(R)
```

The program will display '3', because 3 to the power 5 (3 ↑ 5 or 3*3*3*3*3) is 243.

RANDOM NUMBERS

The 64 has a function which provides random numbers. The function is **RND**(X), which prints a number between 0 and 1. The 'seed' value - X - can be any number or letter, but does directly influence the value of the random number that results. Try **PRINT**ing **RND**(1) a few times. You will get a different number each time.

The random numbers provided by **RND** are not truly 'random' - it is very difficult and expensive to make a machine which will give perfectly random numbers. If you switch your 64 off and on, and immediately type `PRINT RND(1)`, you will get the same number every time (on our 64 it's

0.185564016). The random numbers are calculated from a starting number, or seed, and any seed will always produce the same sequence of numbers. The random number calculation can be controlled by altering the number or letter in the brackets.

You can give the computer a new seed for its random number generator by putting a negative number in the function. Try `PRINT RND(-X)`. The resulting number is not much use, but the effect of the command is to give a new seed number to the random number generator. Try **PRINT**ing five numbers with **RND**(1) and make a note of them. If you then re-seed the random number generator with 1 by **PRINT**ing **RND**(-1) again, and re**PRINT** five numbers with **RND**(1), you will get the same sequence as before. Each negative number sets off a different sequence.

There is a way of obtaining a more nearly random sequence of 'random' numbers. If the seed given to the random number generator is different each time, the sequences will be different. We can get an unknown, varying seed by using the 64's built in timer. The preset variable TI (or TIME) is incremented automatically every 60th of a second. If you use TI to seed the random number generator you will obtain a different number every time. This means that:

```
PRINT RND(-TI)
```

will give an unknown seed to the random number generator.

To sum up, if the number given to the **RND** function is greater than zero, a number from a sequence of 'random' numbers is returned by the

function. If the number given to the function is negative, a new sequence is produced.

Dice Throwing

We can use random numbers in programs to imitate the throwing of dice. Try this:

```
10   A = RND(-TI)
20   PRINT "{CLS}PRESS ANY KEY TO
     THROW THE DICE"
30   GET K$:IF K$="" THEN 30
40   PRINT "{CLS}"
50   DICE = INT( (RND(0)*6) + 1)
60   D$ = "YOUR NUMBER IS" +
     STR$(DICE)
70   PRINT D$
80   FOR DELAY = 0 TO 500:NEXT
90   GOTO 20
```

This program will throw the die every time you press a key. Pressing the STOP key will stop the program.

DEFINING YOUR OWN FUNCTIONS

Commodore BASIC allows you to define functions of your own, which can be used throughout a program. Functions are defined by the command **DEF FN**. They can then be used like any other functions in calculations and tests.

```
10   DEF FNA(N) = π * N↑2
20   PRINT "PROGRAM TO CALCULATE
     AREAS OF CIRCLES"
30   PRINT
40   INPUT "ENTER THE RADIUS";R
50   PRINT: PRINT "AREA OF A CIRCLE
     RADIUS "R" IS "FNA(R)
```

```
60    IF FNA(R) < 100000 THEN 30
```

This program uses a function FN AREA which is defined in line 10. The variable N in the definition is a 'dummy' variable - any variable name can be used when the function is used later in the program. The function is used in lines 50 and 60.

The name of a function may be any legal variable name. It may have any number of letters, but, as with variables, only the first two characters are noticed by the 64.

SUMMARY

There are several built-in functions in Commodore BASIC which operate on numbers or variables:

SQR(N) calculates the square root of a number N.

ABS(N) returns the absolute value of a number - changing negative numbers into positive ones, and leaving positive numbers unchanged.

INT(N) returns the integer value of N, removing any fractional part.

SGN(N) gives 1 if N is positive, 0 if N is zero and -1 if N is negative.

The trigonometric functions:

SIN(N)

COS(N)

TAN(N)

ATN(N)

return the values for the angle N (which must be in radians).

LOG(N) returns the natural logarithm of a number, N.

RND(N) returns a pseudo random number between 0 and 1.

RND(-N) will re-seed the random number generator

DEF FN allows you to define your own functions within programs.

CHAPTER 9

LOGICAL THINKING

As well as doing arithmetic, the Commodore 64 can perform tests to compare numbers and strings. We have already seen this when using **IF ... THEN**. In this chapter we will examine in more detail the way in which the 64 makes comparisons.

Consider the program line:

```
220  IF A = 3 THEN 500
```

The line instructs the 64 to branch to line 500 if the variable A holds the value 3. What does the 64 do when it encounters this program line?

The first thing the computer must do is decide whether A is equal to 3; or to put it another way, whether 'A = 3' is true. The 64 tests this, and if the expression is true, it returns an answer of -1; if the expression is false, the answer is 0. If the result is true (-1), the commands after **THEN** are obeyed. If the answer is false (0) the program will continue with the next line.

Why are we telling you all this? Because the computer can compare numbers and return true or false values without **IF**. Try this short program:

```
10  A = 3
20  PRINT A = 3
```

The program displays '-1'. If you change line 10 to

```
10  A = 5       (or any other number)
```
the program will print 0.

This applies to all the other comparisons listed in Chapter 5 as conditions for **IF**:

=	Equal
<	Less than
>	Greater than
<>	Not equal
>=	Greater than or equal
<=	Less than or equal

Try this:

```
10  A = 5
20  PRINT A <= 7
```

If you try out the other tests, you will find they all behave in the same way.

LOGIC

If two tests are combined, the same true and false answers are still obtained:

```
10  A = 5
20  PRINT ( A < 7 ) AND ( A > 3 )
```

Now, if the computer evaluates simple relational expressions such as A<7, A>3 as -1 or 0, what happens when two are combined, and what does the **AND** do?

There are three BASIC commands which can be used with relational expressions: **AND**, **OR** and

NOT. Forgetting about numerical representations of true and false for the moment, let's look at what these commands do.

Using **AND** to relate two expressions, as in the example above, it seems fairly obvious that the final result will be true only if both smaller expressions are true. This is in fact what happens. Here is a table of the possible combinations, with the two simple expressions represented by X and Y. (This type of table is called a 'truth table'.)

X	Y	X **AND** Y
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

*Truth Table for **AND***

The command **OR** also does the obvious thing, returning -1 if X or Y or both are true.

X	Y	X **OR** Y
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

*Truth Table for **OR***

NOT changes from true to false, and vice versa, so:

```
10  A = 3
20  PRINT NOT A>10
```

prints -1 (true). Here is the truth table for **NOT**.

X	**NOT** X
TRUE	FALSE
FALSE	TRUE

*Truth Table for **NOT***

All this is fairly easy to follow. But the 64 is using numbers to represent true and false. How does it work out the answers?

To understand this, you must understand how the computer stores numbers in binary notation (Appendix 6 contains an explanation). **AND**, **OR** and **NOT** are actually applied by the 64 to the binary representations of the numbers. The truth tables for the three operators remain the same, but true and false are replaced by 1 and 0. The truth table for **AND** therefore changes to the form shown on page 86.

Each number contains more than one binary digit (or 'bit'). The 64 applies **AND**, **OR** and **NOT** to the numbers by comparing bits at the same place in each number. Suppose A and B were two 4-bit numbers:

A = 1 1 0 0 B = 1 0 0 1

A	B	A **AND** B
1	1	1
1	0	0
0	1	0
0	0	0

*Numerical Truth Table for **AND***

The digits for (A **AND** B) are found by comparing the corresponding digits of A and B. So, bit 1 of A **AND** B (counting from the right) is (0 **AND** 1), which is 0. Bit 2 is 0 **AND** 0 which is also 0. Proceeding thus, we find

A **AND** B = 1 0 0 0

By similar means we can show that if P is 1010 and Q is 0011, P **OR** Q is 1011; and if Z is 0101, **NOT** Z is 1010.

Returning to the computer's comparisons of true and false, we can now see why the numbers 0 and -1 are used. To represent negative numbers the machine uses the **two's complement** system . This means that -1 is stored as 1111, while 0 is stored as 0000. (The computer actually uses 16 bits to represent each, but it's easier to follow with only 4). This means that true is a number which is **all ones**, and false is **all zeros**, so the **AND**, **OR** and **NOT** commands can work on the individual bits and produce the right answer.

It may seem that all this knowledge is of little use when programming in BASIC, but it can be used to simplify some programs which involve tests.

```
320  IF (A+B)>6 THEN Z=7
330  IF (A+B)<=6 THEN Z=0
```

could be replaced by the single line:

```
320  Z = -((A+B)>6)*7
```

and the two lines:

```
500  IF P=Q THEN T=5
510  IF P<>Q THEN T=3
```

are equivalent to:

```
500  T = 3-2*(P=Q)
```

AND can be used to keep numbers within certain limits by **masking**. Think about what would happen if any binary number was **AND**ed with the number 00001111. The four highest bits of the result (the left four) would be zero, whatever the other number, because 0 **AND** X is always 0. The four lower bits would be the same as those of the other number. For example:

10101010 AND 00001111 = 00001010

We can therefore use **AND** to copy only a part of a number. Look at this example:

```
230  A = A + 1
240  IF A > 15 THEN A = 0
```

15 in binary is 00001111. If A is increased to 16 (00010000) then A **AND** 15 will be 0. We could replace the two program lines with

```
230  A = (A+1) AND 15
```

These uses of relations and logical operators may seem more confusing than using **IF** to perform tests, but they are also faster, which means that in a long program, or one which will loop many times, considerable time savings can be made.

SUMMARY

The Commodore 64 can not only handle numerical calculations: it can solve logical problems too. Relational tests such as those performed after **IF** are represented numerically:

True is represented by the number -1.

False is represented by the number 0.

There are three logical operators:

AND A **AND** B is true if both A and B are true.

OR A **OR** B is true if either A is true or B is true or both are true.

NOT **NOT** A is true if A is false.

CHAPTER 10

MEMORY MANAGEMENT

The memory of a computer is built up from a large number of storage units, each of which can hold one number. Each storage unit is called a *memory location*, and location has a unique reference number called its *address*. In a microcomputer like the Commodore 64, each memory location can hold an eight-bit binary number, which can have a value between 0 and 255. Larger numbers need more than one location. (If you are not sure about binary numbers, they are explained in Appendix 6.) Eight bits of storage are called a *byte* in computer jargon, so the memory locations of the 64 hold one byte of data each.

The microprocessor used in the Commodore 64 (the 6510) is able to make use of 65535 memory locations (65535 is 2^{16}, or 64k). You will have noticed that the message displayed when you switch on the computer announces that 38911 bytes are free for use by BASIC programs. What happens to the others?

The first 1k of memory is used as a storage area by the machine code programs that interpret your BASIC programs and operate the machine. The next 1000 bytes are used to store the display picture - 1 byte for every character. A second block of memory, locations 55296 to 56295, holds the colour data for each character.

BASIC programs begin at location 2048, and all the memory up to 40k is free for BASIC programs and

variables. The remainder of the memory contains the routines that operate the computer and run BASIC programs.

There are two types of memory used in a computer. Random access memory, or RAM, is used to store data, and its contents can be changed as the data changes. Read only memory, or ROM, cannot be altered, and is used to store the programs needed for the computer to work at all; the BASIC interpreter for example.

There are two BASIC commands, **POKE** and **PEEK**, which are used to look at individual memory locations. **PEEK** is used to find out the contents of a location, and **POKE** writes a new number into the memory. You can try out **PEEK** and **POKE** by using them to control the display. Clear the screen, move the cursor down 1 line, and type:

```
POKE 1024, 1
POKE 55296, 1
```

The first **POKE** puts the code for an A in the first position of the screen. The second puts the code for white in the corresponding position in the colour memory (these will be explained in the next chapter). You should see a white A at the top of the screen. If you now type:

```
PRINT PEEK(1024)
```

the computer will print the answer 1, telling you that the number in location 1024 is a 1 (it should be, as you've just put it there!).

What makes the **POKE** and **PEEK** commands so important is that not all the memory addresses of the 64 are used for memory. Some of the addresses are used to point to the control registers of the

special chips which control the sound and graphics facilities. This means that the chips can be controlled simply by writing a number into these control registers using the **POKE** command. You discovered in Chapter 3 that the screen and border colours could be changed by **POKE**ing numbers into locations 53280 and 53281. These locations are two of the registers of the Video Interface Chip, or VIC, which provides all the display functions of the Commodore 64. In the following chapters we will describe how to use the VIC, and also the Sound Interface Device, or SID, which manages the sound and music facilities of the 64.

EMPTY SPACE

As you write longer programs, you will find that even the 38k of program space provided by the 64 is not unlimited. If a program uses large arrays for data storage, the memory can be eaten up at an alarming rate. The function **FRE(X)** tells you how much memory is left unused.

If the amount free is greater than 32767, **FRE(X)** will return a negative number, so use the formula:

```
PRINT FRE(X) + 65535
```

The argument of the function has no effect - any number or variable can be used.

If you check the memory immediately after switching on your 64, the free space will be 38908 bytes. Enter a program, and try it again, and a smaller number will be printed. If you **RUN** the program and use **FRE(X)** a third time the number printed will be smaller still, as the 64 has taken memory space to store the variables.

THE CLR COMMAND

The **CLR** command may occasionally be useful. The command clears all the variables stored by a program, but leaves the program itself intact. You can then start a program with the maximum amount of memory free. **CLR** also resets the variable pointers, and should therefore be used after moving the top of BASIC memory by **POKE**ing new values into locations 55 and 56. This technique is used in Chapter 12 to protect relocated character sets from BASIC programs.

CHAPTER 11

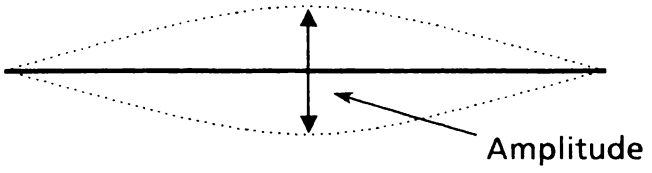
SOUND AND MUSIC

The Commodore 64 has exceptional sound generating facilities. The built-in Sound Interface Device (SID) is a synthesiser on a chip, with three *voices*, a range of eight octaves, and extensive control of the *waveforms* and *envelopes* of each voice. It is possible to *filter* the sound, and even to mix sound generated by the 64 with sound fed in from an external source.

The 64 has no built-in loudspeaker, but uses the loudspeaker of the television or monitor to which it is connected, and can also be played through a hi-fi system.

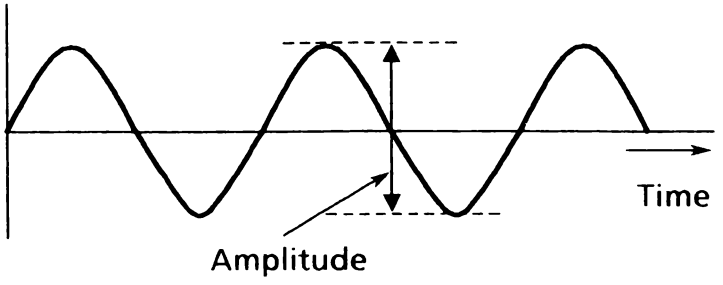
WHAT IS SOUND?

Sound is the effect on our ears of vibrations in the air caused by a vibrating object such as a guitar string or a loudspeaker.



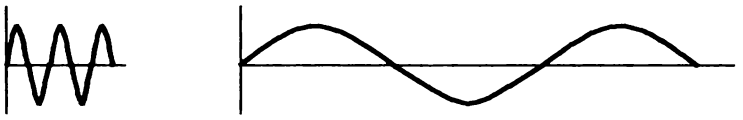
Vibrating Guitar String

If we plot a graph of the way a vibrating object moves over time we get an undulating shape which

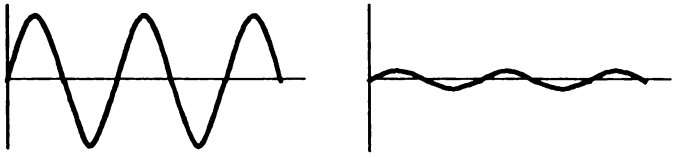


Simple Wave

is called a 'wave'. What you hear depends on the properties of the wave: the rate at which the object vibrates (the frequency of the vibration) determines the pitch of the sound, and the amplitude determines the volume.



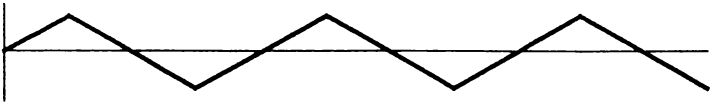
High and Low Frequencies



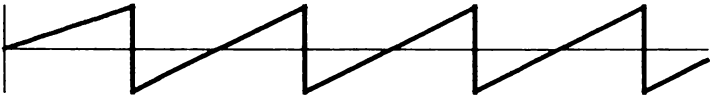
Large and Small Amplitudes (Loud and Soft)

Not all sounds have the same shape of wave, and this wave-shape, or waveform, has a great effect on the quality of the sound. The Commodore 64 can

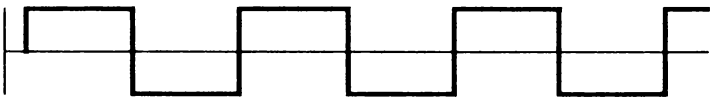
generate four different waveforms, which are shown in the diagrams:



Triangle Waveform



Sawtooth Waveform

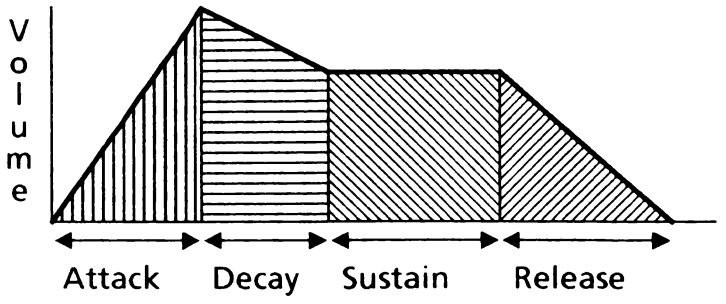


Pulse Waveform



Noise Waveform

The final property of sounds which can alter the way we hear them is the variation of the volume over time. Four parameters are used to describe the shape of the 'envelope' of the sound wave: Attack, Decay, Sustain and Release. Attack is the rate at which the sound rises from nothing to full volume when the note begins. Decay is the rate of falling off to a steady volume. Sustain is the volume at which the sound steadies, and Release describes the rate at which the sound falls off to nothing. The diagram on the next page shows the envelope properties.



The Envelope Parameters

USING THE SOUND GENERATOR

Because the Commodore BASIC does not support special commands to do the job, the sound has to be controlled by **POKE** commands direct to the SID chip. This is not as difficult as it seems, but it does make programs using sound look confusing.

Each of the three voices is controlled by seven registers of the SID chip. These control the frequency of the generated note, its waveform and the envelope shape of the sound. There are a further eight registers which control the volume of the three voices together, and the filter which may be used to modify the sound once it has been defined.

The registers controlling the first voice are at memory locations 54272 to 54278. Try this program to see how these registers are used (don't bother to type in all the **REM** lines):

```
10    REM  FIRST SOUND PROGRAM
20    REM  USES VOICE 1 ONLY
100   REM  SET FREQUENCY
110   POKE 54272, 37
120   POKE 54273, 17
200   REM  SET VOLUME
210   POKE 54296, 15
```

```
300   REM  SET ATTACK/DECAY &
      SUSTAIN/RELEASE
310   POKE 54277,54
320   POKE 54278,168
400   REM  TURN ON THE SOUND WITH
      SAWTOOTH WAVEFORM
410   POKE 54276, 33
500   REM  WAIT A WHILE
510   FOR T = 1 TO 500 : NEXT T
600   REM  SWITCH OFF SOUND
610   POKE 54276, 32
620   POKE 54296, 0
```

When you **RUN** the program you should hear a note from the loudspeaker of your television (make sure the volume control is not turned right down).

CONTROLLING THE SID CHIP

The registers of the SID chip controlling the three voices are as follows:

Voice 1	Voice 2	Voice 3	No.	Property Controlled
54272	54279	54286	0	Frequency - Low Byte
54273	54280	54287	1	Frequency - High Byte
54274	54281	54288	2	Pulse Waveform Properties - Low Byte
54275	54282	54289	3	Pulse Waveform Properties - High Byte
54276	54283	54290	4	Control Register - Waveform and On/Off
54277	54284	54291	5	Attack and Decay
54278	54285	54292	6	Sustain and Release

Registers 0 and 1 (which are locations 54272 and 54273 for Voice 1, for example) control the frequency of the sound. The frequency heard relates to the number in the registers like this:

Frequency = Number * 0.06 (beats/second)

There is no need for complex calculations to work out notes as there is a table on page 152 of the Commodore 64 User Manual which gives the numbers for all notes over a range of 8 octaves. The frequency used in the previous program gave the note middle C.

Register 4 selects the waveform to be used. The possible values are:

17	Triangle
33	Sawtooth
65	Pulse
129	Noise

To see in more detail how these registers are used we will use the previous program to test the effects of altering the parameters of the sound.

FREQUENCY

Save the program onto tape - you will need it again - and then change the program by typing in the following alterations.

```
105   INPUT "FREQUENCY NUMBER"; F
110   POKE 54272, F - INT(F/256)*256
120   POKE 54273, F/256
1000  GOTO 100
```

Try entering different frequency numbers and see how the sound changes; the higher the number the higher the note. You won't hear much for values of

less than about 500, and the highest possible note is 65535. Notice the effect of doubling the number, or of increasing it by 1/2. Try the sequence 4000, 5000, 6000, 8000. To stop the program hold down RUN/STOP and tap RESTORE.

To hear the full range of frequencies, type in these new lines:

```
50    F = 250
105   F = F * 2↑(1/12)
510   FOR T = 1 TO 50 : NEXT
550   IF F < 61800 THEN 105
```

Delete line 1000, and **RUN** the program.

WAVEFORMS

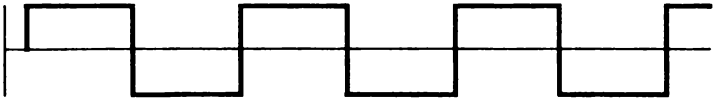
The program uses the Sawtooth waveform which gives a slightly buzzy tone. To hear the Triangle waveform, modify the program again by typing:

```
410   POKE 54276, 17
```

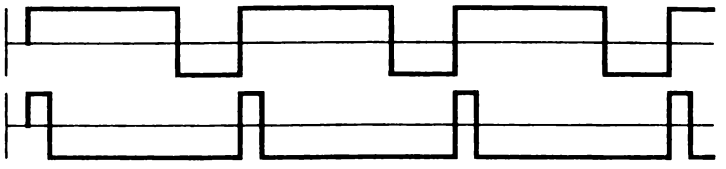
To hear the Noise waveform, alter line 410 to:

```
410   POKE 54276, 129
```

The fourth waveform, Pulse, is slightly more complicated than the others. You will remember that the waveform is:

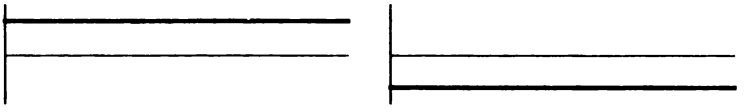


It is possible to alter the proportions of the wave by altering the width of the peaks, to produce a variety of alternative waveforms.



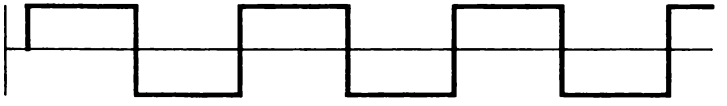
Alternative Pulse Waveforms

This is done by specifying the width of the peak as a proportion of the whole cycle. The proportion is set by **POKE**ing a number between 0 and 4095 into registers 2 and 3 for the voice concerned (locations 54274 and 54275 for Voice 1). A value of 0 or 4095 would give a straight line:



Pulse width 4095 *Pulse width 0*

A value of 2047 would make the peak width 50% of the total, giving:



Pulse Width = 2047 (50%)

To try out the pulse waveform, reload the first program and make the following changes.

```
350   REM  SET PULSE WIDTH
360   INPUT "PULSE WIDTH %"; P
370   P = P*40.95
380   POKE 54274, P AND 255
390   POKE 54275, P/256
410   POKE 54276, 65
610   POKE 54276, 64
```

```
1000  GOTO 360
```

The program requires inputs between 0 and 100, representing the pulse width as a percentage of the total. Try a few different values. The smoothest tone is given by a value of 50, which gives the balanced waveform with equal high and low times. As the numbers increase or decrease from 50, the sound becomes sharper and more 'tinny'.

VOLUME

The volume of the the sound of all three voices is controlled by location 54296. The possible values are from 0 to 15, with 0 giving no sound and 15 maximum volume. Reload the first program again and type these lines:

```
210   INPUT "VOLUME"; V
220   POKE 54296, V
1000  GOTO 210
```

ENVELOPE

The final way of altering the sound of a voice channel is by altering the Envelope, or volume profile. This is done with the four parameters Attack, Decay, Sustain and Release. For the last time, reload the first program. Type in the modifications below:

```
310   INPUT "ATTACK"; A
320   INPUT "DECAY"; D
330   INPUT "SUSTAIN"; S
340   INPUT "RELEASE"; R
350   POKE 54277, ((A AND 15) * 16)
      OR (D AND 15)
360   POKE 54278, ((S AND 15) * 16)
      OR (R AND 15)
510   FOR T = 1 TO 500: NEXT T
```

```
620   GOTO 310
```

Attack and Decay are controlled by register 5 for each voice (54277 for Voice 1). The four most significant bits of the register control Attack; the four least significant bits control Decay. Sustain and Release are similarly controlled by register 6 for each voice (for Voice 1 it's 54278), with Sustain controlled by the most significant bits.

RUN the program and experiment to find the effects of different envelope parameters. Each of the four variables can have a value from 0 to 15. The value used for Sustain controls the volume of the note after the Attack and Decay phases. The durations of Attack, Decay and Release for the range of values which can be set are shown below:

VALUE	ATTACK	DECAY	RELEASE
0	2 ms	6 ms	6 ms
1	8 ms	24 ms	24 ms
2	16 ms	48 ms	48 ms
3	24 ms	72 ms	72 ms
4	38 ms	114 ms	114 ms
5	56 ms	168 ms	168 ms
6	68 ms	204 ms	204 ms
7	80 ms	240 ms	240 ms
8	100 ms	300 ms	300 ms
9	250 ms	750 ms	750 ms
10	500 ms	1.5 s	1.5 s
11	800 ms	2.4 s	2.4 s
12	1 s	3 s	3 s
13	3 s	9 s	9 s
14	5 s	15 s	15 s
15	8 s	24 s	24 s

Attack, Decay and Release

FILTERING

In addition to the three voices, the SID chip has a built-in filtering facility which filters the output from all three voice channels together. The filter is controlled by the four registers 54293 to 54296. Three different filters may be used, either singly or in combination. The three are a High-pass filter,



High-Pass Filter

which allows high frequency sounds to be output but suppresses the lower frequencies, a Low-pass filter which suppresses high frequencies, and a



Low-Pass Filter

Band-pass filter which allows only the frequencies



Band-Pass Filter

within a limited range to pass. Two filters may be combined to produce more complex filtering.

The filters are selected by bits 4, 5 and 6 of location 54296. You will remember that this is the location

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
Noise	Pulse	Saw-tooth	Tri-angle	Test	Ring Mod	Sync	Gate
128	64	32	16	8	4	2	1

The Voice Control Register

The four most significant bits of the register control the waveform of the voice. Only one of these bits should be set to '1' at any time.

TEST switches off the oscillator if bit 3 is set to 1. This is of little use in sound generation.

RING MOD and SYNC allow the production of more complex waveforms than the standard waveforms available. These are described below.

The GATE bit turns the sound on and off. When GATE is set to 1, the Attack/Decay/Sustain cycle starts. The note is sustained until GATE is set to 0, when the Release phase of the note begins. It is important not to alter the waveform when clearing GATE, or the sound will change during the Release phase.

SOUND EFFECTS

The SID chip can not only be used for playing music, but for producing all kinds of sound effects and strange noises. By varying the frequency while a note is playing, siren effects can be achieved. It is possible to alter the waveform while a note is playing by setting the appropriate bits of the control register (remember not to clear the GATE bit of the register, or the sound will be cut off). The SID chip has extra facilities for producing more complex sounds than those we have heard so

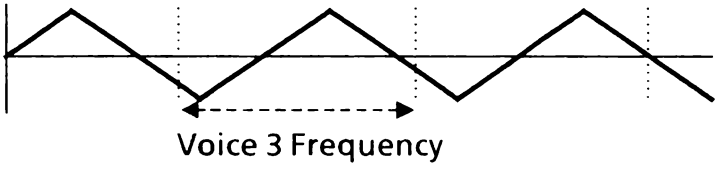
far; these are accessed by using the RING MOD and SYNC bits of the voice control registers.

SYNC

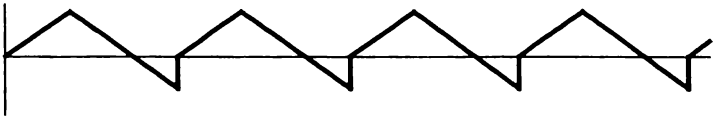
The SYNC bit of the control register of each voice synchronises the output of the voice with the frequency of one of the other voices. The voices are paired thus:

Voice 1 is synchronised with Voice 3
Voice 2 is synchronised with Voice 1
Voice 3 is synchronised with Voice 2

The SYNC function controls the waveform of the voice by repeatedly restarting the wave at the frequency of the controlling voice. If Voice 1 is producing a triangle wave like this:

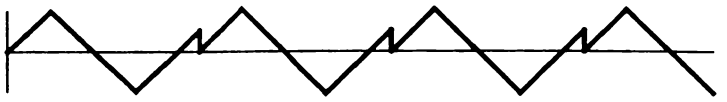


and Voice 3 has the frequency indicated, then if Voice 1 is synchronised with Voice 3 the output will look like:



That is, the waveform becomes a repetition of part of the Voice 1 waveform at the frequency of Voice 3. If the frequency of Voice 3 is greater than that of Voice 1 similar effects result, as shown overleaf.

The waveform of the synchronised voice can be set to any one of the four possibilities, though best



results are obtained from the triangle and sawtooth waveforms. The properties of the synchronising voice other than its frequency have no effect at all on the sound produced. The use of SYNC can produce many new waveforms of great tonal interest as the two frequencies are varied with respect to each other. The following program will allow you to try out different combinations of frequencies in the two voices.

```
10    INPUT"VOICE 1";V1
20    V3=35
30    FOR S=54272 TO 54296: POKE
      S,0: NEXT
40    POKE 54296, 15
50    POKE 54273, V1
60    POKE 54287, V3
70    POKE 54277, 84
80    POKE 54291, 84
90    POKE 54278, 168
100   POKE 54292, 168
110   POKE 54276, 19
120   FOR S=1 TO 1000: NEXT
200   POKE 54276, 0
210   POKE 54290, 0
500   GOTO 10
```

Try altering the frequency of Voice 1. Notice that when the frequency is equal to that of Voice 3, the sound is unaltered, and when the Voice 1 frequency is a multiple or a factor of the Voice 3 frequency the sound is less interesting. Best results are achieved when the frequencies are close to each other.

RING MODULATION

If bit 3 of a voice control register is set, the output of the voice is ring modulated with the frequency of one of the other voices, resulting in a non-harmonic mixture of frequencies. To use ring modulation, the triangle waveform must be used for the voice to be modulated. The properties of the modulating voice, other than its frequency, have no effect. As for SYNC, the voices pair thus:

MODULATED VOICE	MODULATING FREQUENCY
Voice 1	Voice 3
Voice 2	Voice 1
Voice 3	Voice 2

Ring Modulation - The Voice Pairings

Try out the effect of setting RING MOD by altering the SYNC program thus:

```
110 POKE 54276,21
```

Note again that for best results the two frequencies should be close to each other but not identical, and neither should be a multiple of the other.

OTHER SID REGISTERS

There are four registers on the SID chip which have not yet been mentioned. These are:

54297 POT X
54298 POT Y
54299 OSC 3
54300 ENV 3

POT X and POT Y are used with games paddles, and have no application to music. OSC 3 holds the level of the output signal from Voice 3, and ENV 3 contains the volume of Voice 3 as modified by the envelope. These registers can be used to modify the volume of the SID, or the frequency of one of the voices. To test the effects of this, modify the SYNC program by deleting lines 10, 20 and 500, and adding the following lines:

```
50    POKE 54273, 35
60    POKE 54287, 3
110   POKE 54276, 33
120   POKE 54290, 64
130   POKE 54272, PEEK(54299)
140   GET K$: IF K$="" THEN 130
```

The program uses the level of Voice 3, read from 54299, to alter the frequency of Voice 1. As it is written, the program uses the Pulse waveform in Voice 3, and modifies the low byte of the Voice 1 frequency. Try altering the waveform, by altering the number **POKE**d into 54290 in line 120 to 16, 32 or 128. The effects can be made much more dramatic by using OSC 3 to control the high byte of the Voice 1 frequency. Alter line 130 to:

```
130 POKE 54273, PEEK(54299)
```

The effect of using a noise waveform for Voice 3 is worth hearing!

To hear the effect of using ENV 3, make these modifications to the program:

```
80    POKE 54291, 170
100   POKE 54292, 0
130   POKE 54273, PEEK(54300)
```

Here the envelope of Voice 3 is used to control the high byte of the frequency of Voice 1. The Attack

and Decay values for Voice 3 are initially set to 10. Sustain is set to zero, and Release therefore has no effect. The values of Attack and Decay can be controlled by altering line 80. You will find that values below 8 for Attack and Decay are too fast for the effect to be noticeable.

We have now dealt with all the functions of the SID chip. The table in Appendix 7 summarises the functions of all the registers.

MAKING MUSIC

Now that we have dealt with the theory of Commodore 64 sound generation, we will look at the practical applications of the SID chip. The first and most obvious use of the synthesiser chip is as a synthesiser. The example program below turns your 64 into an electronic organ, giving you full control over the waveforms of the voices and the filtering:

```
10    REM ********************
20    REM ***                ***
30    REM *** SYNTHESISER ***
40    REM ***                ***
50    REM ********************
60    REM
70    REM

990   REM ***********************
992   REM * SET UP NOTE TABLE *
994   REM ***********************
996   REM

1000  DIM N(255,1)
1010  READ K
1020  IF K = 0 THEN 1100
1030  READ N(K,0), N(K,1)
1040  GOTO 1010
```

```
1090  REM
1092  REM ************************
1094  REM * CLEAR SID REGISTERS *
1096  REM ************************
1098  REM

1100  FOR L = 54272 TO 54296: POKE
      L, 0: NEXT
1110  SID = 54272
1120  VOL = 6
1130  FOR VV=1 TO 3: F$(VV)="N":
      NEXT

1990  REM
1992  REM ***************
1994  REM * FIRST MENU *
1996  REM ***************
1998  REM

2000  PRINT "{CLS}{CD}{CD}" TAB(15)
      "{RVS} SYNTHESISER {ROF}"
2010  PRINT: PRINT: PRINT "  VOICE
      {RVS}1{ROF}/{RVS}2{ROF}/{RVS}3
      {ROF}"
2020  PRINT: PRINT: PRINT "  FILTER
      {RVS}F{ROF}"
2030  PRINT: PRINT: PRINT "  PLAY
      {RVS}P{ROF}"
2100  GET K$: IF K$ = "" THEN 2100
2110  IF K$="F" THEN GOSUB 3000:
      GOTO2000
2120  IF K$="1"THEN VV=1: V=0: GOSUB
      2500: GOTO 2000
2130  IF K$="2"THEN VV=2: V=7: GOSUB
      2500: GOTO 2000
2140  IF K$="3"THEN VV=3: V=14:
      GOSUB 2500: GOTO 2000
2150  IF K$ = "P" THEN PRINT
      "{CLS}": GOSUB 5000: GOTO 2000
2160  IF K$ = "{F8}" THEN POKE
      S+24,0: END
```

```
2200  GOTO 2100

2490  REM
2492  REM ************************
2494  REM * SET VOICE PROPERTIES *
2496  REM ************************
2498  REM

2500  PRINT "{CLS}{CD}{CD}" TAB(12)
      "{RVS} VOICE PROPERTIES {ROF}"
2510  PRINT: PRINT "VOICE"VV;
2520  PRINT TAB(20) "WAS " W$(VV)
2530  PRINT: PRINT "WAVEFORM:"
      TAB(10) "{RVS}S{ROF}AWTOOTH"
2540  PRINT TAB(10)
      "{RVS}T{ROF}RIANGLE"
2550  PRINT TAB(10)
      "{RVS}P{ROF}ULSE"
2560  PRINT TAB(10)
      "{RVS}N{ROF}OISE"
2600  GET K$: IF K$ = "" THEN 2600
2610  IF K$ = "S" THEN W(VV) = 32:
      W$(VV) = "{RVS}SAWTOOTH": GOTO
      2800
2620  IF K$ = "T" THEN W(VV) = 16:
      W$(VV) = "{RVS}TRIANGLE": GOTO
      2800
2630  IF K$ = "N" THEN W(VV) = 128:
      W$(VV) = "{RVS}NOISE": GOTO
      2800
2640  IF K$ <> "P" THEN 2600
2650  W$(VV) = "{RVS}PULSE"
2700  PRINT: PRINT TAB(5) W$(VV):
      PRINT
2710  W(VV) = 64
2720  P = P(VV)
2730  PRINT "PULSE % (WAS" P(VV)
      ")";: INPUT P
2740  IF P < 0 OR P > 100 THEN PRINT
      "{CU}";: GOTO 2710
2750  P(VV) = P
```

```
2760  P = P(VV)*40.95
2770  POKE SID+V+2, (P AND 255)
2780  POKE SID+V+3, INT(P/256)
2790  GOTO 2810
2800  PRINT: PRINT TAB(5) W$(VV):
      PRINT
2810  PRINT "ATTACK   (WAS"
      A(VV)")";: INPUT A(VV): A =
      (A(VV) AND 15)*16
2820  PRINT "DECAY    (WAS"
      D(VV)")";: INPUT D(VV): D =
      (D(VV) AND 15)
2830  PRINT "SUSTAIN (WAS" S(VV)
      ")";: INPUT S(VV): S=(S(VV)
      AND 15)*16
2840  PRINT "RELEASE (WAS" R(VV)
      ")";: INPUT R(VV): R=(R AND
      15)
2850  POKE SID+V+5,A+D
2860  POKE SID+V+6,S+R
2870  PRINT"{CD}FILTER
      {RVS}Y{ROF}/{RVS}N{ROF} (WAS "
      F$(VV) " )";: INPUT F$(VV)
2880  IF F$(VV)="Y" THEN RF=RF OR
      (2↑(VV- 1)):GOTO2910
2890  IF F$(VV) < > "N" THEN
      PRINT"{CU}{CU}{CU}":GOTO 2870
2900  RF=RF AND (255-2↑(VV-1))
2910  POKE SID+23,RF
2950  RETURN

2990  REM
2992  REM ************************
2994  REM * SET FILTER PROPERTIES *
2996  REM ************************
2998  REM

3000  PRINT "{CLS}{CD}{CD}" TAB(12)
      "{RVS}FILTER PROPERTIES"
```

```
3010  PRINT: PRINT "HIGH PASS
      {RVS}Y{ROF}/{RVS}N{ROF} (WAS
      "HP$" )";: INPUT HP$
3020  VOL = VOL AND 191
3030  IF HP$="Y" THEN VOL=VOL OR 64
3040  PRINT: PRINT "BAND PASS
      {RVS}Y{ROF}/{RVS}N{ROF} (WAS
      "BP$" )";: INPUT BP$
3050  VOL=VOL AND 223
3060  IF BP$="Y"THENVOL=VOL OR 32
3070  PRINT: PRINT "LOW  PASS
      {RVS}Y{ROF}/{RVS}N{ROF} (WAS
      "LP$" )";:INPUT LP$
3080  VOL = VOL AND 239
3090  IF LP$="Y" THEN VOL=VOL OR 16
3100  PRINT: PRINT "FILTER FREQUENCY
      (WAS " FQ " )";: INPUT FQ
3110  IF FQ > 2048 OR FQ < 0 THEN
      PRINT "{CU}{CU}{CU}": GOTO
      3100
3120  POKE SID+21,FQ AND 7
3130  POKE SID+22,INT(FQ/8)
3150  PRINT: PRINT"RESONANCE
       (WAS " RS " )";: INPUT RS
3160  IF RS < 0 OR RS > 15 THEN 3150
3170  POKE S+23,(RF AND 15) OR 16*RS
3500  RETURN

4990  REM
4992  REM      ********
4994  REM      * PLAY *
4996  REM      ********
4998  REM

5000  PRINT "{HOM}{CD}{CD}" TAB(18)
      "RPLAY"
5010  PRINT
      "{HOM}{CD}{CD}{CD}{CD}{CD}{CD}
      {CD}  PLAYING VOICE" VV
5020  PRINT "{CD}{CD}{CD}{CD}{CD}
      {RVS}F1{ROF} VOICE 1"
```

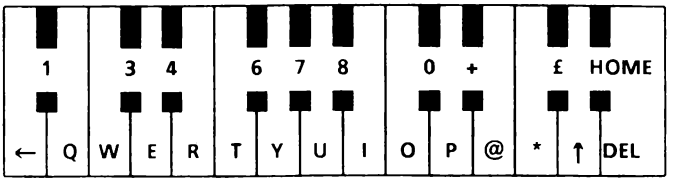
```
5030  PRINT: PRINT "  {RVS}F3{ROF}
      VOICE 2"
5040  PRINT: PRINT "  {RVS}F5{ROF}
      VOICE 3"
5050  POKE SID+24,VOL
5060  PRINT "{HOM}{CD}{CD}{CD}{CD}
      VOLUME"VOL AND 15"{CL} "
5070  GET K$: IF K$="" THEN 5070
5080  HI=N(ASC(K$),1)
5090  IF HI=0 THEN 5150
5100  POKE SID+V+4, W(VV)
5110  POKE SID+V+1, HI
5120  POKE SID+V, N(ASC(K$),0)
5130  POKE SID+V+4, W(VV)+1
5140  GOTO 5070
5150  VX=VOL AND 15
5160  IF K$ <> "{CU}" THEN 5180
5170  IF VX < 15 THEN VX = VX+1:
      GOTO 5200
5180  IF K$ <> "{CD}"THEN 5300
5190  IF VX > 0 THEN VX  = VX-1
5200  VOL = (VOL AND 240) OR VX
5210  GOTO 5050
5300  IF K$="{F1}" THEN POKE
      SID+V+4, W(VV): VV=1: V=0:
      GOTO 5010
5310  IF K$="{F3}" THEN POKE
      SID+V+4, W(VV): VV=2: V=7:
      GOTO 5010
5320  IF K$="{F5}" THEN POKE
      SID+V+4, W(VV): VV=3: V=14:
      GOTO 5010
5400  IF K$=CHR$(13) THEN POKE
      SID+24, 0: RETURN
5500  GOTO 5070

60000 DATA 81,233,7, 87,97,8,
      51,225,8
60010 DATA 69,104,9, 52,247,9,
      82,143,10
```

```
60020 DATA 84,48,11, 54,218,11,
      89,143,12
60030 DATA 55,78,13, 85,24,14,
      56,239,14
60040 DATA 73,210,15, 79,195,16,
      48,195,17
60050 DATA 80,209,18, 43,239,19,
      64,31,21
60060 DATA 42,96,22, 92,181,23,
      94,30,25
60070 DATA 19,156,26, 20,49,28
60080 DATA 95,12,7, 49,119,7
60100 DATA 0
```

The program gives you control over the settings of the three voices and the filter, so that you can experiment with different tones. The method of processing the key strokes and turning them into notes uses a two dimensional array to hold the values of the high and low frequency bytes of each note. The array has 255 elements, corresponding to the ASCII codes of the keys (not all of which are used). This allows the note to be played after only one **IF...THEN** test, to see if the key pressed was a valid one. This use of an array is wasteful of memory; about 1K of memory is reserved for the array and not used; but it has the advantage of speed, as a quick response to the keyboard is essential in this application.

The top two rows of the 64's keyboard are used as the 'organ' keyboard. The diagram overleaf shows the relation of the 64 keyboard to the organ keyboard.

The information relating the keys to the notes is held in the **DATA** statements at the end of the program (lines 60000-60100). The data is loaded into the array N(255,1) by lines 1000 to 1040. Lines 1100 to 1130 initialise the program variables. The first menu page is controlled by the



lines from 2000 onward. The program offers the choice between altering the properties of a voice, setting the filter or playing a tune. You must of course set a voice before trying to play anything. Pressing F8 (the SHIFT key and function key 7) will end the program.

Lines 2500 to 2950 give you control over the properties of the three voices. You may enter new settings, or keep the existing settings by simply pressing RETURN. Lines 3000 to 3500 allow you to set up the filter in a similar manner.

The routine in lines 5000 to 5500 handles the playing of notes. The top two rows of the keyboard act as the synthesiser keyboard. You can switch between voices using the function keys F1, F3 and F5. The volume is controlled by the Cursor Up and Down key - Cursor Up for louder, Cursor Down for softer.

CHAPTER 12

CHARACTER GRAPHICS

The simplest way to produce graphics displays on the Commodore 64 is to use the special characters available from the keyboard. These characters can be obtained by holding down either a SHIFT key and pressing the character key to obtain the right-hand symbol of the two shown on the face of the key, or by holding down the Commodore logo key and pressing the character key to obtain the left-hand symbol of the two. You can produce simple but effective displays by using these symbols and the cursor control characters in PRINT commands. Try this short program. Shifted characters are represented by {X}, characters requiring the Commodore key are shown as [X]. The program should print a smiling face. The symbol ₀ is used to represent a space:

```
10      PRINT"{O}[YYYYY]{P}
20      PRINT"[H]{W}₀₀₀{W}[N]
30      PRINT"[H]₀₀{Q}₀₀[N]
40      PRINT"[H]₀₀₀₀₀[N]
50      PRINT"{M}₀{J*K}₀[N]
60      PRINT"₀{M}[PPP]{N}
```

CHARACTERS AND NUMBERS

In Chapter 7 the ASCII code for representing characters by numbers was introduced. The screen displays of the Commodore 64 are stored in the memory of the machine as a block of 1000 memory locations, each holding a number which indicates the character at the corresponding point on the

screen. This method of storing the display data is called 'memory mapping', as the screen memory area acts as a map of the display. A second area of memory, also of 1000 locations, holds information on the colour of each character. (There are diagrams - or 'maps' - of the screen memory areas in Appendix 9).

The numbers used to indicate the characters on the screen are not the same as those used by the **CHR$** command described in Chapter 7, as the ASCII character set includes control characters such as the cursor characters while the screen character set uses all 256 numbers to represent displayable symbols. There are two sequences of characters, one for displaying upper case letters and graphics, the other for upper and lower case displays. The two character sets and their numbering systems are shown in the table in Appendix 10. The numbers for the colours are shown in the table below. The numbers are the same as those given in Chapter 3 for the background and border colours, and are shown again in the table opposite and in Appendix 5.

In normal use, the screen memory of the Commodore 64 is in store locations 1024 to 2023, while the colour memory extends from location 55296 to 56295. Using this information you can display characters on the screen without using **PRINT**, by **POKE**ing the character number into screen memory, and **POKE**ing a number into the corresponding location of the colour memory. For example:

```
POKE 1024, 33: POKE 55296, 1
```

would display a white exclamation mark at the top left corner of the screen. This method of controlling the display gives you better control over the positioning of the symbols on the screen than the

PRINT command, and can make altering a display much faster.

CODE	COLOUR	CODE	COLOUR
0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light Red
3	Cyan	11	Dark Grey
4	Purple	12	Mid Grey
5	Green	13	Light Green
6	Blue	14	Light Blue
7	Yellow	15	Light Grey

The Colour Codes

CODEBREAKER

This program plays the well known game. You have ten attempts to find out the secret code which the 64 generates, guided by the clues which the program displays after each attempt. Simple character graphics are used to create the display.

```
1       REM ***************
2       REM *               *
3       REM * CODEBREAKER *
4       REM *               *
5       REM ***************
6       REM

7       CM=55296: SM=1024
10      POKE 53280, 11: POKE 53281, 12
```

```
20    PRINT "{CLS}{CD}" TAB(14)
      "{BLU}CODEBREAKER{GRY}":
      PRINT: PRINT
30    PRINT "THE COMPUTER WILL
      SELECT A CODE OF FOUR"
35    PRINT "COLOURS, CHOSEN FROM 8.
      EACH COLOUR MAY";
40    PRINT "BE USED MORE THAN
      ONCE:";
41    PRINT " {RVS}{BLK} {ROF}
      {RVS}{GRN} {ROF} {RVS}{RED}
      {ROF} {RVS}{BLU} {ROF}{GRY}":
      PRINT
45    PRINT "YOU MUST CRACK THE CODE
      IN LESS THAN 10"
50    PRINT "ATTEMPTS": PRINT
55    PRINT "AFTER EACH ATTEMPT YOU
      WILL BE GIVEN A"
60    PRINT "CLUE OF UP TO 4 BLACK
      AND WHITE DISCS:": PRINT
65    PRINT "A {BLK}●{GRY} MEANS A
      CORRECT COLOUR IN THE RIGHT
      PLACE.": PRINT
70    PRINT "A {WHI}●{GRY} MEANS A
      CORRECT COLOUR IN THE WRONG
      PLACE.": PRINT
75    PRINT "USE THE KEYS 1 TO 8 TO
      ENTER YOUR GUESS": PRINT:
      PRINT
85    PRINT TAB(9) "{RED}{RVS}PRESS
      ANY KEY TO START{ROF}";
90    GETA$: IF A$="" THEN 90
100   GOSUB 1000
110   FOR G=0 TO 9: G$="": CO$=C$
115   PRINT TAB(10) G+1;
120   GOSUB 2000
130   NEXT
140   Z$ = "{RVS}{RED}   YOU LOSE
      {ROF}": GOTO 2420

993   REM
```

```
994   REM ************
995   REM *            *
996   REM * SET CODE   *
997   REM *            *
998   REM ************
999   REM

1000  C$ = ""
1010  FOR I=1 TO 4
1020  C$ = C$ + RIGHT$( STR$( INT(
      RND(0)*8)+1),1)
1030  NEXT I
1040  PRINT "{CLS}{CD}" TAB(14)
      "{BLU}CODEBREAKER{GRY}":
      PRINT: PRINT
1050  RETURN

1993  REM
1994  REM ***************
1995  REM *               *
1996  REM * INPUT GUESS   *
1997  REM *               *
1998  REM ***************
1999  REM

2000  GN$ = "INPUT GUESS " +
      RIGHT$(STR$(GUESS),1)
2020  FOR I=0 TO 3
2030  GET I$
2040  IF I$<"1" OR I$>"8" THEN 2030
2050  POKE CM+176+(I*2)+(G*80),
      VAL(I$)-1
2060  POKE SM+176+(I*2)+(G*80), 160
2080  G$ = G$+I$
2090  NEXT I

2092  REM
2093  REM ********************
2094  REM *                    *
2095  REM * RIGHT COLOUR       *
2096  REM * AND RIGHT PLACE    *
```

```
2097  REM *                    *
2098  REM ********************
2099  REM

2100  FOR I=1 TO 4
2110  IF MID$(G$,I,1) <>
      MID$(CODE$,I,1) THEN 2150
2120  C1 = C1+1
2130  G$ = LEFT$(G$,I-1) + "9" +
      MID$(G$,I+1)
2140  CO$ = LEFT$(CO$,I-1) + "9" +
      MID$(CO$,I+1)
2150  NEXT I

2192  REM
2193  REM ********************
2194  REM *                    *
2195  REM * RIGHT COLOUR      *
2196  REM * BUT WRONG PLACE   *
2197  REM *                    *
2198  REM ********************
2199  REM

2200  FOR I=1 TO 4
2210  FOR J=1 TO 4
2220  IF MID$(G$,I,1) =MID$(CO$,J,1)
      AND MID$(G$,I,1)<>"9" THEN
      C2=C2+1: GOTO 2240
2230  GOTO 2260
2240  G$ = LEFT$(G$,I-1) + "9" +
      MID$(G$,I+1)
2250  CO$ = LEFT$(CO$,J-1) + "9" +
      MID$(CO$,J+1)
2260  NEXT J
2270  NEXT I

2293  REM
2294  REM ****************
2295  REM *                *
2296  REM * DISPLAY CLUE   *
2297  REM *                *
```

```
2298  REM ****************
2299  REM

2300  IF C1=0 THEN 2320
2310  FOR I=1 TO C1: CLUE$= CLUE$ +
      "{BLK}●{GRY}": NEXT
2320  IF C2=0 THEN 2340
2330  FOR I=1TO C2: CLUE$ = CLUE$ +
      "{WHI}●{GRY}": NEXT
2340  PRINT TAB(30) CLUE$ "{CD}"
2350  IF C1=4 THEN 2400
2360  C1=0: C2=0: CLUE$="": RETURN

2393  REM
2394  REM ***************
2395  REM *               *
2396  REM * REVEAL CODE *
2397  REM *               *
2398  REM ***************
2399  REM

2400  Z$ = "{RVS}{RED}    YOU WIN
      {ROF}"
2420  PRINT " THE CODE WAS  ";
2430  FOR I=1 TO 4
2440  POKE CM+974+(I*2),
      VAL(MID$(C$,I,1))-1
2445  POKE SM+974+(I*2), 160
2450  NEXT
2460  PRINT TAB(27) "ANOTHER Y/N?";
2465  PRINT "{HOM}{CD}" TAB(13) Z$;
2470  GETA$: IF A$="" THEN 2470
2480  IF A$="Y" THEN 100
2490  PRINT "{CLS}": END
```

Lines 10 to 85 display the instructions with and example. (The solid circle character in lines 65 and 70 - and lines 2310 and 2330 - is obtained by holding down either SHIFT key and pressing Q.) Lines 1000 to 1050 set up a code of four numbers

between 1 and 8. Each number may appear more than once in the code.

You enter your guess in lines 2000 to 2090, pressing the keys between 1 and 8. After each keypress the appropriate colour code is placed in the colour memory, and a reversed space (character 160) is placed in screen memory. After four consecutive keypresses the guess is checked. Lines 2100 to 2150 check for a correct colour in the right place. C1 is incremented for every such item in the guess, and the item is set to '9' to indicate that there is no need to check it further.

Lines 2200 to 2250 check for an item of the right colour in the wrong place. Every such item in the guess is set to '9' once it has been recorded to avoid considering it more than once.

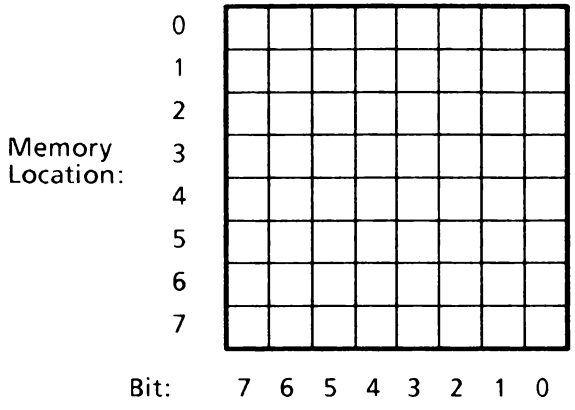
The routine at 2300 to 2360 builds up a string CLUE$ containing a black spot for every correct colour in the right place, and a white spot for every correct colour in the wrong place. Note that the clue gives no information on the position of the correct items - that would make things too easy!

If you crack the code, or have run out of guesses, the routine at 2400 to 2480 lets you know and reveals the code. You may then play another game, or stop the program.

CHARACTER DEFINITIONS

The shapes of all the displayable characters are defined by numbers held in a special area of memory. The 64 allows you to create new symbols by entering new definitions for the characters to replace those held in memory. Each symbol in the character set is defined by the contents of eight memory locations, which correspond to a grid of eight rows; each row consisting of eight dots which

may be set to the foreground or the background colour. (Each of these dots is called a *pixel*, which is



The Character Grid

a contraction of *picture element.* The display is built up from 64000 of these pixels.) Each location defines one row of the character, and each bit of the number in the location represents one pixel. If the bit is set to 1, the pixel is displayed in the foreground colour; if the bit is set to 0 the pixel remains in the background colour.

In normal use, the character definitions are held in 4K of read-only memory (ROM) beginning at location 53248. There are two sets of 256 characters: this first set includes graphics symbols and upper case letters; the second set (which begins at 55296) has both upper and lower case letters, with fewer graphics characters. To find the eight bytes defining any character, multiply the code number of the character by eight, and add the start location of the character set:

Character location = (Code * 8) + Start of set

So, the definition of the letter X (character code 24 - see Appendix 10) in the first character set is given by:

$$\text{Location} = (24 * 8) + 53248 = 53440$$

There is one problem to be overcome before you can read the character definitions. The Commodore 64 uses the memory area at 53248 for two purposes; to hold the character definitions and also to hold some of the input/output routines of the operating system. (This doesn't mean there are two sets of numbers in each location: there are actually two units of memory which take turns in using the same address.) To read the character memory, you must first disable the input/output activities and switch in the character ROM. This can only be done within a program, as the keyboard will not work once you have disabled the input/output routines.

To disable input/output and switch in character ROM, use the two commands:

```
POKE 56334, PEEK(56334) AND 254
POKE 1, PEEK(1) AND 251
```

To return to normal after reading the character memory, use the commands:

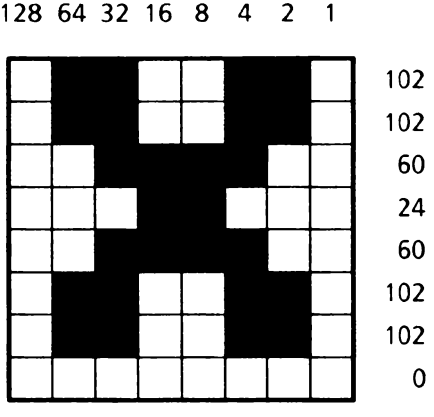
```
POKE 1, PEEK(1) OR 4
POKE 56334, PEEK(56334) OR 1
```

This short program will display the eight numbers defining the X.

```
10    POKE 56334, PEEK(56334) AND
      254
20    POKE 1, PEEK(1) AND 251
30    FOR A = 0 TO 7
40    C(A) = PEEK(A+53440)
```

```
50    NEXT A
60    POKE 1, PEEK(1) OR 4
70    POKE 56334, PEEK(56334) OR 1
100   FOR B=0 TO 7
110   PRINT C(B)
120   NEXT B
```

The program should display the numbers 102, 102, 60, 24, 60, 102, 102, 0. Having found the eight



The Character X

numbers, we can convert them to binary form and fill in the grid to produce a letter X.

Notice that the bottom line of the character is left blank. This is to leave a space between lines of text on the screen. You can inspect the other characters by modifying the character number used by the program.

REDEFINING CHARACTERS

You may be wondering how it is possible to create new characters when the standard character definitions are held in ROM - it's difficult to write

new numbers into read-only memory! The answer is that we have to tell the 64 to look somewhere else for the character shapes, and define the new characters in RAM. The video of the 64 is all controlled by the Video Interface Chip (or VIC), and this chip has control registers which tell it where to find the screen and the character sets. To make use of user defined characters you must first copy all the characters in a set into the RAM, and tell the VIC chip where to find them. You can then alter some (or all) of the characters to your own designs.

The control register of the VIC which holds the position of the character set is at memory location 53272. Only the lower 4 bits of the register form the character pointer, the other four are used to record the position in memory of the screen. The VIC chip can only look at 16K of the memory at once, so the character set and the screen must be in the same 16K bank. As the 64 has 64K of memory, there are four possible alternatives. Which of the four banks is addressed by the chip is set by the sequence:

```
POKE 56578, PEEK(56578) OR 3
POKE 56576,(PEEK(56576)AND 252) OR X
```

where X is a number from 0 to 3, with the meanings shown in the table opposite.

We said above that in normal use the screen memory is at 1024, and the characters begin at 53248. These numbers are definitely not in the same 16k bank, but this is a special case. The ROM holding the standard symbols is in fact very cunningly arranged to appear in two other places as well: at 4096 (in the first bank) and at 40864 (in the third). Because of the organisation of the hardware, the VIC chip can find the ROM at three

X	BANK	MEMORY RANGE
3	0	0 - 16383
2	1	16384 - 32767
1	2	32768 - 49151
0	3	49152 - 65535

VIC Bank Selection

places, while the processor uses these memory areas for something else! (There is a diagram of the 64's memory - a *memory map* - in Appendix 11.)

Having set the bank you want the VIC to address, you can set the position of the character set by the command:

```
POKE 53272,(PEEK(53272)AND 240) OR C
```

C must be an even number between 0 and 14, and has the meaning shown in the table overleaf.

It is best to move the character memory by using a program, because if you try moving it by typing the command at the keyboard all the displayed characters will turn into garbage. The next program alters the position of character memory to 14336, and copies the first character set into these locations:

```
100   REM MOVE CHARACTER SET
110   POKE 52,56: POKE 56,56
120   POKE 53272, (PEEK(53272) AND
      240) OR 14
130   POKE 56334, PEEK(56334) AND
      254
140   POKE 1, PEEK(1) AND 251
150   FOR X = 0 TO 2047
160   POKE 14336+X, PEEK(53248+X)
```

C	START OF CHARACTER MEMORY			
	BANK 0	BANK 1	BANK 2	BANK 3
0	0	16384	32768	49152
2	2048	18432	34816	51200
4	4096	20480	36864	53248
6	6144	22528	38912	55296
8	8192	24576	40960	57344
10	10240	26624	43008	59392
12	12288	28672	45056	61440
14	14336	30720	47104	63488

Character Memory Positions

```
170   NEXT X
180   POKE 1, PEEK(1) OR 4
190   POKE 56334, PEEK(56334) OR 1
```

There is an extra pair of **POKE** commands in line 110 of the program which we have not dealt with before. These reduce the amount of the memory used by the BASIC program so that it does not store data over the new character set.

If you run this program, you should see any characters on the screen turn to garbage when the memory pointers are switched and then return to normal as the character data is copied into the new area. You will now be able to modify the characters by **POKE**ing new values into the memory. With the character set in the new position, the formula for finding a character will be:

Character Location = 14336 + (8*Character Code)

If you add these lines to the program and then **RUN** 1000, all the characters will be turned upside down - you have redefined them!

```
200   END
1000  REM  INVERT CHARACTERS
1010  FOR X = 14336 TO 16376 STEP 8
1020  FOR Y = 0 TO 7
1030  C(Y)=PEEK(X+Y)
1040  NEXT Y
1050  FOR Z=0 TO 7
1060  POKE X+7-Z, C(Z)
1070  NEXT Z
1080  NEXT X
```

RUNning the first program again will restore the original characters at the new location.

MOVING THE SCREEN

Moving the character memory to the first 16K bank has the advantage that you don't have to move the screen memory, but it means that you can not use more than 12K of the memory for BASIC programs, while 24K lies unused. To make better use of the memory you must move both the character memory and the screen to a higher bank. (Refer to the memory map in Appendix 11.)

The screen position within a bank is controlled by the four most significant bytes of the VIC control register at location 53272. To move the screen to another location, use the command:

```
POKE 53272, (PEEK(53272)AND 15) OR S
```

S indicates the location within the bank of the beginning of screen memory and has the meaning given in the table overleaf.

S	START OF SCREEN MEMORY			
	BANK 0	BANK 1	BANK 2	BANK 3
0	0	16384	32768	49152
16	1024	17408	33792	50176
32	2048	18432	34816	51200
48	3072	19456	35840	52224
64	4096	20480	36864	53248
80	5120	21504	37888	54272
96	6144	22528	38912	55296
112	7168	23552	39936	56320
128	8192	24576	40960	57344
144	9216	25600	41984	58368
160	10240	26624	43008	59392
176	11264	27648	44032	60416
192	12288	28672	45056	61440
208	13312	29696	46080	62464
224	14336	30720	47104	63488
240	15360	31744	48128	64512

Screen Memory Positions

When you move the screen, as well as telling the VIC chip where it is, you must tell the 64's operating system where it has gone, so that the **PRINT** command will write to the correct area. Location 648 records the address of the beginning of the screen, and must be **POKE**d with the address of the screen divided by 256, thus:

```
POKE 648, (Screen Start)/256
```

CHARACTER GENERATOR PROGRAM

This program allows you to design new characters on the screen and add them to the character set. The characters are created by filling in a large grid on the screen, which is then used to program the character.

```
10    REM ************************
11    REM *                      *
12    REM * CHARACTER GENERATOR *
13    REM *                      *
14    REM ************************

100   POKE 52,108: POKE 56,108: CLR
110   GOSUB 20000: REM SETTING UP
      ROUTINES
120   GOTO 10000: REM  MAIN MENU

190   REM ************************
192   REM * REDEFINE CHARACTERS *
194   REM ************************

200   EX=0: PRINT "{CLS}"
210   INPUT "CHARACTER CODE"; C
220   IF C < 0 OR C > 255 THEN 210
230   SX = 28672+C*8
240   SC = 27648:CC= SC+331
295   REM  STORE CHARACTER DATA IN
      ARRAY
300   PRINT "{CLS}"
310   FOR R=1 TO 8:GOSUB 1000:NEXT R
395   REM  DISPLAY CHARACTER
400   GOSUB 2000
410   IF F=1 THEN END
420   X=1: R=1
500   B = PEEK(CC)
510   REM  MODIFY CHARACTER
520   GOSUB 3000: IF EX = 1 THEN
      RETURN
530   IF CL=1 THEN CL=0: GOTO 400
```

```
540   CC = SC+X+40*(R+7)+10
550   GOTO 500

990   REM ************************
992   REM * STORE CHARACTER DATA *
994   REM *      IN ARRAY         *
996   REM ************************

1000  N = PEEK( SX+R-1 )
1010  X=8
1020  A%(R,9) = N: REM  VALUE OF ROW
1030  REM STORE BIT PATTERN
1040  IF N/2 = INT(N/2) THEN
      A%(R,X)=0: GOTO 1050
1045  A%(R,X) = 1
1050  N = INT(N/2): X=X-1
1060  IF N >=1 THEN 1040
1070  FOR I=X TO 1 STEP -1
1080  A%(R,I) = 0
1090  NEXT I
1100  RETURN

1990  REM **********************
1992  REM * DISPLAY CHARACTER *
1994  REM **********************

2000  PRINT "{CLS}";
2003  FOR I=55627 TO 55907 STEP 40
2005  FOR II=0 TO 7: POKE I+II, 1:
      NEXT: NEXT
2010  FOR R=1 TO 8
2020  FOR X=1 TO 8
2030  IF A%(R,X)=1 THEN CS = 160:
      GOTO 2040
2035  CS=46
2040  POKE SC+(X+10+(R+7)*40), CS
2050  NEXT X
2060  POKE 214, R+6: PRINT: PRINT
      TAB(24) STR$(A%(R,9))
2070  NEXT R
```

```
2080  RETURN

2990  REM ********************
2992  REM * MOVE CURSOR AND *
2994  REM *  CHANGE PIXELS  *
2996  REM ********************

3000  REM CHECK KEYBOARD
3010  GET K$
3020  REM FLASH CURSOR
3030  CH = PEEK(CC): C%=(CH OR 128)-
      (CH AND 128)
3040  POKE CC, C%:FOR I1=1 TO
      100:NEXT

3050  REM       CHECK KEY INPUT
3060  IF K$="" THEN 3010
3080  IF K$="{CL}" AND X>1 THEN POKE
      CC, B: X=X-1: RETURN: REM LEFT
3090  IF K$="{CR}" AND X<8 THEN POKE
      CC,B: X=X+1: RETURN: REM RIGHT
3100  IF K$="{CD}" AND R<8 THEN POKE
      CC,B: R=R+1: RETURN: REM DOWN
3110  IF K$="{CU}" AND R>1 THEN POKE
      CC,B: R=R-1: RETURN: REM UP
3120  IF K$=" " THEN A%(R,X)  =  1-
      A%(R,X): GOTO 3200: REM CHANGE
      PIXEL
3130  IF K$="P" THEN 4000:REM
      REPROGRAM CHARACTER
3140  IF K$="{CLS}" THEN 3300:REM
      CLEAR CHARACTER
3150  IF K$="{HOM}" THEN X=1: R=1:
      POKECC,B: RETURN
3160  RETURN

3190  REM ****************
3192  REM * CHANGE PIXEL *
3194  REM ****************
```

```
3200  IF B=160 THEN POKE CC, 46:
      RETURN
3210  POKE CC, 160: RETURN

3290  REM ********************
3292  REM *  CLEAR CHARACTER *
3294  REM ********************

3300  FOR R=1 TO 8: FOR X=1 TO 9
3310  A%(R,X)=0
3320  NEXT:NEXT
3330  CL=1:RETURN

3990  REM ************************
3992  REM * REPROGRAM CHARACTER *
3994  REM ************************

4000  M$ = "REPROGRAMMING CHARACTER"
      + STR$(C)
4010  POKE 214,21: PRINT: PRINT
      TAB(5) M$
4020  FOR R=1 TO 8
4030  FOR X=1 TO 8
4040  D = D+(2↑(8-X)) * A%(R,X)
4050  NEXT X
4060  POKE SX+R-1, D
4070  D=0: NEXT R
4080  EX=1
4100  RETURN

9990  REM *************
9992  REM * MAIN MENU *
9994  REM *************

10000 PRINT "{CLS}{CD}{CD}" TAB(12)
      "CHARACTER GENERATOR"
10010 PRINT: PRINT: PRINT
      "{RVS}C{ROF}HANGE CHARACTERS"
10020 PRINT: PRINT: PRINT
      "{RVS}L{ROF}OAD CHARACTER SET
      FROM TAPE"
```

```
10030 PRINT: PRINT: PRINT
      "{RVS}S{ROF}AVE CHARACTER SET
      TO TAPE"
10040 PRINT: PRINT: PRINT
      "{RVS}F1{ROF}  END PROGRAM"

10090 REM  READ KEYBOARD
10100 GET K$: IF K$="" THEN 10110
10110 IF K$="C" THEN GOSUB 200: GOTO
      10000: REM  CHANGE CHARACTERS
10120 IF K$="L" THEN GOSUB 11000:
      GOTO 10000: REM  LOAD
      CHARACTER SET
10130 IF K$="S" THEN GOSUB 12000:
      GOTO 10000: REM  SAVE
      CHARACTER SET
10140 IF K$="{F1}" THEN END
10150 GOTO 10100

10990 REM ************************
10992 REM * LOAD CHARACTER SET *
10994 REM *      FROM TAPE      *
10996 REM ************************

11000 PRINT "{CLS}{CD}{CD}": OPEN
      1,1,0, "CHARACTERS"
11010 FOR C=0 TO 2047
11020 GET#1, C$: IF C$="" THEN
      C$=CHR$(0)
11030 POKE 28672+C, ASC(C$)
11040 NEXT C
11100 CLOSE 1
11500 RETURN

11990 REM ************************
11992 REM * SAVE CHARACTER SET *
11994 REM ************************

12000 PRINT "{CLS}{CD}{CD}": OPEN
      1,1,1, "CHARACTERS"
12010 FOR C=0 TO 2047
```

```
12020 C$ = CHR$(PEEK(28672+C));
12030 PRINT#1, C$
12040 NEXT C
12100 CLOSE 1
12500 RETURN

19990 REM ************************
19995 REM * SETTING UP ROUTINES *
20000 REM ************************

20005 DIM A%(8,9)
20010 POKE 56578, (PEEK(56578) AND
      252) OR 3
20020 POKE 56576, (PEEK(56576) AND
      252) OR 2
20030 POKE 53272, (PEEK(53272) AND
      240) OR 12
20040 POKE 53272, (PEEK(53272) AND
      15) OR 176
20050 POKE 648, 108
20060 PRINT
      "{CLS}{CD}{CD}{CR}{CR}COPYING
      CHARACTER SET"
20070 FOR I=0 TO 255: POKE 28048+I,
      I: POKE 55696+I, 3: NEXT
20080 POKE 56334, PEEK(56334)AND 254
20090 POKE 1, PEEK(1) AND 251
20100 FOR I=0 TO 2047
20110 POKE 28672+I, PEEK(53248+I)
20120 NEXT I
20150 POKE 1, PEEK(1) OR 4
20160 POKE 56334, PEEK(56334) OR 1
20500 RETURN
```

The program allows you to redefine any character, and to save character sets on tape and reLOAD them. You can use the new character sets with other programs.

The first thing the progam does is copy the character set into RAM, and move the screen

memory to give a large amount of memory still free for use by BASIC. Lines 20005 to 20500 do this, displaying the character set as it is copied. Lines 10000 to 10150 then display the menu, giving you the option to change a character or to **LOAD** or **SAVE** a character set. Pressing the function key F1 stops the program.

The redefining of characters is controlled by the routine from 200 to 550. When you input the code of the character you wish to alter, the line 310 calls the routine at lines 1000 to 1100 to store the data of the character in an array. The character is then displayed on the screen by the routine at 2000.

The modification of the character is performed by the section of program at 3000. You may move the cursor around the character grid with the cursor keys. The HOME key returns the cursor to the top left corner of the grid. To erase all the data for a character, use the CLR key.

When you have finished designing the new character, pressing P will store the new definition in character memory. Lines 4000 to 4100 perform this function.

SAVEing a character set onto tape is performed by the subroutine at 12000. The character sets are reloaded into the machine by the routine at 11000. The bytes of data are stored on the tape as single character string variables, as these take less tape than integer or real variables. When loading from tape, the 64 reads a CHR$(0) as a null character, so the test in line 11020 is included to correct this.

To use the new characters you create with other programs, use this program to create the characters, or to **LOAD** them from tape, then stop the program and type NEW. The new character set

will remain, and you can **LOAD** and **RUN** other programs.

WARNING

Do not reset the 64 by using RUN/STOP and RESTORE. The new characters will be lost, but the screen will not be properly reset, and you will not be able to see what you are typing.

MULTI COLOUR CHARACTERS

In multi colour character mode, four colours may be used for each character, instead of the two allowed in normal character mode. The horizontal resolution (but not the width) of the characters is reduced to half that of a normal character - a multi colour character has four columns of definable pixels which are twice as wide as normal. Multi colour character mode is enabled by setting bit 4 of the VIC control register at location 53270, as follows:

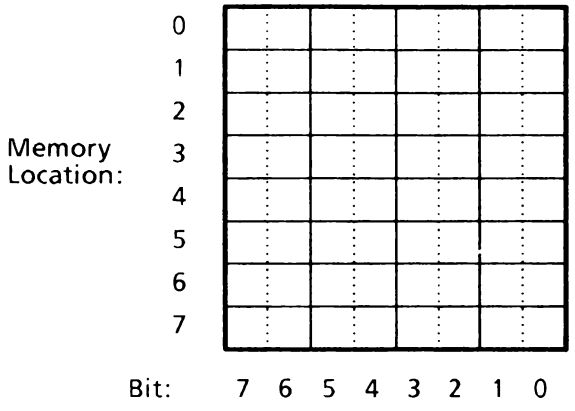
```
POKE 53270, PEEK(53270) OR 16
```

and disabled by:

```
POKE 53270, PEEK(53270) AND 239
```

Once multi colour character mode is enabled for the whole display, each character may be set to multi colour character mode or to standard mode. If the foreground colour of the character is between 0 and 7, the character appears as normal. If the foreground colour is 8 - 15, the character is displayed in multi colour mode.

A multi colour character is defined in a similar way to a normal character by eight bytes of the character memory, which represent a grid of eight rows. In multi colour character mode, however,



The Multi Colour Character Grid

each pixel in the character corresponds to two bits of the memory location defining the row. There are four possible combinations of two bits - 00, 01, 10, and 11 - and each combination specifies a different colour. If the bit pair of a pixel is set to 00, the colour used is the screen background colour, referred to as Background 0, which is defined by location 53281. The combination 01 specifies the colour stored in the Background 1 register at 53282, and 10 gives the colour in Background 2 at 53283. If the bit pair is set to 11, the pixel is set to the colour specified by the lower three bits of the colour memory for the character (the fourth bit sets multi colour character mode on or off for the character). The colour memory extends from 55296 to 56295.

A final variant of character graphics, less complex than multi colour character mode and very much easier to use, is Extended Background Colour Mode.

EXTENDED BACKGROUND COLOUR MODE

Extended background colour mode allows you to use four different background colours for the characters, instead of just one. Extended background colour mode is enabled by setting bit 6 of the VIC control register at 53265, by the command:

```
POKE 53265, PEEK(53265) OR 64
```

and is disabled by:

```
POKE 53265, PEEK(53265) AND 191
```

When extended background colour mode is set, you are restricted to using the first 64 characters of the character set, as the two highest bits of the character code in screen memory are used to control the background colour of the character. The colours are stored in locations 53281 to 53284.

CHARACTER	COLOUR CODE	COLOUR REGISTER
0 - 63	00	53281
64 - 127	01	53282
128 - 191	10	53283
192 - 255	11	53284

Try this short routine to see the effect of extended background colour mode:

```
5     POKE 53265, PEEK(53265) OR 64
```

```
10     FOR C=54296 TO 54299
20     POKE C, 14
30     NEXT C
50     POKE 1024, 1
60     POKE 1025, 65
70     POKE 1026, 129
80     POKE 1027, 193
```

Lines 10 to 30 of the routine set the foreground colour of the first four characters of the display. The first four screen locations are then set to the character code for A with each of the four background colours. When you **RUN** the program you should see four letter As, each with a different background colour.

One advantage of extended background colour mode, which makes it very easy to use, is that you can produce all four background colours without having to **POKE** numbers onto the screen. Normal typing will give you the first background colour, and typing with the SHIFT key engaged gives the second colour. If you hold down the CTRL key and press RVS ON, you will find that the third and fourth background colours become available when typing without or with the SHIFT key. This makes extended background colour mode a useful feature for producing colourful displays with very little effort.

CHAPTER 13

HIGH RESOLUTION GRAPHICS

This chapter deals with the creation of high resolution displays using Bit Mapped mode - a feature not documented in the Users Guide, but which is one of the most powerful features of the machine.

As described in Chapter 12, the standard Commodore 64 screen display, with 25 rows of 40 characters, occupies one thousand bytes of memory. The data describing the characters which can be displayed is held in an area of memory known as character memory, where each character description takes up a block of eight bytes. Each pixel comprising that character is defined by a single bit in one of these eight bytes.

In Bit Mapped mode, each of these pixels is individually addressable, allowing you to create high resolution pictures. The use of bit mapped mode is very similar to User Defined character mode in that you can consider the screen to comprise 1000 'blank' characters awaiting definition.

There are two types of Bit Mapped mode :

STANDARD - giving a resolution of 320 * 200 pixels in two colours

MULTI COLOUR - giving a resolution of 160 * 200 pixels in four colours.

STANDARD BIT MAPPED MODE

The display mode is governed by the VIC control register at location 53265. To select Standard bit mapped mode, we must set bit 5 in this location, using a statement like this:

```
POKE 53265, PEEK(53265) OR 32
```

Since there are 1000 possible character locations with 8 bytes needed to define the data for each, we will need a memory area of 8000 bytes for our bit mapped mode display. We can allocate this by changing the VIC II memory pointer register at location 53272. A convenient place to start the bit mapped display is location 8192, (see Appendix 11) and to do this we can use an instruction like this:

```
POKE 53272,PEEK(53272) OR 8
```

This leaves about 8k free for programs.

CLEARING THE DISPLAY

Before we can create a bit mapped mode display, we must first clear the memory area set aside for the display, with a statement like:

```
FOR Z = 8192 TO 16191:POKE Z,0: NEXT
```

Let's convert these statements into a program:

```
10    POKE 53272,PEEK(53272) OR 8
20    POKE 53265,PEEK(53265) OR 32
30    FOR Z=8192 TO 16192: POKE Z,0:
      NEXT
```

Notice how the screen fills with random patterns d colours, before the memory is cleared by line

30. As the memory is cleared, you may see blocks of colour remaining - these correspond to any data left on the standard display when you typed **RUN** (the program listing for example). This is because characters in the standard screen memory (from 1024 to 2048) are now being interpreted as colour data for the bit mapped mode screen. To remove these the program must clear this area of memory to a single colour.

COLOUR IN BIT MAPPED MODE

The colour of bit mapped mode displays is controlled by data in the 1000 standard screen memory locations in the following way:

7	6	5	4	3	2	1	0
FOREGROUND COLOUR				BACKGROUND COLOUR			

For every byte in standard screen memory, bits 0 to 3 determine the background colour and bits 4 to 7 determine the foreground colour in that 8 by 8 pixel area of the display. Since four bits can describe sixteen possibilities, bit mapped mode displays can be in any of the CBM-64's sixteen possible colours, with the restriction that only one combination of colours is possible in each 8 by 8 section of the screen. As an example, we will calculate the appropriate code for colour memory to set the display to a blue background with white pixels. From the table in Appendix 5, we see that the code for white is 1, and the code for blue is 6, which means that since we want a blue background bits 1 and 2 must be set. For a white foreground, bit 4 must be set. The number describing this colour combination is calculated like this :

$2\uparrow 1 + 2\uparrow 2 + 2\uparrow 4 = 22$

To select this colour combination, add the line:

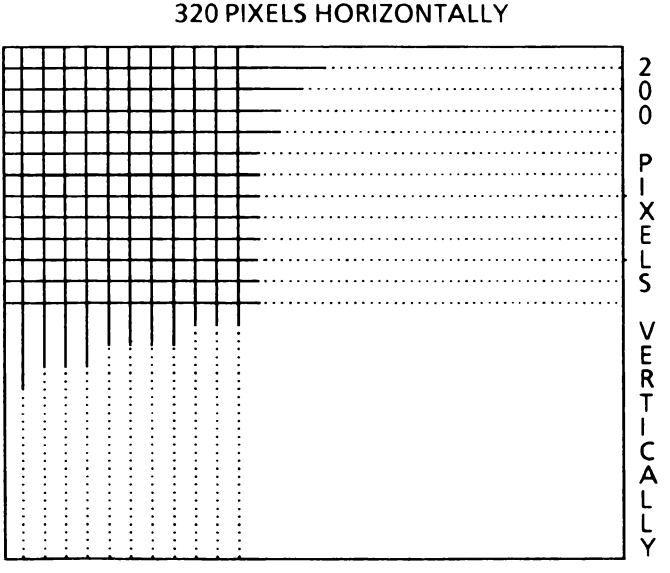
```
40    FOR Z=1024 TO 2023:POKE
      Z,22:NEXT
```

HIGH RESOLUTION DISPLAYS

Now that we have selected bit mapped mode, to create a high resolution display we need to know how to select individual bits from the 8000 bytes of display memory, and set those bits corresponding to the pixels we want to display. To do this we will first examine how the 8000 byte display memory is arranged.

BIT MAPPED MODE DISPLAY MEMORY

The Bit Mapped mode display is set out like this:



The arrangement of the first 16 rows of display memory for the Bit Mapped mode display looks like this:

0	8	16	24	.	.	.	....	312
1	9	17	25	.	.	.	....	313
2	10	18	26	.	.	.	....	314
3	.	.	.	.	.	.	....	315
4	.	.	.	.	.	.	....	316
5	.	.	.	.	.	.	....	317
6	.	.	.	.	.	.	....	318
7	.	.	.	.	.	.	....	319
320	328	336	344	.	.	.	....	632
321	329	.	.	.	.	.	....	633
322	.	.	.	.	.	.	....	634
323	.	.	.	.	.	.	....	635
324	.	.	.	.	.	.	....	636
325	.	.	.	.	.	.	....	637
326	.	.	.	.	.	.	....	638
327	335	343	351	359	367	375	....	639

where the numbers represent bytes of memory, and a vertical group of eight such bytes represents one character position of the standard 25 line character mode display. Notice that the memory isn't set out in an immediately obvious way, so that to be able to address any bit in this memory requires some calculation - fortunately something your CBM-64 can do for you!

If we describe the position of a pixel in terms of its X and Y coordinates, we can calculate for the appropriate byte:

a) Its row in the display memory:

ROW = INT (Y/8)

b) Its column position within that row:

COL = INT (X/8)

c) Its line position within that row:

LINE = Y AND 7

We can also define the appropriate bit within that byte:

BIT = 7-(X AND 7)

We can combine all these calculations in a short subroutine which will calculate the position of, and display, any pixel given its X and Y coordinates.

```
999   REM DISPLAY PIXEL
1000  ROW = INT (Y/8)
1010  COL = INT (X/8)
1020  LINE = Y AND 7
1030  BIT = 7 - (X AND 7)
1040  BYTE = 8192 + ROW*320 + COL*8
      + LINE
1050  POKE BYTE,PEEK(BYTE)OR 2↑BIT
```

To see this subroutine in action, here is a short program using what we have learned so far to simulate a cloudless summer's night!

The Sky at Night

```
10    POKE 53272,PEEK(53272) OR 8
20    POKE 53265,PEEK(53265) OR 32
30    FOR I = 8192 TO 16192:POKE
      I,0:NEXT
40    FOR I = 1024 TO 2023:POKE
      I,22:NEXT
50    FOR J = 0 TO 200
60    X = INT(RND(1)*320)
70    Y = INT(RND(1)*200)
80    GOSUB 1000
90    NEXT J
100   GOTO 100
```

```
999   REM DISPLAY PIXEL

1000  ROW = INT (Y/8)
1010  COL = INT (X/8)
1020  LINE = Y AND 7
1030  BIT = 7 - (X AND 7)
1040  BYTE = 8192 + ROW*320 + COL*8
      + LINE
1050  POKE BYTE,PEEK(BYTE)OR 2↑BIT
1060  RETURN
```

In a more practical vein, we can use the pixel plotting subroutine to draw lines in bit mapped mode. For example, by changing lines 50 to 80 and deleting line 90 in the last program we can draw a horizontal line across the middle of the screen:

```
50    Y = 100
60    FOR X =0 TO 319
70    GOSUB 1000
80    NEXT X
```

Similarly, we can draw a vertical line:

```
50    X = 100
60    FOR Y = 0 TO 199
70    GOSUB 1000
80    NEXT Y
```

By adding a **STEP** to line 60, we can draw dotted lines - try changing line 60 in the last program to:

```
60    FOR Y = 0 TO 219 STEP 3
```

A further modification will allow you to draw circles and ellipses:

```
50    R = 50
60    FOR I = 0 TO 2*π STEP .01
70    X=R * SIN(I)+160
80    Y=R * COS(I)+100
```

```
90    GOSUB 1000
100   NEXT
110   GOTO 110
```

This program will draw a circle of radius R (set in line 50) with its centre at the middle of the screen.

To convert the program to draw an ellipse simply add a reducing factor to either line 70 or line 80:

```
70    X = 0.75 * R * SIN(I)+160
```

The following modifications will cause the computer to draw a spiral in the centre of the screen:

```
50    R = 50
60    FOR J = 0 TO 500
70    R = R - 0.1
80    X = 160 + R*SIN(J/32*π)
90    Y = 100 + R*COS(J/32*π)
100   GOSUB 1000
110   NEXT J
120   GOTO 120
```

You will notice that these programs do not run very quickly, because the CBM-64 is not equipped with BASIC commands to control its high resolution displays. To speed up the process would require the use of machine code.

As mentioned above, we are not restricted to one foreground and one background colour over the entire screen, only in each 8 by 8 pixel block. To demonstrate this, here is a program which will draw different coloured lines across the screen:

```
10    POKE 53272,PEEK(53272) OR 8
20    POKE 53265,PEEK(53265) OR 32
30    FOR I = 8192 TO 16191
35    POKE I,0:NEXT I
```

```
40   FOR I = 1024 TO 2023
50   POKE I,13+X
60   IF I/40=INT(I/40) THEN X = X+16
70   IF X=256 THEN X=0
80   NEXT I
100  FOR Y=4 TO 124 STEP 8
110  FOR X=0 TO 39
120  BYTE = 8192+INT(Y/8)*320+X*8
130  POKE BYTE,255
140  NEXT X,Y
150  GOTO 150
```

Lines 40 to 80 set the foreground colour to light green in the normal way but the background colour is changed every 40 bytes (i.e. every screen line) by adding 16 to the variable X. Adding 16 has the effect of incrementing the most significant 4 bits of that byte, thereby increasing the value of the foreground colour code by one. Consequently, each line is drawn in a different colour.

Another point to notice about this program is that the lines are drawn much more quickly than in previous examples. This is because instead of calculating the position of every bit and setting it to 1, each byte is set to 255 (thus setting each bit to 1) since we are only drawing straight lines. This trick makes the drawing of horizontal lines in high resolution mode tolerably fast, but of course is no use in drawing any other type of line.

Here is a short program which allows you to draw on the standard bit mapped mode screen.

The F1 function key toggles between draw and erase modes (indicated by the border colour changing from green to red); the cursor keys move the cursor around the screen. Because of the speed limitations mentioned above, the program is slow to use and further reinforces the point that without

machine code, high resolution graphics aren't very fast on the 64!

```
100  POKE 53272, PEEK(53272) OR 8
110  POKE 53265, PEEK(53265) OR 32
120  FOR I = 8192 TO 16191
130  POKE I,0:NEXT I
140  FOR I = 1024 TO 2023
150  POKE I,22: NEXT
493  REM
494  REM *****************
495  REM *                 *
496  REM * GET KEY PRESS *
497  REM *                 *
498  REM *****************
499  REM
500  CC = PEEK(BY)
505  GET K$: IF K$="" THEN POKE BY,
     PEEK(BY) OR 2↑BI: POKE BY, PEEK
     (BY) AND 255-2↑BI:GOTO 505
510  IF K$="{CU}" AND Y>0 THEN Y=Y-
     1:GOSUB 1000
520  IF K$="{CD}" AND Y<199 THEN
     Y=Y+1: GOSUB 1000
530  IF K$="{CR}" AND X<319 THEN
     X=X+1: GOSUB 1000
540  IF K$="{CL}" AND X>0 THEN X=X-
     1: GOSUB 1000
550  IF K$="{F1}" THEN MO=1-MO: POKE
     53280, 2-(MO=1)*3): GOTO 505
560  GOTO 500
993  REM
994  REM **************
995  REM *              *
996  REM * PLOT PIXEL *
997  REM *              *
998  REM **************
999  REM
1000 POKE BY, CC: CH = INT(X/8)
1010 RO = INT(Y/8)
1020 LN = Y AND 7
```

```
1030 BY = 8192 + RO*320 + 8*CH + LN
1040 BI = 7-(X AND 7)
1050 IF MO=0 THEN POKE BY,PEEK (BY)
     OR 2↑BI
1060 IF MO=1 THEN POKE BY,PEEK (BY)
     AND 255-2↑BI
1070 RETURN
```

Whilst standard bit mapped mode allows very high resolution graphics, it doesn't allow much flexibility in terms of colour - each pixel in every 8 by 8 block must be in the same colour. The 64 offers a way round this with multi colour bit mapped mode.

Here again there is a parallel with multi colour characters in that up to four colours are allowed in each 8 by 8 block, and adjacent pixels may be different colours. The drawback is that horizontal resolution is halved, with the display width being reduced to 160 pixels - each pixel occupies twice the normal width, however, so the width of the whole display does not change.

MULTI COLOUR BIT MAPPED GRAPHICS

To allow the display of up to four foreground colours the VIC II chip treats the data in the 8000 byte display memory in a slightly different way.

Colours in multi colour bit mapped mode are selected from either:

a) Background register 0 (53281).

b) The most significant nybble of screen memory (a *nybble* is four bits - half a byte).

c) The least significant nybble of screen memory.

d) The least significant nybble of colour memory.

Creating a display in multi colour bit mapped mode requires a lot of planning, since each of the 1000 screen character locations can contain up to 4 independent colours. It is best to draw your design on graph paper first, decide on which areas will be the same colour, and calculate the bit patterns for the appropriate pixels according to this table. The final step would be to calculate data from the groups of 4 bit pairs, and use a program to **POKE** this data into the appropriate byte of bit mapped mode screen memory to recreate your picture.

BIT PATTERN	SOURCE OF COLOUR DATA
00	BACKROUND REGISTER 0
01	HIGH NYBBLE OF SCREEN MEMORY
10	LOW NYBBLE OF SCREEN MEMORY
11	LOW NYBBLE OF COLOUR MEMORY

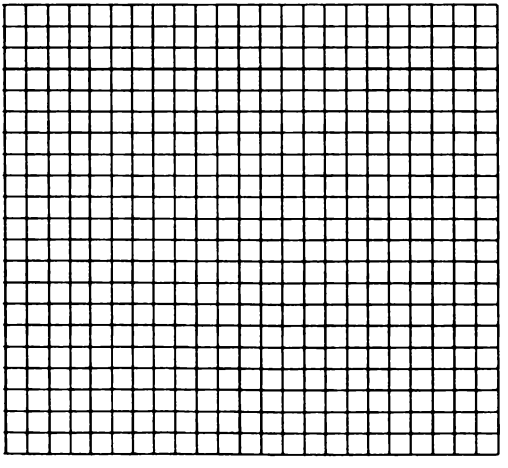
The technique is much the same as for multi colour characters, and for multi colour sprites, as we shall see in the next chapter.

CHAPTER 14

SPRITES

Sprites, or Moveable Object Blocks as they are sometimes known, are a special breed of user defined characters - that is they operate on similar principles, but are endowed with a few extra facilities which make them a flexible and simple way of creating moving high resolution displays. The CBM-64 allows you to use up to 8 sprites at a time, each of which can be in any one of 16 colours.

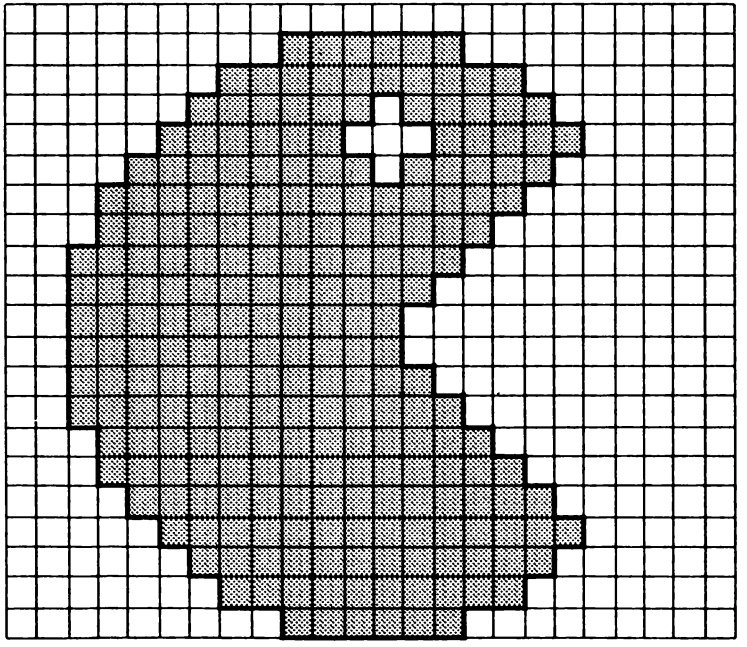
One of the reasons sprites are so useful is that you can move them around the screen very easily, simply by giving them an X and Y coordinate based on the 320 by 200 pixel bit mapped display and the VIC II chip does all the rest. This makes moving graphics in BASIC very easy to create.



The Sprite Grid

Like a user defined character, the shape of a sprite is defined by a block of memory locations, each bit of which determines the state (on or off) of a pixel. However, sprites are rather larger than user defined characters comprising 24 horizontal pixels by 21 vertical, like the one shown on the previous page.

Let's start by creating a sprite and see what we can do with it. The sprite is designed by filling in the squares on the grid which correspond to the pixels of the sprite we want to be illuminated.



To convert the familiar character above into data describing the sprite we use exactly the same technique as for user defined characters.

The difference is that instead of having 64 pixels described in eight bytes, we have 24 * 21 = 504 pixels. This means we will need 63 bytes to store the data describing our sprite. The bytes are arranged to read from left to right, so byte 0 contains the data for the top left hand 8 pixels, byte 1 the next 8 on that row, and so on:

BYTE 0	BYTE 1	BYTE 2
BYTE 3	BYTE 4	BYTE 5
:	:	
:	:	:
BYTE 60	BYTE 61	BYTE 62

Rather than go through the process of converting the picture into DATA statements, which is exactly the same as that given for user-defined characters, here is the data for our sprite:

```
0       0       0
0       126     0
1       255     128
3       247     192
7       227     224
15      247     192
31      255     128
31      255     0
63      254     0
63      252     0
63      248     0
63      248     0
63      252     0
254     0       31
255     0       31
255     128     15
255     192     7
255     224     3
```

255	192	1
255	128	0
126	0	0

Having defined the sprite in terms of 63 bytes of data, we need to store the data somewhere and tell the VIC II chip where to find it.

STORING SPRITE DATA

When allocating space for sprite data, we must bear in mind that the VIC II chip can only access one bank of 16k bytes at a time, and that the data must be kept in an area which is protected from BASIC.

One convenient place is the cassette input buffer, which is an area of memory set aside for temporary storage of data on its way to and from the cassette unit. The cassette buffer is located between address 828 and 1019 giving us 191 bytes of 'safe' memory. In fact there is a little more since the eight bytes before the tape buffer and the four after it are unused - 203 bytes in all.

To indicate to the VIC II chip the whereabouts of the sprite data, 8 bytes of memory have been set aside to serve as pointers to the beginning of the data. This area of memory is the eight bytes immediately after the screen memory; in normal circumstances this means locations 2040 to 2047. Each of these locations contains a 'pointer' to the start of a 64 byte block of memory containing sprite data. The pointer for Sprite 0 is at 2040, for Sprite 1 at 2041 and so on.

To indicate a block of memory in the cassette buffer, starting at location 832 the pointer would contain the value 832 (13*64). All that now remains is for us to **POKE** the data into the appropriate locations, set the pointer for Sprite 0 to

point to it, and tell the VIC II chip to display the sprite.

The VIC II chip provides control over which sprites are visible by means of the SPRITE ENABLE REGISTER at location 53269. Each bit in this register controls one sprite - if it is set to 1, the sprite is visible; a 0 means it is invisible. So, a given sprite may be made visible with a BASIC statement:

```
POKE 53269,PEEK(53269) OR 2↑N
```

where N is the sprite number (0 to 7).

Turning a sprite off involves setting the appropriate bit to 0, like this:

```
POKE 53269,PEEK(53269) AND 255-2↑N
```

Before we discuss sprites in greater detail, let's put what we have learned so far into program form:

```
10   REM CREATE SPRITE
20   FOR Z=0TO63:REM READ SPRITE
     DATA
30   READ D
40   POKE 832+Z,D:REM CASS. BUFFER
50   NEXT Z
60   POKE 2040,13: REM POINT TO DATA
70   POKE 53269,1: REM SPRITE 0 ON
80   END
999  REM DATA FOR SPRITE 0
1000 DATA 0,0,0,0,126,0,1,255
1010 DATA 128,3,247,192,7,227
1020 DATA 224,15,247,192,31
1030 DATA 255,128,31,255,0,63
1040 DATA 254,0,63,252,0,63
1050 DATA 248,0,63,248,0,63,252
1060 DATA 0,63,254,0,31,255,0
1070 DATA 31,255,128,15,255,192
```

```
1080 DATA 7,255,224,3,255,192
1090 DATA 1,255,128,0,126,0,0
```

If you **RUN** the program, you will not be able to see the sprite. This is because we have not indicated to the VIC II chip where we want the sprite to be displayed. To do this, type in the following commands in immediate mode:

```
POKE 53248,160:POKE 53249,110
```

- the sprite becomes visible! The two locations we **POKE**d were the SPRITE POSITION REGISTERS for Sprite 0. The numbers we used told the VIC II chip to display the sprite at a position 160 pixels from the left of the display, and 110 pixels from the top. Try **POKE**ing the position registers with other numbers and notice how the sprite moves almost instantaneously.

If you use very small or very large numbers for the Y position register (53249), you will notice that not all of the sprite is visible. This is because there are only 220 possible locations in the vertical direction on the screen, and the Y position register can contain numbers between 0 and 255. This means that a value of less than 30 in the Y position register will display the sprite above the top of the screen, and a value greater than 249 will put it out of sight beyond the bottom.

You may also come across another problem - how to get the sprite over to the extreme right hand edge of the screen. The position registers can only hold numbers up to 255, and the screen is 320 pixels wide. This problem is solved by the use of a further register at location 53264, known as the X MSB REGISTER. Each bit in this register corresponds to one sprite - if it is set, that sprites X Position Register value is added to 255 to calculate

its position on the screen. Try this in immediate mode by setting the X position register to 30 :

```
POKE 53248,30
```

and setting the X msb register for Sprite 0:

```
POKE 53264,1
```

-the sprite jumps from one side of the screen to the other.

SPRITE COLOURS.

Each sprite can be displayed in any of 16 possible colours and the colour is controlled by yet another register. For Sprite 0, the SPRITE COLOUR REGISTER is at location 53287. Try **POKE**ing this location with a few values (0-15) and watch the colours change. Notice that the screen background shows through those parts of the sprite where no pixels are defined.

The following program will display all 8 sprites in different colours and move them around the screen. Notice that all sprites are the same design because every sprite data pointer is set to point at the same data by lines 70 and 80.

A FLOCK OF SPRITES

```
10    PRINT "{CLS}"
20    R = 53248: X = 100:Y = 100
30    FOR I=0 TO 63
40    READ D
50    POKE 832+I,D
60    NEXT I
70    FOR I=2040 TO 2047
80    POKE I,13:NEXT I
90    POKE 53269,255
100   FOR N=0 TO 15 STEP 2
```

```
110   X=X+(RND(1)*20)-(RND(1)*20)
120   Y=Y+(RND(1)*20)-(RND(1)*20)
130   IF X>255 OR X<0 THEN X=100
140   IF Y>249 OR Y<30 THEN Y=100
150   POKE 53248+N,X
160   POKE 53249+N,Y
170   NEXT N
180   GOTO 100
1000  DATA 0,0,0,0,126,0,1,255,128
1010  DATA 3,247,192,7,227,224,15
1020  DATA 247,192,31,255,128,31
1030  DATA 255,0,63,254,0,63,252,0
1040  DATA 63,248,0,63,248,0,63,252
1050  DATA 0,63,254,0,31,255,0,31
1060  DATA 255,128,15,255,192,7,255
1070  DATA 224,3,255,192,1,255
1080  DATA 128,0,126,0,0
```

SPRITE PRIORITIES

You will notice that when two sprites collide they cross over each other, the lowest numbered sprite appearing to cross IN FRONT of the other. This sprite to sprite display priority is pre-determined, the lowest numbered sprite will always cross on top of the other.

It is possible to control the sprite to background priority, so that sprites can be made to appear to cross in front of or behind other screen data, by using the SPRITE TO BACKGROUND REGISTER at 53275. If the bit associated with a sprite in this register is a 1, then it will cross 'behind' any background data; if it is 0 then the sprite has a higher priority than the background data and will pass in front of it. The following program demonstrates this:

```
5     PRINT"{CLS}"
10    FOR I=0 TO 63
20    READ D
```

```
30    POKE 832+I,D:NEXTI
40    POKE 2040,13:POKE 2041,14
50    POKE 53287,0:POKE 53288,1
60    POKE 53248,0:POKE 53249,140
70    POKE 53250,90:POKE 53251,140
75    POKE 53264,2
80    POKE 53269,3

100   W$=" {RED}{RVS}      {ROF}
      {PUR}{RVS}     {ROF}
      {GRN}{ROF}     {RVS}
      {YEL}{ROF}     {RVS}"
110   PRINT"{HOM}{CD}{CD}{CD}{CD}"
120   FOR I=0 TO 11:PRINT W$:NEXT
200   FOR C=1 TO 345
210   X0=PEEK(53248)
220   X1=PEEK(53250)
230   IF X0=255 THEN X0=0: POKE
      53264, 1
240   IF X1=0 THEN X1=255: POKE
      53264, 0
250   X0=X0+1
260   X1=X1-1
270   POKE 53248,X0
280   POKE 53250,X1
290   IF X0/80=INT(X0/80) THEN POKE
      53275,1-PEEK(53275)
300   NEXT C

1000  DATA 24,24,24,24,60,24,24,126
1010  DATA 24,12,255,48,15,255,240,3
1020  DATA 255,192,7,255,224,15,255
1030  DATA 240,24,126,24,48,126,12,96
1040  DATA 126,6,255,255,255,255,255
1050  DATA 255,255,255,255,255,255,
           255
1060  DATA 127,255,254,56,60,28,28,
      24
1070   DATA  56,14,0,112,28,0,56,56,0,
       28,0
```

Notice how the black sprite is made to move 'in and out' of the coloured bars, whilst the white one always has higher priority than the data and therefore passes in front of all the coloured bars.

EXPANDING SPRITES

There are two further registers associated with sprites which allow you to expand them in the X and Y directions. Each bit in these two registers controls the size of one sprite in each direction - a 0 means the sprite is normal size in that direction; a 1 means it is twice the size. To expand a sprite, use the following statement:

X direction: `POKE 53277,PEEK(53277) OR 2↑N`
Y direction: `POKE 53271,PEEK(53271) OR 2↑N`

where N is the number of the sprite.

To revert to normal size :

X direction: `POKE 53277,PEEK(53277) AND 255-2↑N`
Y direction: `POKE 53271,PEEK(53271) AND 255-2↑N`

Try these commands in immediate mode to see their effect.

COLLISION DETECTION

Another useful feature of sprites is that you can determine when one sprite comes into contact with another sprite or with other data on the screen.

When a collision between sprites is detected, the bits appropriate to those sprites in the SPRITE TO SPRITE COLLISION REGISTER at location 53278

are set to 1, and will remain so until that register is examined with a **PEEK** command.

Collisions between sprites and screen data are indicated in the SPRITE TO DATA COLLISION REGISTER at location 53279. A collision is deemed to have occured when any non-zero part of a sprite overlaps another non zero-part of data. The following short game shows this happen.

You are in control of a toboggan hurtling down the Cresta Run. You move left and right using the two cursor keys, trying to avoid the obstacles strewn along the course whilst keeping on the track:

CRESTA RUN

```
20    GOSUB 500
100   GET K$:IF K$="" THEN 160
110   IF K$="{CD}" THEN X=X-5
120   IF K$="{CR}" THEN X=X+5
130   IF X>250 THEN X=0:POKE 53264,1
      :GOTO 150
140   IF X=0 THEN X=255:POKE 53264,0
150   POKE 53248,X
160   OB=INT(RND(0)*4*DF)
170   OP=INT(RND(0)*WD)
180   IF OB=1 THEN O$="{BLK}↑{WHT}"
      :GOTO 190
185   O$=" "
190   PRINT TAB(10+I)"●"SPC(OP)O$
      SPC(WD-OP)"●"
200   S=S+1:T=T+1
210   IF PEEK(53279)=1 THEN POKE
      53287,11:H=H+1:GOTO230
220   POKE 53287,7
230   IF S<SS THEN 280
240   IF DI=0 THEN 270
250   I=I-1:IF I=0 THEN DI=1-DI
260   GOTO 280
270   I=I+1:IF I=BB THEN S=0:DI=1-DI
```

```
280   IF S=0 THEN SS=INT(RND(0)*DF*
      15)
290   IF T=300 THEN 310
300   GOTO 100
310   PRINT"{CLS} GAME OVER"
320   PRINT:PRINT"YOUR SCORE WAS
      "INT((T-H)/T*100)"%"
330   PRINT:PRINT"ANOTHER GO (Y/N)?"
340   GET A$:IF A$="" THEN 340
350   IF A$="Y" THEN RUN
360   END
500   PRINT"{CLS}{WHT}"
510   INPUT"DIFFICULTY (1=EASY TO
      5=HARD)";DF
520   IF DF<1 OR DF>5 THEN 500
530   DF=6-DF
540   WD=10:SS=50:BB=10
550   FOR J=0 TO 63:READ D
560   POKE 832+J,D:NEXT J
570   POKE 2040,13
580   X=140:POKE 53248,X
590   POKE 53249,100
600   POKE 53287,7
610   POKE 53269,1
620   RETURN
1000  DATA 0,193,128,0,193,128
1010  DATA 0,193,128,0,255,128
1020  DATA 0,255,128,0,255,128
1030  DATA 0,136,128,0,156,128
1040  DATA 0,156,128,0,136,128
1050  DATA 0,255,128,0,255,128
1060  DATA 0,156,128,0,156,128
1070  DATA 0,156,128,0,156,128
1080  DATA 0,156,128,0,221,128
1090  DATA 0,127,0,0,62,0
1100  DATA 0,28,0,0
```

Notice how the colour of the toboggan changes whenever it 'hits' the walls of the track or an obstacle. The collision is detected in line 210 which

increments the counter H (the number of collisions) and changes the colour of Sprite 0 (the toboggan) to grey. The Sprite to Data collision register is checked each time the track is moved, and the toboggan remains grey for as long as it is in contact with the obstacle. The other thing to notice about the program is that the toboggan can only move horizontally - it is the track which moves. The program relies on the screen scrolling to give the illusion of movement, since this is performed by the operating system much more quickly then we could with a BASIC program.

MULTI COLOUR SPRITES

In the same way that we can have multi colour characters, we can choose any sprite to be in multi colour mode, with the same trade off of resolution for colour as with multi colour characters. That is Multi Colour Sprites can be in up to four different colours, but horizontal resolution is halved, so multi colour sprites are 12 * 21 pixels in size.

In multi colour mode, sprite data is interpreted differently in that a pair of bits is required to specify a pixel - each coloured 'pixel' in fact consists of two pixels side by side, each pair being called a Bit Pair. The Bit Pair can be selectively coloured using one of the four combinations of 0 and 1, as shown in the table on the next page.

To select multi colour mode for a sprite, the appropriate bit in the SPRITE MULTI COLOUR REGISTER must be set to 1, using a command such as:

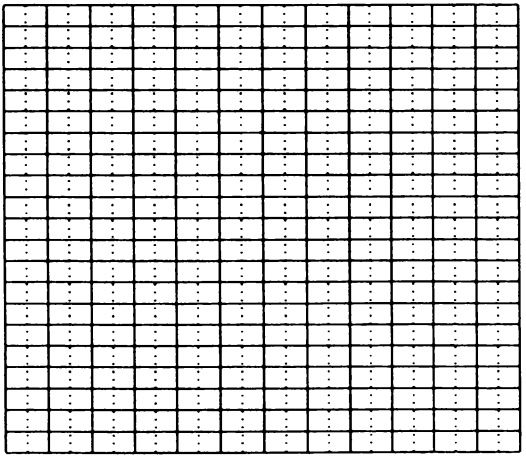
```
POKE 53276,PEEK(53276) OR 2↑N
```

BIT PAIR	COLOUR
00	Screen Colour
01	Multi Colour Register 0
10	Sprite Colour Register
11	Multi Colour Register 1

N is the number of the sprite to be changed to multi colour mode. To return the sprite to standard mode this bit must be set to 0:

```
POKE 53276,PEEK(53276) AND 255-2↑N
```

Multi colour sprites behave in exactly the same way as standard sprites - the only difference is in the way they are defined. Let's take an example to make this clear.



Multicolour Sprite Grid

CREATING A MULTI COLOUR SPRITE

We can use the same 24 by 21 grid we used earlier to plan the design, but remember that each pixel is now defined by two squares of the grid. Opposite is a design for a face which we will turn into a multi colour sprite. Notice that the pixels are twice the size of those in our earlier designs and that there are four different types of shading corresponding to the four different colours available for multi colour sprites. We can allocate each of these colours to a particular register:

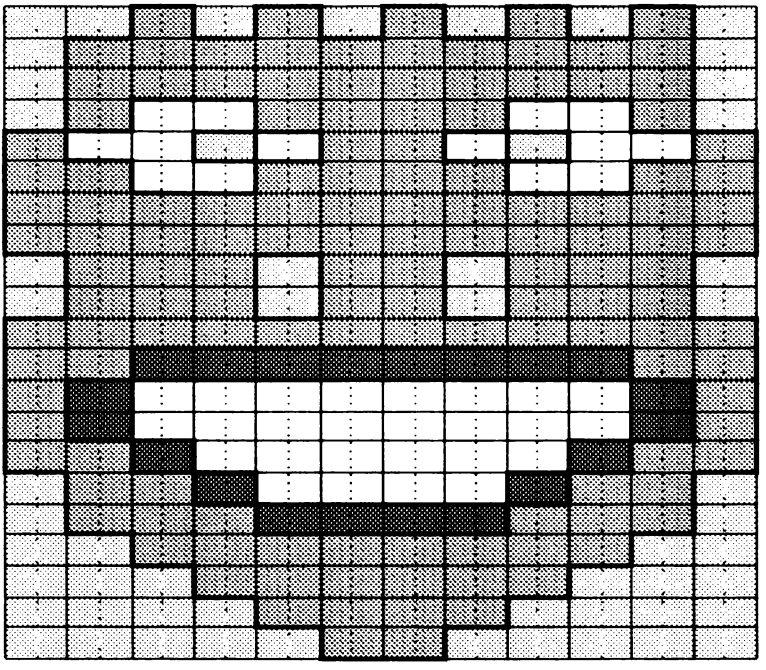
BACKGROUND	SCREEN COLOUR	00
FACE	SPRITE COLOUR	10
EYES	REGISTER 0	01
LIPS	REGISTER 1	11

The numbers in the right hand column are the bit pairs which specify the register from which the colour of a given pixel is drawn. This means that all pixels containing the bit pair 11 will be displayed in the colour currently held in Multi Colour register 1, whilst all those set to 00 will be in the current screen colour.

We can now fill each area of the design with the bit pairs appropriate to their colour.

This information is converted into 63 bytes of data in the usual way. In order to see the difference between multi colour mode sprites and the standard variety here is a short program which displays the multicoloured face:

```
5     PRINT "{CLS}":POKE 53281,0
10    FOR I=0 TO 63:READ D
20    POKE 832+I,D:NEXT
```



	00	Screen Colour:	Blue
	01	Multicolour Sprite Colour 0:	White
	10	Normal Sprite Colour:	Green
	11	Multicolour Sprite Colour 1:	Red

Multicolour Sprite

```
30    FOR I=2040 TO 2047
40    POKE I,13:NEXT
50    POKE 53276,255
60    FOR I=0 TO 15
```

```
70    POKE 53248+I,50+(10*I)
80    NEXT I
90    FOR I=0 TO 7
100   POKE 53287+I,5+I
110   NEXT I
120   POKE 53285,1
130   POKE 53286,2
140   POKE 53269,255
1000  DATA 8,136,136,42,170
1010  DATA 168,42,170,168,37
1020  DATA 170,88,148,105,22
1030  DATA 165,170,90,170,170
1040  DATA 170,170,170,170,42
1050  DATA 40,168,42,40,168
1060  DATA 170,170,170,175,255
1070  DATA 250,181,85,94,181
1080  DATA 85,94,173,85,122,43
1090  DATA 85,232,42,255,168
1100  DATA 10,170,160,2,170
1110  DATA 128,0,170,0,0,40,0,0
```

Finally, a word about animation.

ANIMATION

Because of the way in which the VIC II chip is directed to the start of a block of sprite data, simple animation is just a matter of setting up several blocks of data corresponding to the various stages of movement, then changing the value of the sprite pointer to point to each block in turn. The change in the appearance is instantaneous, and so the illusion of movement is easily created. For instance, our first character would look much more convincing if his mouth opened and closed as he moved, and his mouth always faced his direction of motion. Here is a program which does just that and moves the sprite around the screen:

```
10    PRINT"{CLS}"
20    POKE 53281,15
```

```
30   FORI=0TO191:READ D:POKE 832+I
     ,D :NEXT
40   POKE 2040,13 :POKE 53269,1
50   C=0:POKE53287,C
100  FOR Y=50 TO 200 STEP 50
110  POKE 53249,Y
130  GOSUB 500
135  IF Y=50 THEN POKE 53277,1
140  IF Y=100 THENPOKE 53277,0:POKE
     53271,1
145  IF Y=150 THEN POKE 53277,1
150  D=1-D
155  C=C+2
160  POKE 53287,C
165  NEXT Y
170  POKE 53269,0
180  POKE 53271,0: POKE 53277,0:END
500  IF D=1 THEN 600
510  POKE 2040,13-((MODE=1)*2)
520  POKE 53248,X
530  IF X=90 AND PEEK(53264)=1 THEN
     700
540  X = X+5
550  IF X=255 THEN X=0:POKE
     53264,1:POKE  3248,X
560  IF X/15 = INT(X/15)THEN MODE =
     1-MODE
570  GOTO 510
600  POKE 2040,14-((MODE=1)*1)
610  X = X-5
620  POKE 53248,X
630  IF X=0 AND PEEK(53264)=0 THEN
     700
640  IF X=0 THEN X=255:POKE 53264,0
     :POKE 53248,X
650  IF X/15=INT(X/15)THEN MODE = 1-
     MODE
660  GOTO 600
700  RETURN
997  REM
998  REM RIGHT FACING SPRITE
```

```
999  REM
1000 DATA 0,0,0,0,126,0,1,255
1010 DATA 128,3,247,192,7,227
1020 DATA 224,15,247,192,31
1030 DATA 255,128,31,255,0,63
1040 DATA 254,0,63,252,0,63
1050 DATA 248,0,63,248,0,63
1060 DATA 252,0,63,254,0,31
1070 DATA 255,0,31,255,128,15
1080 DATA 255,192,7,255,224,3
1090 DATA 255,192,1,255,128
1095 DATA 0,126,0,0
1097 REM
1098 REM LEFT FACING SPRITE
1099 REM
2000 DATA 0,0,0,0,126,0,1,255
2010 DATA 128,3,247,192,7,227
2020 DATA 224,3,247,240,1,255
2030 DATA 248,0,255,248,0,127
2040 DATA 252,0,63,252,0,31
2050 DATA 252,0,31,252,0,63,252
2060 DATA 0,127,252,0,255,248
2070 DATA 1,255,248,3,255,240
2080 DATA 7,255,224,3,255,192
2090 DATA 1,255,128,0,126,0,0
2097 REM
2098 REM CLOSED MOUTH
2099 REM
3000 DATA 0,0,0,0,126,0,1,255
3010 DATA 128,3,247,192,7,227
3020 DATA 224,15,247,240,31,255
3030 DATA 248,31,255,248,63,255
3040 DATA 252,63,255,252,63
3050 DATA 255,252,63,255,252
3060 DATA 63,255,252,63,255
3070 DATA 252,31,255,248,31
3080 DATA 255,248,15,255,24
3090 DATA 7,255,224,3,255,192
3095 DATA 1,255,128,0,126,0,0
```

Lines 10 to 50 set up the various sprite registers, and **POKE** the data into the cassette buffer.

Lines 100 to 200 move the sprite down the screen by incrementing the Y position register in multiples of 50, and expand the sprite and change its colour, depending upon its position down the screen.

Lines 500 to 700 move the sprite across the screen and close and open the mouth.

SPRITE CREATION

The process of defining sprites is very laborious, and, like user defined characters, is one with which your computer can assist. The following program will allow you to create and modify your sprites, save the data on tape and many other useful things, so it's well worth persevering in typing it all in!

```
1     REM ******************
2     REM *                  *
3     REM * SPRITE EDITOR *
4     REM *                  *
5     REM ******************
6     REM
7     REM
10    DIM D%(23,20),DA%(64)
20    CS=1065:CX=0:CY=0:CC=CS:CD%(0)=
      43:CD%(1)=81
30    POKE53280,13:POKE53281,12:PRINT
      "{GR3}"
40    RS=53248:DS=12288:SN=0
50    B$="
                  {CU}":REM 37 SPACES
60    MO=1
93    REM
94    REM **************
```

```
95   REM *              *
96   REM * PRINT GRID *
97   REM *              *
98   REM **************
99   REM
100  PRINT"{CLS}"
110  G$="  ++++++++++++++++++++++++"
120  FOR I=0 TO 20
130  PRINT G$:NEXT I
140  CD = PEEK(CS)
150  PRINT "{HOM}"
160  FOR I=1 TO 8:PRINT TAB(32)I
170  IF I<8 THEN PRINT:PRINT
180  NEXT I
493  REM
494  REM ****************************
495  REM *                          *
496  REM * SET UP SPRITE REGISTERS *
497  REM *                          *
498  REM ****************************
499  REM
500  FOR I=0 TO 7:REM DO EACH SPRITE
510  POKE 53248+2*I,50:REM  X POS
520  POKE 53249+2*I,52+(25*I):REM Y
     POS
530  POKE 53287+I,1:REM COLOUR
540  POKE 2040+I,192+I:REM POINTERS
550  NEXT I
560  POKE 53264,255:REM X MSB
570  POKE 53269,255:REM ENABLE
580  POKE 53277,0:REM NO X EXPANSION
590  POKE 53271,0:REM NO Y EXPANSION
600  POKE 53276,0:REM MULTI COL. OFF
650  PRINT SPC(6)"{RVS}P{ROF}ROGRAM
     {RVS}L{ROF}OAD {RVS}S{ROF}AVE"
993  REM
994  REM ******************
995  REM *                *
996  REM * KEYBOARD INPUT *
997  REM *                *
998  REM ******************
```

```
999  REM
1000 GET K$:IF K$="" THEN 1500
1005 K = ASC(K$)
1010 IF K$="{CR}" AND CX<23 THEN
     CX=CX+1:POKE CC,CD
1020 IF K$="{CL}" AND CX>0 THEN
     CX=CX-1:POKE CC,CD
1030 IF K$="{CD}" AND CY<20 THEN
     CY=CY+1:POKE CC,CD
1040 IF K$="{CU}" AND CY>0 THEN
     CY=CY-1:POKE CC,CD
1060 IF K$=" " THEN SD=1-SD:CD=
     CD%(SD)
1070 IF K$="P" THEN SA=56222:L=6:
     CO=2:GOSUB 1200:GOSUB 3000:
     CO=11:GOSUB 1200
1090 IF K$="L"THEN 4100
1095 IF K$="S"THEN 4000
1100 IF K>48 AND K<57 THEN GOSUB
     2000
1110 IF K>32 AND K<41 THEN GOSUB
     2500
1120 CC=CS+CX+(40*CY):D%(CX,CY)=SD
1130 POKE CC,CD:GOTO 1000
1200 FOR I=0 TO L
1210 POKE SA+I,CO:NEXT:RETURN
1493 REM
1494 REM ****************
1495 REM *                *
1496 REM * FLASH CURSOR *
1497 REM *                *
1498 REM ****************
1499 REM
1500 CT = PEEK(CC)
1510 C%=(CT OR 128)-(CT AND 128)
1520 POKE CC,C%
1530 FOR I=0 TO 100:NEXT
1540 GOTO 1000
1600 PRINT SPC(6)"{RVS}P{ROF}ROGRAM
     {RVS}L{ROF}OAD {RVS}S{ROF}AVE"
     :RETURN
```

```
1993 REM
1994 REM ********************
1995 REM *                  *
1996 REM * SPRITE SELECTION *
1997 REM *                  *
1998 REM ********************
1999 REM
2000 IF SN=0 THEN 2020
2010 POKE SS+54272,11
2020 SN = K-48: SS=CS-88+(SN*120):
     POKE SS+54272,3
2030 GOSUB 3500:RETURN
2493 REM
2494 REM **********************
2495 REM *                    *
2496 REM * CLEAR SPRITE DATA *
2497 REM *                    *
2498 REM **********************
2499 REM
2500 PRINT"{HOM}{22 * CD}"
2510 PRINT B$:PRINT" ARE YOU SURE ?"
2520 GETA$:IF A$="" THEN 2520
2530 IF A$<> "Y" THEN PRINT"{CU}"B$:
     GOSUB 1600:RETURN
2540 PRINT"{CU}"B$:PRINT" CLEARING
     DATA"
2550 FOR I=0 TO 63:POKE DS+((K-
     33)*64)+I,0:NEXT I
2560 PRINT"{CU}"B$:GOSUB 1600:
     RETURN
2993 REM
2994 REM ****************************
2995 REM *                          *
2996 REM * CONVERT GRID TO DATA     *
2997 REM *                          *
2998 REM ****************************
2999 REM
3000 I = 0:FOR Y=0 TO 20
3010 FOR BYTE=0 TO 2:DA%(I)=0
3020 FOR BIT=0 TO 7
```

```
3030 DA%(I)=DA%(I)+(2↑(7-BIT))*-
     (PEEK(CS+(BIT)+(8*BYTE)+(40*Y))
     =81)
3040 NEXT BIT
3041 POKE DS+((SN-1)*64)+I,DA%(I):
     I=I+1
3050 NEXT BYTE
3060 NEXT Y
3120 RETURN
3493 REM
3494 REM ************************
3495 REM *                      *
3496 REM * CONVERT DATA TO GRID *
3497 REM *                      *
3498 REM ************************
3499 REM
3500 FOR I=0 TO 63:DA%(I)=PEEK
     (DS+((SN-1)*64)+I):NEXT:I=0
3510 FOR Y=0 TO 20
3520 FOR BYTE=0 TO 2
3525 N = DA%(I)
3530 FOR BIT=7 TO 0 STEP-1
3540 B2 = 2↑BIT
3550 IF N - B2 >=0 THEN CH=81:N=N-
     B2:GOTO 3570
3560 CH = 43
3570 POKE CS+(7-BIT)+(8*BYTE)
     +(40*Y),CH
3590 NEXT BIT:I = I+1:NEXT BYTE
3600 NEXT Y:RETURN
3993 REM
3994 REM ********************
3995 REM *                    *
3996 REM * SAVE SPRITE DATA *
3997 REM *                    *
3998 REM ********************
3999 REM
4000 PRINT"{HOM}{23 * CD}"
4010 PRINT B$
4020 INPUT" FILE NAME";N$
4025 PRINT"{CU}"B$"{CU}"
```

```
4030 OPEN 1,1,1,N$
4035 PRINT"{CU}"B$
4040 FOR I=0 TO 63:PRINT#1,CHR$
     (DA%(I));:NEXT
4050 PRINT"{CU}"B$"{CD}"
4060 CLOSE 1
4070 GOTO 650
4093 REM
4094 REM ********************
4095 REM *                  *
4096 REM * LOAD SPRITE DATA *
4097 REM *                  *
4098 REM ********************
4099 REM
4100 PRINT "{HOM}{23 * CD}"
4110 PRINT B$
4120 INPUT" WHICH SPRITE (1-8)";SN
4122 PRINT"{CU}"B$
4125 INPUT" FILE NAME";N$
4130 PRINT"{CU}"B$"{CU}{CU}"
4140 OPEN 1,1,0,N$
4145 PRINT"{CU}"B$
4150 FOR I=0 TO 63
4151 GET#1,D$
4152 IF D$ ="" THEN D$ = CHR$(0)
4153 DA%(I) = ASC(D$)
4155 POKE DS+(SN-1)*64+I,DA%(I):NEXT
4157 PRINT"{CU}"B$"{CD}"
4160 CLOSE 1
4170 GOTO 650
```

How to use the program

The program draws a large 24 by 21 grid on the left hand side of the screen upon which sprites are created. The eight sprites, numbered 1 to 8, are displayed down the right hand edge of the display, and will contain random data when the program is

first **RUN**. To clear any sprite data area, hold down the SHIFT key and press the sprite number key.

To create a sprite, the current sprite data must be loaded into the grid by pressing the appropriate number key. When loaded, the sprite will be displayed on the grid ready for editing.

The flashing cursor is moved around the grid with the cursor keys and can be set to erase or plot 'pixels' by using the space bar which toggles between the two options.

When a sprite has been created, the appropriate data is placed in the sprite data area by pressing 'P'.

Sprites can be **SAVE**d on tape and **LOAD**ed by pressing 'S' or 'L' respectively, and following the prompts displayed on the screen.

CHAPTER 15

PERMANENT STORAGE

The 64 is able to store programs onto cassette tapes or floppy disks, and to LOAD and RUN programs from tape, disk or plug-in cartridge. Tapes and disks may also be used to store data for future use.

PROGRAMS ON TAPE

To load a program from tape, put the cassette in the tape unit and rewind it to the beginning. Type the command:

```
LOAD "NAME"
```

and press RETURN. 'NAME' should be the name of the program you wish to load. If you want the first program on the tape just type:

```
LOAD ""
```

The 64 will respond to these commands by displaying the message:

```
PRESS PLAY ON TAPE.
```

Start the tape by pressing the PLAY key. The screen will go blank and the tape will start to move. When the program is reached, the tape will stop moving and the message:

```
FOUND NAME
```

will be displayed on the screen. After a few seconds

the screen will go blank again and the program will be loaded. You can cut short the pause by pressing the Commodore key (C=) when the program name is displayed. If you do not wish to load the program, press RUN/STOP.

To **LOAD** and **RUN** the first program on a cassette, hold down a SHIFT key and press RUN/STOP. The 64 will again ask you to start the tape, and will then display the name of the first program found. When the program has been **LOAD**ed it will start running automatically.

Saving Programs

To save the program in the computer onto tape, put a blank cassette into the tape unit and wind it forwards past the transparent leader. Type the command:

```
SAVE "NAME"
```

The computer will respond with:

```
PRESS PLAY AND RECORD ON TAPE
```

Start the tape by pressing the RECORD and PLAY keys together. The screen will go blank, and the program will be **SAVE**d. The name you use for the **SAVE**d program can be any combination of characters up to a maximum length of 16.

Checking Programs

To check that a program has been saved accurately, use the **VERIFY** command. Rewind the tape to the beginning of the program, and type:

```
VERIFY "NAME"
```

The message:

PRESS PLAY ON TAPE

will be displayed on the screen. Start the tape, and the screen will go blank while the 64 searches for the program. When the program is found, the message:

FOUND NAME

is displayed. After a few seconds the screen will blank again, and the program on tape will be compared with the program in the computer. When the program has been checked, the message 'OK' will be displayed if the two programs were identical, or '?VERIFY ERROR' if the two did not match.

PROGRAMS ON DISK

Loading and saving programs from disk is very similar to using the tape. The same commands are used, with the suffix ',8' added which instructs the 64 to use the disk drive instead of the cassette unit. To load a program from disk, put the disk into the drive and close the door. Type the command:

```
LOAD "NAME",  8
```

and press RETURN. The computer will respond with the messages:

SEARCHING FOR NAME
LOADING NAME

and then READY when the process is complete.

To save a program onto the disk, type:

```
SAVE "NAME", 8
```

The 64 will display:

```
SAVING NAME
```

and READY when it has finished.

The **VERIFY** command can also be used with disks. To verify a program, type:

```
VERIFY "NAME", 8
```

The 64 will display:

```
SEARCHING FOR NAME
VERIFYING
```

and either OK if the program on disc matched the one in the 64, or ?VERIFY ERROR if the two were different.

PROGRAMS ON CARTRIDGES

Programs on cartridges are the easiest of all to use. First, you must **switch off** the 64, or it may be damaged. Then plug the cartridge into the socket on the right at the rear of the machine, and switch on again. Follow the instructions enclosed with the cartridge to start the program.

FILE HANDLING

As well as loading and saving programs, the Commodore 64 can use the cassette unit to store and reload data generated by programs. The data is stored in *files*, and can be in numerical or string form. To create a file, the command **OPEN** is used:

```
OPEN F, D, S, "NAME"
```

The parameter F is the *file number*, a reference number used to identify which file is being used by each file command (there may be a number of files in use at the same time). D is the *device number*, which is always 1 for the tape unit - other peripherals have different numbers. S is called a *secondary address*, and can have three meanings when using tape files. If S is zero, the file data is to be read from the tape; if S is one, data is to be written to the tape, and if S is two, data is written to the tape and a special "end of tape" marker is written at the end of the file.

To write data to the tape the command **PRINT#** is used. This is very similar to the **PRINT** command, except that data is written to the tape instead of to the screen. The command must be followed by the file number of the file to which the data is to be written:

```
PRINT#1, X
```

would write the value of the variable X to file number 1.

There are two commands which read data from files: **INPUT#** and **GET#**. Again, the commands are similar to their keyboard reading counterparts. **INPUT#** reads a number of bytes of data terminating in a carriage return, whereas **GET#** reads single bytes from the tape. The file number must be specified when using these commands also.

CLOSE is used to terminate the use of a file. When reading a file, **CLOSE** simply closes the input channel, whereas when writing to a tape file, the **CLOSE** command writes an "End of File" or "End of Tape" marker onto the tape.

The following short program shows the use of **OPEN**, **PRINT#**, **INPUT#** and **CLOSE**.

```
10    OPEN 1, 1, 1, "TEST"
20    PRINT#1, "THIS IS A TEST FILE"
30    CLOSE 1
40    STOP
100   OPEN 3, 1, 0, "TEST"
110   INPUT#1, A$
120   CLOSE 1
130   PRINT A$
```

Put a blank tape into the tape unit and wind it forwards past the transparent leader tape. **RUN** the program, and the 64 will display the message:

PRESS PLAY AND RECORD ON TAPE

as happens when **SAVE**ing programs. Start the tape, and the file will be created and the data stored. The program will stop with the message "BREAK IN 40".

Rewind the tape and type RUN 100. The message

PRESS PLAY ON TAPE

is displayed. When you set the tape going the data file will be **OPEN**ed and read, and the data printed on the screen.

This technique may be used to store any amount of data onto tape. You can save numerical data as well as strings in the same way.

Punctuation marks - commas and semi-colons - may be used with **PRINT#** as with **PRINT**. Without punctuation, **PRINT#** puts a carriage return character after each item of data written to the tape, and it these carriage returns are used by the **INPUT#** statement to distinguish between

adjacent items of data. Using commas or semi-colons suppresses the carriage returns with the result that the **INPUT#** statement will not read the data properly. This is where **GET#** is useful, reading single bytes from the file without regard for carriage returns or any other markers. If you need to store single byte numbers, then writing them to the tape with **PRINT#;** and reading them with **GET#** makes very efficient use of the tape, as all the space is taken up by data with no separating characters. This method is used in the Character Generator program in Chapter 12 to save character sets onto tape.

OTHER PERIPHERAL COMMANDS

STATUS, or **ST**, is a system variable which provides information about tape files. The variable has eight possible meanings, which are shown in the table:

STATUS	MEANING
1, 2	OK
4	SHORT BLOCK
8	LONG BLOCK
16	UNRECOVERABLE READ ERROR
32	CHECKSUM ERROR
64	END OF FILE
-128	END OF TAPE

One last peripheral handling instruction is **CMD**. This diverts the normal screen output to a specified output channel or data file. The command is most often used to list data to a printer, but can also be used to record programs listings on tape, perhaps

for incorporation in a word processor document. (All the programs in this book were transferred to a word processor in this way).

To list a program to a printer use the command sequence:

```
OPEN 1, 4
CMD 1
LIST
CLOSE 1
```

The listing will be diverted from the screen to the printer. Similarly, a program can be listed to a tape file by the sequence:

```
OPEN 3, 1, 1, "LISTING"
CMD 3
LIST
CLOSE 3
```

Program listings saved in this way can not be reloaded and **RUN**, so always make sure you have a copy of the program **SAVE**d in the normal way.

DATAFILE

This program uses the file handling commands to maintain a simple database on tape. The program is written as an address book, but there are many other applications. As well as loading and saving the data, the program allows you to insert and remove entries, and to search through the index for specific items.

```
1     REM ******************
2     REM *                *
3     REM *  64 DATAFILE   *
4     REM *                *
5     REM ******************
```

```
10    GOSUB 50000:  REM  SET UP

90    REM ****************
92    REM *                *
94    REM *  FIRST MENU    *
96    REM *                *
98    REM ****************

100   PRINT "{CLS}{CD} {RVS} 64
      DATAFILE {ROF}"
110   PRINT TAB(10)
      "{CD}{CD}{RVS}L{ROF}OAD FILE
120   PRINT TAB(10)
      "{CD}{CD}{RVS}N{ROF}EW  FILE
130   PRINT
      TAB(10)"{CD}{CD}{RVS}F1{ROF}:
       END"
200   GET K$: IF K$="" THEN 200
210   IF K$="L" THEN GOSUB 10000:
      GOSUB 1000: GOTO 100
220   IF K$="N" THEN GOSUB 11000:
      GOSUB 1000: GOTO 100
230   IF K$ <> "{F1}" THEN 200
240   PRINT "{CLS}": END
900   IF EX=1 THEN RETURN

990   REM ******************
992   REM *                  *
994   REM *  INSPECT FILE    *
996   REM *                  *
998   REM ******************

1000  GOSUB 22000: REM PRINT SCREEN
      HEADER
1010  PRINT: PRINT: PRINT TAB(10)
      "{RVS}E{ROF}NTER NEW DATA"
1020  PRINT: PRINT TAB(10)
      "{RVS}F{ROF}IND ITEM"
1030  PRINT: PRINT TAB(10)
      "{RVS}L{ROF}IST ALL ITEMS"
```

```
1040  PRINT: PRINT TAB(10)
      "{RVS}S{ROF}AVE FILE TO TAPE"
1050  PRINT: PRINT TAB(10)
      "{RVS}F1{ROF}: RETURN TO FIRST
      MENU"
1100  GET K$: IFK$="" THEN 1100
1110  IF K$="E" THEN GOSUB 2000:
      GOTO 1000
1120  IF K$="F" THEN GOSUB 3000:
      GOTO 1000
1130  IF K$="L" THEN GOSUB 4000:
      GOTO 1000
1140  IF K$="S" THEN EX=0: GOSUB
      5000: GOTO 900
1150  IF K$ <> "{F1}" THEN 1100

1190  REM
1192  REM *** SAFETY CHECK ***
1194  REM

1200  GOSUB 22000: REM PRINT SCREEN
      HEADER
1210  PRINT "{CD}{CD}  {RVS}DO YOU
      WANT TO SAVE THE FILE?{ROF}"
1220  PRINT TAB(10)
      "{CD}{CD}{RVS}Y{ROF}ES"
1225  PRINT TAB(10)
      "{CD}{RVS}N{ROF}O"
1230  GET K$: IF K$="" THEN 1230
1240  IF K$="Y" THEN GOSUB 5000:
      GOTO 1000
1250  IF K$="N" THEN RETURN
1260  GOTO 1230

1990  REM ******************
1992  REM *                *
1994  REM * ENTER NEW DATA *
1996  REM *                *
1998  REM ******************
```

```
2000  IF P < EL THEN 2100: REM CHECK
      SPACE
2010  PRINT "{CLS}" TAB(10)
      "{CD}{CD}{CD}{RVS} INDEX FULL
      {ROF}"
2020  W=3: GOSUB 30000: REM 3 SECOND
      PAUSE
2030  RETURN

2090  REM
2092  REM *** ADD NEW ITEM ***
2094  REM

2100  GOSUB 22000: REM SCREEN HEADER
2110  PRINT: PRINT: PRINT " ENTER
      DATA{CD}"
2120  FOR I=0 TO FLDS
2130  PRINT F$(I);: INPUT A$(P,I)
2140  NEXT I
2150  P = P+1

2190  REM
2192  REM *** DISPLAY NEW ITEM ***
2194  REM

2200  GOSUB 22000: REM SCREEN HEADER
2210  ID = P-1
2220  GOSUB 20000: REM DISPLAY ITEM
2230  PRINT "{CD}{CD}  PRESS
      {RVS}RETURN{ROF} TO ACCEPT
      THIS ITEM"
2240  PRINT "  OR {RVS}DEL{ROF} TO
      DELETE IT"
2250  GET K$: IF K$="" THEN2250
2260  IF K$=RET$ THEN RETURN
2270  IF K$ <> DEL$ THEN 2250
2280  GOSUB 21000: REM DELETE ITEM
2300  RETURN

2990  REM ************
2992  REM *           *
```

```
2994  REM * FIND ITEM *
2996  REM *           *
2998  REM *************

3000  GOSUB 22000: REM SCREEN HEADER
3010  IF P=0 THEN 3700: REM CHECK
      FOR ENTRIES
3020  PRINT: INPUT "FIND"; FI$
3030  PRINT "{CD}IN:";
3040  FOR I=0 TO FLDS
3050  PRINT TAB(10) I+1, F$(I)
3060  NEXT I
3100  GET K$: IF K$="" THEN 3100
3110  IF ASC(K$) < 49 OR ASC(K$) >
      49+FLDS THEN 3100
3120  F = ASC(K$)-49
3130  PRINT "{CLS}{CD}{CD}
      SEARCHING..."
3190  REM  CLEAR FOUND ITEM POINTER
3200  FOR I=0 TO P
3210  FP(I) = -1
3220  NEXT
3230  FF=0

3290  REM
3292  REM *** DO THE SEARCH ***
3294  REM

3300  FOR I=0 TO P
3310  IF A$(I,F)=FI$ THEN FP(FF)=I:
      FF=FF+1
3320  NEXT I
3400  PRINT "{CLS}{CD}{CD}{CD}   " FF
      " ITEMS FOUND"
3410  IF FF=0 THEN W=3: GOSUB 30000:
      RETURN: REM WAIT 3 SECONDS

3490  REM
3492  REM ** DISPLAY FOUND ITEMS **
3494  REM
```

```
3500  FOR I=0 TO FF-1
3510  ID = FP(I)
3520  GOSUB 20000: REM DISPLAY ITEM
3530  PRINT "{CD} PRESS {RVS}N{ROF}
      FOR NEXT ITEM"
3535  PRINT "{CD} OR {RVS}DEL{ROF}
      TO DELETE THIS ONE"
3540  GET K$: IF K$="" THEN 3540
3550  IF K$="N" THEN 3610
3560  IF K$ <> DEL$ THEN 3540
3570  GOSUB 21000: REM DELETE ITEM
3580  FOR II=0 TO FF-1
3590  FP(II) = FP(II)-1: REM RESET
      FOUND POINTERS
3600  NEXT II
3610  NEXT I
3620  RETURN

3690  REM
3692  REM IF NO ITEMS PRINT MESSAGE
3694  REM

3700  PRINT TAB(5) "{CD}{RVS} THERE
      ARE NO ITEMS TO FIND! {ROF}"
3710  W=3: GOSUB 30000: REM WAIT 3
      SECONDS
3720  RETURN

3990  REM **********************
3992  REM *                    *
3994  REM * LIST ALL ENTRIES   *
3996  REM *                    *
3998  REM **********************

4000  GOSUB 22000: REM SCREEN HEADER
4010  PRINT "{CD}{CD} LIST"
4020  IF P=0 THEN 4300: REM CHECK IF
      ANY ENTRIES
4090  REM DISPLAY ENTRIES
4100  ID=0
4110  GOSUB 20000: REM DISPLAY ITEM
```

```
4120  PRINT: PRINT "{CD} PRESS
      {RVS}N{ROF} FOR NEXT ITEM,"
4130  PRINT: PRINT " {RVS}DEL{ROF}
      TO DELETE THIS ONE"
4140  PRINT: PRINT " OR {RVS}F1{ROF}
      TO END
4150  GET K$: IF K$="" THEN 4150
4160  IF K$=DEL$ THEN GOSUB 21000:
      GOTO 4180: REM DELETE ITEM
4170  ID = ID+1
4180  IF K$ <> "{F1}" AND ID<P THEN
      4110
4200  RETURN

4290  REM
4292  REM  *** IF NO ENTRIES ***
4293  REM  *** PRINT MESSAGE ***
4294  REM

4300  PRINT TAB(5)"{CD}{CD}{RVS}
      THERE IS NOTHING TO LIST
      {ROF}"
4310  W=3: GOSUB 30000: REM WAIT 3
      SECS
4320  RETURN

4990  REM **********************
4992  REM *                    *
4994  REM * SAVE FILE TO TAPE  *
4996  REM *                    *
4998  REM **********************

5000  GOSUB 22000: REM SCREEN HEADER
5010  PRINT "{CD}{CD} SAVE FILE"
5020  PRINT: PRINT: INPUT " FILE
      NAME"; FL$
5030  PRINT"{CD}{CD}PREPARE CASSETTE
      THEN PRESS {RVS}RETURN{ROF}
5035  GET K$: IF K$ <> RET$ THEN
      5035
5040  OPEN 1, 1, 1, FL$
```

```
5050  FOR II=0 TO P-1
5060  FOR JJ=0 TO FLDS
5070  PRINT#1, A$(II,JJ)
5080  NEXT JJ, II
5090  CLOSE 1
5100  PRINT "{CD} PRESS {RVS}C{ROF}
      TO CONTINUE"
5110  PRINT "{CD} OR {RVS}F1{ROF} TO
      END"
5120  GET K$: IF K$="" THEN 5120
5130  IF K$="C" THEN RETURN
5140  IF K$="{F1}" THEN EX=1: RETURN
5150  GOTO 5120

9990  REM ************************
9992  REM *                      *
9994  REM * LOAD FILE FROM TAPE  *
9996  REM *                      *
9998  REM ************************

10000 PRINT "{CLS}{CD} {RVS} 64
      DATAFILE {ROF}"
10010 PRINT "{CD}{CD} LOAD FILE"
10020 FL$=""
10030 PRINT: PRINT: INPUT " FILE
      NAME";FL$
10040 PRINT"{CD}{CD} SET UP CASSETTE
      THEN PRESS {RVS}RETURN{ROF}"
10050 GET K$: IF K$ <> RET$ THEN
      10050
10060 OPEN 1, 1, 0, FL$
10070 P=0
10080 FOR JJ=0 TO FLDS
10090 INPUT#1, A$(P,JJ)
10110 NEXT JJ
10120 P=P+1
10130 IF P<EL AND ST<>64 THEN 10080
10140 CLOSE 1
10200 RETURN

10990 REM ************
```

```
10992 REM *           *
10994 REM * NEW FILE *
10996 REM *           *
10998 REM ************

11000 PRINT "{CLS}{CD} {RVS} 64
      DATAFILE {ROF}"
11010 PRINT "{CD}{CD} NEW FILE
11020 FOR II=0 TO EL
11030 FOR JJ=0 TO FLDS
11040 A$(II,JJ) = ""
11050 NEXT JJ, II
11060 P=0
11100 RETURN

19990 REM ********************
19992 REM *                  *
19994 REM * DISPLAY AN ITEM *
19996 REM *                  *
19998 REM ********************

20000 GOSUB 22000: REM SCREEN HEADER
20010 PRINT: PRINT " " A$(ID, 0)
20020 FOR II=1 TO FLDS
20030 PRINT: PRINT TAB(5) A$(ID,II)
20040 NEXT II
20050 RETURN

20990 REM ******************
20992 REM *                *
20994 REM * DELETE AN ITEM *
20996 REM *                *
20998 REM ******************

21000 PRINT"{CD}{RVS}DELETING THIS
      ITEM{ROF}"
21010 FOR II=ID TO P-1
21020 FOR JJ=0 TO FLDS
21030 A$(II,JJ) = A$(II+1,JJ)
21040 NEXT JJ, II
21050 P = P-1
```

```
21060 PRINT"{CD}ITEM DELETED"
21070 RETURN

21990 REM ****************
21992 REM *              *
21994 REM * SCREEN HEADER *
21996 REM *              *
21998 REM ****************

22000 PRINT "{CLS}{CD} {RVS} 64
      DATAFILE {ROF}" TAB(26) P "
      ENTRIES"
22010 RETURN

29990 REM ****************
29992 REM *              *
29994 REM *  WAIT ROUTINE *
29996 REM *              *
29998 REM ****************

30000 TI$="000000"
30010 IF TI < W*60 THEN 30010
30020 RETURN

49990 REM ****************
49992 REM *              *
49994 REM * INITIAL SET-UP *
49996 REM *              *
49998 REM ****************

50000 EL    = 100
50010 FLDS  = 6
50020 F$(0) = "SURNAME"
50030 F$(1) = "FIRST NAME"
50040 F$(2) = "HOUSE AND STREET"
50050 F$(3) = "TOWN"
50060 F$(4) = "COUNTY"
50070 F$(5) = "POST CODE"
50080 F$(6) = "PHONE NUMBER"
50100 DIM A$(EL, FLDS)
50110 DIM FP(EL)
```

```
50200 RET$ = CHR$(13)
50210 DEL$ = CHR$(20)
50300 RETURN
```

Program Description

The program is controlled using *menus*. A list of options is displayed, and you select the one you want by pressing the appropriate key - usually the first letter of the option title.

The program is written in *modules*. A main menu routine calls a number of subroutines which are independent of each other and perform different data-management tasks: entering new data, listing the data and so on.

The first instruction of the program calls the subroutine at 50000 which sets up the array in which the data is stored and the other main variables of the program.

Lines 100 - 240 display an initial menu offering the options:

Load a file from tape
Create a new file
Stop the program

Pressing N for New or L for Load carries out the selected task and then subroutine 1000 (INSPECT FILE) is called. This subroutine is the heart of the program. A second menu is displayed, offering the options:

Enter new data
Find an item of data
List all the items
Save the file on tape
Return to the first menu

and the appropriate subroutine is called when an item is selected. If F1 is pressed the user is asked whether the file should be saved on the tape before the program returns to the first menu.

The subsidiary routines of the program are:

ENTER NEW DATA (2000) adds new data to the file.

FIND ITEM (3000) searches for specified items.

LIST ALL ITEMS (4000) lists all the entries in the file.

SAVE FILE (5000) saves the file on tape.

LOAD FILE (10000) loads a file from the tape.

NEW FILE (11000) blanks out the file and sets the entry counter (P) to zero.

DISPLAY ITEM (20000) displays one entry of the file.

DELETE ITEM (21000) deletes an entry.

SCREEN HEADER (22000) prints a message at the top of the screen.

The file is stored in the array A$, which is dimensioned in 50100. The other main variables used are:

P	Pointer to the first free entry (and therefore also a count of the number of entries in the file).
EL	The maximum number of entries.

FLDS	The number of 'fields' in each entry - the second dimension of A$.
F$( )	The names of the fields.
FL$	The filename when loading or saving the data.
ID	The item to be displayed by DISPLAY ITEM.
FF	The number of entries found in FIND ITEM.
FP( )	Pointers to the found items.

I, II, J and JJ are used as counters in loops.

To alter the program to store different information, change the names of the fields in lines 50020 onwards. If you alter the number of fields, change FLDS in line 50010 to the number of the last field.

If you want to add more features they may easily be 'plugged in'. For example, a routine to sort the entries into alphabetical order (perhaps based on the string sorting program in Chapter 7 - page 66) could be added as subroutine 6000 by adding two lines to INSPECT FILE to display an extra menu item and call the subroutine.

CHAPTER 16

ADVANCED TECHNIQUES

The previous chapters cover all you need to write many programs for your 64. This chapter decribes some final BASIC commands which you will find useful as you write more ambitious programs, and rounds off with some general advice on how to write good programs.

MACHINE CODE SUBROUTINES

The 64's microprocessor does not understand BASIC programs without some assistance; it understands a much cruder set of instructions called **machine code** or **machine language**. BASIC programs can be run only because the 64 has some machine code programs permanently stored in read-only memory (ROM) which interpret the BASIC. Writing programs in machine code is more difficult than using BASIC, but can be very worthwhile as machine code programs run very much faster. Machine code is beyond the scope of this book, but there are several books available about the 64's microprocessor (the 6510, which is a variant of the popular 6502 and has the same instruction code) as it is very common in personal computers. The companion volume to this book, *The Advanced Commodore 64 Handbook*, describes the use of machine code with the 64, and gives useful routines to extend the 64's facilities.

You can enter machine code routines from BASIC using the command **SYS**, giving the start address

in memory of the machine code routine. For example:

```
SYS 2048
```

will perform the same function as holding down the RUN/STOP key and pressing RESTORE, and:

```
SYS 64738
```

will call the power-on setting up routines which clear the memory and print the start-up message. **BE CAREFUL** using this - it will destroy any program you have in the 64 at the time you use it.

Machine code routines may also be called by the instruction **USR( )**. The instruction is used like a BASIC function. For example:

```
1000  A = USR(X)
```

The number in brackets is stored in the 64's floating point accumulator, and then a previously defined machine code routine is called. At the end of the routine, the number now in the floating point accumulator is returned to be stored in the variable.

The address of the machine code to be run is defined by **POKE**ing the low and high bytes of the start address into locations 785 and 786. For example,

```
POKE 785, 60: POKE 786, 3
```

would cause subsequent **USR( )** instructions to run the code starting at address 828 (the beginning of the cassette buffer).

THE WAIT COMMAND

The **WAIT** command is **not** a pause command, as it is on many machines. The command has the format:

```
WAIT MEM, X, Y
```

(the Y is optional) and works as follows:

The contents of memory location MEM are exclusively ORed with the variable Y, and the result is ANDed with the variable X. If the result is zero, the 64 checks the memory location again, until a non-zero result is obtained, when the program continues. In other words, the program is halted until the contents of a memory location become equal to a specfied number.

WAIT is almost never used. The command does nothing that can not be acheived using **PEEK** and **IF**, and it has the disadvantage that the STOP key can not be used to end the program while the **WAIT** command is in operation. This means that if the right answer is not found, the 64 can lock up, and only RUN/STOP and RESTORE will let you escape. **WAIT** is only useful when a very fast response is required, as for example when the computer is being used to control other equipment.

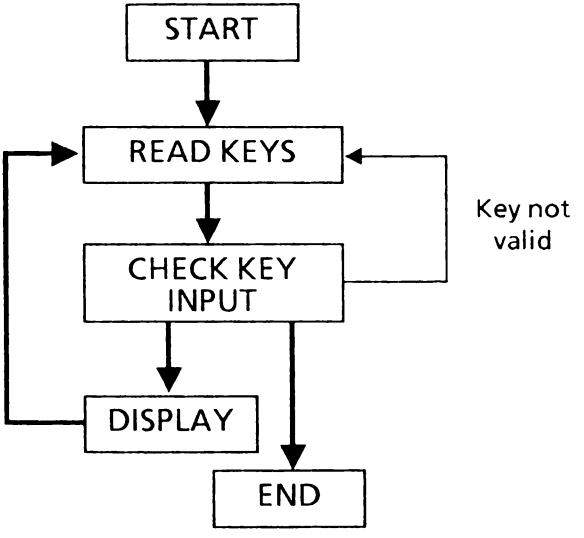
DESIGNING GOOD PROGRAMS

It is easy to write simple BASIC programs, and after some practice you will find it fairly easy to write quite complicated routines. However, unless you plan your programs carefully, a long program can get very messy and it can be very difficult to find all the mistakes you are bound to make.

To write complex programs successfully you must design them carefully, following a few simple rules.

The phrase **Structured Programming** is often used to describe these rules ; all this means is that programs designed in accordance with these rules have a clear and logical structure, and the flow through the program is easy to follow. You may be put off by hearing people say that BASIC is not a suitable language for structured programming. Don't let this discourage you. While BASIC may lack the elegance of some other languages, it is still possible to use it to write good programs by following the simple guidelines set out here:

1 Always plan out the program on paper. Decide what you want the program to do, identifying the various tasks it will have to perform, and divide the program into sections which correspond to the tasks. For example, you might have one section of program to get data from the keyboard, another to sort the data, a third to display results, and so on.



Sketch programs before writing the BASIC

2 It is often useful to draw sketches of the way the tasks fit together.

3 When you have got the overall structure of the program sorted out, work out in more detail what each section should be doing, and how it should be done.

4 Write the BASIC routines for each section of the program - writing them on paper and **not** typing straight into the 64. Try to write each section of the program so that it is as independent as possible of the other sections, and arrange the routines with a clear sequential flow from beginning to end: avoid jumping around with **GOTO**. Use a different set of line numbers for each section, beginning each at a round number of so many thousands or tens of thousands.

5 Now type the program into the 64.

6 The program will not work! At this stage you can sort out all the SYNTAX ERRORs easily. If you have designed the program well, with different sections doing different tasks, it will be much easier to find the more subtle errors, as you will be able to isolate the faulty bit of the program without difficulty.

If you follow these guidelines, you will save a lot of time in getting your programs to work, and you will have more time for actually using them.

APPENDIX 1

ABBREVIATIONS

LISTING	MEANING	KEYS TO PRESS
{BLK}	BLACK	CTRL and 1
{WHT}	WHITE	CTRL and 2
{RED}	RED	CTRL and 3
{CYN}	CYAN	CTRL and 4
{PUR}	PURPLE	CTRL and 5
{GRN}	GREEN	CTRL and 6
{BLU}	BLUE	CTRL and 7
{YEL}	YELLOW	CTRL and 8
{ORG}	ORANGE	CBM and 1
{BRN}	BROWN	CBM and 2
{LTRED}	LIGHT RED	CBM and 3
{DKGRY}	DARK GREY	CBM and 4
{GRY}	GREY	CBM and 5
{LTGRN}	LIGHT GREEN	CBM and 6
{LTBLU}	LIGHT BLUE	CBM and 7
{LTGRY}	LIGHT GREY	CBM and 8
{RVS}	REVERSE VIDEO ON	CTRL and 9
{ROF}	REVERSE VIDEO OFF	CTRL and 0
{CLS}	CLEAR SCREEN	SHIFT and CLR / HOME
{HOM}	CURSOR HOME	CLR / HOME
{CU}	CURSOR UP	SHIFT and ↑ CRSR ↓
{CD}	CURSOR DOWN	↑ CRSR ↓
{CL}	CURSOR LEFT	SHIFT and CRSR ⇆
{CR}	CURSOR RIGHT	CRSR ⇆

APPENDIX 2

BASIC COMMANDS

ABS (N)

Returns the absolute value of a number (removing the minus sign).
See Chapter 8

AND

Logical operator. Returns the value TRUE if both operands are true (1).

A = B AND C A is TRUE if B and C are both TRUE.

Can also operate on binary values of numbers.
Chapter 9

ASC (C$)

Function. Returns the ASCII code of a character, or of the first character of a string.
Chapter 7

ATN (N)

Function. Gives the angle in radians whose arctangent is N.
Chapter 8

CHR$ (N)

Converts ASCII codes to characters in string form.
Chapter 7

CLOSE (N) — Closes a channel, N, to a peripheral device (disk, printer, etc)
Chapter 15

CLR — Clears variables, arrays, etc, from memory, and makes the memory available to BASIC programs.
Chapter 16

CMD (N) — Transfers output to the specified file number. Any output normally printed on the screen will be directed to the device specified, in the same format.
Chapter 15

CONT — Restarts program after a break. Only possible if no alterations have been made to the program.
Chapter 5

COS (N) — Gives cosine of angle, which must be given in radians.
Chapter 8

DATA — Marks a list of string or numeric data written into a program. The items must be separated by commas.
Chapter 6

DEF FNA(N) — Defines a user - definable function.
Chapter 8

DIM A(L,M)

Dimensions arrays. Arrays of one dimension with up to 10 elements may be used without DIM.
Chapter 4

END

Ends program. Program may be restarted using CONT.
Chapter 5

EXP (N)

Function returning exponential of N (e^N). Acts as natural antilog function.
Chapter 8

FNA (N)

Calls function defined by DEF FN.
Chapter 8

FOR

Begins loop. For example:

```
FOR N = A TO B STEP C
```

All lines as far as NEXT command are repeated with value of N increased each time from A to B in steps of C. STEP may be omitted, in which case variable is increased by 1.
Chapter 5

FRE (0)

Returns the amount of memory at present unused by BASIC.
Chapter 10

GET N

Used for single character input. Assigns key value to variable. The variable may

be a string variable, in which case any key may be pressed, or a number variable, for which number keys only may be pressed.
Chapter 5

GET # (N) — Reads a single character from the specified file or device. Similar to GET, above.
Chapter 15

GOSUB — Program branches to a subroutine at the specified line, returning to instruction after GOSUB when RETURN is encountered.
Chapter 5

GOTO — Program branches to the specified line.
Chapter 5

IF (condition) **THEN** (action) — If the condition is true the action after THEN is carried out, otherwise the program continues at the next line.
Chapter 5 and Chapter 9

INPUT N
INPUT "data";N — Prompts the operator for an input and assigns it to a variable. The variable may be a number or string: the input data must correspond.
Chapter 4

INPUT #(N) — Retrieves data from the specified file number. Data is in the form of strings up to 80 characters in length,

delimited by CHR$(13) , (,), (;) or (:).
Chapter 15

INT (N) — Returns integer component of real number. For example:

```
INT(3.75)
```

returns 3.
Chapter 8

**LEFT$**(C$,N) — Returns the first N characters of the string.
Chapter 7

LEN(C$) — Gives the number of characters in the string.
Chapter 7

LET — Optional. May be used when assigning values to variables:

```
LET P = 5   and

P = 5
```

have the same effect.

LIST — Lists the specified lines of the program on to the screen.
Chapter 4

LOAD A$, D, A — Reads program file A$ from device number D (if unspecified, default=1, the tape unit). Secondary address A specifies start address of program.
Chapter 15

LOG (N) — Returns the base 10 logarithm of a number.
Chapter 8

**MID$** (C$,X,N) — Returns N characters of string beginning at the Xth character.
Chapter 7

NEW — Clears a program and its variables from memory.
Chapter 4

NEXT N — Marks end of loop begun by FOR. The variable need not be specified.
Chapter 5

NOT — Logical operator. Reverses truth of expression (e.g. NOT TRUE returns FALSE). Can also be applied to binary values of numbers.
Chapter 9

ON N **GOSUB** — Program branches to Nth subroutine in list. If N is larger than number of items in list no branch occurs.
Chapter 5

ON N **GOTO** — Program branches to Nth destination in list. If N larger than number of items in list no branch occurs.
Chapter 5

OR — Logical operator. Returns value TRUE if either or both operands are true. Thus:

```
A = B OR C
```

A is true if B or C is true. Can also act on binary values of numbers.
Chapter 9

PEEK (LOC) — Gives the contents of memory location LOC.
Chapter 15

POKE LOC,N — Puts value of N into memory location LOC. N must have value 0 - 255.
Chapter 15

POS (0) — Gives the present horizontal position of the cursor on the display. The number in brackets has no significance.
Chapter 4

PRINT — Puts data, numbers or characters on the screen. May be written as '?'.
Chapter 3

PRINT # N — Writes data to the specified file.
Chapter 15

READ — Copies items from DATA statements into variables.
Chapter 6

REM — Allows remarks to be inserted in programs as an aid to clarity. Remarks are ignored when program is run.
Chapter 4

RESTORE — Returns READ pointer to first DATA item.
Chapter 6

RETURN — Marks end of subroutine. Program returns to the instruction after the GOSUB instruction which called the subroutine.
Chapter 5

**RIGHT$** (C$,N) — Returns the N rightmost characters of string C$.
Chapter 7

RND (N) — If N > 0, returns random number.

If N < 0, reseeds random number generator.
Chapter 8

RUN — Begins execution of BASIC program. All variables are cleared. A line number may be given, otherwise execution begins at the lowest numbered line.
Chapter 4

SAVE A$, D, A
Stores the program currently in memory onto tape or disk., with the name A$. Device number D specifies tape(1) or disk(8).
Chapter 15

SGN (N)
Returns 1, 0 or -1 according to whether number is positive, zero or negative.
Chapter 8

SIN
Function. Returns the sine of a number in radians.
Chapter 8

SPC (N)
Prints N spaces on the screen. N must be 0-255.
Chapter 4

SQR (N)
Returns the square root of the argument.
Chapter 8

STATUS
Returns status of the last Input / Output operation. Abbreviated to **ST**.
Chapter 8

STOP
Stops execution of program. A message 'BREAK IN xxx' is printed. Program may be restarted using CONT.
Chapter 4

STR$ (N)
Converts numbers to strings of numeric characters.
Chapter 7

SYS (N) — Calls a machine code subroutine at the specified address.
Chapter 16

TAB (N) — Moves the cursor N places from the left edge of the screen in a PRINT statement.
Chapter 4

TAN (N) — Gives the tangent of an angle, which must be in radians.
Chapter 8

THEN — See IF.
Chapter 5

TIME — Abbreviated to **TI**. Reads the system clock.
Chapter 5

TIME$** — Abbreviated to **TI$. Returns the time since power up or reset in Hours, Minutes and Seconds.
Chapter 5

TO — See FOR.
Chapter 5

USR (N) — Calls machine code routine at address previously specified. The value in brackets is placed in the floating point accumulator. Returns the contents of the floating point accumulator after subroutine is run.
Chapter 16

VAL (C$)	Converts string of number characters to number. Chapter 7
VERIFY	Compares a specified program file with the program currently in memory to check for correct saving. Chapter 15
WAIT LOC,X,Y	Program pauses until the contents of a memory location change in a specified way. Chapter 16

APPENDIX 3

BASIC ERROR MESSAGES

If the CBM 64 encounters a command which it is unable to execute, or a number it cannot handle, an error message will be displayed.

If the error occurred while running a program, the program will stop, and a message of the form

***** ERROR IN XXXX

will be displayed, where ***** represents the type of error, and XXXX is the program line at which the error has occurred. The program is retained in the computer, as are the values assigned to all variables at the time of the error.

In immediate mode, the error message takes the form

***** ERROR

The following descriptions explain the error messages, and the possible reasons for them:

BAD DATA

The data received from a file is not of the type expected. For example, the file contains string data but the program is trying to input numeric data.

BAD SUBSCRIPT

An array element outside the dimensions of that array has been accessed, or the wrong number of dimensions has been used, for example:

`Z(9,9,9) = A` .... when Z is a two-dimensional array.

BREAK

Displayed when program is halted with the RUN/STOP key.

CAN'T CONTINUE

You may not continue the **RUN**ning of a program, using the command **CONT**, if the program has been edited in any way.

DEVICE NOT PRESENT

Displayed when the specified device is not connected to the computer, and an attempt has been made to access files on that device.

DIVISION BY ZERO

It is impossible to divide by zero.

EXTRA IGNORED

An attempt has been made to enter too many items in response to an **INPUT** statement. The extra data is not considered.

FILE NOT FOUND

With tape systems, this means that an end of file marker has been encountered. In the case of disk units the file does not exist on this disk.

FILE NOT OPEN

A file which has not previously been **OPEN**ed has been specified in a file handling command.

FORMULA TOO COMPLEX

A series of string handling functions are too complex to be carried out in one step. They must be split into smaller steps.

ILLEGAL DIRECT

The following BASIC commands may not be used in Immediate mode:

DEF FN
GET
GET #
INPUT
INPUT #

ILLEGAL QUANTITY

A parameter passed to a mathematical or string function was outside the allowed range for that parameter.

LOAD

The computer has been unable to **LOAD** the program from tape and has aborted the attempt.

NEXT WITHOUT FOR

The variable in a **NEXT** statement has no corresponding **FOR** statement or **FOR ... NEXT** loops have been incorrectly nested.

NOT INPUT FILE

An attempt has been made to read data from a file which has previously been specified as a write only file.

NOT OUTPUT FILE

An attempt has been made to write data to a file which has previously been specified as a read only file.

OUT OF DATA

This error occurs when a **READ** command is executed, but there is either no **DATA**, or all the **DATA** has previously been read. The program either tried to read too much data, or there was insufficient data in the **DATA** statement.

OUT OF MEMORY

Either the program is too large, or you have used too many variables, too many **FOR ... NEXT** loops, too many **GOSUB**s, or you have allocated too much space for arrays with the **DIM** command.

OVERFLOW

The result of a calculation, or a directly entered number, was larger than 1.70141 * E38. If an underflow occurs, the result is given as zero, and *no* error message is displayed.

REDIM'D ARRAY

After an array was **DIM**ensioned, a statement **DIM**ensioning the same array was encountered.

REDO FROM START

A string function has been entered when the program was executing an **INPUT** statement. The message will be repeated until a number is entered.

RETURN WITHOUT GOSUB

An attempt has been made to execute a **RETURN** statement which was not preceded by a **GOSUB** statement.

STRING TOO LONG

A string has been concatenated - made up from several smaller strings joined together - and its length has exceeded 255 characters.

SYNTAX

This can be caused by :

Use of a non-existent (or mispelled) BASIC command
Missing brackets
Incorrect punctuation
Missing parameters
Illegal characters

TYPE MISMATCH

An attempt has been made to assign a numeric value to a string variable, or vice-versa . A numeric argument was passed to a function requiring a string argument, or vice-versa. For example:

```
PRINT ASC(A) instead of
PRINT ASC("A")

A=Z$ instead of A=Z
or A$=2 instead of A$="2"
```

UNDEF'D FUNCTION

An attempt to reference an undefined user defined function has been made.

UNDEF'D STATEMENT

The program attempted to **GOTO**, **GOSUB**, **THEN** or **RUN** a non-existent statement.

VERIFY

The program on tape is not the same as the program currently in memory.

APPENDIX 4

SPEEDING UP PROGRAMS

There are several things you can do to increase the running speed of your BASIC programs. Unfortunately, this is usually at the expense of clarity, and you may find it useful to keep a 'slow', but easy-to-follow version of your program should you wish to amend it at a later date.

1 Remove all unnecessary spaces, **REM**s and indentation from the program. A small speed increase will result, because BASIC will not have to skip over redundant spaces to find executable commands.

2 Always use variables instead of constants. The CBM 64 can handle variables much more rapidly than numbers. This is especially important in **FOR ... NEXT** loops.

3 Use as many statements per line (separated by ':') as possible.

4 Re-use the same variables whenever possible.

5 Use the zero elements of arrays when possible.

6 Assign often - used variables early on in the program. The 64 stores all variables in a table. The first declared variables are the first in the table and are found more quickly.

7 Put all subroutines near the start of the program. The computer searches through the whole program for a subroutine each time a **GOSUB** command is executed, and subroutines having low line numbers will be found and executed more quickly than those at the end of the program.

8 Omit the variable after **NEXT** in **FOR... NEXT** loops.

9 In programs where the keyboard is not often used, the routines which scan the keyboard can be disabled within a program with a line :

```
100 POKE 56334,PEEK(56334) AND
    254:POKE 1,PEEK(1) AND 251
```

and enabled each time you want to use **INPUT** or **GET** by

```
200 POKE 1,PEEK(1) OR 4:POKE
    56334,PEEK(56334) OR 1
```

Remember to include an enable statement before the program ends, to allow you to regain control of the computer!

10 Programs involving large amounts of calculation but not requiring the screen display may be speeded up by turning off the screen, with a line such as :

```
100 POKE 53265,PEEK(53265)AND 239
```

The screen can be turned back on to display the results of the calculations with a statement:

```
200 POKE 53265,PEEK(53265) OR 16
```

APPENDIX 5

COLOUR CODES

NOTE Only colours 0 to 7 can be used in multi-colour character mode

COLOUR	CODE
BLACK	0
WHITE	1
RED	2
CYAN	3
PURPLE	4
GREEN	5
BLUE	6
YELLOW	7
ORANGE	8
BROWN	9
LIGHT RED	10
LIGHT GREY	11
MEDIUM GREY	12
LIGHT GREEN	13
LIGHT BLUE	14
DARK GREY	15

APPENDIX 6

NUMBERING SYSTEMS

Computers store and operate upon numbers in a different way from humans - they use a numbering system known as **Binary Notation**.

Binary notation is a means of representing quantities with groups of 1s and 0s. We are more used to a system called Decimal Notation, in which quantities can be represented by combinations of up to ten symbols (the numbers 0 to 9).

Computers use the binary system because they are able to recognise and differentiate between only two states - ON and OFF. These two states can conveniently be represented by 1 (ON) and 0 (OFF).

A single 1 or 0 is called a **BI**nary digi**T**, or **BIT**, and computers store data in the form of groups of eight of these bits, known as **BYTES**.

In the computer memory, one memory location is able to store one Byte of data (eight bits). A collection of 1024 of these bytes is called a **KILOBYTE**, or **k** for short. We can get an idea of the data storage capacity of a computer from the number of k of memory it has (48k, for example, is 48 * 1024 * 8 bits).

We have described a byte as a collection of eight bits, like this:

```
11111111
```

This is an 8-bit binary number, which represents 255 in decimal notation. To see how this is so, we must first examine the decimal number and see what it means.

```
H  T  U
2  5  5
```

means:

```
2*100 + 5*10 + 5*1
```

In other words, each digit is worth 10 times the one to its right.

Binary notation uses this same 'place value' principle, except each bit in a binary number is worth **double** that to its right. We can assign values to the eight bits in the same way as the 'hundreds, tens and units' assigned to the digits of a decimal number.

```
128 64  32  16  8   4   2   1
  1  1   1   1  1   1   1   1
```

By adding up, we can see why this number represents 255:

```
1 *128
1 * 64
1 * 32
1 * 16
1 *  8
1 *  4
1 *  2
1 *  1   +
-------
   255
```

You'll notice that 255 is the biggest number we can represent with an 8-bit binary number. Hence this

is the largest number we can store in a single memory location.

As a further example, let's take the decimal number 170. To find its binary representation, we continuously divide by two, and the remainder becomes a bit in the binary number.

```
170/2  =  85  remainder  0
 85/2  =  42       "     1
 42/2  =  21       "     0
 21/2  =  10       "     1
 10/2  =   5       "     0
  5/2  =   2       "     1
  2/2  =   1       "     0
  1/2  =   0       "     1
```

giving us the binary number

```
10101010        (reading upwards)
```

ADDITION OF BINARY NUMBERS

Binary numbers can be added together in the same way as decimal numbers. An example will make this clear.

To perform the sum:

```
105  +
 19
---
124
```

we add up the digits in each column to form that digit of the answer. If the result of this addition is greater than 9, we generate a carry into the next column. This principle also applies to binary numbers. Let's perform the same calculation with the binary forms of 105 and 19:

```
01101001   +
00010011
--------
01111100
```

In the case of binary addition we generate a carry when adding 1 to 1 (in columns 1 and 2 in our example).

NEGATIVE BINARY NUMBERS

You might have wondered how the computer can recognise negative numbers, since it can only tell the difference between on and off. This is achieved by a method known as 'two's complement' notation, which uses one bit of the binary number (the most significant bit, often labelled bit 7) as a sign bit.

To subtract two numbers in binary, we form the two's complement of the number to be subtracted, then **add** it to the other number.

As an example we'll subtract 50 from 100 :

```
100  -      01100100
 50         00110010
----        --------
 50         00110010
```

(the answer we want).

To perform the sum in binary, first we find the two's complement of 50, by changing all the 0s to 1s and all the 1s to 0s, then adding 1.

```
50 in binary is:          00110010
Change 1s to 0s:          11001101
Adding 1 gives            --------
Its two's complement      11001110
```

Now add this to the binary for 100:

```
          01100100
          11001110
      1   00110010
```

The result is binary 50 - the method worked.

Notice that a carry was generated, indicating that the answer is positive. A consequence of using bit 7 as a sign bit is that the range of numbers we can represent with an 8-bit number is restricted to

-128 to +127

HEXADECIMAL

Manipulating numbers in binary is a lot easier for a computer than it is for a human, and one way in which binary numbers can be made more digestible is by representing them in hexadecimal notation, or **hex.**

Hex is a system of counting in base 16, using the symbols 0 to 9 and A to F as follows

```
DECIMAL 0 1...9 10 11...15 16 17

HEX     0 1...9  A  B... F 10 11
```

Thus FF (hex) represents 255 (decimal) and 11111111(binary).

You will frequently encounter references to hex, usually as memory addresses, because it is so convenient. (Which of these is easiest to recognise ? 1111111111111111 or 65535 or #FFFF).

APPENDIX 7

SID REGISTERS

ADDRESS	7	6	5	4	3	2	1	0	REGISTER FUNCTION
									VOICE 1
54272	F 7	F 6	F 5	F 4	F 3	F 2	F 1	F 0	LOW FREQ
54273	F 15	F 14	F 13	F 12	F 11	F 10	F 9	F 8	HIGH FREQ
54274	PW 7	PW 6	PW 5	PW 4	PW 3	PW 2	PW 1	PW 0	PULSE WIDTH LOW
54275	-	-	-	-	PW11	PW10	PW 9	PW 8	PULSE WIDTH HIGH
54276	NOI	PUL	SAW	TRI	TEST	RING	SYNC	GATE	CONTROL REGISTER
54277	ATK 3	ATK 2	ATK 1	ATK 0	DEC 3	DEC 2	DEC 1	DEC 0	ATTACK / DECAY
54278	SUS 3	SUS 2	SUS 1	SUS 0	REL 3	REL 2	REL 1	REL 0	SUSTAIN / RELEASE
									VOICE 2
54279	F 7	F 6	F 5	F 4	F 3	F 2	F 1	F 0	LOW FREQ
54280	F 15	F 14	F 13	F 12	F 11	F 10	F 9	F 8	HIGH FREQ
54281	PW 7	PW 6	PW 5	PW 4	PW 3	PW 2	PW 1	PW 0	PULSE WIDTH LOW
54282	-	-	-	-	PW11	PW10	PW 9	PW 8	PULSE WIDTH HIGH
54283	NOI	PUL	SAW	TRI	TEST	RING	SYNC	GATE	CONTROL REGISTER
54284	ATK3	ATK2	ATK1	ATK0	DEC 3	DEC 2	DEC 1	DEC 0	ATTACK / DECAY
54285	SUS 3	SUS 2	SUS 1	SUS 0	REL3	REL 2	REL 1	REL 0	SUSTAIN / RELEASE

ADDRESS	7	6	5	4	3	2	1	0	REGISTER FUNCTION
									VOICE 3
54286	F 7	F 6	F 5	F 4	F 3	F 2	F 1	F 0	LOW FREQ
54287	F 15	F 14	F 13	F 12	F 11	F 10	F 9	F 8	HIGH FREQ
54288	PW 7	PW 6	PW 5	PW 4	PW 3	PW 2	PW 1	PW 0	PULSE WIDTH LOW
54289	-	-	-	-	PW11	PW10	PW 9	PW 8	PULSE WIDTH HIGH
54290	NOI	PUL	SAW	TRI	TEST	RING	SYNC	GATE	CONTROL REGISTER
54291	ATK 3	ATK 2	ATK 1	ATK 0	DEC 3	DEC 2	DEC 1	DEC 0	ATTACK / DECAY
54292	SUS 3	SUS 2	SUS 1	SUS 0	REL 3	REL 2	REL 1	REL 0	SUSTAIN / RELEASE
									FILTER
54293	-	-	-	-	-	FC 2	FC 1	FC 0	LOW FILTER
54294	FC 10	FC 9	FC 8	FC 7	FC 6	FC 5	FC 4	FC 3	HIGH FILTER
54295	RES 3	RES 2	RES 1	RES 0	FILTX	FILT 3	FILT 2	FILT 1	RESONANCE / FILTER
54296	3 OFF	HP	BP	LP	VOL 3	VOL 2	VOL 1	VOL 0	MODE / VOLUME
									MISCELLANEOUS
54297	PX 7	PX 6	PX 5	PX 4	PX 3	PX 2	PX 1	PX 0	POT X
54298	PY 7	PY 6	PY 5	PY 4	PY 3	PY 2	PY 1	PY 0	POT Y
54299	OSC 7	OSC 6	OSC 5	OSC 4	OSC 3	OSC 2	OSC 1	OSC 0	OSCIL 3 / RANDOM
54300	ENV 7	ENV 6	ENV 5	ENV 4	ENV 3	ENV 2	ENV 1	ENV 0	VOICE 3 ENVELOPE

APPENDIX 8

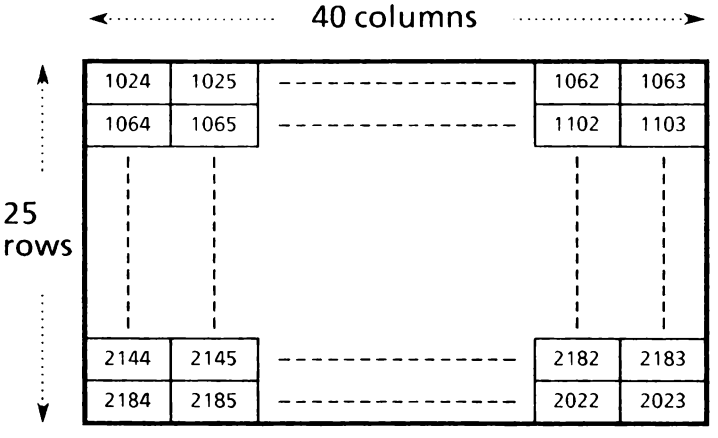
VIC II REGISTERS

ADDRESS	FUNCTION
53248	SPRITE 0 X POSITION
53249	SPRITE 0 Y POSITION
53250	SPRITE 1 X POSITION
53251	SPRITE 1 Y POSITION
53252	SPRITE 2 X POSITION
53253	SPRITE 2 Y POSITION
53254	SPRITE 3 X POSITION
53255	SPRITE 3 Y POSITION
53256	SPRITE 4 X POSITION
53257	SPRITE 4 Y POSITION
53258	SPRITE 5 X POSITION
53259	SPRITE 5 Y POSITION
53260	SPRITE 6 X POSITION
53261	SPRITE 6 Y POSITION
53262	SPRITE 7 X POSITION
53263	SPRITE 7 Y POSITION
53264	SPRITE X POSITION MSB
53265	CONTROL REGISTER
53266	RASTER REGISTER
53267	LIGHT PEN X POSITION
53268	LIGHT PEN Y POSITION
53269	SPRITE ENABLE

53270	CONTROL REGISTER
53271	SPRITE Y EXPAND
53272	VIDEO MEMORY POINTERS
53273	INTERRUPT REGISTER
53274	ENABLE INTERRUPT
53275	SPRITE TO DATA PRIORITY
53276	SPRITE MULTI COLOUR
53277	SPRITE X EXPAND
53278	SPRITE - SPRITE COLLISION
53279	SPRITE - DATA COLLISION
53280	BORDER COLOUR
53281	BACKGOUND COLOUR 0
53282	BACKGOUND COLOUR 1
53283	BACKGOUND COLOUR 2
53284	BACKGOUND COLOUR 3
53285	SPRITE MUTLTICOLOUR 0
53286	SPRITE MUTLTICOLOUR 1
53287	SPRITE 0 COLOUR
53288	SPRITE 1 COLOUR
53289	SPRITE 2 COLOUR
53290	SPRITE 3 COLOUR
53291	SPRITE 4 COLOUR
53292	SPRITE 5 COLOUR
53293	SPRITE 6 COLOUR
53294	SPRITE 7 COLOUR

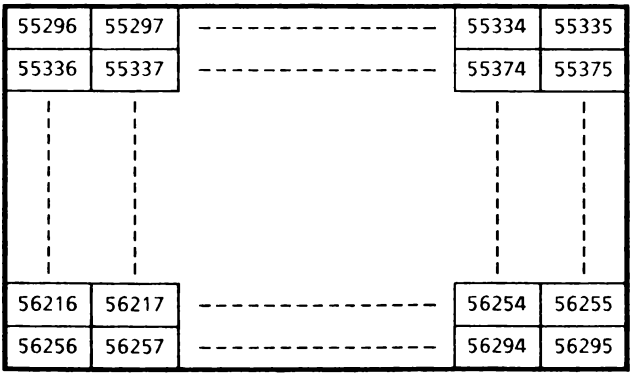
APPENDIX 9

SCREEN MEMORY MAPS

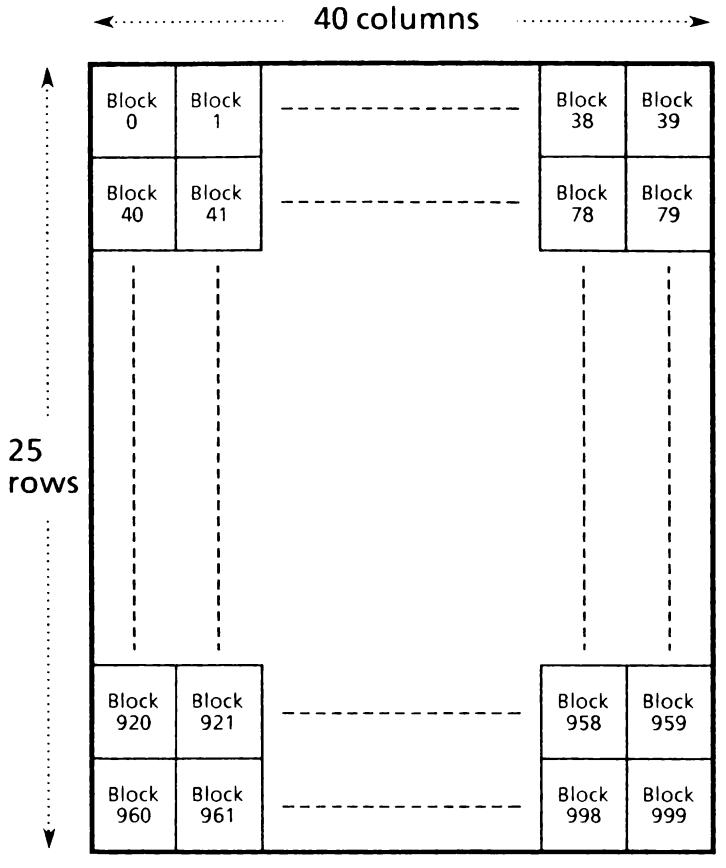


The Character Screen Memory

Each block represents 1 memory location



The Colour Memory



The Bit Mapped Screen Memory

Each block corresponds to eight memory locations laid out like this:

The address of any byte is:
Screen start + 8*Block No + Byte No

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

APPENDIX 10

CHARACTER CODES

SCREEN DISPLAY CODES

POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2
0	@	@	21	U	u	42	*	:
1	A	a	22	V	v	43	+	+
2	B	b	23	W	w	44	,	,
3	C	c	24	X	x	45	-	-
4	D	d	25	Y	y	46		.
5	E	e	26	Z	z	47	/	/
6	F	f	27	[	[	48	0	0
7	G	g	28	£	£	49	1	1
8	H	h	29	]	]	50	2	2
9	I	i	30	↑	↑	51	3	3
10	J	j	31	←	←	52	4	4
11	K	k	32	SPACE	SPACE	53	5	5
12	L	l	33	!	!	54	6	6
13	M	m	34	"	"	55	7	7
14	N	n	35	#	#	56	8	8
15	O	o	36	$	$	57	9	9
16	P	p	37	%	%	58	:	:
17	Q	q	38	&	&	59	;	;
18	R	r	39	'	'	60	<	<
19	S	s	40	(	(	61	=	=
20	T	t	41	)	)	62	>	>

POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2
63	?	?	85		U	107		
64			86		V	108		
65		A	87		W	109		
66		B	88		X	110		
67		C	89		Y	111		
68		D	90		Z	112		
69		E	91			113		
70		F	92			114		
71		G	93			115		
72		H	94			116		
73		I	95			117		
74		J	96	SPACE	SPACE	118		
75		K	97			119		
76		L	98			120		
77		M	99			121		
78		N	100			122		
79		O	101			123		
80		P	102			124		
81		Q	103			125		
82		R	104			126		
83		S	105			127		
84		T	106					

The codes from 128 to 255 give the same characters in inverse video.

ASCII AND CHR$ CODES

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
0		28	RED	56	8
1		29	CRSR RIGHT	57	9
2		30	GREEN	58	:
3		31	BLUE	59	;
4		32	SPACE	60	<
5	WHITE	33	!	61	=
6		34	"	62	>
7		35	#	63	?
8	Dis SHIFT C =	36	$	64	@
9	En SHIFT C =	37	%	65	A
10		38	&	66	B
11		39	'	67	C
12		40	(	68	D
13	RETURN	41	)	69	E
14	LOWER CASE	42	*	70	F
15		43	+	71	G
16		44	,	72	H
17	CRSR DOWN	45	-	73	I
18	RVS ON	46		74	J
19	HOME	47	/	75	K
20	DEL	48	0	76	L
21		49	1	77	M
22		50	2	78	N
23		51	3	79	O
24		52	4	80	P
25		53	5	81	Q
26		54	6	82	R
27		55	7	83	S

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
84	T	113		142	UPPER CASE
85	U	114		143	
86	V	115		144	BLACK
87	W	116		145	CRSR UP
88	X	117		146	RVS OFF
89	Y	118		147	CLR
90	Z	119		148	INST
91	[	120		149	
92	£	121		150	
93	]	122		151	
94	↑	123		152	
95	←	124		153	
96		125		154	
97		126	π	155	
98		127		156	PURPLE
99		128		157	CRSR LEFT
100		129		158	YELLOW
101		130		159	CYAN
102		131		160	SPACE
103		132		161	
104		133	F1	162	
105		134	F3	163	
106		135	F5	164	
107		136	F7	165	
108		137	F2	166	
109		138	F4	167	
110		139	F6	168	
111		140	F8	169	
112		141	SHIFT RETURN	170	

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
171		178		185	
172		179		186	
173		180		187	
174		181		188	
175		182		189	
176		183		190	
177		184		191	

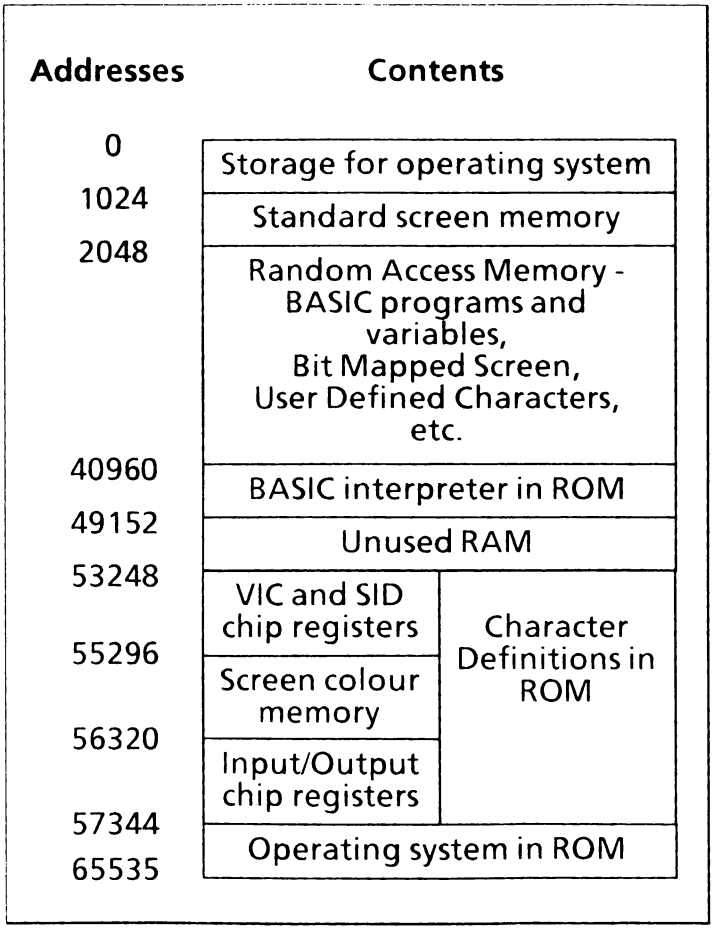
Characters 192 to 223 are the same as 96 to 127.

Characters 224 to 254 are the same as 160 to 190.

Character 255 is the same as 126.

APPENDIX 11

MEMORY MAP



INDEX

Abbreviations 209
ABS 74
AND 83
Animation 174
Arrays 27
Arithmetic 11
Assembly Language 204
ASC 62
ASCII code
Attack 95
ATN 75

BASIC 1
  Commands 210
Binary numbers 230
Bit 127, 230
Bit mapped mode 146
Boolean algebra 82
Branching 37
Byte 89

Cassette unit 4
Channels 188
Character sets 128
CHR$ 62
Clock 38
CLOSE 188
CLR 92
CLR/HOME key 10
Collision detection 167
CMD 190
Colour 12
Colour memory 120
CONT 49
CTRL key 12
COS 75
Cursor 8
Cursor keys 9

DATA 54
Decay 95
DEF FN 79
DEL key 10
DIM 29

Editing 10
END 48
Envelope 95
Error messages 221
EXP 77

Files 187
Filter 103
FN 79
FOR 34
FRE 91
Functions 73

GET 52
GET# 188
GOSUB 46
GOTO 37
Graphics 119
  Symbols 15
  Bit mapped 146

Hexadecimal numbers 234
Hierarchy of operations 12

IF 43
Immediate mode 8
INPUT 29
INPUT# 188
INST key 10
INT 74
Integer variables 25

Jumps 37

Keyboard 8
Keywords, BASIC 210

Languages 1
LEFT$ 60
LEN 59
LET 214
LIST 19
LOAD 184
Loading programs 184
LOG 76
Loops 34

Lower case 14

Machine code 204
Memory maps 239, 246
Memory use 89, 246
Microprocessor 204
MID$ 61
Modulation 109
Monitor 7
Music 93

NEW 21
NEXT 34
NOT 84
Numbering systems 230

ON ... GOSUB 49
ON ... GOTO 49
OPEN 187
Operating System 204
OR 84

PEEK 90
Pixel 127, 158
POKE 13, 90
POS 57
PRINT 8, 55
PRINT# 188
Program 18

RAM 90
Random numbers 77
READ 53
Real variables 25
Register 91
Release 95
REM 32
Reserved words 24, 210
RESTORE 55
RESTORE key 11,14
RETURN 47
Reverse video 15
RIGHT$ 60
RND 77
ROM 90
RUN 19
RUN/STOP key 11

SAVE 185
Saving programs 185
Screen memory 120,133,239
SGN 75
SID registers 96, 235
SIN 75
SPC 56
Sprites 158
SQR 73
STATUS 190
STEP 35
STOP 49
Stopping programs
String variables 26
STR$ 63
Structured programming 207
Subroutines 46
Sustain 95
Syntax errors 221
SYS 204

TAB 57
TAN 75
Television 4
THEN 43
TIME 39
TIME$ 39
TO 34

Upper case 14
User defined functions 79
User defined characters 129
USR 205

VAL 63
Variables 22
VERIFY 185
VIC 130

WAIT 206

The Commodore 64 Handbook offers its readers the opportunity of harnessing the power of one of the newest and most exciting microcomputers with its advanced sound and graphics facilities. Peter Lupton and Frazer Robinson are both professional technical authors and their clear step-by-step introduction opens the Commodore 64 to the beginner by starting with first principles and developing to explain the more advanced features and capabilities of the micro. The wealth of hints and tips, exciting programs and applications makes the book essential reading for even the experienced programmer. There are lengthy sections on sound and graphic capabilities which are particularly impressive features of the machine.

The Commodore 64 Handbook is a must for anyone who wants to take their Commodore 64 to the limits – and beyond!

For those who want to explore their Commodore 64 in even more depth there is a companion volume, *The Advanced Commodore 64 Handbook*, by the same authors.

ISBN 0 7126 0419 7
CENTURY COMMUNICATIONS LTD

£5.95



This was brought to you

from the archives of

http://retro-commodore.eu