CENTURY
COMMUNICATIONS

COMMODORE 64 OMNIBUS

A Comprehensive Programming Handbook

PETER LUPTON & FRAZER ROBINSON

THE COMMODORE 64 OMNIBUS

PETER LUPTON & FRAZER ROBINSON

CENTURY COMMUNICATIONS
LONDON

First published in Great Britain in 1985
by Century Communications Ltd,
12 - 13 Greek Street,
London W1V 5LE

ISBN 0 7126 06769

Printed in Great Britain in 1984 by
Billing & Sons Limited, Worcester.

Typeset by Spokesman, Bracknell

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

CONTENTS

PART 1
INTRODUCING COMMODORE 64 PROGRAMMING

PART 2
ADVANCED COMMODORE 64 PROGRAMMING

APPENDICES

ACKNOWLEDGEMENTS

It is customary at this point to mention all those people without whose help this book would not have been possible. We would like to acknowledge the support and enthusiasm of Peter Rhodes without whom none of our books would have been possible. We haven't been able to say so until now - Thanks Peter!

INTRODUCTION

This book is an omnibus edition of our two books,

The Commodore 64 Handbook
The Advanced Commodore 64 Handbook

We have taken the opportunity to correct one or two errors which crept into the first editions, and to clarify some obscurities noticed by our readers. The combined volumes form a complete guide to the Commodore 64.

The first section of this book provides a complete introduction to BASIC and the features of the 64, while the second introduces machine code (providing a set of routines to speed up graphics) and describes the use of disk drives and printers.

With this book at your side you will be fully armed for the great adventure of 64 programming!

Peter Lupton and Frazer Robinson 1984

THE COMMODORE 64 OMNIBUS

PART 1

INTRODUCING COMMODORE 64 PROGRAMMING

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.)

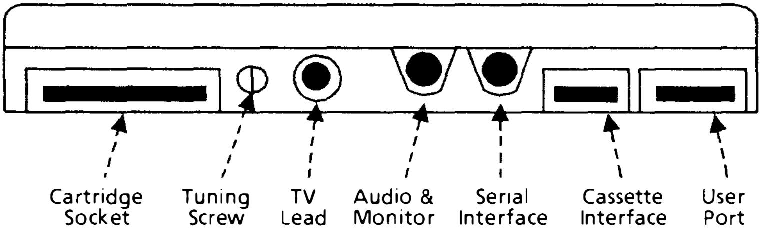
A programming language is much simpler than a human language because computers, for all their power, are not as good at understanding languages as people are. The BASIC language used by the 64 has less than 80 words. The rules for combining the words – the 'grammar' of the language – are much more strict 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!

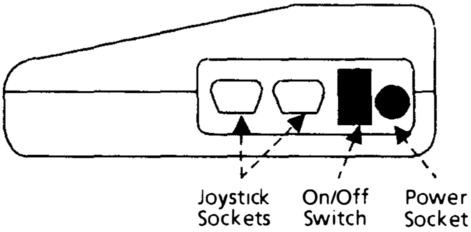
CHAPTER 2

SETTING UP YOUR 64

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



The Commodore 64 Connectors - Rear View



The Commodore 64 Connectors - Side View

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.

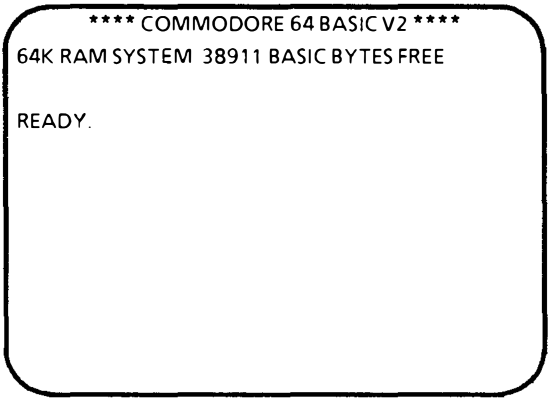
Second, switch on the disk drive and then the printer. (Remember that you should never switch a disk unit on or off with disks inside it.)

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:



```
     **** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM  38911 BASIC BYTES FREE

READY.
```

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:

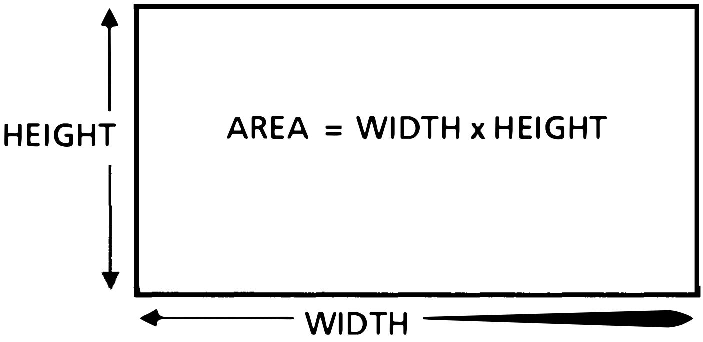
```
PRINT 3+5
```

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	BRIAN
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) = "BRIAN"
```

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

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

Array P$

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 the example overleaf:

```
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.

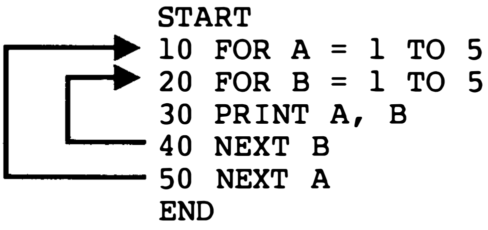
The variables used in **FOR ... NEXT** loops may be real or integer variables, but **not** string variables, and **not** elements of arrays.

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.

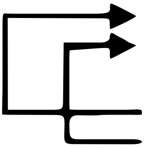


```
START
10 FOR A = 1 TO 5
20 FOR B = 1 TO 5
30 PRINT A, B
40 NEXT B
50 NEXT A
END
```

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 first item in the **PRINT** statement appears in column 0. The next item – the first after a comma – appears in column 10. The other items appear in columns 20 and 30.

The semicolon (;) can be used in a similar fashion. This leaves no space between successive items in the **PRINT** statement.

```
PRINT "A"; "B"; "C"; "D"; "E"; "F"
```

will display:

```
ABCDEF
```

The same rules apply when printing numbers, and real and integer variables, except that a space is printed on either side of each number. The leading space takes the minus sign when negative numbers are displayed.

The effect of commas and semicolons is not confined to the **PRINT** statement in which they appear. If you end one **PRINT** statement with a comma or semicolon then the data printed by the next **PRINT** command will appear on the same line, with spaces as appropriate.

You can exercise further control over the formatting of screen displays using two more BASIC commands – **SPC** and **TAB**.

SPC(N)

SPC(N) tells the 64 to **PRINT** N spaces.

```
PRINT "{CLS}I AM" SPC(10) "CBM 64"
```

will print 10 spaces between 'I AM' and 'CBM 64'.

TAB(N)

The **TAB** command instructs the 64 to move the cursor N spaces from the left side of the screen before printing. For example,

```
PRINT TAB(17) "CBM 64"
```

displays 'CBM 64' in the middle of a screen line.

The difference between **SPC** and **TAB** is straightforward. **TAB** moves the cursor to a column position on what is, in effect, its current line – so has a maximum value of 39. **SPC**, on the other hand, moves the cursor onwards by up to 255 spaces, so can embrace several lines if necessary. What neither do is to overprint blank spaces – so that they will not delete data which is left on the screen when new data is **PRINT**ed.

There is a function **POS** corresponding to the **TAB** command which returns the number of the column in which the cursor is placed. For example;

```
PRINT SPC(10);: PRINT POS (0)
```

gives the value 10, since the cursor was in column 10 after the first **PRINT** command. (The number in brackets has no significance – any number or letter can be used.)

CURSOR CONTROL

Just as we can use the cursor keys to move the cursor round the screen, so can we use a program to move it. The 64 interprets four characters as control codes governing the cursor movement.

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 16 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$(Z):NEXT Z
80    FOR Z=0 TO 9
90    READ D(Z),A(Z):NEXTZ
93    REM
94    REM ****************
95    REM *              *
96    REM * DRAW GALLOWS *
97    REM *              *
98    REM ****************
99    REM
100   G1$=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 D(Z) and the 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 16.

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)

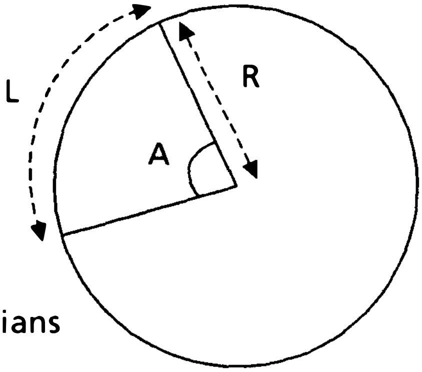
COS(N)

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 (2πR), so it follows that 360 degrees is equivalent to 2π radians.

From this we can work out that 1 degree is 2π/360 radians (about 0.175), and 1 radian is 360/2π 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 **PRINTing 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 of random numbers 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 10 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 10.) 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

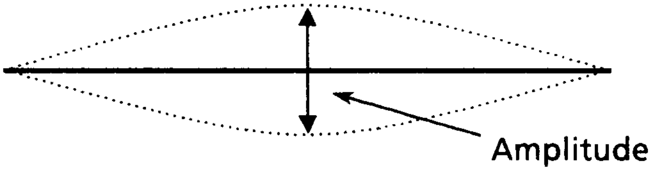
SOUND AND MUSIC - PART 1

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.

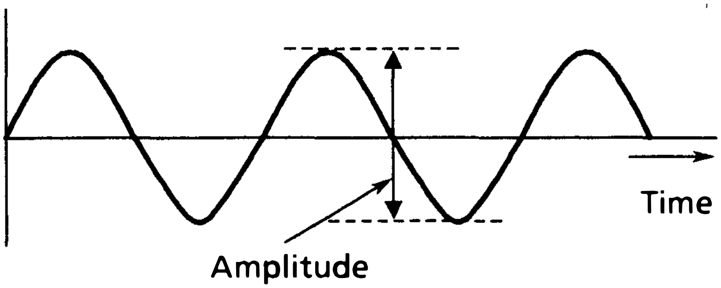
WHAT IS SOUND?

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.



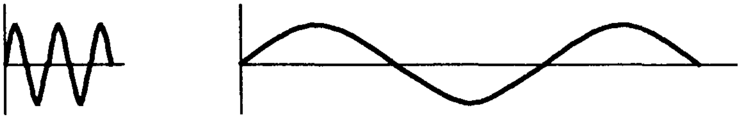
Vibrating Guitar String

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

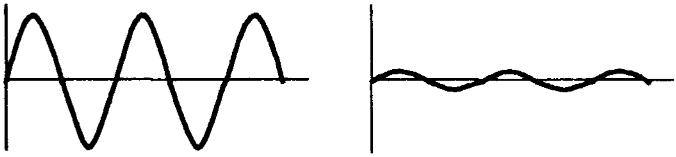


Simple Wave

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.

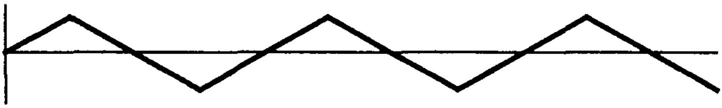


High and Low Frequencies

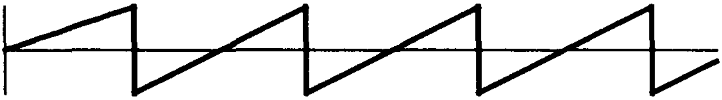


Large and Small Amplitudes (Loud and Soft)

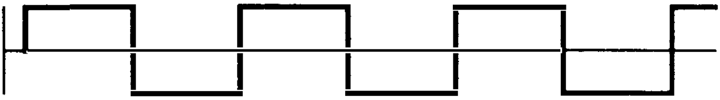
Not all sounds have the same shape of wave, and this wave-shape, or waveform, has a great effect on the quality of the sound. The Commodore 64 can generate four different waveforms, which are shown in the diagrams:



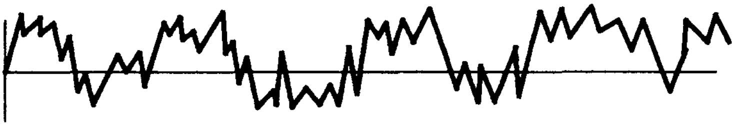
Triangle Waveform



Sawtooth Waveform

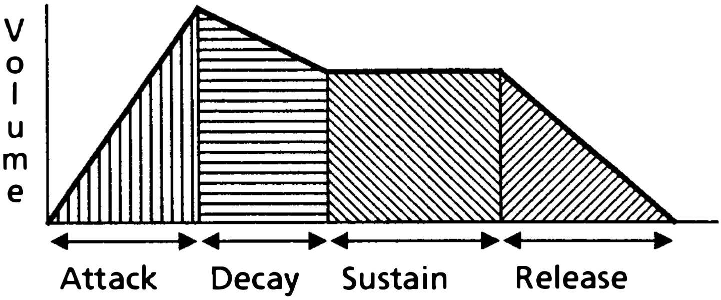


Pulse Waveform



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

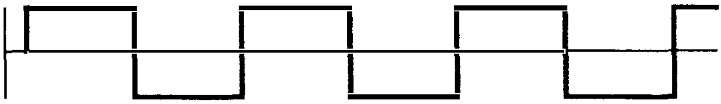
The program uses the Sawtooth waveform which gives a slightly buzzy tone. To hear the Triangle waveform, modify the program again by typing:

```
410   POKE 54276, 17
```

To hear the Noise waveform, alter line 410 to:

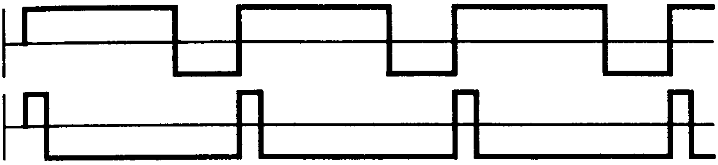
```
410   POKE 54276, 129
```

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



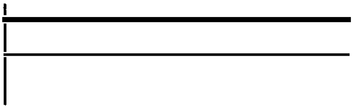
It is possible to alter the proportions of the wave by altering the width of the peaks, to produce a variety of alternative waveforms.

This is done by specifying the width of the peak as a proportion of the whole cycle. The proportion is set

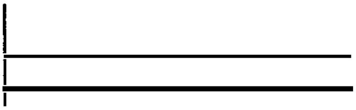


Alternative Pulse Waveforms

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:

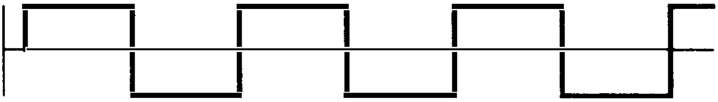


Pulse width 4095



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.

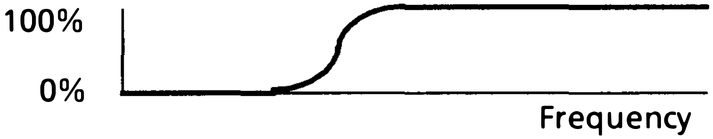
RUN the program and experiment to find the effects of different envelope parameters. Each of the four variables can have a value from 0 to 15. The value used for Sustain controls the volume of the note after the Attack and Decay phases. The durations of Attack, Decay and Release for the range of values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

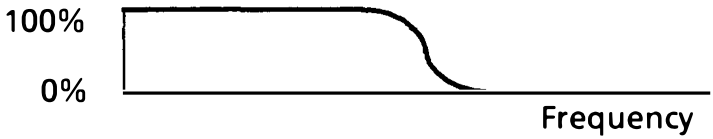
FILTERING

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,



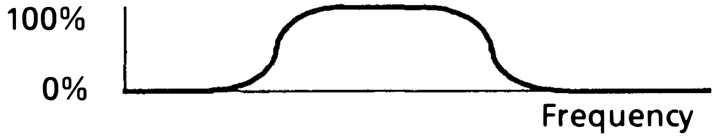
High-Pass Filter

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

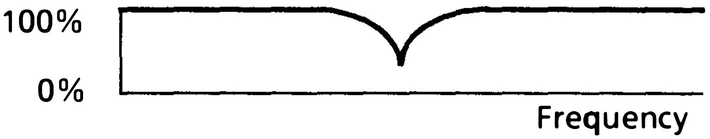
Band–pass filter which allows only the frequencies



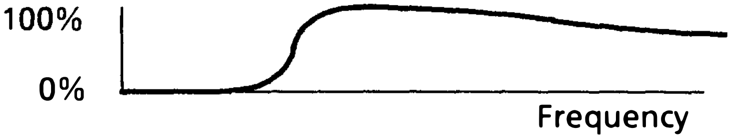
Band-Pass Filter

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 controlling the volume, but as the volume can only vary between 0 and 15, only the lower four bits of the register are needed to hold the volume setting. The full details of the use of location 54296 are shown in the table.



High-Pass and Low-Pass Filters Combined



High-Pass and Band-Pass Filters Combined

BIT	MEANING	VALUE
0 - 3	Volume	0 - 15
4	Low Pass Filter	16
5	Band Pass Filter	32
6	High Pass Filter	64
7	Turn off Voice 3	128

Setting Filters – Location 54296

So, to select the High pass filter and a volume of 7, the register must be POKEd with a value of 64 + 7, which is 71.

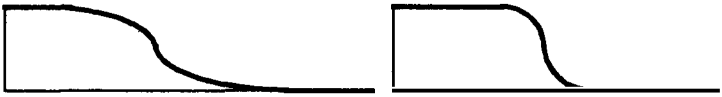
The cut-off frequency of the filter is controlled by the registers at locations 54293 and 54294. There are 11 bits altogether for the frequency setting; the lowest 3 bits of 54293 hold the 3 least significant bits of the frequency, while the eight most significant bits are held in 54294.

Location 54295 is used for two purposes: to select which of the three voices are to be filtered, and to set a resonance value.

BIT	MEANING	VALUE
0	Filter Voice 1	1
1	Filter Voice 2	2
2	Filter Voice 3	4
3	Filter External Signal	8
4 - 7	Resonance	16 - 240

Selecting Filters – Location 54295

The resonance value controls the sharpness of the cut–off. The higher the number, the sharper is the filter cut–off of the unwanted frequencies.



Low and High Resonance FIlters

THE CONTROL REGISTER

The control register (register 4 for each voice, location 54276 for Voice 1) does more than control the waveform used by the voice. Each of the eight bits of the register has a different meaning.

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

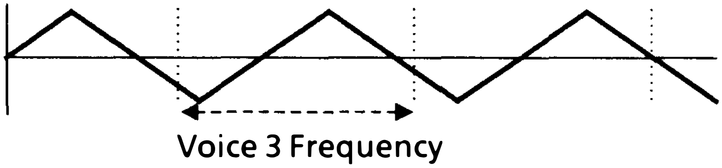
The SID chip can not only be used for playing music, but for producing all kinds of sound effects and strange noises. By varying the frequency while a note is playing, siren effects can be achieved. It is possible to alter the waveform while a note is playing by setting the appropriate bits of the control 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

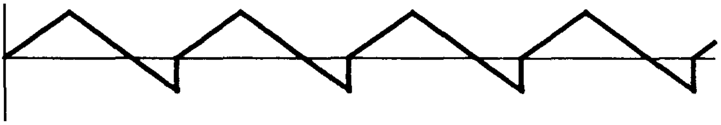
The SYNC bit of the control register of each voice synchronises the output of the voice with the frequency of one of the other voices. The voices are paired thus:

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:

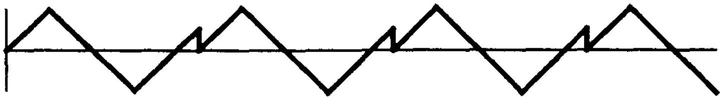


and Voice 3 has the frequency indicated, then if Voice 1 is synchronised with Voice 3 the output will look like:



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 POKEd 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 13 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"
```

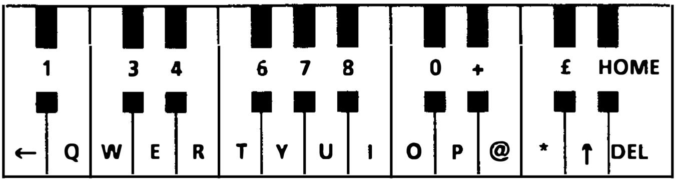
```
5030  PRINT: PRINT "  {RVS}F3{ROF}
      VOICE 2"
5040  PRINT: PRINT "  {RVS}F5{ROF}
      VOICE 3"
5050  POKE SID+24,VOL
5060  PRINT "{HOM}{CD}{CD}{CD}{CD}
      VOLUME"VOL AND 15"{CL} "
5070  GET K$: IF K$="" THEN 5070
5080  HI=N(ASC(K$),1)
5090  IF HI=0 THEN 5150
5100  POKE SID+V+4, W(VV)
5110  POKE SID+V+1, HI
5120  POKE SID+V, N(ASC(K$),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.

You will find more about making music with the the SID chip in Chapter 23.

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 ₒ is used to represent a space:

```
10      PRINT"{O}[YYYYY]{P}
20      PRINT"[H]{W}ₒₒₒ{W}[N]
30      PRINT"[H]ₒₒ{Q}ₒₒ[N]
40      PRINT"[H]ₒₒₒₒₒ[N]
50      PRINT"{M}ₒ{J*K}ₒ[N]
60      PRINT"ₒ{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 17)

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 16. 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 8.

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=l TO 4
2110  IF MID$(G$,I,l) <>
      MID$(CODE$,I,l) THEN 2150
2120  Cl = Cl+l
2130  G$ = LEFT$(G$,I-l) + "9" +
      MID$(G$,I+l)
2140  CO$ = LEFT$(CO$,I-l) + "9" +
      MID$(CO$,I+l)
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=l TO 4
2210  FOR J=l TO 4
2220  IF MID$(G$,I,l) =MID$(CO$,J,l)
      AND MID$(G$,I,l)<>"9" THEN
      C2=C2+l: GOTO 2240
2230  GOTO 2260
2240  G$ = LEFT$(G$,I-l) + "9" +
      MID$(G$,I+l)
2250  CO$ = LEFT$(CO$,J-l) + "9" +
      MID$(CO$,J+l)
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.

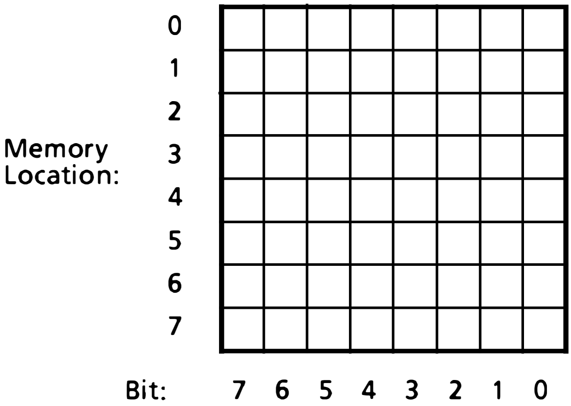
Lines 2200 to 2250 check for an item of the right colour in the wrong place. Every such item in the guess is set to '9' once it has been recorded to avoid considering it more than once.

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 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.



The Character Grid

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 16) in the first character set is given by:

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:

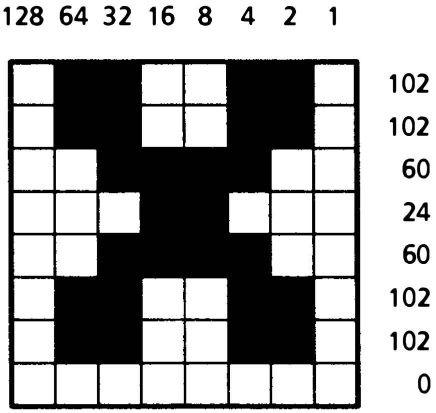
```
POKE 1, PEEK(1) OR 4
POKE 56334, PEEK(56334) OR 1
```

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



The Character 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 places, while the processor uses these memory

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

VIC Bank Selection

areas for something else! (There is a diagram of the 64's memory – a *memory map* – in Appendix 17.)

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)
```

	START OF CHARACTER MEMORY			
C	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 17.)

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:

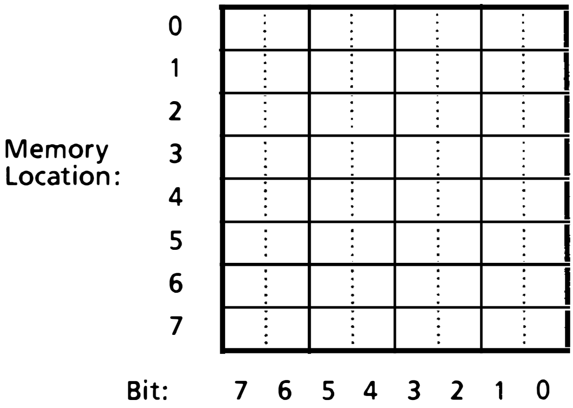
```
   POKE 53270, PEEK(53270) OR 16
```

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 17) 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 and 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↑1 + 2↑2 + 2↑4 = 22

To select this colour combination, add the line:

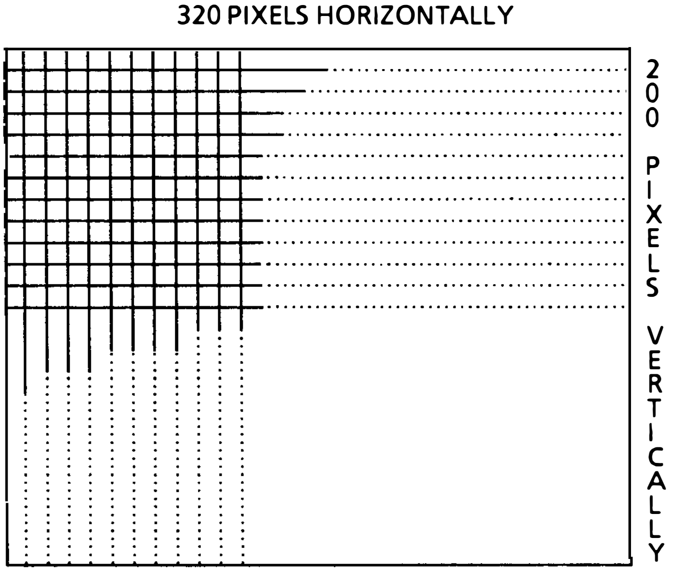
```
40    FOR Z=1024 TO 2023:POKE
      Z,22:NEXT
```

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

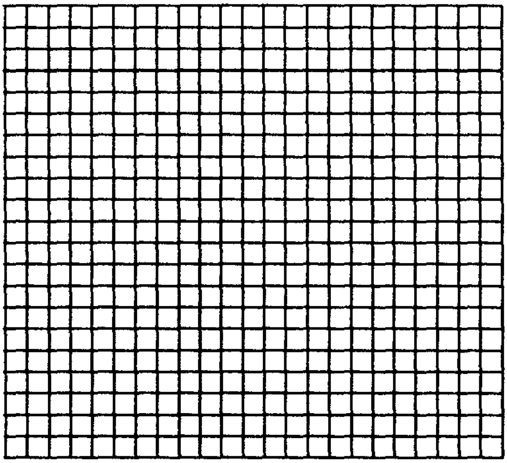
The technique is much the same as for multi colour characters, and for multi colour sprites, as we shall see in the next chapter.

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.

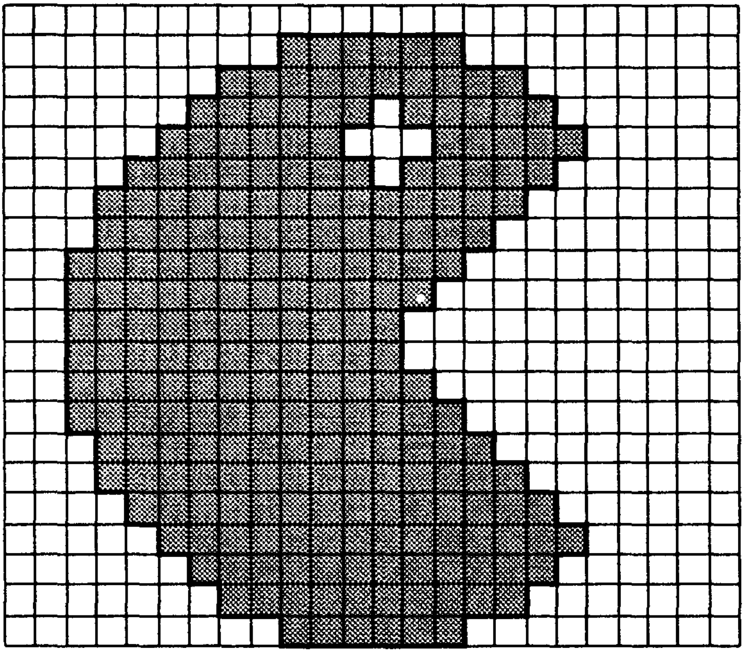
One of the reasons sprites are so useful is that you can move them around the screen very easily, simply by giving them an X and Y coordinate based on the 320 by 200 pixel bit mapped display and the VIC II chip does all the rest. This makes moving graphics in BASIC very easy to create.



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 POKEing 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:

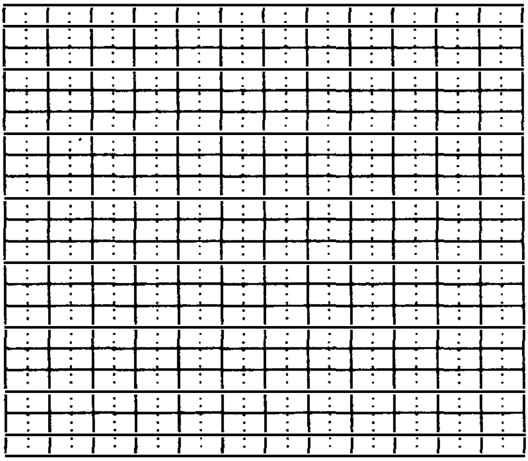
```
   POKE 53276,PEEK(53276) OR 2↑N
```

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:

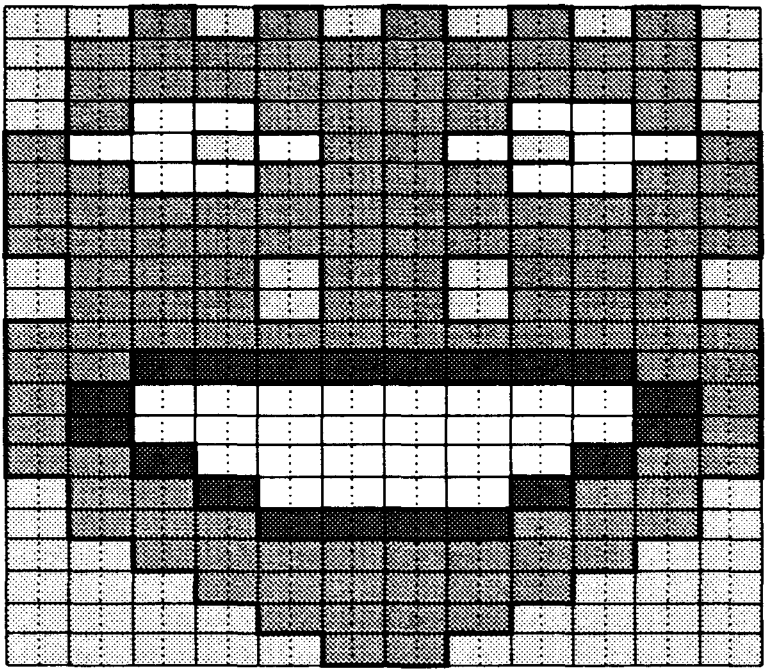
BACKGROUND	SCREEN COLOUR	00
FACE	SPRITE COLOUR	10
EYES	REGISTER 0	01
LIPS	REGISTER 1	11

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 LOADed 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 SAVEd. The name you use for the SAVEd 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 disk matched the one in the 64, or ?VERIFY ERROR if the two were different.

Full details of all aspects of disk use are given in Part 2, Chapter 24.

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 semicolons 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.

This chapter introduces the BASIC commands used to call machine code programs. In Part 2, Chapter 18 describes the use of machine code with the 64, and Chapters 19 to 21 give some powerful 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.

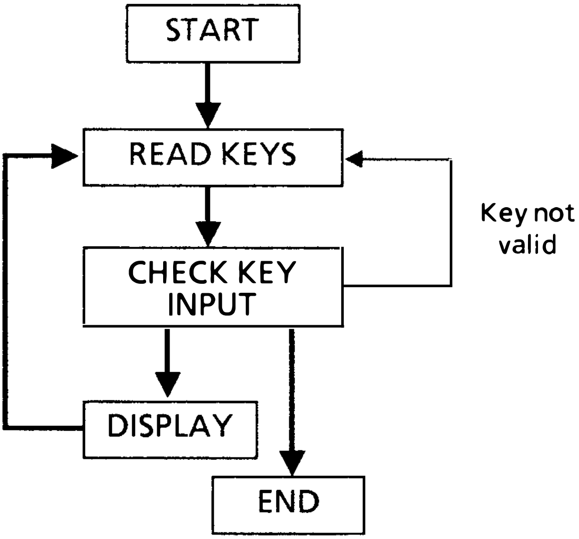
DESIGNING GOOD PROGRAMS

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.

THE COMMODORE 64 OMNIBUS

PART 2

ADVANCED COMMODORE 64 PROGRAMMING

CHAPTER 17

BASIC AND HOW IT WORKS

You are probably aware that the microprocessor in your 64 doesn't understand BASIC and needs to be programmed in its own language – machine code. To enable the 64 to run your BASIC programs it comes equipped with a large machine code program called the *BASIC interpreter*. As the name suggests this program interprets a BASIC program – converting it into a series of machine actions, the instructions for which are subroutines of the interpreter program.

Machine code is covered in more detail later in the next chapter; this chapter deals with how the 64 stores and runs BASIC programs and shows how you can get more out of your 64 by understanding the techniques involved.

Interpreter and Kernal ROMs

The ROM memory in the 64 contains two machine code programs – the basic interpreter mentioned above and the *kernal*. The kernal is responsible for all the functions needed to operate the 64 – reading the keyboard, arranging the memory, operating the screen editor and so forth. Both the interpreter and the kernal reserve some of the RAM for their own use, generally speaking in the first 2k of memory.

HOW BASIC IS STORED

When you type in a line of a BASIC program, routines in the kernal take the character for each key press and store it in screen memory. When you press RETURN the entire line is copied from the screen into memory. The BASIC interpreter reads the first few characters of the line to see if they form a line number and, if so, the line is inserted into the stored program. This system allows you to edit a line using the cursor keys without retyping the entire line.

BASIC programs are normally stored starting from location 2048. To see this type in the following one line program and then enter the immediate mode commands following it.

```
10 PRINT "TEST": GOTO 10

FOR Z=2048 TO 2068:PRINT Z,PEEK(Z):
NEXT Z
```

The display will look like the two left hand columns in the list below if you have typed the program in exactly as it appears:

ADDRESS	CONTENTS	COMMENT
2048	0	The first byte of a program is always zero
2049	19	Low byte of link pointer
2050	8	High byte of link pointer
2051	10	Low byte of line number
2052	0	High byte of line number

2053	153	PRINT
2054	32	(space)
2055	34	"
2056	84	T
2057	69	E
2058	83	S
2059	84	T
2060	34	"
2061	58	:
2062	137	GOTO
2063	32	(space)
2064	49	1
2065	48	0
2066	0	Null
2067	0	Null
2068	0	Null

The comment to the right of each item in the list show what each location contributes to the program line.

The contents of location 2048 are always zero if a BASIC program is stored in the 64.

Locations 2049 to 2065 hold the BASIC line and have the following functions:

The first two bytes, 2049 and 2050, indicate the position of the next line of BASIC – more about this later.

Bytes 3 and 4 of the line (2051 and 2052) are the line number in the order low byte, high byte. In this case the number is 10 – that is 10 + 256*0.

The next byte contains the value 153 which is a shorthand for **PRINT**. This is because the 64 stores BASIC programs in a compressed form using a system of *tokens* whereby single byte codes are used to represent BASIC reserved words. This method offers significant memory savings over storing commands as a series of ASCII characters and also increases the speed at which programs run. When you pressed RETURN and entered the program line the kernal program recognised the **PRINT** as BASIC reserved word by comparing it with a list kept in ROM starting at location 41118. The token 153, used to replace the **PRINT** command, represents the position of that command in the list with 128 added to it, so **PRINT** is the 25th item in the list. The following program displays a section of the BASIC reserved word table from the ROM.

```
10    REM DISPLAY RESERVED WORDS
15    PRINT "{CLS}"CHR$(14)
20    FOR Z=41118 TO 41250
30    A$ = CHR$(PEEK(Z))
40    PRINT A$;
50    IF ASC(A$)<91 THEN 70
60    N=N+1 : PRINT CHR$(13);N,
70    NEXT Z
```

You will see that **PRINT** is the 25th item in the list as mentioned above.

When a program is listed the tokens are converted back into the BASIC words they represent, making the system transparent to the user.

Another advantage of the token system is that it allows you to type BASIC commands in an abbreviated form, since only the first two or three characters are checked by the interpreter. A full list of BASIC reserved words, abbreviations and tokens is given in Appendix 3.

The next eight bytes in the list are simply the ASCII representation of the characters you typed as part of the program line.

Location 2062 is another token, this time 137 representing the **GOTO** command, and is followed by a space. The next two bytes are the line number following the **GOTO** command, 10, but stored in their ASCII form, *not* as a binary number. The end of a BASIC program is indicated by three consecutive zeros.

Let's return to the two bytes labelled 'link pointer' at the start of the list. These two bytes form the low and high bytes respectively of the address where the next line of the program is stored. In this case they point to address 2067 (19 + 8*256 = 2067), which is the end of the program.

To see how the link pointer works, try adding the following line to the program and examining memory locations 2066 to 2074.

```
20    REM
```

```
FOR Z=2066 TO 2074:PRINT Z, PEEK(Z):
NEXT Z
```

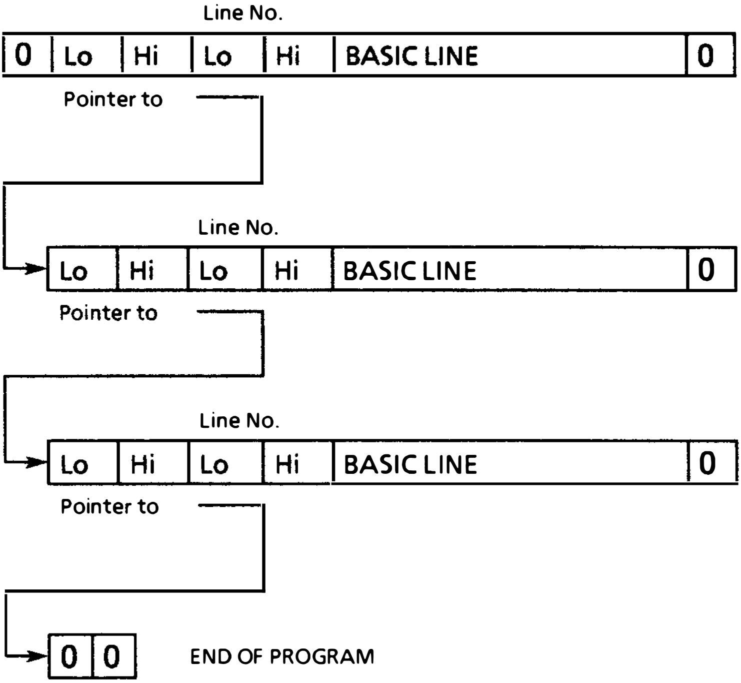
You will get a list like this:

2066	0	A null for the end of line 10
2067	25	Low byte of link pointer
2068	8	High byte of link pointer
2069	20	Low byte of line number
2070	0	High byte of line number
2071	143	REM token
2072	0	Null
2073	0	Null
2074	0	Null

You should be able to see that there is now only one zero after line 10, and that location 2067 contains the low byte of the link pointer to the new end of the program. Remember that the link pointer for line 10 pointed to location 2067 ?

As before the third and fourth bytes contain the low byte and the high byte of the line number, and 143 is the token for **REM**. The program again terminates in three consecutive zeros.

The diagram opposite summarises how BASIC programs are stored.



The linked nature of a BASIC program

Renumbering Programs

We can make use of this knowledge to renumber BASIC programs. When a program is being developed it is often useful to be able to 'open up' gaps between lines to fit in new lines, and the next program allows you to do just that.

```
60000 REM RENUMBER
60010 INPUT"START";S
60020 INPUT"END";E
60030 INPUT"NEW START";NS
60040 INPUT"INCREMENT";I
60050 A=2049
60060 Q=PEEK(A+2)+256*PEEK(A+3)
60070 IF Q<S THEN 61000
```

```
60080 IF Q>E THEN PRINT"FINISHED":
      END
60090 POKE A+2,NS-INT(NS/256)*256
60095 POKE A+3,INT(NS/256):NS=NS+I
61000 A=PEEK(A)+256*PEEK(A+1)
61010 IF A=0 THEN END
61020 GOTO 60060
```

The program is numbered so it can form the last part of any program you are developing. Whenever you need to renumber simply type:

```
RUN 60000
```

and enter the line numbers of the beginning and end of the program to be renumbered and the new start number and increment.

This is a very crude and simple renumbering program and doesn't take account of the line numbers following **GOTO**, **GOSUB** and **THEN** commands. To write a program to alter these is more difficult because these line numbers are stored in their ASCII form and so occupy a varying number of bytes. This means that if a line number specified by a **GOTO** command is three figures long and is required to become a four figure number when the program is renumbered, the entire BASIC program would need to be moved up in memory by one byte to create space. Such a program would not be impossible to write in BASIC but it would be quite slow!

A possible way around this would be always to start your program numbering at line 10000 - thereby ensuring that every **GOSUB**, **GOTO** or **THEN** reference was a five figure number. If you want to try this, refer to the table of BASIC commands and their tokens in Appendix 3.

EXECUTING A BASIC PROGRAM

A BASIC program stored in memory is of little use until it can be executed. This is achieved by typing RUN and pressing RETURN. The RUN is interpreted as an immediate command and the appropriate part of the BASIC interpreter program is called. This subroutine initialises all pointers in zero page and closes all open channels. The first line of the program is then interpreted by loading it a character at a time into the accumulator and passing control to the appropriate subroutines as a command is found. This process is carried out by a small machine code subroutine called CHRGET which is copied from ROM into zero page RAM (locations 115 to 138) when the 64 is switched on.

Below is a disassembly listing of CHRGET:

```
START    115    E6 7A          INC   $7A
         117    D0 02          BNE   $02
         119    E6 7B          INC   $7B
FETCH    121    AD B7 12       LDA   $12B7
         124    C9 3A          CMP   #$3A
         126    B0 0A          BCS   RETURN
         128    C9 20          CMP   #$20
         130    F0 EF          BEQ   START
         132    38             SEC
         133    E9 30          SBC   #$30
         135    38             SEC
         136    E9 D0          SBC   #$D0
RETURN   138    60             RTS
```

The first operation of CHRGET is to increment the low byte of the character pointer – two locations pointing to the next character of the BASIC text to be accessed. If incrementing the low byte of the character pointer causes an overflow, then the high byte is incremented. The character pointer is at locations $7A and $7B (122 and 123) which form part of the CHRGET program, so the program is self modifying! The section of the program labelled FETCH reads the character pointed to by the character pointer into the accumulator. This location is variable and the value in the listing above is what happened to be present in our 64 when the listing was created.

If the character is a colon (:), signifying another statement on this line, then control jumps back to other routines in the interpreter which execute the command just fetched before returning for the next one.

If a space is encountered then the next character is read in.

This routine is very important since it gives you a chance to intercept the processing of BASIC programs and implement additional features, by replacing the code at the beginning of the routine by two JSR commands calling two of your own routines. The second routine simply copies that part of CHRGET overwritten by the JSR instructions while the first is the new user defined code.

This technique can be used to add new commands to the BASIC language as the following simple example shows:

ADDING NEW COMMANDS TO BASIC

To add a new command to BASIC there are two steps to follow:

a) Modify the first three bytes of CHRGET to perform a JSR to the routine which interprets and carries out the new command.

b) Modify the next three bytes of CHRGET to perform a JSR to a subroutine which is a copy of the first six bytes of the original CHRGET routine.

For example, suppose you wanted to add a command which caused a short tone to be emitted by the TV speaker, as a warning in case of errors.

Here is a short machine code routine which performs the function which could be called BEEP.

```
START     LDA #37       ;set up SID
          STA 54271     ;registers
          LDA #100
          STA 54273
          LDA #15
          STA 54296
          LDA #54
          STA 54277
          LDA #168
          STA 54278
          LDA #33
          STA 54276
          LDA #122      ;store into
          STA 162       ;Jiffy clock
DELAY     LDA 162       ;loop until
          BPL DELAY     ;zero
QUIET     LDA #32       ;turn off the
          STA 54276     ;noise
          LDA #0
          STA 54296
          RTS
```

Even if you are not familiar with machine code you might recognise that the program simply sets up the necessary SID registers on channel one to make a tone, uses the jiffy clock to create a short pause and then turns the sound off.

To incorporate the new command into BASIC, this machine code must be preceded by some code to recognise the new command. In this example we will use the # symbol as the new command, to avoid confusing the issue with subroutines to decode the characters in a command.

Here is the program which modifies the CHRGET routine and incorporates a 'BEEP' command into BASIC:

```
1 *=$C000
10 CHRGET=115
90 !
95 !
100             LDY #0          ;Modify the
110 INIT        LDA TABLE,Y     ;first six
120             STA CHRGET,Y    ;bytes of
130             INY             ;CHRGET
140             CPY #6          ;routine
150             BNE INIT
160             RTS
190 !
191 !
200 PROG        JSR READ        ;get char
205             CMP #35         ;is it a #?
210             BNE RETURN      ;no
220             JSR BEEP        ;yes! BEEP
225             JSR COPY        ;update $7A
230 RETURN      RTS             ;and $7B
290 !
291 !
300 READ        LDA $7A         ;copy $7A
301             STA R2+1        ;and $7B
302             LDA $7B         ;into new
303             STA R2+2        ;read
309             INC R2+1        ;routine &
310             BNE R2          ;get next
320             INC R2+2        ;character
330 R2          LDA $0800
340             RTS
390 !
391 !
500 COPY        INC $7A         ;copy of
```

```
510             BNE EXIT        ;start of
520             INC $7B         ;CHRGET
530 EXIT        RTS             ;routine
590 !
591 !
800 BEEP        LDA #37
805             STA 54271
810             LDA #100
815             STA 54273
820             LDA #15
825             STA 54296
830             LDA #54
835             STA 54277
840             LDA #168
845             STA 54278
850             LDA #33
855             STA 54276
860             LDA #122
865             STA 162
870 DELAY       LDA 162
875             BPL DELAY
880 QUIET       LDA #32
885             STA 54276
890             LDA #0
895             STA 54296
900             RTS
990 !
991 !
1000 TABLE      JSR PROG        ;new start
1010            JSR COPY        ;of CHRGET
```

Below is a BASIC loader for the machine code:

```
5     REM BASIC LOADER FOR BEEP
      COMMAND
10    FOR Z=49152 TO 49264
20    READ X:POKE Z,X:NEXT Z
20000 DATA 160,0,185,106,192,153,
      115,0,200,192,6,208,245,96,32,
      28,192
```

```
20010 DATA 201,35,208,6,32,57,192,
      32,50,192,96,165,122,141,47,19
      2,165
20020 DATA 123,141,48,192,238,47,
      192,208,3,238,48,192,173,97,8,
      96,230
20030 DATA 122,208,2,230,123,96,169,
      37,141,255,211,169,100,141,1,2
      12,169
20040 DATA 15,141,24,212,169,54,141,
      5,212,169,168,141,6,212,169,33
      ,141
20050 DATA 4,212,169,122,133,162,
      165,162,16,252,169,32,141,4,21
      2,169,0
20060 DATA 141,24,212,96,32,14,192,
      32,50,192,64
```

NOTE The code occupies some of the memory used by the graphics routines, which will be overwritten by this program. To start the program type:

```
SYS 49152
```

From now on every time a # is detected in the program, the BEEP routine will be called. The # can be treated like any other BASIC command with one exception – it must not be the first command on a line.

How the Program Works

Upon initialisation the program modifies the first six bytes of the CHRGET routine to read:

```
JSR $C00E

JSR $C032
```

The routine keeps its own pointer to the current byte of the BASIC program in the READ subroutine, and this is updated to keep up with that used by CHRGET. If a # symbol is detected, the BEEP subroutine is called, the CHRGET pointers are incremented to avoid reading the # twice and control is returned to the CHRGET routine. If the character isn't a # then control is passed back to CHRGET.

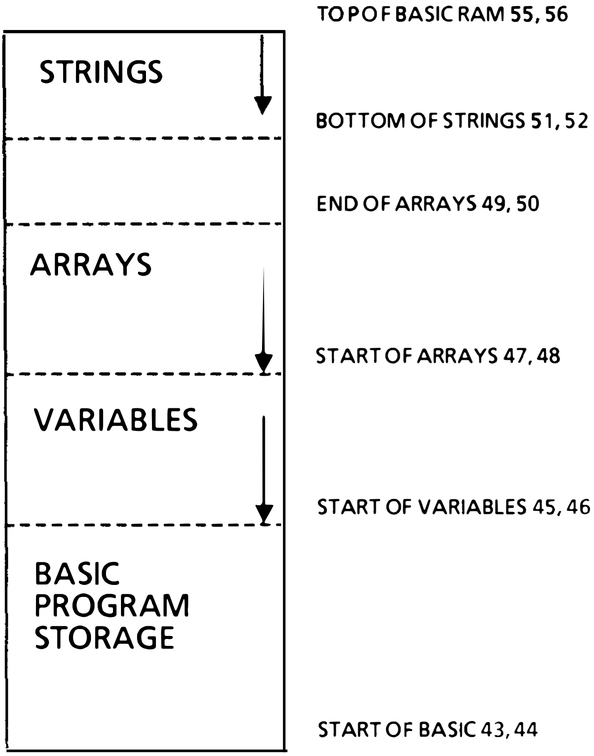
To add a number of commands to BASIC would involve storing them in a table, and preceding each with a #, then adding code between lines 210 and 220 to compare the characters after the # with the commands in the table. If a match was found then the appropriate subroutine would be called.

VARIABLE STORAGE IN BASIC

One of the things the BASIC interpreter must take care of is the handling of variables. It must decide where to store them and have an indication as to what type of variable they are. This section illustrates the various types of variable, the form in which they are stored in the 64 and their location in RAM. The diagram overleaf illustrates the arrangement of memory when a BASIC program is stored and the zero page locations which contain the start address of the various areas of memory.

SIMPLE VARIABLES

Simple variables are stored in memory starting immediately after the BASIC program, at the location pointed to by locations 45 and 46. Each variable occupies seven bytes of memory but these bytes are used differently by the different types of variable.



Memory Allocation for a BASIC program

Integer Variables

The diagram shows how the seven bytes are used by an integer variable.

VARIABLE NAME		VARIABLE VALUE		NULL	NULL	NULL
1st + 128	2nd + 128	High Byte	Low Byte	0	0	0

Format of an integer variable

Note that three bytes are unused, and so the use of integer variables doesn't result in a memory saving over floating point variables.

Floating Point Variables

These are stored as an exponent and a mantissa, so the number is stored in the form mantissa * 2↑ exponent.

The mantissa of a floating point variable is stored in packed Binary Coded Decimal form which gives 8 bit precision with the four bytes allocated. The first bit of the first byte of the mantissa is a sign bit.

VARIABLE NAME		VARIABLE VALUE				
1st	2nd	Expone- -nt	Mantiss- -a	Mantiss- -a	Mantiss- -a	Mantiss- -a

Format of a floating point variable

String Variables

A string variable is stored in two parts: as an entry in the variable table as illustrated in the diagram below, and as the characters comprising the string, which are stored in a separate area of memory, pointed to by the entry in the variable table.

VARIABLE NAME		POINTER TO CHARACTERS			NULL	NULL
1st	2nd + 128	No of Chars	Low Byte	High Byte	0	0

Format of a string variable

Because the number of characters in the string is contained in one byte in the variable table entry the maximum number of characters in a string is 255.

String variables may also be defined in a program by such statements as:

```
A$="HOW LONG IS A PIECE OF STRING"
```

In this case the entry for A$ in the variable table would point to a location within the BASIC program text.

Type in the following program exactly as it appears to see the various variable types and their storage mechanisms.

```
10    PRINT "START OF VARIABLES
      BEFORE=";:PRINT PEEK(45)+256*
      PEEK(46)
20    PRINT "START OF ARRAYS
      BEFORE=";:PRINT PEEK(47)+256*
      PEEK(48)
30    A% = 100
40    B = 1000
50    C$ = "CBM-64"
60    D$ = C$+C$
70    PRINT"START OF ARRAYS
      AFTER=";:PRINT PEEK(47)+256*
      PEEK(48)
```

If you run the program you will see that before the variables are created, the start of variables and the start of arrays are at the same location. However when the variables have been created by the program there is a difference of 28 bytes between the two – since each variable occupies 7 bytes and the program created 4 variables.

Type in the following command to examine the variable area:

```
FOR Z=2265 TO 2292 :PRINTZ,PEEK(Z):
NEXT Z
```

You should get a display like this:

```
2265      193      A (65 + 128) 1st char of name
2266      128      Blank 2nd character of name
2267      0        High byte of integer variable
2268      100      Low byte of integer variable
2269      0        Null
2270      0        Null
2271      0        Null

2272      66       B (66) 1st char of name
2273      0        Blank 2nd char of name
2274      138      128 + exponent
2275      122      1st byte of mantissa
2276      0        2nd byte of mantissa
2277      0        3rd byte of mantissa
2278      0        4th byte of mantissa

2279      67       C (67) first character of name
2280      128      Blank 2nd character of name
2281      6        Number of characters
2282      142      Low & high bytes of pointer
2283      8        to program area
2284      0        Null
2285      0        Null

2286      68       D(68) first character of name
2287      128      Blank 2nd character of name
2288      12       Number of characters
2289      244      Low & high bytes of pointer
2290      159      to string variable area
2291      0        Null
2292      0        Null
```

From the comments added to the list you should be able to see that the arrangement of data in the variable area corresponds to that decribed above.

One of the uses to which we can put this knowledge is in a short routine to dump the names of all the variables used in a program. Such a routine can be a valuable aid when developing programs, and could be made more useful by getting it to give the values assigned to each variable at the time of the dump. The following program will print a list of all variables used in a program.

```
1     REM VARIABLE DUMP
10000 Z1=PEEK(45)+256*PEEK(46)
10010 Z2=PEEK(47)+256*PEEK(48)-21
10020 FOR Z3=Z1 TO Z2 STEP 7
10030 Z4=PEEK(Z3):Z5=PEEK(Z3+1)
10040 IF Z4>127 AND Z5>127 THEN
      PRINT CHR$(Z4-128)CHR$(Z5-
      128);"%":GOTO 10070
10050 IF Z4<127 AND Z5<127 THEN
      PRINT CHR$(Z4)CHR$(Z5):GOTO
      10070
10060 PRINT CHR$(Z4)CHR$(Z5-128);"$"
10070 NEXT
```

To use the program simply append it to your own program and type in direct mode `GOTO 10000` when you need a variable dump.

For optimum efficiency it should be written in machine code, and could then be called with a SYS whenever required during the debugging stage of your BASIC program.

Array Variables

All three types of variable may be stored in array form, in which case different methods are used for allocating memory. Arrays are stored immediately above simple variables, starting from the address pointed to by locations 47 and 48. The array type (floating point, integer or string) is indicated by the way in which the array name is stored – in exactly the same way as with simple variables. The arrangement of an array in memory is illustrated below.

ARRAY HEADER	Element 0	Element 1	Element 2	Element 3	Element 4

Format of an array

NOTE: Elements are stored in reverse order in string arrays.

The Array Header

The header format is the same regardless of the array type, and occupies seven bytes plus an extra two bytes for each dimension beyond 1.

ARRAY NAME CHARACTERS		TOTAL BYTES				
1st	2nd	Low Byte	High Byte	No. of DIMensions	Size of Nth DIMension	Size of N-1th DIMension

Format of an array header

Bytes one and two contain the array name and bytes three and four store the amount of memory occupied by the array in low byte, high byte order. The fifth byte contains the number of dimensions in the array.

In a one dimensional array bytes six and seven contain the size of the array as specified in the **DIM** statement that created it. If no **DIM** statement exists then the default number is 10. For arrays with more than one dimension, the header is larger by two bytes per additional dimension and these two bytes contain the size of the extra dimension. The dimension bytes are stored in reverse order in the header to that in which they appear in the **DIM** statement that created them.

Array Elements

The way in which array elements are arranged in memory is illustrated in the following diagram:

FLOATING POINT	Expone- -nt	Mantiss- -a 1	Mantiss- -a 2	Mantiss- -a 3	Mantiss- -a 4
INTEGER	High Byte	Low Byte			
STRING	No. of Chars.	High Byte	Low Byte		
		Pointer			

Format of array elements

Floating point array elements occupy 5 bytes, string elements occupy 3 bytes and integer array elements need only two bytes

The next program creates an array and the immediate mode commands following it display the area of memory in which it is stored.

```
10    DIM AB(10,20)

FOR Z=2067 TO 2085:PRINT Z,PEEK(Z):
NEXT Z
```

You should get a list like this:

```
2067      90      These seven bytes
2068      0       are the loop counter, Z,
2069      140     used to display
2070      1       the list.
2071      112
2072      0
```

```
2073      0

2074      65      A  The array name
2075      66      B

2076      140     The number of bytes used, low
2077      4       and high. 4*256+140=1164

2078      2       The number of dimensions

2079      0       Size of second dimension, high
2080      21      and low bytes. 0*256+21=21

2081      0       Size of the first dimension, high
2082      11      and low bytes. 0*256+11=11

2083      0       This is the start of the area of
2084      0       memory in which the array
2085      0       elements are stored
```

CHAPTER 18

MACHINE CODE ON THE 64

Although the 64 runs programs in BASIC, this is not the 'natural' language of the microprocessor (a 6510) at the heart of the computer. When you switch on the 64 it automatically begins running a very sophisticated set of programs which are written in the machine language of the 6510. It is these programs which interpret the BASIC commands, provide the screen editing facility and handle the cassette, disk drive and printer.

The machine language of a microprocessor is a set of instructions which can be interpreted directly by the electronics within the microchip. Each type of processor has its own instruction set – the 6510 is a variant of the well known 6502 processor and shares the same instructions.

If the processor can interpret machine language programs directly, why go to the trouble of providing a second language, BASIC, for the 64? The reason is that machine language instructions are less powerful than BASIC instructions, and programs are therefore less easy to write, as it may take several machine code instructions to perform the equivalent of one BASIC command.

When computers were first developed all programs for them were written in machine code, but it was soon realised that this made programming very tedious. "High-level" languages such as BASIC were developed to make programming easier by allowing the programmer to concentrate on the problem to be solved without getting bogged down in the details of writing the code.

THE 6510 MICROPROCESSOR

The electronics of a microprocessor are extremely complicated, but from the machine code programmer's point of view the 6510 is a fairly simple device. The 6510 has three 8-bit data registers, the *accumulator*, and the *X* and *Y index registers*; and three control registers, the *processor status register*, the *stack pointer* and the *program counter* (which unlike the others is a 16-bit register). These registers are similar to storage locations in memory but are built into the microprocessor chip, and are used to hold the data being processed by the 6510.

Accumulator

This is the most used of all the registers. Almost all the data processed by the computer passes through this register. The accumulator is used in all calculations – addition, subtraction and logical operations.

X and Y Index Registers

The index registers are so called because they may be used in a variety of ways to index tables of data in the memory. They also play a useful role as loop counters and temporary storage registers.

Processor Status Register P

This is not so much a register as a collection of single bits which act as 'flags' to indicate various conditions of the processor. These flags are:

C The Carry flag. Used in addition and subtraction operations.

Z The Zero flag. Indicates that the result of the last operation was zero.

N Indicates a negative result.

V Indicates an overflow or underflow in signed arithmetic.

D Sets Decimal mode.

I Interrupt disable.

B Break instruction just performed.

These seven flags form the processor status register, which is an 8-bit register thus:

BIT 7							BIT 0
N	V		B	D	I	Z	C

Bit 5 is not used.

The 6510 has a number of instructions which test the states of these flags and cause the program to branch if a flag is in a certain state. These instructions correspond loosely to the IF ... THEN ... GOTO of BASIC, and play an important part in all machine code programs.

Program Counter PC

This 16-bit register stores the address in memory of the next instruction to be executed.

Stack Pointer S

This is an eight bit register used to index the memory area in which subroutine return addresses are stored.

THE 6510 INSTRUCTION SET

The instructions in the machine code instruction set include commands for moving data between the 64's memory and the three data registers, for performing arithmetic and logical operations on the contents of the accumulator, and for testing the results of these operations.

All 6510 instructions take the form of single byte numbers, but may be followed by one or two further bytes as an operand. For example,

```
    169    1
```

means 'load the accumulator register with the number 1'. Instructions are usually expressed in hexadecimal as this is more convenient. The instruction given above would look like this in hexadecimal:

```
    $A9    $01
```

The $ prefix indicates that the numbers are in hex.

Instructions may be entered in a number of ways. The simplest method is to **POKE** the numbers into store using a BASIC program. This method is unbelievably tedious and error-prone for all but the shortest of programs. For more complex programs an *assembler* should be used. This is a special program which allows you to enter and amend a sequence of machine code instructions using mnemonics to represent the instructions, and then translates these into the numbers which the microprocessor can read. For example, the instruction above could be entered in the form:

```
LDA    #$01
```

where **LDA** means **LoaD** the **Accumulator**, and #$01 is the number to be loaded. The # is used to distinguish a number from an address in assembly language, while the $ symbol again indicates a hexadecimal number.

The instructions of the 6510 are described below.

Transferring Data

The first group of instructions move data between the 6510 registers and the memory.

LDA This instruction copies a byte of data from memory into the accumulator.

STA Copies the accumulator to memory.

For example:

```
LDA $0401

STA $0402
```

copies data from location $0401 (which in decimal is 1025) into the accumulator and then stores it in location $0402 (1026).

There are a number of different forms of the **LDA** and **STA** instructions, which differ in the way the memory is used. These different forms are called *addressing modes.*

Absolute Addressing Mode

Instructions using this addressing mode give the address of the memory location as a two byte number following the instruction code. For example:

```
LDA $0401
```

The code for the **LDA** instruction is $AD (or 173), so the instruction is coded as:

```
AD 01 04
```

(The low byte of the address is always given first.)

Similarly:

```
STA $0402
```

STA has the code $8D, so the instruction is:

```
8D 02 04
```

Immediate Addressing Mode

In this mode the data is read not from a specified memory location, but from the location after the program instruction. This is used to load the accumulator with known values. The # symbol is used to signify that the number following the instruction, called the *operand*, is to be interpreted as a number:

```
LDA #1
```

means load the accumulator with the number 1. The code for the instruction in $A9, so the full instruction would be:

```
A9 01
```

There is no corresponding **STA** instruction – it would be meaningless!

Zero Page Addressing Mode

Here the memory location indicated is in page zero of the memory (the first 256 bytes, $0000 to $00FF, for which the high byte of the address is $00).

```
LDA $7D
```

loads data from $007D. The code is $A5, so the full instruction is:

```
A5 7D
```

This is equivalent to the absolute addressed:

```
LDA $007D        AD 7D 00
```

but is faster in operation, and takes less storage space. Because zero-page addressing is faster it is useful to store frequently used data in page zero.

Other data transfer instructions, **LDX**, **LDY**, **STX** and **STY** transfer data to and from the X and Y index registers. These instructions have the following codes:

MODE	LDX	LDY	STX	STY
ABSOLUTE	AE	AC	8E	8C
IMMEDIATE	A2	A0	–	–
ZERO PAGE	A6	A4	86	84

Further instructions move data between the accumulator and the X and Y registers. These are single byte instructions with no operands:

	CODE	FUNCTION
TAX	AA	Copies Accumulator to X register
TAY	A8	Copies A to Y
TXA	8A	Copies X to A
TYA	9A	Copies Y to A

Arithmetic

The 6510 can perform addition and subtraction operations. Multiply and divide must be calculated by program, as instructions are not provided.

Addition is performed by the command **ADC**. This adds a number in memory to the number in the accumulator, storing the result in the accumulator. If the carry flag in the processor status register was set before the operation, 1 is added to the result. If the result is greater then 255, the least significant eight bits remain in the accumulator and the carry flag is set; otherwise it is cleared.

The use of the carry flag allows numbers of two bytes or more to be added together, with the carry operating in the same way as when you add two numbers with pencil and paper.

To add two single byte numbers stored in, say, $1000 and $1001:

```
CLC               Clear the carry flag.

LDA $1000         Load first number.
```

```
ADC $1001         Add second number.

STA $1002         Store result.
```

The result is stored in $1002. The two original numbers are unaffected by the operation.

Notice that the carry flag was cleared before the addition. This should always be done before an addition, as the flag may have been set by a previous operation, and this would give the wrong result for the addition. The code for CLC is $18.

Like **LDA** and **STA**, **ADC** can be used in several addressing modes:

Absolute Mode	ADC<address>	6D
Immediate	ADC#<number>	69
Zero Page	ADC<zero page address>	65

Larger numbers can be added in a similar way to that shown in the example above. Suppose there are two 3 byte numbers stored in locations $A1, $A2, $A3 and $A4, $A5, $A6, and the sum of these is to be stored in $B1, $B2, $B3. The method is as follows:

```
CLC         Clear carry flag
LDA  $A1    Load lowest byte of first number
ADC  $A4    Add lowest byte of second number
STA  $B1    Store lowest byte of sum
LDA  $A2    Load second byte of first number
ADC  $A5    Add second byte of second number
STA  $B2    Store second byte of sum
LDA  $A3    Load highest byte of first number
ADC  $A6    Add highest byte of second number
STA  $B3    Store highest byte of sum
```

Just as in normal arithmetic, the lowest parts of the numbers (the least significant bytes) are added first.

Subtraction

ADC is complemented by a subtraction command **SBC**, which subtracts a number from that in the accumulator. This is used in a similar fashion to the **ADC** command, and again the carry flag is used when handling large numbers. There is an important difference – the carry flag must be *set* before the subtraction, and will be cleared to indicate a 'borrow' when the result of the subtraction is less than zero. If the flag is clear before subtraction, 1 is subtracted from the result. The different addressing modes are again available:

Absolute	SBC <address>	ED
Immediate	SBC# <number>	E9
Zero Page	SBC<zero page address>	E5

This example subtracts 1 from a two byte number stored in $0401, $0402:

```
SEC            Set carry = clear borrow
LDA  $0401     Load low byte
SBC  #1        Subtract 1
STA  $0401     Store new low byte
LDA  $0402     Load high byte
SBC  #0        Subtract 0
STA  $0402     Store result
```

The last 3 instructions are not as pointless as they may seem. If the low byte was zero to start with, subtracting 1 gives 255 ($FF) and clears the carry flag (sets 'borrow'). The last three instructions load the high byte of the original number and subtract

0, but if the carry flag has been cleared, 1 will be subtracted from the result. So, if locations $0401 and$0402 had originally contained the number $2900, the result would be $28FF.

Addition and subtraction can be performed only on numbers in the accumulator. There are no corresponding commands for numbers held in the X and Y registers.

Incrementing and Decrementing

You can add or subtract 1 to a number in memory or in the X or Y register using Increment and Decrement instructions:

	CODE	FUNCTION
INX	E8	Adds 1 to X register
DEX	CA	Subtracts 1 from X register
INY	C8	Adds 1 to Y
DEY	88	Subtracts 1 from Y

These are single byte instructions with no operand. The two instructions **INC** and **DEC** perform similar operations on numbers in the memory.

`INC <address>` Adds 1 to the contents of that address

`DEC <address>` Subtracts 1 from the contents of the address

INC and **DEC** do not alter the accumulator or the X and Y registers.

None of these increment and decrement instructions has any effect on the carry flag. The Z flag will be set if the result of any of these operations is zero, and the N flag will be set if Bit 7 of the result is set. (Z and N are similarly affected by **ADC** and **SBC**.)

Branches

The 6510 has a number of conditional branch instructions which test a flag in the P register and cause the program to branch if the tested flag is set or clear. For example, the instruction **BEQ** causes a branch if the Z flag is set, that is if the result of the last operation performed was zero. **BNE** does the opposite: the program branches if Z is clear, after a non-zero result.

The instructions have single byte operands which give the branch destination relative to the current program position. If the operand is between 0 and 127 the jump is forwards; if the operand is between 128 and 255 the jump is backwards. An operand of 128 causes a jump to the instruction whose address is 128 less than the current position in the program, and the size of the jump gets less as the number increases to 255. So,

```
    BNE 20
```

causes a jump forward of 20 bytes if the zero flag is set and:

```
    BNE 230
```

causes a jump back of 26 bytes.

A vital point to remember is that the jump is not measured from the address of the branch instruction but from that of the next instruction. This is because the program counter is increased to point to the next instruction before the branch instruction is processed.

Branch instructions are indispensable in programs of any length, as they are the machine code equivalents of BASIC commands like **IF ... THEN ... GOTO**, allowing programs to make decisions and act accordingly. The instructions also provide a convenient way of creating loops. Consider the following:

```
   . . .
   . . .
   LDX #10       A2 0A
   DEX           CA
   BNE 253       D0 FD
   LDA $ABCD     AD CD AB
   . . .
   . . . etc.
```

This loop is repeated 10 times, until the Z flag is set as the value in the X register becomes zero, at which point the program continues. The loop in the example is literally a waste of time, doing nothing more than delay the program slightly. There could however be a number of instructions between the beginning of the loop and the **DEX** instruction.

An important point to remember is that the branch instruction must follow immediately after the instruction which creates the condition under test. Any instruction which modifies any of the 6510 data registers (and quite a few which don't) will modify the flags in the processor status register, so the flags are constantly changing. If there are any instructions between the one whose effect you wish to test and the test itself, the flags may change again before the test is performed. For example, in a loop ending:

```
 . . .
 . . .
 DEX
```

```
LDA $1234
BNE . . .
```

the flags will be set by the **DEX** instruction, but immediately changed by the **LDA** instruction to indicate the nature of the number loaded into the accumulator.

The full list of branch instructions is:

	CODE	OPERATION
BEQ	F0	Branch on result = 0 (Z set)
BNE	D0	Branch on result< >0
BCS	90	Branch if carry set
BCC	B0	Branch if carry clear
BMI	30	Branch if negative (N set)
BPL	10	Branch if positive (N clear)
BVS	70	Branch if V set (overflow)
BVC	50	Branch if V clear

These instructions test the four flags Z, C, N and V.

The Z or zero flag indicates a result of zero, either in the accumulator, or the X or Y register, or if the **INC** or **DEC** instruction was used, in the memory location concerned.

The carry flag may be set or cleared by program. It is also altered by addition and subtraction instructions, and by the register shift instructions which are described later.

The N flag is a copy of Bit 7 of the register last altered. In signed arithmetic this bit indicates the sign of the number - set for negative, clear for positive.

The V flag indicates an overflow in twos complement arithmetic. The flag may be cleared by the **CLV** instruction.

Comparisons

The three instructions **CMP**, **CPX** and **CPY** are used to compare registers with numbers in memory. These instructions do not alter either memory or the register, but the Z, C and N flags are altered to indicate the result of the comparison.

CMP compares the accumulator with another number. For example:

```
    CMP #1
```

compares the contents of the accumulator with the number 1. The comparison leaves the flags in the state they would be in after setting the C flag and subtracting the number from the accumulator. That is:

C is set if A >= Number, otherwise C is clear.

Z is set if A = Number, otherwise Z is clear.

N is set if the operation (A−Number) would leave a 1 in Bit 7.

The instructions **CPX** and **CPY** are similar, except that the X or Y register is tested instead of the accumulator. All these instructions may test memory or numbers using the addressing modes described for **LDA**.

MODE	CMP	CPX	CPY
ABSOLUTE	CD	EC	CC
IMMEDIATE	C9	E0	C0
ZERO PAGE	C5	E4	C4

ADDRESSING MODES

Many of the 6510 instructions can take several forms, providing different ways for specifying the memory location to be used by the instruction. These different forms of the instructions are said to use different addressing modes, because they address the memory in different ways. We have already introduced three addressing modes: absolute addressing, zero page addressing and immediate addressing. These and the other modes are explained below.

Absolute Addressing

This mode uses a two byte operand after the instruction code. The two bytes are the low and high bytes of the address of the data in memory. Absolute addressing is indicated by the full address after the instruction:

```
    LDA  $ABCD
```

Zero Page Addressing

This mode uses a one-byte operand following the instruction code. This byte is the low part of an address in page zero ($0000 to $00FF). Instructions in this mode are written thus:

```
    LDA  $C7
```

Indexed Addressing

Indexed addressing uses the X or Y index register to modify the address given after the instructon, so that the address of the data is Operand+X or Operand+Y. For example:

Absolute:

```
LDA $ABCD
```
loads the contents of $ABCD

Absolute Indexed:

```
LDA $ABCD,X
```
loads the contents of $ABCD+X.

The Y index register can also be used in a similar manner.

As well as absolute addressing there are also zero page indexed addressing modes. For example:

```
LDA $AB,X
```

loads the accumulator with the contents of $00AB+X.

Indirect Addressing

In the indirect addressing modes the indicated memory location is not used to store the data, but holds the low byte of the address of another location where the data is to be placed. The high byte of the address is held in the next location. Only one 6510 instruction, the jump instruction **JMP**, can use simple indirect addressing, but many instructions use indexed forms of indirect addressing.

Indirect Indexed & Indexed Indirect Addressing

In indirect indexed addressing the single operand byte following the instruction indicates the first of a pair of zero page locations whose contents form a pointer to the target location. The contents of the Y index register are added to the pointer to find the final address.

Indexed indirect addressing uses the X register to index the zero page address in which the pointer is to be found. So:

Indirect Indexed:

`LDA (TABLE),Y` Reads the zero-page locations TABLE and TABLE+1 and adds Y to the contents to produce the address of the data.

Indexed Indirect:

`LDA (TABLE,X)` Adds X to the address TABLE to find the zero page location where the data pointer is held.

OTHER ADDRESSING MODES

Implied addressing

No address at all is specified, as in **CLC**, **RTS** etc.

Relative Addressing

The address is given as an offset from the current program address. This is the addressing mode used by the relative branch instructions. For example:

```
BCS 35
```

means branch to the program instruction at the address 35 bytes above the current address in the program counter.

Accumulator Addressing

The instruction acts only on the accumulator. Again no operand is used. This addressing mode is used only by the register shift instructions.

JUMPS AND SUBROUTINES

In addition to the relative branch instructions described above, there is an absolute jump instruction, **JMP**. This causes the program to jump to another location. There are two addressing modes:

Absolute:

`JMP $ABCD` The program continues at the location specified in the next two bytes, in this case $ABCD.

Indirect:

`JMP ($ABCD)` The program jumps to the address whose low byte is contained in location $ABCD, and whose high byte is contained in location $ABCE.

A similar instruction, **JSR**, is used to call subroutines. This may be used only in the absolute addressing mode, for which the instruction code is $20.

`JSR $C000` Jumps to the subroutine at $C000.

Before jumping to the subroutine, the 6510 stores the current value of the program counter in a special area of memory called the *stack*, so that it may be restored on completion of the subroutine. The command **RTS** causes the return from subroutine; the 6510 reads the first two bytes on the stack and resets the program counter to that address.

The stack is held in page 1 of the 64's memory, locations $0100 to $01FF. The stack is designed to be used as a last-in first-out store, and the *stack pointer* register is used to indicate the first free location in the stack. When the 6510 is reset, the pointer is set to $FF (the high byte is always $01 and can be disregarded), and the pointer is decremented as the data is added by subroutine calls, and is incremented as data is removed by **RTS** commands, so that it always indicates the first free location, to which the next byte to be added will be written. This allows the nesting of subroutines; as subroutines are called the return addresses are added to the stack, which fills down from $01FF. Data is removed in reverse order, and the stack empties up to $01FF.

If the stack pointer should reach zero and a further subroutine call is made, the pointer would return to $1FF, and the oldest items in the stack would be overwritten, which would cause havoc! This is unlikely to happen except in very complicated programs; most of the time you can forget about the working of the stack and let it look after itself.

Data may be written to the stack by a program, using the instruction **PHA**, which 'pushes' the contents of the accumulator onto the stack. This can be useful if you want to put a number on one side for a moment while performing another calculation. Numbers are pulled from the stack by the instruction **PLA**.

It is also possible to store the processor status register P on the stack and recall it, using **PHP** and **PLP**.

	CODE	OPERATION
PHA	48	A to STACK
PLA	68	STACK to A
PHP	08	P to STACK
PLP	28	STACK to P

The stack pointer is automatically adjusted by the 6510 when these instructions are used. Don't forget to take things off the stack in the reverse order to that in which you put them on.

Be careful using these instructions. Remember that they all use the same stack as is used to store subroutine return addresses. This means that you must be careful not to execute a **RTS** instruction between writing data to the stack and reading it back, or the program will return to the wrong place. You can of course call a subroutine after writing the data; it will return without any problem. What you must not do is return from a subroutine which has pushed data onto the stack before taking the data back off the stack.

Two further instructions, TSX and TXS, allow you to modify the stack pointer.

TSX	BA	Copies the stack pointer to the X register

TXS	9A	Copies the X register to the stack pointer

This means you could, if you felt confident, maintain two or more stacks, but this is not advisable. There would be a slight speed advantage in using an area of page 1 as a table, indexed by the stack pointer, but it would be both easier and less hazardous to use indirect indexed addressing in another part of the memory.

INTERRUPTS

A microprocessor in a computer such as the 64 has several jobs to do. As well as running the BASIC interpreter program, the screen must be managed, the keyboard checked for keypresses, and many other routine operations performed. The ideal microprocessor would be able to run many separate programs at the same time, but such a processor does not exist. Instead, the 6510 has a facility which allows programs to be interrupted while another program is run and then restarted at the point at which they were stopped.

Two of the 40 pins on the 6510 chip are *interrupt request* pins. Peripheral devices applying signals to one or other of these pins will stop the 6510 in the middle of whatever it is doing, and divert it to another piece of program. The 6510 will be returned to the original program by an instruction at the end of the interrupt program. The two pins are similar in use, the difference being that the IRQ (interrupt request) pin is ignored if the I bit of the processor status register has been set by an SEI instruction, whereas the NMI (non-maskable interrupt) pin can not be ignored.

When a suitable signal is applied to the IRQ pin the 6510 finishes the current instruction, stores the P register and the program counter on the stack, sets the interrupt disable flag (I) in the P register and jumps to the program at the address held in locations $FFFE and $FFFF. The processor continues to execute the program from this point until a **RTI** instruction is reached. The **RTI** (return from interrupt) acts in a similar way to **RTS**, but restores the P register from the stack as well as the program counter.

Interrupts in response to the **NMI** pin are similar, except that the start address for the interrupt program is held in locations $FFFA and $FFFB.

The interrupt feature allows the effects of several programs running at once. The 64 has a *clock* circuit which interrupts the 6510 every 1/60 second to call the keyboard scanning routine. This checks the keyboard and puts the ASCII code of any key held down into the keyboard buffer before **RTI**ing. The effect of this is that the rest of the BASIC interpreting routines do not need any complex subroutines to read the keys; they just look in the keyboard buffer (this is what the BASIC **GET** command does).

Interrupts are discussed further in Chapter 21, which shows how interrupt programs may be used to modify the display.

LOGIC

The two logical operations **AND** and **OR**, familiar in BASIC, are also available in machine code, as is a third, the Exclusive OR. The Exclusive OR differs from OR in that 1 OR 1 gives 1, but 1 EOR 1 gives 0. The mnemonics for the three instructions are **AND**, **ORA** and **EOR**.

The truth tables for these operations are:

A	B	A OR B	A EOR B	A AND B
0	0	0	0	0
0	1	1	1	0
1	0	1	1	0
1	1	1	0	1

The instructions all act on the accumulator.

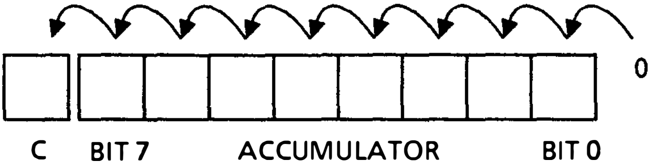
Logical instructions are useful for testing and modifying selected bits within a byte without altering the rest. For example the **AND** instruction may be used to *mask* a part of a byte. To inspect the first four bits of location $1234, the data would be loaded into the accumulator and ANDed with the number 15 (15 in binary is 00001111). As the **AND** operation only sets the bits which are set in both the numbers in the operation, the top four bits of the result will be zero, and the lower four bits will be a copy of the lower four bits of location $1234. The instruction sequence would be:

```
    LDA $1234
    AND #15
```

SHIFTS

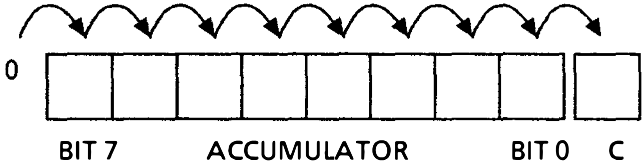
There are four shift instructions available which shift or rotate the bits in the accumulator or memory.

ASL Shifts the accumulator to the left. A zero is placed in Bit 0 and Bit 7 is moved to the carry flag. This is equivalent to multiplying the accumulator by two.



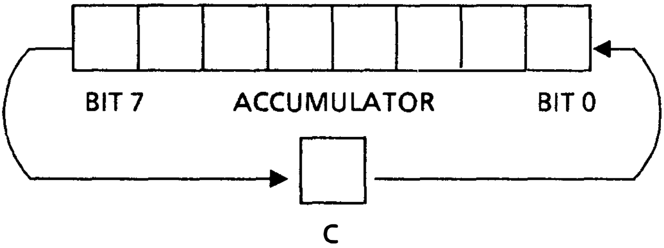
The ASL operation

LSR Shifts the accumulator to the right. A zero is placed in Bit 7, and Bit 0 it moved to the carry flag. This is equivalent to dividing the number in the accumulator by 2.



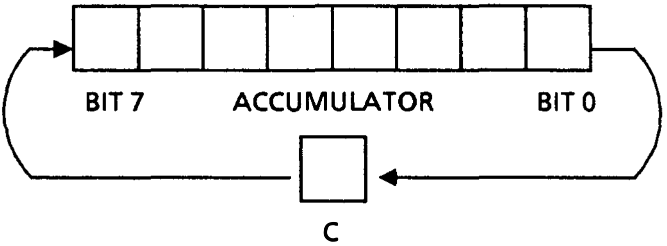
The LSR operation

ROL Rotates the number in the accumulator to the left. The carry flag is copied to Bit 0, and Bit 7 is moved to the carry flag.



The ROL operation

ROR Rotates the accumulator to the right.



The ROR operation

USING MACHINE CODE WITH BASIC

Machine code programs are called from BASIC by the SYS command. This is similar in effect to the GOSUB command except that the subroutine called is a machine code routine at the address specified in the SYS command. Control is returned to BASIC when the routine is completed by the machine code instruction RTS. The SYS command is followed by the address of the start of the machine code program. For example:

```
    SYS 64738
```

calls the routine in ROM which resets the 64 when it is switched on. *Don't* try this if you have a valuable program loaded – it will be erased.

Storing Machine Code Programs

The most convenient place to store most machine code programs is in the 4k block of memory between 49152 ($C000) and 52247 ($CFFF) which is not used by the 64 in the execution of BASIC programs.

If your program is longer than 4k it may be placed lower down in memory in the area usually reserved for BASIC programs. This area normally extends from 2048 to 40759 but you can reserve a part of it for machine code programs by altering the 'top of memory' pointer in locations 55 and 56.

To use this method, work out the new top of memory you need by subtracting the length of your machine code program from 40759, and **POKE** the low byte of that address into location 55 and the high byte into location 56. Finally reset the pointers by typing:

```
CLR
```

The memory above the new 'top of memory' is now protected from being overwritten by BASIC variables.

The Kernal Routines

Many of the subroutines present in the 64's ROM can be used in your machine code programs. Some of the most accessible and well documented form part of the operating system called the kernal.

To enable you to make use of these routines, a table of their start addresses is kept in memory - a feature shared by all Commodore machines. Calling each routine involves a **JSR** to the appropriate part of the table. By arranging the table in this way, programs written on one Commodore machine may be more easily translated to run on another. An example of the use of the kernal routines to send output to a printer may be found in Chapter 26, and a list of the routines and their functions is given in Appendix 11.

Learning to Write Machine Code

This chapter is not intended to teach you all there is to know about machine code programming, rather it introduces the fundamentals of the language. If you wish to study the subject further, there are a number of books aboout the 6502/6510 microprocessors which you may find helpful.

We have written a number of machine code programs for this book, and if you study these you will soon get a feel for the language, and realise that despite first appearances, machine code programming is not really very difficult.

CHAPTER 19

BIT-MAPPED GRAPHICS - PART 1

As well as the normal text display, the 64 can produce a high resolution bit-mapped display in which the pixels correspond to individual bits in a defined area of memory. There are two alternative bit-mapped display modes: standard bit-mapped mode, which has a screen resolution of 320 pixels by 200 pixels in two colours; and multicolour bit-mapped mode which has a lower resolution of 160 by 200 pixels, but in four colours.

The graphics modes are controlled by the VIC–II chip. Bit 5 of the control register at location 53265 selects bit-map mode: if the bit is set, bit-mapped mode is selected; if the bit is clear the 64 display is in text mode. To select multicolour bit-mapped mode, Bit 4 of the second VIC control register at location 53270 must also be set.

Display Memory

When you enable bit-mapped mode, you must also decide where in memory the screen information will be stored. Bit mapped displays need nearly 9k of memory, 8000 bytes for the picture data and 1000 for the text screen memory, which in the bit-mapped modes is used to hold colour information. The VIC chip can only address 16k of memory at a time, so the two blocks of memory needed for bit-mapped displays must lie within one of the four 16k *banks* or sections which make up the 64k of total memory space in the 64.

Within the 16k bank, the 1k of text screen memory may be set to begin at any location whose address is a multiple of 1024, but the 8k bit-map display memory area must begin either at the beginning of the bank, or at the mid point, 8k above the start. These limitations mean that the bit-map screen memory can not be put in the obvious place at the top of the user RAM (from 32k to 40k), as there is no more RAM in this bank into which the text screen memory could be placed. The best place for the bit-map display memory is therefore at the top of the second bank, from 24576 to 32575, with the text screen memory beginning at 23k (23552). This leaves about 21k of memory free for BASIC programs, which should be enough for most purposes.

The bank which the VIC chip addresses is set by two **POKE** commands. First the lowest two bits of the data direction register of the CIA interface chip must be set to 1 by:

```
POKE 56578, PEEK(56578) OR 3
```

and the bank number must be **POKE**d to the lowest two bits of the interface port at 56576:

```
POKE 56576, (PEEK(56576)AND252) OR B
```

where B is the bank number given by the table:

B	BANK	Starting Location
3	0 to 16k	0
2	16k to 32k	16384
1	32k to 48k	32768
0	48k to 64k	49152

Screen Memory Positioning

The position within the 16k bank of the bit-mapped screen is controlled by Bit 3 of location 53272. If the bit is zero the bit-mapped display is placed at the beginning of the bank. If the bit is 1 the screen memory begins at the mid-point of the bank.

The position of the text screen within the bank is controlled by the four highest bits of location 53272. The value of these bits is the number of 1k units by which the start of the screen memory is offset from the beginning of the 16k bank.

To set the bit-map display to begin at 24k, which is the mid-point of the 16k-32k bank, Bit 3 of 53272 must be set. To place the text screen memory at 23k, the four most significant bytes must be set to 23-16, which is 7. The value to be POKEd into location 53272 is therefore 7*16 + 8, which is 120.

So, to set the 64 to the bit-mapped display mode with the display at 24k, we need the following short program:

```
5      REM SELECT BANK 16-32K
10     POKE 56578, PEEK(56578) OR 3
20     POKE 56576, (PEEK(56576) AND
       252) OR 2
30     POKE 53265, PEEK(53265) OR
       32:REM SET BIT MAPPED MODE ON
40     POKE 53272, 120:REM SET MEMORY
       POINTERS
```

Restoring the Text Display

To restore the normal display manually, you can hold down RUN/STOP and tap RESTORE. Restoring the text display by program is a matter of returning all the registers to their normal values - clearing Bit 5 in location 53265 and resetting the address pointers. Add the following lines to the previous program:

```
100   GET K$: IF K$="" THEN 100
200   POKE 53265, PEEK(53265) AND 23
210   POKE 53272, 21
220   POKE 56578, PEEK(56578) OR 3
230   POKE 56576, PEEK(56576) OR 3
```

When run, the program will set up a high-resolution display and wait until you press a key, after which the normal display will be restored.

Clearing the Screen

The display you see when you run the program will be filled with random blocks of colour. These correspond to the data with which the memory happened to be filled when you set bit-mapped mode. To clear the screen, you must clear the memory by **POKE**ing it with zeros.

Colour

The colour of a bit-mapped display is controlled by the numbers in the text screen memory. Each number in this memory area controls the colour of an 8x8 block of pixels on the high resolution display. The 'foreground' colour – the colour displayed for a bit set to 1 in display memory – is controlled by the four highest bits of the screen memory, while the background colour depends on the four lowest bits. For example, to set a foreground colour of red (colour 2) and a white background (colour 1), each location in the text memory must be set to 16*2 + 1, which is 33.

Add the following two lines to the program. Line 50 sets the colours, and line 60 clears the screen memory.

```
50    FOR L=23552 TO 24551: POKE L,
      33: NEXT L
60    FOR M=24576 TO 32575: POKE M,
      0: NEXT
```

The process of setting the colours and clearing the screen takes around 35 seconds - far from instantaneous! The only way of speeding up the screen clearing is to use a machine code program. A program to set the screen mode and clear the screen is described later, but first a few words about multicolour bit-mapped mode.

MULTICOLOUR BIT-MAPPED MODE

Multicolour bit-mapped mode is similar to standard bit-mapped mode, except that the screen resolution is reduced, and that each pixel may take one of four colours. The two extra colours are controlled by the colour memory (always located from 55296 to 56295) and the background colour stored in location 53281. Each pixel is controlled by two bits of data in the display memory as follows:

COLOUR	BIT PATTERN	COLOUR DISPLAYED
0	00	Background - location 53281
1	01	Upper four bits of screen memory
2	10	Lower four bits of screen memory
3	11	Colour memory

To change from standard bit-mapped mode to multicolour bit-mapped mode, Bit 4 of location 53270 must be set, by the command:

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

The multicolour bit must be cleared on returning to text mode by:

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

USING MACHINE CODE WITH GRAPHICS

Because of the speed problems using BASIC to control the graphics, machine code can be very useful. Included in this book are a number of machine code programs to speed up graphics functions, and the first of these sets bit-mapped mode (BMM) or multicolour bit-mapped mode (MCBMM) and clears the bit-mapped screen.

If you have an assembler and are familiar with machine code you can enter the machine code as it is given in this chapter and Chapter 20. Alternatively, you can type in the BASIC loader programs in Appendix 18 which you can use to load the machine code into your 64. These programs comprise a series of **DATA** statements which form the machine code program and a short routine to **READ** the **DATA** and **POKE** it into memory.

```
 10 ! **********************
 20 ! *** BIT-MAP MODES ***
 30 ! **********************
 40 !
 50 * = $C000               Starts at 49152
 60 !
 70 !
 80 COL1 = 686              Colour 1
 90 COL2 = 687              Colour 2 (Background
                            in Standard BMM)
100 COL3 = 688              Colour 3 (MC BMM
                            only)
110 MCBM = 689              1 = BMM, 0 = MC
                            BMM
120 !
130 !
```

```
140 !
160 BMM     LDA #1      Start for BMM
170         STA MCBM
180         BNE CLEAR
190 !
210 MCBMM   LDA #0      Start for MCBMM
220         STA MCBM
230 !
240 ! START BY CLEARING BMM SCREEN
245 !
250 CLEAR   LDA #$60    Wipe 64x256 bytes
                        starting at $6000
260         STA 252
270         LDA #0
280         STA 251
290         LDX #64
300         JSR WIPE
310 ! SET COLOURS
320 COLOUR LDA #92      Put COL1 and COL2
                        into screen mem
330         STA 252
340         LDA #0
350         STA 251
360         LDA COL1    Multiply COL1 by 16
370         ASL A
380         ASL A
390         ASL A
400         ASL A
410         ORA COL2    Add COL2
420         LDX #8
430         JSR WIPE
440 !
450         LDA MCBM
460         BNE HIRES
470         LDA #$D8    If MC, put COL3 into
                        colour memory
480         STA 252
490         LDA #0
500         STA 251
510         LDX #8
520         LDA COL3
```

```
530          JSR WIPE
540 ! NOW SET POINTERS FOR BM MODE
545 !
550 HIRES    LDA 56578   Set bank 2
560          ORA #3
570          STA 56578
580          LDA 56576
590          AND #252
600          ORA #2
610          STA 56576
620          LDA 53265   Set BMM on
630          ORA #32
640          STA 53265
650          LDA #120    Set screen position
660          STA 53272
670 !
680          LDA MCBM
690          BNE BMMEND
700          LDA 53270   If MC, set MC mode
710          ORA #16
720          STA 53270
730 !
740 !
750          BMMEND RTS
760 !
770 !
780 !
790 ! SWITCH TO TEXT MODE
795 !
800 LORES    LDA 56578   Reset all pointers for
                         text
810          ORA #3      Restore Bank 3
820          STA 56578
830          LDA 56576
840          AND #252
850          ORA #3
860          STA 56576
870          LDA 53265   Clear BMM bit
880          AND #223
890          STA 53265
900          LDA 53270   Clear MC bit
```

```
 910          AND #239
 920          STA 53270
 930          LDA #21      Reset pointers
 940          STA 53272
 950          RTS
 955 !
 960 ! WIPE OVER BLOCK WITH
     CHARACTER IN ACCUMULATOR
 965 !
 970 WIPE     LDY #127     Subroutine to fill
                           memory with
                           character in
                           accumulator
 980          F1 STA (251),Y
 990          DEY
1000          BPL F1
1010          PHA
1020          CLC
1030          LDA 251
1040          ADC #128
1050          STA 251
1060          LDA #0
1070          ADC 252
1080          STA 252
1090          PLA
1100          DEX
1110          BNE WIPE
1120          RTS
1130          END
```

The exclamation marks in the listing are used by this assembler to indicate comments, and do not form part of the program.

To use the machine code routine to set bit-mapped mode, POKE the foreground colour into location 686 and the background colour into location 687, and call the routine with the command:

```
SYS 49152
```

To return to text mode, use:

```
SYS 49266
```

To set multicolour bit-mapped mode, store the number of the third colour in location 688, and start the routine at line 210 of the listing using the command:

```
SYS 49159
```

Other call addresses allow you to switch from text to bit-mapped mode without clearing the screen, to clear the screen, or to reset the colours. The full list of entry points is:

BMM	49152	Clear, set BMM and colours
MCBMM	49159	Clear, set MCBMM and colours
CLEAR	49164	Clear BM screen and reset colours
COLOUR	49177	Set colours
HIRES	49221	Set to BM or MCBMM
LORES	49266	Return to text mode

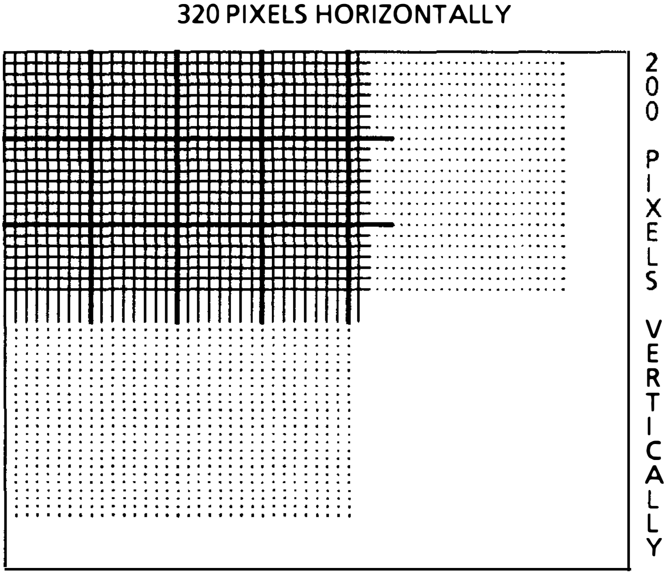
NOTE: Because these routines place the bit-mapped screen in the middle of the BASIC program area, at 24k, there is a risk that the display may be corrupted by the BASIC or by program variables. The bit-mapped screen can be protected from BASIC by resetting the top of BASIC memory thus:

```
POKE 55,0: POKE 56, 92: CLR
```

Which sets the top of memory to 23732 and thus prevents any BASIC programs or variables overwriting the colour memory. A line such as this should be the first in every program which uses bit-mapped graphics.

PLOTTING POINTS

The bit-mapped screen is arranged as a grid of 200 rows of 320 pixels (or 160 pixels in multicolour mode). Each byte in the screen memory represents a row of 8 pixels.



However, the layout of the memory is a little complicated. The top line (Row 0) of the display is mapped onto the display memory 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
```

The bytes are laid out in blocks of eight, corresponding to the characters of a text display. This layout means that plotting points on the bit-mapped screen requires some calculation to convert X and Y co-ordinates to byte addresses. The following formulae can be used to set and clear pixels (the value BASE represents the address of the start of the bit-mapped screen memory).

```
ROW  = INT(Y/8)
CHAR = INT(X/8)
LINE = Y AND 7
BYTE = ROW*320 + CHAR*8 + LINE + BASE
BIT  = 7-(X AND 7)
```

The pixel is set by:

```
POKE BYTE, PEEK(BYTE) OR 2↑BIT
```

and cleared by:

```
POKE BYTE, PEEK(BYTE)AND(255-2↑BIT)
```

The second program in our machine code graphics package plots points on the bit-mapped screen.

```
 10 ! ************
 20 ! *** PLOT ***
 30 ! ************
 40 !
 50 !
 60 * = $C0B8              Starts at 49366
 80 !
 90 !
100 BYTELO = 251
```

```
110 BYTEHI = 252
120 T1     = 253
130 T2     = 254
140 XLO    = 679
150 XHI    = 680
160 Y      = 681
170 !
180 COLNO  = 685
190 MCBM   = 689
200 BASE   = 24576
220 !
230 !
240 !
250 PLOT   LDA Y       Find ROW = Y/8
260        LSR A
270        LSR A
280        LSR A
290        STA ROW
300 !
310        LDA XHI     Find CHAR
320        LSR A
330        LDA XLO
340        ROR A
350        LSR A
360        LDX MCBM    If MC mode then
                       CHAR = X/4
370        BEQ PLOT1
380        LSR A       If not MC mode CHAR
                       = X/8
390 PLOT1  STA CHAR
400
410        LDA Y       LINE = Y AND 7
420        AND #7
430        STA LINE
440 !
450        LDA ROW     ROW = ROW*64
460        STA T1
470        LDA #0
480        STA T2
490        LDX #6
500 P1     JSR TIMES2
```

```
510         DEX
520         BNE P1
530         LDA T2
540         STA BYTEHI
550         LDA T1
560         STA BYTELO
570 !
580         JSR TIMES2 Add ROW * 256
590         JSR TIMES2
600         CLC
610         LDA T1
620         ADC BYTELO
630         STA BYTELO
640         LDA T2
650         ADC BYTEHI
660         STA BYTEHI BYTE now holds
                       ROW*320
670 !
680         LDA #0     Find CHAR*8
690         STA T2
700         LDA CHAR
710         STA T1
720         JSR TIMES2
730         JSR TIMES2
740         JSR TIMES2
750         CLC
760         LDA T1     Add CHAR*8 to
                       BYTE
770         ADC BYTELO
780         STA BYTELO
790         LDA T2
800         ADC BYTEHI
810         STA BYTEHI BYTE now holds
                       ROW*320 + CHAR*8
820 !
830         CLC        Add LINE to BYTE
840         LDA LINE
850         ADC BYTELO
860         STA BYTELO
870         LDA #0
880         ADC BYTEHI
```

```
 890        STA BYTEHI BYTE holds
                       ROW*320 + CHAR*8
                       + LINE
 900 !
 910        CLC        Add screen start
                       address
 920        LDA #<BASE Low byte of BASE
 930        ADC BYTELO
 940        STA BYTELO
 950        LDA #>BASE High byte of BASE
 960        ADC BYTEHI
 970        STA BYTEHI
 975
 980 ! BYTE = BASE + ROW*320 +
     CHAR*8 + LINE
 985
 990        LDA MCBM   Branch if MC mode
                       selected
1000        BEQ MCPLOT
1010 !
1020 !
1040 !
1050 BMPLOT LDA XLO    Normal mode PLOT
1060        AND #7
1070        STA BITT   BITT = X AND 7
1080        SEC
1090        LDA #7
1100        SBC BITT
1110        STA BITT   BITT=7-(X AND 7)
1120 !
1130        CLC        If BITT < > 0 then
                       find 2↑BITT
1140        LDA #1
1150        LDX BITT
1160        BEQ P3
1170 P2     ASL A
1180        DEX
1190        BNE P2     Accumulator now
                       holds 2↑BITT
1200 !
1210 P3     LDY #0
```

```
1220 !
1230        LDX COLNO   Test for Plot or Unplot
1240        BEQ UNPLOT  Branch ifCOLNO=0
1250        ORA (BYTELO),Y
1260        STA (BYTELO),Y   Plot the pixel
1270 !
1280        RTS
1290 !
1310 UNPLOT EOR #$FF    Clear the pixel
1320        AND (BYTELO),Y
1330        STA (BYTELO),Y
1340        RTS         Standard mode plot
                        ends here
1350 !
1360 !
1380 !
1390 MCPLOT LDA XLO     Plot in MC mode
1400        AND #3
1410        STA BITT
1420        SEC
1430        LDA #3
1440        SBC BITT
1450        ASL A
1460        STA BITT    BITT=2*(3-(X AND
                        3)
1470
1480        LDY #0      Move colour number to
1490        LDA COLNO   correct position in byte
1500        AND #3
1510        LDX BITT
1515        BEQ MCP2
1520 MCP1   ASL A
1530        DEX
1540        BNE MCP1
1550 MCP2   STA COLS    COLS = COLNO *
                        2↑BITT
1560 !
1570        LDA #%11111100   Set up mask
                             for pixel
1580        LDX BITT
1585        BEQ MCP4
```

```
1590        SEC
1600 MCP3   ROL A           Rotate mask to fit reqd
                            pixel
1610        DEX
1620        BNE MCP3
1630 !
1640 MCP4   AND (BYTELO),Y  Mask out bits
                            of pixel
1650        ORA COLS        Set new pixel
                            state
1660        STA (BYTELO),Y  Store the byte
1665 !
1670        RTS             End of MC plot
1680 !
1690 !
1700 !  MULTIPLY (T1,T2) BY 2
1710 !
1720 TIMES2 LDA #0
1730        ASL T2
1740        ASL T1
1750        ADC T2
1760        STA T2
1770        RTS
1780 !
1790 !
1800 ROW    NOP
1810 CHAR   NOP
1820 LINE   NOP
1830 BITT   NOP             Called BITT because
1840 COLS   NOP             BIT is a 6510
1850 !                      instruction
1860 !
1870 END
```

To use the program, **POKE** the low byte of the X co-ordinate into location 679, the high byte into location 680, and Y into location 681 (these locations are labelled XLO, XHI and Y in the program above) using the commands:

```
POKE 679, X AND 255
```

```
   POKE 680, X/256
   POKE 681, Y
```

Note that the program does not perform any checks to make sure that the co-ordinates are within the screen limits. Using overlarge values of X or Y is unlikely to have any ill effects other than spoiling the appearance of the display, but you should design your BASIC programs to avoid using incorrect values.

Location 685 (called COLNO in the program) is used to select the colour in which the pixel is plotted. In standard two colour mode, **POKE** 685 with 1 to plot in the foreground colour, or with zero to plot in the background colour (to erase a pixel). In multicolour mode, the colours are numbered from 0 to 3, as shown in the table on page 56.

The PLOT routine begins at location 49336, and so may be run by the command:

```
SYS 49336.
```

CHAPTER 20

BIT MAPPED GRAPHICS - PART 2

In this chapter we look further at the uses and applications of bit-mapped graphics, and introduce line drawing and block fill routines.

DRAWING LINES

Straight lines may be drawn using the equation:

```
Y=M*X+C
```

in which M represents the gradient of the line, and C is a constant. To find the equation of the straight line between the two points X1,Y1 and X2,Y2, we can perform the following calculations:

```
M = DY/DX = (Y2-Y1)/(X2-X1)
C = Y1 - M*X1
```

and then plot the line with:

```
FOR X=X1 TO X2: Y=M*X+C: JSR (PLOT)
```

with a suitable subroutine to plot the pixels.

If you try plotting a few lines by this method you will find that the lines appear broken if Y2-Y1 is greater than X2-X1. If this is so, reverse the algorithm to loop from Y1 to Y2 and calculate the values of X.

The third program in the machine code graphics package draws lines in this manner between two points on the screen:

```
100 !  ************
110 !  *** DRAW ***
120 !  ************
130 !
140 !
150 *  = $C1C8                 Starts at 49608
160 !
170 !
190 !
200 DRAW     LDA X2L           Take copy of line end
                               co-ordinates
210          STA XTL
220          LDA X2H
230          STA XTH
240          LDA Y
250          STA YT
260 !
270          LDA #0
280          STA NEG
290          SEC               DY = ABS(Y2-Y)
300          LDA Y2
310          SBC Y
320          BCS DRAW1
330          LDA NEG           If Y2 <Y then set NEG
340          EOR #1
350          STA NEG
360          SEC
370          LDA Y
380          SBC Y2
390 DRAW1    STA DY
400 !
410          SEC               DX = ABS (X2-X)
420          LDA X2L
430          SBC XL
440          STA DXL
```

```
450         LDA X2H
460         SBC XH
470         BCS DRAW2
480         LDA NEG       If X2<X then switch
                          NEG
490         EOR #1
500         STA NEG
510         SEC
520         LDA XL
530         SBC X2L
540         STA DXL
550         LDA XH
560         SBC X2H
570 DRAW2   STA DXH
580 !
590         BNE DRAW2A
600         LDA DXL
610         BNE DRAW2A
620         LDA #1        If DX=0 then set
                          NEG=1
630         STA NEG
640 !
650 DRAW2A  LDA DY        If DY=0 then set
                          NEG=1
660         BNE DRAW2B
670         LDA #1
680         STA NEG
690 !
700 DRAW2B  LDA DXH       If DX>=DY then
                          SHALLOW else
                          STEEP
710         BNE SHALLOW
720         LDA DXL
730         CMP DY
740         BCS SHALLOW
750         JMP STEEP
760 !
770 SHALLOW SEC           Draw shallow lines
                          with DY/DX<=1
780         LDA X2L       If X2<X1 then swap
                          co-ordinates
```

```
 790         SBC XL
 800         LDA X2H
 810         SBC XH
 820         BCS SHAL1
 830         JSR SWAP
 835 !
 840 SHAL1   LDA DY       Find gradient
                          M=DY/DX
 850         STA DIVLO
 860         LDA #0
 870         STA DIVHI
 880         LDA DXL
 890         STA D1
 900         LDA DXH
 910         STA D2
 920         JSR DIVIDE
 930 !
 940         LDA XL       Calculate C
 950         STA MXL      First find X*M
 960         LDA XH
 970         STA MXH
 980         JSR MULT
 990         STA C
1000         LDA P4
1010         STA CH
1020         LDX NEG
1030         BEQ DRAW3    If M +ve then branch
1040         CLC          C=M*X+Y
1050         LDA C
1060         ADC Y
1070         STA C
1080         LDA CH
1090         ADC #0
1100         STA CH
1110         JMP SHLOOP
1120 DRAW3   LDA Y        C=Y-(M*X)
1130         SBC C
1140         STA C
1150         LDA #0
1160         SBC CH
1170         STA CH
```

```
1180 !
1190 SHLOOP LDA XH       Loop from X to X2
1200        STA MXH
1210        LDA XL
1220        STA MXL
1230        JSR MULT     Find M*X
1240 !
1250        LDX NEG
1260        BNE SHL1
1270        CLC          If M +ve then Y =
                         M*X + C
1280        ADC C
1290        STA Y
1300        JMP SHL2
1310 SHL1   SEC          If M -ve then Y = C -
                         M*X
1320        LDA C
1330        SBC P3
1340        STA Y
1350 SHL2   JSR PLOT     Plot the point
1355 !
1360        CLC          Increment X
1370        LDA XL
1380        ADC #1
1390        STA XL
1400        LDA XH
1410        ADC #0
1420        STA XH
1425 !
1430        SEC          See if X=X2
1440        LDA X2L
1450        SBC XL
1460        LDA X2H
1470        SBC XH
1480        BCS SHLOOP If X2>=X then repeat
1490        JMP TIDY     Finish off
1500 !
1510 !
1520 STEEP  SEC          Draw steep lines with
                         DY/DX>1
1530        LDA Y2
```

```
1540         CMP Y
1550         BCS STEEP1
1560         JSR SWAP    If Y2 <Y then swap
                         coordinates
1565 !
1570 STEEP1  LDA DXL     Find gradient DX/DY
1580         STA DIVLO
1590         LDA DXH
1600         STA DIVHI
1610         LDA DY
1620         STA D1
1630         LDA #0
1640         STA D2
1650         JSR DIVIDE
1655 !
1660         LDA Y       Find Y*M
1670         STA MXL
1680         LDA #0
1690         STA MXH
1700         JSR MULT
1710         STA C
1720         LDX NEG
1730         BEQ ST1
1740         CLC         If M -ve then C= M*Y
                         + X
1750         ADC XL
1760         STA C
1770         LDA #0
1780         ADC XH
1790         STA CH
1800         JMP STLOOP
1810 ST1     LDA XL      If M +ve then C = X -
                         M*Y
1820         SBC C
1830         STA C
1840         LDA XH
1850         SBC #0
1860         STA CH
1865 !
1870 STLOOP  LDA Y       Loop from Y to Y2
1880         STA MXL
```

```
1890          LDA #0
1900          STA MXH
1910          JSR MULT     Find M*Y
1920          LDX NEG
1930          BNE STL1     If slope -ve then
                           branch
1940          CLC          X = M*Y+C
1950          ADC C
1960          STA XL
1970          LDA CH
1980          ADC P4
1990          STA XH
2000          JMP STL2
2005 !
2010 STL1     SEC          X = C-M*Y
2020          LDA C
2030          SBC P3
2040          STA XL
2050          LDA CH
2060          SBC P4
2070          STA XH
2075 !
2080 STL2     JSR PLOT     Plot pixel
2085 !
2090          CLC          If Y <=Y2 then repeat
2100          LDA Y
2110          ADC #1
2120          STA Y
2130          CMP Y2
2140          BCC STLOOP
2150          BEQ STLOOP
2155 !
2156 !
2160 TIDY     LDA XTH      Set X and Y to new
                           line end coordinates
2170          STA XH
2180          LDA XTL
2190          STA XL
2200          LDA YT
2210          STA Y
2220          RTS
```

```
2230 !
2235 !
2240 MULT   LDA Q1     Multiply number in MX by gradient in Q1-Q3
2250        STA M1     Copy gradient to M1-M3
2260        LDA Q2
2270        STA M2
2280        LDA Q3
2290        STA M3
2300        LDA #0     Initialise product P to zero
2310        STA P1
2320        STA P2
2330        STA P3
2340        STA P4
2350        STA MX3
2360        STA MX4
2365 !
2370        LDY #24    Set Y to repeat 24 times
2375 !
2380 MULT1  LSR M3     Shift M to the right
2390        ROR M2
2400        ROR M1
2410        BCC MULT2  If nothing in carry then branch
2420        CLC        Add MX to P
2430        LDA MXL
2440        ADC P1
2450        STA P1
2460        LDA MXH
2470        ADC P2
2480        STA P2
2490        LDA MX3
2500        ADC P3
2510        STA P3
2520        LDA MX4
2530        ADC P4
2540        STA P4
```

```
2545 !
2550 MULT2  ASL MXL     Multiply MX by 2
2560        ROL MXH
2570        ROL MX3
2580        ROL MX4
2585 !
2590        DEY         Repeat if Y <0
2600        BNE MULT1
2605 !
2610        LDA P3      Store result in
                        accumulator
2620        RTS
2630 !
2635 !
2640 DIVIDE LDA #0      Divides DIV by D
2650        STA Q3      First set result Q to
                        zero
2660        STA Q2
2670        STA Q1
2680        STA DIV3
2685 !
2690        LDY #24     Set to repeat 24 times
2695 !
2700        SEC         Subtract D from high
                        bytes of DIV
2710        LDA DIVHI
2720        SBC D1
2730        STA DIVHI
2740        LDA DIV3
2750        SBC D2
2760        STA DIV3
2765 !
2770 DVD1   PHP         Save P register for
                        later
2775 !
2780        ROL Q1      Rotate Q: mult by 2
                        and add carry
2790        ROL Q2
2800        ROL Q3
2810        ASL DIVLO   Multiply Div by 2
2820        ROL DIVHI
```

```
2830            ROL DIV3
2835 !
2840            PLP          Reload P register
2850            BCC DVD2     If DIV-D was <0 then
                             branch
2855 !
2860            LDA DIVHI    Subtract D from new
                             value of DIV
2870            SBC D1
2880            STA DIVHI
2890            LDA DIV3
2900            SBC D2
2910            STA DIV3
2920            CLV
2930            BVC DVD3     Jump on to DVD3
2935 !
2940 DVD2       LDA DIVHI    Add D to new value of
                             DIV
2950            ADC D1
2960            STA DIVHI
2970            LDA DIV3
2980            ADC D2
2990            STA DIV3
2885 !
3000 DVD3       DEY
3010            BNE DVD1     Repeat loop 24 times
3015 !
3020            ROL Q1       Multiply result by 2
3030            ROL Q2
3040            ROL Q3
3050            RTS
3060 !
3070 !
3080 SWAP LDA X2L            Swap (X,Y) with
                             (X2,Y2)
3090            LDY XL
3100            STA XL
3110            STY X2L
3120            LDA X2H
3130            LDY XH
3140            STA XH
```

```
3150        STY X2H
3160        LDA Y2
3170        LDY Y
3180        STA Y
3190        STY Y2
3200        RTS
3210 !
3220 ! Label definitions
3225 !
3230 PLOT   = $C0B8
3240 XL     = 679
3250 XH     = 680
3260 Y      = 681
3270 X2L    = 682
3280 X2H    = 683
3290 Y2     = 684
3300 XTL    NOP
3310 XTH    NOP
3320 YT     NOP
3330 DXL    NOP
3340 DXH    NOP
3350 DY     NOP
3360 D1     NOP
3370 D2     NOP
3380 DIVLO  NOP
3390 DIVHI  NOP
3400 DIV3   NOP
3410 MXL    NOP
3420 MXH    NOP
3430 MX3    NOP
3440 MX4    NOP
3450 C      NOP
3460 CH     NOP
3470 Q1     NOP
3480 Q2     NOP
3490 Q3     NOP
3500 M1     NOP
3510 M2     NOP
3520 M3     NOP
3530 P1     NOP
3540 P2     NOP
```

```
3550 P3      NOP
3560 P4      NOP
3570 NEG     NOP
3580 END
```

To use the program to draw from (X,Y) to (X2,Y2), **POKE** the low byte of X into 679, the high byte of X into 680, and Y into 681. Similarly **POKE** X2 into 682 and 683, and Y2 into 684. Set the colour in which the line is to be drawn by **POKE**ing a suitable number into 685, in the same way as with the PLOT routine. The program works equally well in either screen mode. The program begins at location 49608, and so is run by:

```
SYS 49608
```

Like the PLOT routine, this program does not include any checks to ensure that the lines do not run off the screen. You will find that if a line runs off one side of the screen it will reappear at the other, and it is unlikely that any memory other than the display memory will be corrupted. You should however include suitable checks in your BASIC programs to avoid spoiling the displays.

The program works by first checking the gradient of the line. If the line is shallow (the gradient is less than or equal to 1), the line is drawn by the SHALLOW routine which scans from X to X2, finding Y for each value of X by the formula Y=M*X+C. If the line is steep, the STEEP routine is used to draw the line by scanning from Y to Y2 and finding X by each point with the formula X=M*Y+C.

The subroutine DIVIDE is used to calculate the gradient. The number stored in DIV (DIVLO, DIVHI and DIV3) is divided by the number in D (D1 and D2), and the result (the quotient) is stored in Q (Q1 to Q3). The lower two bytes of the result are the fractional part of the gradient: as the program is designed always to draw lines with a gradient less than 1 the fractional part is very important.

The MULT subroutine multiplies the gradient stored in Q by the value of X or Y stored in MX (MXLO, MXHI, MX3 and MX4) and stores the product in P (P1 to P4). Again the lower two bytes of the result are the fractional part, so the X or Y coordinate to be plotted is found in P3, with the highest bit in P4 if the product represents the X coordinate of a steep line.

FILLING BLOCKS

The fourth machine code program in the graphics set is a FILL routine, to plot all the pixels in a given rectangle in one colour. The routine is very short, the machine code equivalent of a BASIC program such as:

```
10    FOR X = X1 TO X2
20    POKE 679, X AND 255: POKE 680,
      X/256
30    FOR Y = Y1 TO Y2
40    POKE 681, Y
50    SYS 49336
60    NEXT Y
70    NEXT X
```

The machine code routine is:

```
10 ! ************
20 ! *** FILL ***
30 ! ************
40 !
50 !
60 * = $C4E2             Starts at50402
70 !
```

```
 90 !
100 !
110 XL      = 679
120 XH      = 680
130 Y       = 681
140 !
150 X2L     = 682
160 X2H     = 683
170 Y2      = 684
180 !
190 PLOT    = $C0B8
200 !
210 !
230 !
240 FILL    LDA Y       Save Y to reset for
                        each loop
250         STA YT
260 !
280 FILL1   JSR PLOT    Plot point
290         CLC
300         LDA Y       Increment Y
310         ADC #1
320         STA Y
330 !
340         LDA Y2
350         CMP Y       Repeat loop
360         BCS FILL1   until Y > Y2
370 !
380         LDA YT      Reset Y
390         STA Y
400         CLC
410         LDA XL      Increment X
420         ADC #1
430         STA XL
440         LDA XH
450         ADC #0
460         STA XH
470 !
480         SEC         Find X2-X
490         LDA X2H
500         SBC XH
```

```
510          LDA X2L
520          SBC XL
530          BCS FILL1   Repeat loop if
                         X<=X2
540          RTS
550 !
560 !
570 YT       NOP
```

The FILL routine uses the same locations as the Draw routine. **POKE** the X co-ordinate of the top left-hand corner of the block to be filled into locations 679 and 680, and the Y co-ordinate into location 681. The co-ordinates of the bottom right-hand corner of the block should be **POKEd** into location 682, 683 and 684. Location 685 is again used to select the colour.

Using the Graphics Routines

To finish off, here are some short programs which show the use of all the machine code routines we have introduced in the last two chapters. Before running them you must load the graphics machine code into the 64 - see Appendix 18 for details of how to do this. The graphics routines are split into four separate BASIC loader programs which should each be run in accordance with the instructions in Appendix 18. In addition to the four graphics routines, the first program below requires the mixed mode machine code described in Chapter 21 to run – a loader program for this is also given in Appendix 18.

Graphics Demonstration Program

The first program shows the routines in use and also uses the mixed mode display routine described in Chapter 21.

```
100   REM **********************
```

```
110   REM *** GRAPHICS DEMO ***
120   REM *********************
130   REM
140   REM GRAPHICS PACKAGE MUST BE
      LOADED
150   REM BEFORE THIS PROGRAM IS RUN
160   REM
170   REM
1000  REM SET MULTICOLOUR MODE
1010  POKE 53281,15:POKE 686,2:POKE
      687,6:POKE 688,0
1020  SYS 49159
1030  REM SET SPLIT SCREEN
1040  POKE 50518,225:SYS 50468
1050  PRINT"{CLS}{23 * CD}"
1060  PRINT"   {RVS}MULTI COLOUR BIT
      MAPPED DISPLAY{ROF}"
1070  FOR T=1 TO 1000: NEXT T
2000  REM PLOT POINTS
2005  PRINT"{CLS}{23 * CD}"
2010  PRINT"        {RVS}PLOTTING
      POINTS "
2020  FOR A=0 TO 2*π STEP 0.1
2030  X=100+50*SIN(A)
2040  Y=60-50*COS(A)
2050  GOSUB 2500
2060  NEXT A
2090  FOR T = 1 TO 1000: NEXT T
2100  GOTO 3000
2500  POKE 679,X
2510  POKE 681,Y
2520  POKE 685,1
2530  SYS 49336
2540  RETURN
3000  REM DRAW LINES
3010  PRINT"{CLS}{23 * CD}"
3020  PRINT"        {RVS}DRAWING
      LINES"
3030  X1=10:Y2=150: POKE 685,2
3040  FOR Y1=10 TO 150 STEP 5
3050  X2=Y1
```

```
3060  POKE 679,X1:POKE 681,Y1:POKE
      682,X2:POKE 684,Y2
3070  SYS 49608
3080  NEXT Y1
3090  FOR T = 1 TO 1000: NEXT T
4000  REM FILL BLOCK
4010  PRINT"{CLS}{23 * CD}"
4020  PRINT"        {RVS}FILLING
      BLOCKS"
4030  POKE 685,1
4040  POKE 679,70: POKE681,30
4050  POKE 682,130: POKE684,90
4060  SYS 50402
4070  FOR T = 1 TO 1000: NEXT:END
```

The program will end with the computer in mixed display mode. Use RUN/STOP and RESTORE to return to normal text mode.

Plotting Points

This program uses the PLOT routine to draw a pattern on the screen.

```
10    REM PLOT PATTERN
20    POKE 685,1:POKE 686,2:POKE
      687,7
30    SYS 49152:REM SET UP BMM
35    J=0:K=80
40    FOR N=0 TO 2*Π STEP .05
50    X = 160+J*SIN(N)
60    Y = 95+K*COS(N)
70    POKE 679,X:POKE 681,Y
80    SYS 49336:REM PLOT IT!
90    NEXT N
100   J = J+10:K = K-10
110   IF K >-10 THEN 40
120   GET K$: IF K$="" THEN 120
130   SYS 49266:REM BACK TO TEXT
```

Headache!

This program uses the DRAW routine to create a pattern on the multi-colour bit-mapped mode screen and demonstrates the effect of changing the colours using the COLOUR routine.

```
4      H=180:W=140:XC=80:YC=100:S=10
5      POKE1020,0:POKE1021,96
6      POKE685,1
10     POKE 686,2:POKE 687,1:POKE
       688,6:POKE 53281,12: SYS 49159
15     FOR Y=YC-H/2 TO YC+H/2 STEP S
20     POKE 679,(XC-W/2)AND 255:POKE
       680,(XC-W/2)/256:POKE 681,Y
30     POKE 682,XC AND 255:POKE 683,
       XC/256:POKE 684,YC
40     SYS 49608:POKE 685,(PEEK(685)
       +1-(PEEK(685)=3))AND 3
45     NEXT
50     FOR X=XC-W/2+S TO XC+W/2 STEP
       S
60     POKE 679,X AND 255:POKE 680,
       X/256:POKE 681,YC+H/2
70     POKE 682,XC AND 255:POKE 683,
       XC/256:POKE 684,YC
80     SYS 49608:POKE 685,(PEEK(685)+
       1-(PEEK(685)=3))AND 3
90     NEXT
115    FOR Y=YC+H/2-S TO YC-H/2 STEP-
       S
120    POKE 679,(XC+W/2) AND 255:POKE
       680,(XC+W/2)/256:POKE 681,Y
130    POKE 682,XC AND 255:POKE 683,
       XC/256:POKE 684,YC
140    SYS 49608:POKE 685,(PEEK(685)+
       1-(PEEK(685)=3))AND 3
145    NEXT
150    FOR X=XC+W/2-S TO XC-W/2 STEP-
       S
```

```
160   POKE 679,X AND 255:POKE 680,
      X/256:POKE 681,YC-H/2
170   POKE 682,XC AND 255:POKE 683,
      XC/256:POKE 684,YC
180   SYS 49608:POKE 685,(PEEK(685)+
      1-(PEEK(685)=3))AND 3
185   NEXT
190   FOR CC=1 TO 96
191   POKE 53280,(PEEK(53280)+1) AND
      15
192   POKE 686,(PEEK(686)+1) AND 15
193   POKE 687,(PEEK(687)+1) AND 15
194   POKE 688,(PEEK(688)+1)AND 15
195   POKE 53281,(PEEK(53281)+1) AND
      15
196   SYS 49177
197   FOR T=1 TO 200:NEXT:NEXT
200   GET A$:IFA$="" THEN 200
210   SYS 49266
220   END
```

Lace

This program uses the DRAW routine to create a lace pattern on the standard bit-mapped screen.

```
10    DIM A(36), B(36)
20    POKE 686,6: POKE 687,3 : SYS
      49152
30    L=120: J=80
40    FOR H=1 TO 5
50    FOR N=1 TO 36
60    K = N/18*π
70    A(N)= 128+L*SIN(K):
      B(N)=88+J*COS(K)
100   NEXT N
110   FOR N=1 TO 36
120   M = N+12
130   IF M>36 THEN M=M-36
160   POKE 679,A(N) AND 255: POKE
      680,A(N)/256: POKE 681,B(N)
```

```
170   POKE 682,A(M) AND 255: POKE
      683,A(M)/256: POKE684,B(M):
      POKE 685,1
180   SYS 49608
190   NEXT N
200   L=L/2: J=J/2
210   NEXT H
500   GET K$: IF K$=""THEN 500
510   SYS 49266:END
```

J R's Hat

This program uses the DRAW routine to draw a three dimensional graph – it takes quite a while to run.

```
100   DIM UB(424), LB(424)
110   XC=320:YC=115:XR=175:ZR= 120
120   H=40:W=0.043:XA=107
200   FOR S=1 TO 424
210   UB(S)=0:LB(S)=1000
220   NEXT S
300   POKE 686,2: POKE 687,1:SYS
      49152
500   FOR Z=-ZR+1 TO ZR-1 STEP 5
510   XL=INT(XR*SQR(1-(Z*Z)/
      (ZR*ZR))+.5)
520   X=-XL
530   Y=H*SIN(W*SQR(X*X+Z*Z))
540   X1=X+XC+Z
550   Y1=INT(199-(YC+Y+Z/2)+.5)
600   FOR X=-XL+1 TO XL-1
610   Y=H*SIN(W*SQR(X*X+Z*Z))
620   X2=XC+X+Z
630   Y2=INT(199-(YC+Y+Z/2)+.5)
640   IF Y2>=LB(X2-XA) THEN 680
650   LB(X2-XA)=Y2
660   IF UB(X2-XA)=0 THEN UB(X2-XA)
      = Y2
670   GOTO 700
680   IF Y2<=UB(X2-XA) THEN 730
```

```
690   UB(X2-XA)=Y2
700   POKE 679,(X1/2) AND 255:POKE
      680,(X1/2)/256:POKE 681,Y1
710   POKE 682,(X2/2) AND 255:POKE
      683,(X2/2)/256:POKE 684,Y2
720   POKE 685,1:SYS 49608
730   X1=X2
740   Y1=Y2
750   NEXT X
760   NEXT Z
1000  GET K$:IF K$="" THEN 1000
1010  SYS 49266
```

CHAPTER 21

DISPLAY INTERRUPTS

The VIC-II chip is a very powerful device and allows many different graphics displays to be created. It is possible to extend these capabilities to enable the creation of displays using several different character sets, or more than eight sprites for example - you can even have mixed text and graphics.

This chapter deals with the techniques used in creating such displays, but to understand fully you must know how the VIC-II chip generates its displays, and how a TV or monitor works.

TV Pictures

The pictures on a TV screen are created by an electron beam which is directed at the phosphor coated inner surface of the screen. Where this beam strikes the screen the phosphor glows. To create a full picture the electron beam scans across the screen in rows, varying in intensity as it goes. This variation in intensity is dictated by information from the TV transmitter and produces a corresponding variation in the intensity with which the phosphor glows. When the electron beam reaches the edge of the screen it is turned off and the process starts again from a position just below the last starting position. This process, called *raster scanning*, continues until the bottom of the screen is reached, at which point it starts all over again at the top. The process must happen many times a second to create a picture that doesn't flicker.

To create a colour picture, the screen is coated with three different types of phosphor which emit the colours red, blue and green when struck by the electron beam. The different phosphors are distributed in tiny dots or blocks over the screen and the colour TV signal must contain information about which of these points the electron beam should strike, as well as luminance information.

In generating its displays the VIC-II chip takes data from video memory and uses it to create a signal which controls the electron beam in the TV in much the same way as a transmitted TV signal does.

The Raster Register

One of the VIC-II registers, the *raster register* (location 53266), is concerned with the current raster position (position down the screen) of the scaning electron beam. It has two functions depending on whether data is written to it or read from it.

If you read the raster register the number returned is the bottom eight bits of the current raster position, as shown in this program:

```
10    PRINT"{CLS}";PEEK(53266):
      GOTO 10
```

Obviously this number is changing very rapidly and incidentally is a good source of random numbers. Since the raster position can be greater than 255, a ninth bit is required and this is Bit 7 of the control register at location 53265.

If data is written to the raster register, it is stored within the VIC chip and used in a raster compare operation – the current raster position is compared with the stored value, and when the two are equal this fact is indicated in the *interrupt register*.

The Interrupt Status Register

When the current raster position equals the stored value, Bit 0 of the interrupt status register is set (Bit 7 is also set for any VIC interrupt). Bit 0 will remain set until you clear it by writing a 1 to that bit.

So far we have seen how we can get an indication of when the scanning electron beam reaches a given point on the screen, but to use this knowledge we must make use of another VIC register – the *interrupt enable register*.

The Interrupt Enable Register

If Bit 0 of this register is set when a raster interrupt is indicated in the interrupt status register, an interrupt will be generated and the 6510 will begin to execute a machine code program whose start address is stored in locations $FFFE and $FFFF (65534 and 65535). Before returning to whatever program was being executed at the time of the interrupt, a program whose start address is stored in locations 788 and 789 is run. If we change the contents of 788 and 789 (the Interrupt ReQuest vectors) to point to a routine of our own, we can alter the display midway through a scan, and so create the effects mentioned at the beginning of the chapter.

To illustrate the technique, here is a short machine code program which changes the background colour of the display half way down the screen. If you don't have an assembler, type in the BASIC loader program which follows the assembly language listing.

```
100     IRQLO=788         IRQ vectors
110     IRQHI=789
120     RASREG=53266      Raster register
130     IENREG=53274      Interrupt enable
140     SCREEN=53281      Screen colour reg
150     INTREG=53273      Interrupt status
160     IRQVEC=59953      default IRQ vector
170     !
180     SETUP SEI         disable interrupts
190     LDA #<PROG        load IRQ vectors with
200     STA IRQLO         address of new program
210     LDA #>PROG
220     STA IRQHI
230     LDA #1            set bit 0 of interrupt enable
240     STA IENREG         register
242     LDA #0            initialise raster register
244     STA RASREG
245     LDA 53265         including bit 8!
246     AND #127
247     STA 53265
250     CLI               enable interrupts
260     RTS               back to BASIC
270   !
280   ! if we get here an interrupt has occurred.
290   !
300   !
PROG    LDA INTREG        examine interrupt register
350     AND #1            for bit 0=1?
360     BEQ NORMAL        if not, exit
370     STA INTREG        raster interrupt
380     LDA RASREG        - clear RASREG
382     BNE RR            if< >0 then branch
384     LDA #145          next interrupt is half way
386     STA RASREG        down the screen
388     STA SCREEN        change screen colour
390     JMP RR2           exit
392     RR LDA #0         next interrupt is at the top
394     STA RASREG        of the screen
```

```
396     STA SCREEN      change colour
420     RR2 PLA         restore values in registers
421     TAY             before interrupt
422     PLA
423     TAX
424     PLA
425     RTI             end
NORMAL JMP  IRQVEC      handle other  interrupts
```

Here is the BASIC loader for the program, which loads the code into the cassette buffer, runs it and clears itself from memory.

```
5      REM BASIC LOADER FOR INTERRUPT
       DEMO
10     FOR Z=828 TO 901
20     READ D
30     POKE Z,D
40     NEXT Z
50     SYS 828:NEW
20000 DATA 120,169,91,141,20,3,169,
       3,141,21,3,169,1,141,26,208
20010 DATA 169,0,141,18,208,173,17,
       208,41,127,141,17,208,88,96,17
       3
20020 DATA 25,208,41,1,240,33,141,
       25,208,173,18,208,208,11,169,1
       45
20030 DATA 141,18,208,141,33,208,76,
       125,3,169,0,141,18,208,141,33
20040 DATA 208,104,168,104,170,104,
       64,76,49,234
```

How the Program works

Lines 180 to 260 load the IRQ vectors with the start address of the new routine and initialise the registers. Notice that the I flag is set to prevent any interrupts occurring while the program is running (and cleared afterwards!).

The interrupt program itself starts at line 340 by checking for Bit 0 of the interrupt register being set. If it is not then the interrupt wasn't generated by a raster compare routine and control is passed to the normal interrupt routines by line 440.

If a raster interrupt has occurred, Bit 0 of the interrupt register is cleared and the contents of the raster register are examined.

If the interrupt occurred at the top of the screen then the raster register will contain zero. In this case the raster register is loaded with 145 (corresponding to a position half way down the screen - the position where we want the next interrupt to occur) and the screen colour is changed to white.

If the interrupt was at the middle of the screen (i.e. the raster register contained 145), the raster register is cleared so that the next interrupt happens at the top of the screen, and the screen colour is set to 0 which is black.

Before returning from the routine, the accumulator, X and Y registers are retrieved from the stack where they were placed by the 64's interrupt handling routines.

If you run the program you will see that the top half of the screen will be white while the bottom is black. Once running it will have no effect on your BASIC programs, and can be disabled by pressing RUN/STOP and RESTORE.

You will notice that the screen flickers slightly and that the rate of flicker increases when you press a key. This is because each time you press a key an interrupt is generated, calling the new interrupt routine before control is passed to the 64's routine at 59953. If the running of this routine coincides with the processing of a key press then a raster interrupt will be 'missed' and the screen will flicker. This effect is particularly noticeable with this example, but can be minimised as you will see in later examples.

As we mentioned earlier raster interrupts can be used to create effects not possible by any other means. For example you could display text in different fonts on different parts of the screen by having several character sets in RAM and arranging your interrupt routine to change the character set pointer when a raster interrupt occurs. This technique is used in the next program to display 8 standard sprites and 8 multicolour sprites on the screen at the same time!

An Abundance of Sprites!

```
10          ;**************
15          ;*            *
20          ;* 16 sprites *
25          ;*            *
30          ;**************
40          ;
50          POINTER=2040
60          YPOS=53249
70          SMCREG=53276
100         IRQLO=788
110         IRQHI=789
120         RASREG=53266
130         IENREG=53274
150         INTREG=53273
160         IRQVEC=59953
170         *=$C000
175 ;
180 SET     SEI
190         LDA #<PROG
200         STA IRQLO
210         LDA #>PROG
220         STA IRQHI
```

```
230      LDA #1
240      STA IENREG
242      LDA #0
244      STA RASREG
245      LDA 53265
246      AND #127
247      STA 53265
250      CLI
260      RTS
270 !
300 PROG LDA INTREG
310      AND #1
320      BEQ NOR
330      STA INTREG
340      LDA RASREG
350      BNE MULT
360      LDA #145
370      STA RASREG
380      LDX #7
390 STD  LDA #14
400      STA POINTER,X
410      DEX
420      BPL STD
421      LDX #14
422      LDA #70
424 Y1   STA YPOS,X
425      DEX
426      DEX
427      BPL Y1
430      LDA #0
440      STA SMCREG
450      JMP EXIT
500 MULT LDA #0
510      STA RASREG
520      LDX #7
540      LDA #13
550 LOOP STA POINTER,X
570      DEX
580      BPL LOOP
581      LDA #200
582      LDX #14
```

```
584 Y2   STA YPOS,X
585      DEX
586      DEX
587      BPL Y2
590      LDA #255
600      STA SMCREG
610 EXIT PLA
620      TAY
630      PLA
640      TAX
650      PLA
660      RTI
670 NOR  JMP IRQVEC
```

The next program is a BASIC loader for the machine code.

NOTE: The code occupies the same area of memory as the graphics routines, which will be overwritten.

```
10    REM LOADER FOR 16 SPRITE DEMO
20    FOR Z=49152 TO 49271
30    READ D:POKE Z,D
40    NEXT Z
50    REM
20000 DATA 120,169,31,141,20,3,169,
      192,141,21,3,169,1,141,26,208
20010 DATA 169,0,141,18,208,173,
      17,208,41,127,141,17,208,88,96
      ,173
20020 DATA 25,208,41,1,240,79,
      141,25,208,173,18,208,208,34,
      169,145
20030 DATA 141,18,208,162,7,169,
      14,157,248,7,202,16,248,162,14
      ,169
20040 DATA 70,157,1,208,202,202,16,
      249,169,0,141,28,208,76,111,
      192
```

```
20050 DATA 169,0,141,18,208,162,7,
      169,13,157,248,7,202,16,250,
      169
20060 DATA 200,162,14,157,1,208,202,
      202,16,249,169,255,141,28,208,
      104
20070 DATA 168,104,170,104,64,76,
      49,234
```

You will see that the program is very similar to the last one, the only difference is that when the scan is half-way down the screen the sprite Y position registers are changed to 200, the sprite data pointers are switched to use another set of data and multicolour mode is selected. To see the effect of the interrupt program, enter it followed by this BASIC program:

```
1     REM 16 SPRITES ON ONE SCREEN
      DEMO
5     POKE 53281,0
10    FOR I=0 TO 127:READ D
20    POKE 832+I,D:NEXT
30    FOR I=2040 TO 2047
40    POKE I,14:NEXT
60    FOR I=0 TO 14 STEP 2
70    POKE 53248+I,25+(15*I)
75    POKE 53248+I+1,70
80    NEXT I
90    FOR I=0 TO 7
100   POKE 53287+I,25+I
110   NEXT:POKE 53294,2
120   POKE 53285,1
130   POKE 53286,2
140   POKE 53269,255
150   PRINT"{CLS}{YEL}HERE ARE 8
      STANDARD SPRITES"
160   FOR DE=0 TO 1000:NEXT
170   SYS 49152
```

```
180   PRINT"{15 * CD}HERE ARE 8
      M{WHT}U{RED}L{CYN}T{PUR}I{GRN}
      C{BLU}O{YEL}L{ORG}O{BRN}U{YEL}
      R SPRITES"
190   PRINT"{CD}{CD}{CD}{CD}{CD}
      {CD}{WHT}IT'S ALL DONE WITH
      INTERRUPTS!!"
200   GOTO 200
998   REM
999   REM DATA FOR MULTI COLOR
      SPRITES
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
1120  REM
1999  REM DATA FOR STANDARD SPRITES
2000  DATA 24,24,24,24,60,24,24,126
2010  DATA 24,12,255,48,15,255,240,3
2020  DATA 255,192,7,255,224,15,255
2030  DATA 40,24,126,24,48,126,12,96
2040  DATA 126,6,255,255,255,255,255
2050  DATA 255,255,255,255,255,255,
      255
2060  DATA 127,255,254,56,60,28,28,
      24
2070  DATA 56,14,0,112,28,0,56,56,0,
      28,0
```

You could use this method to get even more sprites on one display, but you would need a machine code program to move them around satisfactorily.

MIXED MODE DISPLAYS

One of the disadvantages of the 64's display compared with some other micros is that you cannot display text at the same time as high resolution graphics. Raster interrupts can be put to use here by switching between bit-mapped mode and standard text mode at the appropriate place on the screen. That is how the split screen in the program demonstrating the graphics routines in Chapter 20 was created.

Here is a listing of the split screen interrupt program and a BASIC loader:

```
100       IRQLO = 788
110       IRQHI = 789
120      RASREG = 53266
130      IENREG = 53274
140      INTREG = 53273
150      IRQVEC = 59953
160       LORES = 49266
170       HIRES = 49221
200 * = $C512
210 SETUP SEI
220         LDA #<PROG
225         STA IRQLO
230         LDA #>PROG
240         STA IRQHI
250         LDA #1
255         STA IENREG
260         LDA #0
265         STA RASREG
270         LDA 53265
280         AND #127
290         STA 53265
300         CLI
310         RTS
500 PROG    LDA INTREG
510         AND #1
```

```
520        BEQ NORM
530        STA INTREG
540        LDA RASREG
550        BNE TEXT
560        JSR HIRES
565        LDA #200
566        STA RASREG
570        JMP EXIT
600 TEXT   JSR LORES
610        LDA #0
620        STA RASREG
700 EXIT   PLA
710        TAY
720        PLA
730        TAX
740        PLA
750        RTI
800 NORM   JMP IRQVEC
```

The routine is loaded by the following BASIC program and is located immediately after the graphics routines, which obviously must be in situ for it to work.

```
10    REM MIX MODE LOADER
20    FOR Z=50468 TO 50541
30    READ D:POKE Z,D
40    NEXT Z
20000 DATA 120,169,67,141,20,3,169,
      197,141,21,3,169,1,141,26,208
20010 DATA 169,0,141,18,208,173,17,
      208,41,127,141,17,208,88,96,
      173
20020 DATA 25,208,41,1,240,33,141,
      25,208,173,18,208,208,11,32,69
20030 DATA 192,169,200,141,18,208,
      76,83,197,32,114,192,169,0,141
      ,18
```

```
20040 DATA 208,104,168,104,170,104,
      64,76,49,234
```

To initialise the routine type:

```
SYS 50450
```

The display will now comprise a graphics screen with a six line text window at the foot of the screen.

We have mentioned just a few of the ways raster interrupts can be used to create more interesting displays – it's up to you to adapt the principles to suit your own applications.

OTHER INTERRUPTS

In addition to setting Bit 0 when the current raster position equals the stored raster position, the interrupt status register provides an indication of sprite collisions.

Bit 1 is set when a sprite to data collision occurs and will remain set until cleared. If the corresponding bit in the interrupt enable register is set, an interrupt will be generated. Similarly, Bit 2 will be set when a sprite to sprite collision occurs.

No information is provided as to which sprite(s) are involved – that must be done by your program. But using interrupts means your program does not have to be constantly checking the sprite collision registers so it will run more quickly.

Interrupts are used in many ways by computers – another example is in the program in Chapter 22, where movement of the joystick generates an interrupt. This allows very smooth motion of the cursor, and means that other programs can run at the same time.

CHAPTER 22

PROGRAMS AND PEOPLE

Writing programs to be used by other people is a very different art from the writing of programs which only you will use. Most people who use programs written by others are not skilled in the use of computers, and may not be familiar with the typewriter keyboard. In this chapter we will look at a few ideas for making programs easier to use – making them 'user friendly' as the jargon puts it.

USER FRIENDLINESS

There are two sides to making programs user-friendly. They must allow the user to select options easily and quickly, and they must also tell the user what options are available at all times. A program may have supporting documentation, but the screen display should present enough information for the program to be usable without referring to the notes once the basics have been learnt.

Menus

One of the simplest and most effective ways of assisting the user to control the program is the *menu*. This is a list of the options available, with some means of indicating how to select an option. The simplest form presents the list with a key legend next to each item, indicating that the key should be pressed to select that feature. The function keys of the 64 work well with this type of menu; other possibilities are to use the initial letter of each option as the selecting key, or to use numbers.

A second form of menu displays the list as before, but indicates the selected item by displaying it in a different colour, or in inverse video. One or two keys are then used to move the selection up and down the list, and a third key acts as the "go" button, indicating that the selected item is to be acted on. The cursor keys and the RETURN key are probaby best in this application.

Mice and Pointers

The typewriter keyboard is far from the ideal device for communicating with a computer for the newcomer. Many people have no experience of using a keyboard, and find using programs very difficult if they are constantly searching for the right key to press. In addition, the keyboard is not always the best way of controlling a computer, even for an expert. To get round these problems, there are a number of different devices now being used to allow the operator to make inputs by pointing at symbols on the screen.

Perhaps the best known of these devices at the moment is the *mouse*, a small box with a ball-bearing underneath which moves a cursor around the screen as the mouse is rolled about on the desk. The mouse usually has one or two buttons on the top which are used to make selections when the cursor reaches the right point on the screen.

Other pointing devices which perform a similar function are the *light-pen*, which detects the light emitted by the screen and so may be used to point directly at the screen, and the *tracker ball*, which rolls a cursor round the screen. The latest innovation is the touch sensitive screen, which allows the humble human finger to select objects on the display.

The type of display used with these devices may be very different from that used in ordinary textual menus. If the user is making selections by pointing at the screen, the display describing what is available need not use text, but can use small pictorial symbols to indicate options. These symbols are called *icons*, and are used in several of the most advanced business micros now on the market.

SKETCHPAD PROGRAM

This program demonstrates the use of icons on the 64, using a joystick to draw pictures in multicolour bit-mapped mode.

```
90    PRINT "{CLS}{WHT}{CD}" TAB(12)
      "{RVS} 64 SKETCH PAD {ROF}":
      PRINT: PRINT: PRINT
100   C0=6:C1=10:C2=1:C3=0:IC=1
105   POKE 53281,C0: POKE 686,C1:
      POKE 687,C2: POKE 688,C3
110   FB=2
120   REM READ SPRITE DATA
130   FOR I=0 TO 383: READ X: POKE
      23168+I,X: NEXT
140   REM READ MACHINE CODE DATA
150   FOR I=0 TO 202:READ X: POKE
      50542+I,X: NEXT
160   REM SET SPRITE Y COORDS
170   FOR I=53251 TO 53263 STEP 2:
      POKE I,225: NEXT
180   FOR I=0 TO 12 STEP 2: POKE
      53250+I,(24+24*I) AND 255:
      NEXT
990   REM MAIN MENU
```

```
1000  PRINT "{CLS}{WHT}{CD}" TAB(12)
      "{RVS} 64 SKETCH PAD {ROF}":
      PRINT: PRINT: PRINT
1010  PRINT TAB(5) "SHOW CURRENT
      PICTURE   ... {RVS} F1 {ROF}":
      PRINT: PRINT
1020  PRINT TAB(5) "NEW PICTURE
             ... {LTRED}{RVS} F2
      {ROF}{WHT}": PRINT: PRINT
1030  PRINT TAB(5) "LOAD PICTURE
      FROM DISK ... {RVS} F3 {ROF}":
      PRINT: PRINT
1040  PRINT TAB(5) "SAVE PICTURE TO
      DISK   ... {RVS} F5 {ROF}":
      PRINT: PRINT
1050  PRINT TAB(5) "CHANGE COLOURS
             ... {RVS} F7 {ROF}":
      PRINT: PRINT
1060  PRINT TAB(5) "STOP PROGRAM
             ... {LTRED}{RVS} F8
      {ROF}{WHT}"
1100  GET K$: IF K$="" THEN 1100
1110  IF K$="{F1}" THEN SYS 49177:
      GOSUB 2010: GOTO 1000
1120  IF K$="{F2}" THEN GOSUB 2000:
      GOTO 1000
1130  IF K$="{F3}" THEN GOSUB 5000:
      SYS 49177: GOSUB 2010: GOTO
      1000
1140  IF K$="{F5}" THEN GOSUB 6000:
      GOTO 1000
1150  IF K$="{F7}" THEN GOSUB 7000:
      GOTO 1000
1160  IF K$="{F8}" THEN END
1170  GOTO 1100
1990  REM BIT-MAPPED DRAWING
2000  SYS 49159: REM TURN ON MC BIT
      MAP DISPLAY
2010  REM NOW SET SPRITE DATA
      POINTERS
2020  POKE 24568,106:POKE 24569,108
```

```
2030  FOR I=24570 TO 24572: POKE I,
      107: NEXT
2040  POKE 24573,109: POKE
      24574,110: POKE 24575,111
2045  POKE 53248,172: POKE 53249,120
2050  POKE 53288,IC: POKE 53289,C1:
      POKE 53290,C2: POKE 53291, C3
2055  POKE 53292,IC: POKE 53293,IC:
      POKE 53294,IC:POKE 53287,IC
2060  POKE 53269, 255: REM TURN
      SPRITES ON
2065  POKE 53264,192: REM SPRITE X
      MSB
2070  SYS 50542: POKE690,1: REM TURN
      ON CURSOR MACHINE CODE
2075  C=PEEK(53278)
2080  POKE 53271,0: REM SPRITE Y
      EXPANSION
2090  POKE 53277,2↑(FB+1): REM
      SPRITE Y EXPANSION
2095  POKE 685,FB: POKE 53281,C0
2099  REM ICON HANDLING
2100  C=PEEK(53278): IF C=0 THEN
      2100
2105  IF (PEEK(56320) AND 16)<>0
      THEN 2100
2110  CC = C AND 254
2120  IF CC=2 THEN FB=0: GOTO 2090
2130  IF CC=4 THEN FB=1: GOTO 2090
2140  IF CC=8 THEN FB=2: GOTO 2090
2150  IF CC=16 THEN FB=3: GOTO 2090
2160  IF CC=32 THEN GOSUB 3000:
      GOTO2090
2170  IF CC=64 THEN GOSUB 4000: GOTO
      2090
2180  IF CC<>128 THEN 2100
2190  REM RETURN TO MENU
2200  POKE 53269,0: REM TURN OFF
      SPRITES
2210  SYS 50732: REM TURN OFF CURSOR
      INTERRUPT PROG
```

```
2220  SYS 49266: REM BACK TO TEXT
      MODE
2230  POKE 53281,6
2240  RETURN
2990  REM DRAW LINE
3000  POKE 690,0:POKE 53277,CC
3005  IF (PEEK(56320)AND16) <> 16
      THEN 3005
3010  IF (PEEK(56320)AND16) <> 0
      THEN 3010
3020  X1=PEEK(691): X2=PEEK(692):
      Y1=PEEK(693)
3030  IF (PEEK(56320)AND16) <> 16
      THEN 3030
3040  IF (PEEK(56320)AND16) <> 0
      THEN 3040
3050  XL=PEEK(691): XH=PEEK(692):
      Y=PEEK(693)
3060  GOSUB 3100
3070  IF(PEEK(56320)AND16) <> 16
      THEN POKE 685,0: GOSUB 3100:
      POKE 685,FB: GOTO 3050
3080  POKE 690,1: RETURN
3100  POKE 682,X1: POKE 683,X2: POKE
      684,Y1
3110  POKE 679,XL: POKE 680,XH: POKE
      681,Y
3120  SYS 49608: RETURN
3990  REM FILL BLOCK
4000  POKE 690,0:POKE 53277,CC
4005  IF (PEEK(56320)AND16) <> 16
      THEN 4005
4010  IF (PEEK(56320)AND16) <>0 THEN
      4010
4020  X1=PEEK(691): X2=PEEK(692):
      Y1=PEEK(693)
4030  IF (PEEK(56320)AND16) <>16
      THEN 4030
4040  IF (PEEK(56320)AND16) <>0 THEN
      4040
```

```
4050  XL=PEEK(691): XH=PEEK(692):
      Y=PEEK(693)
4060  GOSUB 4200:POKE 685,0: GOSUB
      4200: POKE685,FB
4070  IF (PEEK(56320)AND16) <>16
      THEN 4050
4080  GOSUB 4200
4090  IF X1+256*X2>XL+256*XH THEN
      T1=X1: T2=X2: X1=XL: X2=XH:
      XL=T1: XH=T1
4100  IF Y1>Y THEN T=Y: Y=Y1: Y1=T
4110  POKE 679,X1:POKE 680,X2: POKE
      681,Y1
4120  POKE 682,XL: POKE 683,XH: POKE
      684,Y
4130  SYS 50402
4140  POKE 690,1: RETURN
4200  POKE 682,X1: POKE 683,X2: POKE
      684,Y1
4210  POKE 679,XL: POKE 680,XH: POKE
      681,Y1
4220  SYS 49608
4230  POKE 682,XL: POKE 683,XH: POKE
      684,Y1
4240  POKE 679,XL: POKE 680,XH: POKE
      681,Y
4250  SYS 49608
4260  POKE 682,XL: POKE 683,XH: POKE
      684,Y
4270  POKE 679,X1: POKE 680,X2: POKE
      681,Y
4280  SYS 49608
4290  POKE 682,X1: POKE 683,X2: POKE
      684,Y
4300  POKE 679,X1: POKE 680,X2: POKE
      681,Y1
4310  SYS 49608: RETURN
4990  REM LOAD PICTURE
5000  PRINT "{CLS}{WHT}{CD}" TAB(12)
      "{RVS} 64 SKETCH PAD {ROF}":
      PRINT: PRINT: PRINT
```

```
5010  PRINT TAB(5) "{RVS} LOAD
      PICTURE {ROF}": PRINT: PRINT
5020  INPUT "     PICTURE TITLE";F$
5030  OPEN 1,8,2,F$+"S,R"
5040  INPUT#1,C0
5050  INPUT#1,C1: POKE 686,C1
5060  INPUT#1,C2: POKE 687,C2
5070  INPUT#1,C3: POKE 688,C3
5080  INPUT#1,IC
5090  FOR I=24576 TO 32575
5100  GET#1, D$
5110  IF D$="" THEN D$=CHR$(0)
5120  POKE I,ASC(D$)
5130  NEXT
5140  CLOSE 1
5150  RETURN
5990  REM SAVE PICTURE
6000  PRINT "{CLS}{WHT}{CD}" TAB(12)
      "{RVS} 64 SKETCH PAD {ROF}":
      PRINT: PRINT: PRINT
6010  PRINT TAB(5)"{RVS} SAVE
      PICTURE {ROF}":PRINT:PRINT
6020  INPUT "     PICTURE TITLE";F$
6030  OPEN 1,8,2,F$+"S,W"
6040  PRINT#1,C0
6050  PRINT#1,C1
6060  PRINT#1,C2
6070  PRINT#1,C3
6080  PRINT#1,IC
6090  FORI=24576TO32575
6100  PRINT#1,CHR$(PEEK(I));
6110  NEXT
6120  PRINT#1:CLOSE1
6130  RETURN
7000  PRINT "{CLS}{CD}": INPUT
      "BACKGROUND";C0
7010  PRINT: INPUT "COLOUR
      1";C1:POKE 686,C1
7020  PRINT: INPUT "COLOUR
      2";C2:POKE 687,C2
```

```
7030  PRINT: INPUT "COLOUR
      3";C3:POKE 688,C3
7040  PRINT: INPUT "ICON COLOUR";IC
7050  RETURN
19990 REM DATA FOR SPRITES
20000 DATA 0,0,0,0,0,0,0,0,0,0
20010 DATA 24,0,15,255,240,12,24,48,
      12,24,48,12,0,48,12,0,48,12,0,
      48,31,0,248
20020 DATA 12,0,48,12,0,48,12,0,48,
      12,24,48,12,24,48,15,255,240,0
      ,24
20030 DATA 0,0,0,0,0,0,0,0,0,0,0
20100 DATA 0,0,0,31,192,60,127,252,
      254,127,255,254,127,255,254,
      255,255
20110 DATA 255,255,255,255,63,255,
      255,63,255,255,127,255,254,
      127,255,254,127
20120 DATA 255,252,127,255,248,63,
      255,252,31,255,252,31,255,254,
      31,255,255
20130 DATA 15,255,255,3,255,231,0,
      63,224,0,7,192,0,0,0,0,31
20140 DATA 192,60,63,252,254,112,
      127,230,96,7,134,224,0,7,
      240,0,3
20150 DATA 48,0,3,48,0,7,112,0,6,96,
      0,14,96,0,28,112,0
20160 DATA 24,48,0,12,24,0,12,24,0,
      6,30,0,63,15,224,127,3
20170 DATA 252,103,0,63,224,0,7,192,
      0,224,0,0,112,0,0,56,0
20180 DATA 0,28,0,0,14,0,0,7,0,0,3,
      128,0,1,192,0,0
20190 DATA 224,0,0,112,0,0,56,0,0,
      28,0,0,14,0,0,7,0
20200 DATA 0,3,128,0,1,192,0,0,224,
      0,0,112,0,0,56,0,0
20210 DATA 28,0,0,14,0,0,0,0,0,0,0,
      63,255,252,63,255,252
```

```
20220 DATA 63,255,252,63,0,252,63,
      63,252,63,63,252,63,63,
      252,63,3
20230 DATA 252,63,63,252,63,63,252,
      63,63,252,63,63,252,63,
      63,252,63
20240 DATA 63,252,63,255,252,63,255,
      252,63,255,252,0,0,0,0,0,0
20250 DATA 0,0,0,0,0,0,0,63,255,252,
      48,0,12,48,255,12,48
20260 DATA 0,12,48,0,12,51,252,204,
      48,0,12,48,0,12,51,252,204
20270 DATA 48,0,12,48,0,12,51,252,
      204,48,0,12,48,0,12,51,252
20280 DATA 204,48,0,12,63,255,252,
      0,0,0,0,0,0,0
24990 REM DATA FOR MACHINE CODE
20500 DATA 120,169,123,141,20,3,169,
      197,141,21,3,88,96,173,0,
      220,74
20510 DATA 176,12,72,173,1,208,201,
      41,144,3,206,1,208,104,
      74,176,12
20520 DATA 72,173,1,208,201,239,176,
      3,238,1,208,104,74,176,35,
      72,173
20530 DATA 16,208,41,1,208,7,173,0,
      208,201,14,144,19,56,173,0,208
20540 DATA 233,1,141,0,208,176,8,
      173,16,208,41,254,141,16,
      208,104,74
20550 DATA 176,29,72,173,16,208,41,
      1,240,7,173,0,208,201,75,
      176,13
20560 DATA 238,0,208,208,8,173,16,
      208,9,1,141,16,208,104,74,
      176,68
20570 DATA 56,173,1,208,233,40,141,
      181,2,56,173,0,208,233,
      12,141,179
```

```
20580 DATA 2,173,16,208,233,0,41,1,
      141,180,2,173,180,2,74,141,180
20590 DATA 2,173,179,2,106,141,179,
      2,173,178,2,240,21,173,
      179,2,141
20600 DATA 167,2,173,180,2,141,168,
      2,173,181,2,141,169,2,32,
      184,192
20610 DATA 76,49,234,120,169,49,141,
      20,3,169,234,141,21,3,88,96
```

How to use the program

This program uses the machine code graphics routines introduced in Chapters 18 and 19. The machine code **must** be loaded into the 64 before this program is run. Appendix 18 gives full instructions on how to do this.

The program requires a joystick, which must be plugged into control port 2 at the right hand side of the 64. The joystick is used to move a cursor around the multicolour bit-mapped display, and the fire button of the joystick plots points on the screen.

When you run the program, after a short pause while the data is read, a menu is displayed offering you the following options, which are selected by the function keys:

Show Current Picture: This will change the display to bit-mapped mode without clearing the bit-mapped screen.

New Picture: This clears and displays the bit-mapped screen.

Load Picture: Loads previously saved pictures from the disk.

Save Picture: Saves the current bit-mapped picture onto the disk.

Change Colours: Allows you to change the colours of the display and the icons.

Stop Program: Stops the program!

Selecting either of the first two options displays a multicolour bit-mapped screen with seven icons at the bottom and a cross-hair cursor in the middle. Moving the joystick will move the cursor, and holding down the fire button will cause points to be plotted under the cursor.

The icons are used to select different line colours, and to use the line drawing and block filling routines. The four left-hand icons represent blobs of paint, and are used to select the line colour. Simply fire at the appropriate icon to change the colour. The selected icon will expand to twice its normal width.

The fifth icon (the line) is used to select line drawing mode. Once this is selected, the next point at which you fire will be the start of the line, and pressing fire again will mark the line end, and the line will be drawn. If you hold down the fire button while marking the line end a line will be flashed on and off, and will follow the cursor around until you release the button, when the line will be drawn in the currently selected colour.

The sixth icon (the F) is used in a similar way to fill blocks on the screen. The two fire pushes after selecting the fill icon will mark diagonally opposite corners of the rectangle to be filled.

The seventh icon represents the main menu, and is used to return there.

The 64's sprites are used as the icons and the cursor. The collisions detection facility of the VIC chip is used to check whether the cursor sprite is over one of the icon sprites.

The program uses an additional machine code routine to move the cursor and to plot on the bit-mapped screen when the joystick fire button is pressed. This machine code is included in the **DATA** statements in lines 20500 to 20610, and is loaded into memory when you run the program. This program is listed below.

```
100 IRQLO   = 788        64 interrupt vector
110 IRQHI   = 789        stored here
120 !
130 XL      = 679        Co-ordinates for plot
140 XH      = 680
150 Y       = 681
160 !
170 CURSORXL  = 691      Cursor sprite co-
                         ordinates
180 CURSORXH  = 692
190 CURSORY   = 693
200 !
210 XMIN    = 14         Max and min
220 XMAX    = 75         cursor co-ordinates
230 YMIN    = 41
240 YMAX    = 239
250 !
260 IRQVEC  =  59953     Normal interrupt
270 !                    address
280 PLOT    = $C0B8      Plot routine
290 PLOTOK = 690         Plot on/off
300 !
```

```
310 JOYREG = 56320        Joystick port
320 !
330 SPRXL  = 53248        Cursor (sprite 0)
340 SPRXH  = 53264        position registers
350 SPRY   = 53249
360 !
370 !
380 * = $C56E             Starts at 50542
390 !
400 !
410 !
420 SETUP   SEI           Reset interrupt vector
430         LDA #<PROG    to point at this
440         STA IRQLO     program
450         LDA #>PROG
460         STA IRQHI
470         CLI
480         RTS
490 !
500 !
510 PROG    LDA JOYREG    Read joystick port
520         LSR A
530         BCS J2        Branch if  bit 0 set
540         PHA
550         LDA SPRY      Stick held up
560         CMP #YMIN     If not at top
570         BCC J1B       move cursor up
580         DEC SPRY
590 J1B     PLA
600 !
610 J2      LSR A         Branch if bit 1 of
620         BCS J3        joystick port set
630         PHA
640         LDA SPRY      Stick held down
650         CMP #YMAX     If not at bottom
660         BCS J2B       move cursor down
670         INC SPRY
680 J2B     PLA
690 !
700 J3      LSR A         Branch if bit 2
710         BCS J4        of joystick port set
```

```
 720          PHA
 730          LDA SPRXH   Stick held left
 740          AND #1      If not at left-hand
 750          BNE J3A     edge of screen
 760          LDA SPRXL   move cursor left
 770          CMP #XMIN
 780          BCC J3B
 790 J3A      SEC
 800          LDA SPRXL
 810          SBC #1
 820          STA SPRXL
 830          BCS J3B
 840          LDA SPRXH
 850          AND #254
 860          STA SPRXH
 870 J3B      PLA
 880 !
 890 J4       LSR A       Branch if bit 3 of
 900          BCS J5      joystick port set
 910          PHA
 920          LDA SPRXH   Stick held right
 930          AND #1      If not at right-hand
 940          BEQ J4A     edge of screen
 950          LDA SPRXL   move cursor right
 960          CMP #XMAX
 970          BCS J4B
 980 J4A      INC SPRXL
 990          BNE J4B
1000          LDA SPRXH
1010          ORA #1
1020          STA SPRXH
1030 J4B      PLA
1040 !
1050 J5       LSR A       Branch if bit 4 of
1060          BCS EXIT    joystick port set
1070          SEC
1080          LDA SPRY    Fire button down
1090          SBC #40     reset plot co-ordinates
1100          STA CURSORY
1110          SEC
1120          LDA SPRXL
```

```
1130            SBC #12
1140            STA CURSORXL
1150            LDA SPRXH
1160            SBC #0
1170            AND #1
1180            STA CURSORXH
1190            LDA CURSORXH
1200            LSR A
1210            STA CURSORXH
1220            LDA CURSORXL
1230            ROR A
1240            STA CURSORXL
1250            LDA PLOTOK Check if plot allowed
1260            BEQ EXIT   If not branch
1270            LDA CURSORXL
1280            STA XL
1290            LDA CURSORXH
1300            STA XH
1310            LDA CURSORY
1320            STA Y
1330            JSR PLOT   Plot pixel
1340 !
1350 !
1360 EXIT       JMP IRQVEC Jump to normal
1370 !                     interrupt routine
1380 !
1390 !
1400 RESET      SEI             Disable routine
1410            LDA #<IRQVEC    by resetting
1420            STA IRQLO       interrupt vector
1430            LDA #>IRQVEC    to point to
1440            STA IRQHI       normal interrupt
1450            CLI             handler
1460            RTS
```

CHAPTER 23

SOUND AND MUSIC - PART 2

Sound adds another dimension to programs – making them more interesting and exciting or more user friendly. This chapter covers some of the ways in which the 64 can be used to play tunes and melodies, by exploiting the capabilites of its built-in synthesiser – the SID chip.

THE SID CHIP

The SID chip provides the 64 with three voices, each with a range of eight octaves, which can be independently programmed to produce an almost infinite variety of waveforms. The sound so produced may be heard either through the TV speaker or fed to a hi-fi system.

Playing Tunes

To get the 64 to play tunes, you must 'feed' the SID chip with the appropriate frequency data for the notes comprising the tune, in the correct order and at the correct speed.

One way of doing this is shown in this example:

```
1       GOSUB 1000
10      LO=INT(RND(1)*256)
20      HI=INT(RND(1)*256)
30      GOSUB 500
40      FOR D=0 TO 50:NEXT D
50      GOTO 10
500     REM CHANGE NOTE
```

```
510   POKE 54272,LO
520   POKE 54273,HI
530   RETURN
999   REM SET UP SID CHIP
1000  POKE 54296,15:REM VOL
1030  POKE 54277,54:REM ATTACK/DECAY
1040  POKE 54278,168:REM SUST/REL
1050  POKE 54276,33:REM WAVEFORM
1060  RETURN
```

This idea can be extended to cover all three channels like this:

```
1     GOSUB 1000:GOSUB 2000:GOSUB
      3000
10    LO=INT(RND(1)*256)
20    HI=INT(RND(1)*256)
30    POKE L1,LO:POKE L2,LO/2:POKE
      L3,LO/4
40    POKE H1,HI:POKE H2,HI/2:POKE
      H3,HI/4
50    FOR D=0TO50:NEXT D
60    GOTO 10
999   REM SET UP SID CHIP REG 1
1000  POKE 54296,15:REM VOL
1010  L1=54272:REM LOW
1020  H1=54273:REM HIGH
1030  POKE 54277,54:REM ATTACK/DECAY
1040  POKE 54278,168:REM SUST/REL
1050  POKE 54276,33:REM WAVEFORM
1060  RETURN
1999  REM SET UP SID CHIP REG 2
2000  L2=54279:REM LOW
2010  H2=54280:REM HIGH
2020  POKE 54284,54:REM ATTACK/DECAY
2030  POKE 54285,168:REM SUST/REL
2040  POKE 54283,33:REM CONTROL REG
2050  RETURN
2099  REM SET UP SID CHIP REG 3
3000  L3=54286:REM LOW
3010  H3=54287:REM HIGH
```

```
3020  POKE 54291,54:REM ATTACK/DECAY
3030  POKE 54292,168:REM SUST/REL
3040  POKE 54290,33:REM WAVEFORM
3050  RETURN
```

Neither of these examples could be described as music – for that we need a more controlled way of calculating the note frequencies.

The table on page 152 of the Commodore 64 User Guide gives a list of frequency values corresponding to musical notes (if you are unfamiliar with musical notation refer to Appendix 9). Using this table you can build up a series of numbers corresponding to the high and low bytes of the frequency of the notes, but can they be stored? One way is to keep them in a string, reading them one at a time and **POKE**ing them into the appropriate SID registers. This technique is illustrated in the following program:

```
1     GOSUB 1000
10    L$="%%%%?**%%/%"
20    H$="{CD}{CD}{CD}{CD}{HOM}
      {RVS}{RVS}{CD}{CD}"+CHR$(16)+"
      {CD}"
30    FOR Q=1 TO LEN(L$)
40    LO=ASC(MID$(L$,Q,1))
50    HI=ASC(MID$(H$,Q,1))
60    GOSUB 500
70    FOR X=0 TO 100:NEXT X
80    HI=0:LO=0:GOSUB 500
90    NEXT Q
100   STOP
500   REM CHANGE NOTE
510   POKE 54272,LO
520   POKE 54273,HI
530   RETURN
999   REM SET UP SID CHIP
1000  POKE 54296,15:REM VOL
1020  POKE 54277,54:REM ATTACK/DECAY
```

```
1030  POKE 54278,168:REM SUST/REL
1040  POKE 54276,33:REM WAVEFORM
1050  RETURN
```

The main limitation of this method is that the notes are all the same length, which renders it impractical for all but the simplest of tunes.

There are several ways of overcoming this. For example, a second string could be used to hold the data for the length of notes. Add these lines to the last program to hear this technique in action.

```
25    L$="32132121214"
70    FOR D=0 TO 100*VAL(MID$(L$,Q,
      1)):NEXT D
```

This method is adequate for simple tunes, but a better way is to hold information concerning notes and note lengths in **DATA** statements. You can see an example of this technique in the program at the end of the chapter, in which data for high and low frequencies is held in **DATA** statements.

Multi Channel Music

You can easily expand the **DATA** statement method to cater for tunes using all three voices, perhaps with different Attack/Decay and Sustain/Release characteristics to simulate different musical instruments. An extra problem occurs when a note on one voice is of a different length to that on another. To overcome this you could use three duration loop counters, one for each voice, or, more simply, reduce each note to the shortest common length and play a note several times if it is longer than this. This technique is illustrated in the program at the end of this chapter.

MUSIC FROM MACHINE CODE PROGRAMS

Using the SID chip from within machine code programs is very easy, since all you have to do is duplicate the BASIC POKEs to achieve the same effect, as the next program illustrates.

```
1      *=$C000
100    VOICE1LOW=54272
110    VOICE1HIGH=54273
120    ATTDEC=54277
130    SUSREL=54278
140    CONTROL=54296
150    NOTES=679
160    !
1000   INIT     LDX #0
1001            LDA #13
1002            STA NOTES
1003            LDA #15
1004            STA CONTROL
1005            LDA #54
1006            STA ATTDEC
1007            LDA #168
1008            STA SUSREL
1009   PLAY     LDA #33
1010            STA 54276
1015            LDA TABLE,X
1020            STA VOICE1LOW
1030            INX
1040            LDA TABLE,X
1050            STA VOICE1HIGH
1060            INX
1070            LDA TABLE,X
1075            INX
1080            TAY
1090            JSR DELAY
1092            LDA #32
1093            STA 54276
1094            LDY #255
1096   LOOP     DEY
1097            BNE LOOP
```

```
1100          DEC NOTES
1110          BNE PLAY
1120   QUIET  LDA #32
1130          STA 54276
1140          LDA #0
1150          STA 54296
1160          RTS
2000   DELAY  LDA #100
2010          STA 162
2015   TIME   LDA 162
2020          BPL TIME
2030   DEY
2040   BNE DELAY
2050   RTS
5000   TABLE  BYT 177,25,2,177,25,2,
              227,22,1,177,25,1,214,28
              ,2,154,21,4
5010          BYT
              63,19,4,37,17,2,37,17,
              2,47,16,1,37,17,1,63,19,
              2,107,14,4
```

However since machine code is so much faster than BASIC you could introduce such tricks as altering the volume of a note or changing the waveform, dynamically. Another possibility is to use interrupts to allow tunes to be played while other actions are being performed – you could even have the 64 play soothing music as you type in a program!!

Calculating Note Frequencies

The frequencies of musical notes are governed by mathematical rules - for example any note is exactly twice the frequency of the same note in the octave below. This means you don't have to rely on a table to calculate the frequencies you need for a particular tune: you can get the 64 to do it for you.

The next program is a simple music editor which allows you to enter a note and its octave in the form C3, which is note C from the third octave, and displays the appropriate frequency numbers for the SID chip to play that note. The note is played and the numbers and notes stored in the array TD%. Sharps and flats can be specified by adding the # or b symbol to the note when you type it in – C4#. The strings describing the notes are stored in the array N$.

```
10    DIM TD%(2,199),N$(199):HI=0:
      LO=0:GOTO 15000
992   REM
993   REM **************************
994   REM *                        *
995   REM * CALCULATE NOTE, OCTAVE *
996   REM *                        *
997   REM **************************
998   REM
1000  N=ASC(LEFT$(Z$,1))-65:IF N<0
      OR N>6 THEN EF=1:GOTO 1040
1010  O=ASC(MID$(Z$,2,1))-48:IF O<0
      OR O>7 THEN EF=1:GOTO 1040
1020  IF LEN(Z$)=2 THEN P=0:GOTO
      1040
1030  P=ASC(RIGHT$(Z$,1)):IF P<>35
      AND P<>45 THEN EF=1
1040  RETURN
1992  REM
1993  REM **************
1994  REM *            *
1995  REM * CALC PITCH *
1996  REM *            *
1997  REM **************
1998  REM
2000  IFN>1 THEN N=N-2:GOTO 2020
2010  N=N+5
2020  N=N*2
2030  IF N>4 THEN N=N-1
2040  IF P=35 THEN N=N+1
```

```
2050  IF P=45 THEN N=N-1
2060  RETURN
2992  REM
2993  REM ********************
2994  REM *                  *
2995  REM * CALC FREQ VALUES *
2996  REM *                  *
2997  REM ********************
2998  REM
3000  F=(274*(2↑O))*2↑(N/12)
3010  HI=INT(F/256)
3020  LO=INT(F-HI*256)
3030  RETURN
9992  REM
9993  REM ********
9994  REM *      *
9995  REM * PLAY *
9996  REM *      *
9997  REM ********
9998  REM
10000 POKE 54277,54:REM ATTACK/DECAY
10010 POKE 54278,168:REM SUST/REL
10020 POKE 54276,33:REM WAVEFORM
10030 POKE 54296,15
10040 POKE 54272,LO
10050 POKE 54273,HI
10060 RETURN
14992 REM
14993 REM ***************
14994 REM *             *
14995 REM * ENTER NOTES *
14996 REM *             *
14997 REM ***************
14998 REM
15000 GOSUB 10000:REM SET UP SID
15010 PRINT"s": INPUT "NOTE";Z$
15020 L=LEN(Z$):IF L<2 OR L>3 THEN
      15000
15030 GOSUB 1000
15040 IF EF=1 THEN EF=0:GOTO 15000
15050 GOSUB 2000
```

```
15060 GOSUB 3000
15070 PRINT"HIGH FREQUENCY=";HI
15080 PRINT"LOW FREQUENCY=";LO
15085 TD%(0,Z)=HI:TD%(1,Z)=LO:
      N$(Z)=Z$
15090 GOSUB 10040: Z=Z+1
15095 PRINT "PRESS ANY KEY TO
      CONTINUE"
15100 GET K$:IF K$="" THEN 15100
15110 HI=0:LO=0:GOSUB 10040:GOTO
      15010
```

PICTURES AT AN EXHIBITION

This program plays a piece of classical music in three part harmony to demonstrate some of the techniques covered in this chapter. A lot of typing is involved but the end result is worth it!

```
1     GOSUB 1000:GOSUB 2000:GOSUB
      3000
10    READ H1,L1,H2,L2,H3,L3
15    IF H1=999 THEN END
20    POKE 54273,H1:POKE 54272,L1
30    POKE 54280,H2:POKE 54279,L2
40    POKE 54287,H3:POKE 54286,L3
50    FOR D=0 TO 120:NEXT D
60    GOTO 10
999   REM SET UP SID CHIP REG 1
1000  POKE 54296,15:REM VOL
1030  POKE 54277,54:REM ATTACK/DECAY
1040  POKE 54278,191:REM SUST/REL
1050  POKE 54276,33:REM WAVEFORM
1060  RETURN
1999  REM SET UP SID CHIP REG 2
2000  POKE 54284,54:REM ATTACK/DECAY
2010  POKE 54285,168:REM SUST/REL
2020  POKE 54283,33:REM CONTROL REG
2030  RETURN
2099  REM SET UP SID CHIP REG 3
3000  POKE 54291,54:REM ATTACK/DECAY
```

```
3010  POKE 54292,168:REM SUST/REL
3020  POKE 54290,33:REM WAVEFORM
3050  RETURN
10000 DATA 12,216,0,0,0,0,12,216,0,
      0,0,0,11,114,0,0,0,0,11,114,0,
      0,0,0
10010 DATA 15,70,0,0,0,0,15,70,0,0,
      0,0,17,37,0,0,0,0,22,227,
      0,0,0,0
10020 DATA 19,63,0,0,0,0,19,63,
      0,0,0,0
10025 REM ***********************
10030 DATA 17,37,0,0,0,0,22,227,
      0,0,0,0,19,63,0,0,0,0,19,63,0,
      0,0,0,15,70,0,0,0,0
10040 DATA 15,70,0,0,0,0,17,37,0,0,
      0,0,17,37,0,0,0,0,12,216,
      0,0,0,0
10050 DATA 12,216,0,0,0,0,11,114,
      0,0,0,0,11,114,0,0,0,0
10055 REM ***********************
10060 DATA 12,216,9,159,7,163,12,
      216,9,159,7,163,11,114,8,
      147,7,53
10070 DATA 11,114,8,147,7,53,15,70,
      9,159,7,163,15,70,9,159,7,163
10080 DATA 17,37,14,107,8,147,22,
      227,14,107,8,147,19,63,14,107,
      11,114
10090 DATA 19,63,14,107,11,114
10095 REM ***********************
10100 DATA 17,37,14,107,8,147,22,
      227,14,107,8,147,19,63,15,
      70,11,114
10110 DATA 19,63,15,70,11,114,15,70,
      12,216,9,159,15,70,12,216,
      9,159
10120 DATA 17,37,12,216,10,205,17,
      37,12,216,10,205,12,216,8,
      147,6,108
```

```
10130 DATA 12,216,8,147,6,108,11,
      114,8,147,7,53,11,114,8,
      147,7,53
10135 REM ************************
10140 DATA 11,114,0,0,0,0,11,114,0,
      0,0,0,12,216,0,0,0,0,12,216,0,
      0,0,0
10150 DATA 9,159,0,0,0,0,9,159,0,0,
      0,0,11,114,0,0,0,0,12,216,
      0,0,0,0
10160 DATA 8,147,0,0,0,0,8,147,0,
      0,0,0
10165 REM ************************
10170 DATA 12,216,0,0,0,0,14,107,0,
      0,0,0,11,114,0,0,0,0,11,114,
      0,0,0,0
10180 DATA 22,227,11,114,5,185,22,
      227,11,114,5,185,19,63,11,
      114,7,163
10190 DATA 19,63,11,114,7,163,17,37,
      11,114,6,108,15,70,11,114,
      6,108
10200 DATA 11,114,0,0,5,185,11,114,
      0,0,5,185
10205 REM ************************
10210 DATA 11,114,0,0,0,0,11,114,0,
      0,0,0,12,216,0,0,0,0,12,216,
      0,0,0,0
10220 DATA 9,159,0,0,0,0,9,159,0,0,
      0,0,11,114,0,0,0,0,12,216,
      0,0,0,0
10230 DATA 10,60,0,0,0,0,10,60,0,
      0,0,0
10235 REM ************************
10240 DATA 15,70,0,0,0,0,17,37,0,0,
      0,0,13,156,0,0,0,0,13,156,
      0,0,0,0
10250 DATA 27,56,13,156,6,206,27,56,
      13,156,6,206,22,227,13,156,
      9,21
```

```
10260 DATA 22,227,13,156,9,21,20,
      100,13,156,7,163,18,42,13,156,
      7,163
10270 DATA 13,156,0,0,6,206,13,156,
      0,0,6,206
10275 REM ************************
10280 DATA 13,156,10,60,3,8,13,156,
      10,60,3,8,15,70,10,60,3,8,15,
      70,10,60,3,8
10290 DATA 13,156,10,60,2,220,13,
      156,10,60,2,220,15,70,10,
      60,3,54
10300 DATA 17,37,10,60,3,54,20,100,
      10,60,3,54,15,70,10,60,3,54
10310 DATA 13,156,10,60,6,16,13,156,
      10,60,6,16
10315 REM ************************
10320 DATA 18,42,13,156,11,114,20,
      100,17,37,13,156,22,227,18,42,
      13,156
10330 DATA 27,56,13,156,0,0,24,63,
      20,100,15,70,22,227,18,42,
      13,156
10340 DATA 20,100,17,37,13,156,24,
      63,0,0,12,32,22,227,18,
      42,15,70
10350 DATA 18,42,0,0,9,21,20,100,17,
      37,13,156,20,100,17,37,13,156
10355 REM ************************
10360 DATA 13,156,10,60,3,8,13,156,
      10,60,3,8,15,70,10,60,3,8,15,
      70,10,60,3,8
10370 DATA 13,156,10,60,2,220,13,
      156,10,60,2,220,15,70,10,
      60,3,8
10380 DATA 17,37,10,60,3,8,20,100,
      10,60,6,16,15,70,7,163,0,0
10385 REM ************************
10390 DATA 17,37,12,216,7,163,17,
      37,12,216,7,163,19,63,12,216,
      7,163
```

```
10400 DATA 19,63,12,216,7,163,17,37,
      12,216,7,53,17,37,12,216,7,53
10410 DATA  19,63,12,216,7,163,22,
      227,12,216,7,163,25,170,12,
      216,7,163
10420 DATA 19,63,12,216,7,163,17,
      37,12,216,15,70,17,37,12,
      216,15,70
10425 REM ************************
10430 DATA 22,227,17,37,14,107,25,
      177,21,154,17,37,28,214,22,
      227,17,37
10440 DATA 34,75,0,0,17,37,30,141,
      25,177,3,54,28,214,22,227,7,53
10450 DATA 25,177,21,154,17,37,30,
      141,0,0,15,70,28,214,22,227,
      19,63
10460 DATA 22,227,0,0,11,114,25,
      177,21,154,17,37,25,177,21,
      154,17,37
10465 REM ************************
10470 DATA 28,214,17,37,14,107,21,
      154,17,37,14,107,22,227,19,
      63,7,163
10480 DATA 22,227,19,63,7,163,28,
      214,17,37,2,220,28,214,17,
      37,2,220
10490 DATA 19,63,15,70,10,205,19,63,
      15,70,10,205,28,214,17,37,
      2,220
10500 DATA 28,214,17,37,2,220,19,63,
      15,70,10,60,19,63,15,70,10,60
10505 REM ************************
10510 DATA 22,227,14,107,11,114,17,
      37,14,107,11,114,19,63,15,
      70,11,114
10520 DATA 19,63,15,70,11,114,22,
      227,14,107,4,208,22,227,14,
      107,4,208
```

```
10530 DATA 19,63,15,70,11,114,19,63,
      15,70,11,114,22,227,14,107,
      4,208
10540 DATA 17,37,14,107,4,208,19,
      63,15,70,11,114,19,63,15,
      70,11,114
10545 REM *************************
10550 DATA 17,37,12,216,10,60,
      17,37,12,216,10,205,14,107,11,
      114,8,147
10560 DATA 14,107,11,114,8,147,15,
      70,11,114,9,159,15,70,11,114,
      9,159
10570 DATA 17,37,12,216,10,60,17,
      37,12,216,10,205,14,107,11,
      114,8,147
10580 DATA 14,107,11,104,8,147,15,
      70,11,114,9,159,19,63,11,114,
      9,159
10585 REM *************************
10590 DATA 17,37,12,216,10,205,17,
      37,12,216,10,205,14,107,11,114
      ,9,159
10600 DATA 14,107,11,114,9,159,17,
      37,12,216,10,205,17,37,12,216,
      10,205
10700 DATA 22,227,11,114,0,0,22,227,
      11,114,0,0,20,100,15,70,12,216
10710 DATA 19,63,0,0,9,159,17,37,14,
      107,11,114,15,70,11,114,9,159
10715 REM *************************
10720 DATA 17,37,11,114,3,155,17,37,
      11,114,3,155,19,63,11,114,4,12
10730 DATA 19,63,11,114,4,12,22,227,
      17,37,14,107,22,227,17,37,
      14,107
10740 DATA 25,177,20,100,15,70,30,
      141,0,0,15,70,22,227,11,114,
      2,220
```

```
10750 DATA 22,227,11,114,2,220,25,
      177,12,216,3,54,25,177,12,216,
      3,54
10755 REM ************************
10760 DATA 22,227,11,114,2,220,22,
      227,11,114,2,220,20,100,15,70,
      12,216
10770 DATA 19,63,0,0,9,159,17,37,14,
      107,11,114,15,70,11,114,9,159
10775 DATA 17,37,11,114,8,147,17,37,
      11,114,8,147,19,63,11,114,
      9,159
10776 DATA 19,63,11,114,9,159,22,
      227,17,37,14,107,22,227,17,37,
      14,107
10777 REM ************************
10780 DATA 25,177,20,100,15,70,30,
      140,15,70,0,0,22,227,11,114,
      2,220
10790 DATA 22,227,11,114,2,220,25,
      177,12,216,3,54,25,177,12,
      216,3,54
10800 DATA 22,227,11,114,2,220,22,
      227,11,114,2,220,12,216,0,0,6,
      108,12,216,0,0
10810 DATA 6,108,11,114,0,0,5,185,
      11,114,0,0,5,185
10815 REM ************************
10820 DATA 25,177,21,154,15,70,30,
      141,0,0,15,70,22,227,11,114,
      2,220
10830 DATA 22,227,11,114,2,220,25,
      177,12,216,3,54,25,177,12,216,
      3,54
10840 DATA 22,227,11,114,2,220,22,
      227,11,114,2,220,12,216,10,60,
      7,163
10850 DATA 12,216,10,60,7,163,11,
      114,8,147,7,53,11,114,8,147,
      7,53
10855 REM ************************
```

```
10860 DATA 15,70,11,114,7,163,15,70,
      11,114,7,163,17,37,14,107,
      11,114
10870 DATA 22,227,0,0,11,114,19,63,
      15,70,11,114,19,63,15,70,
      11,114
10880 DATA 17,37,14,107,11,114,22,
      227,0,0,11,114,19,63,15,70,
      11,114
10890 DATA 19,63,15,70,11,114,15,70,
      11,114,9,159,15,70,11,114,
      9,159
10895 REM ************************
10900 DATA 17,37,12,216,10,60,17,37,
      12,216,10,60,12,216,10,205,
      8,147
10910 DATA 12,216,10,205,8,147,11,
      114,8,147,7,53,11,114,8,147,
      7,53
10920 DATA 12,216,9,159,6,108,12,
      216,9,159,6,108,11,114,8,147,
      7,53
10930 DATA 11,114,8,147,7,53,15,70,
      11,114,7,163,15,70,11,114,
      7,163
10935 REM ************************
10940 DATA 17,37,14,107,11,114,22,
      227,0,0,11,114,19,63,15,70,
      11,114
10950 DATA 19,63,15,70,11,114,15,70,
      12,216,9,159,15,70,12,216,
      9,159
10960 DATA 20,100,17,37,12,216,20,
      100,17,37,12,216,17,37,14,107,
      11,114
10970 DATA 17,37,14,107,11,114,15,
      70,11,114,9,159,15,70,11,114,
      9,159
50000 DATA 0,0,0,0,0,0,999,0,0,0,0,0
```

The frequency data is stored in the order Low Byte, High Byte for voices one, two and three.

Arranging music, especially classical music, for the 64 often involves a good deal of pruning to make it work with only three voices – very little of it was composed with the SID chip in mind!

CHAPTER 24

THE 1541 DISK DRIVE

The 1541 disk drive provides a means of storing and retrieving programs and data in a more flexible way than is possible with the cassette unit. The storage medium is the floppy disk, a flexible disk of plastic coated with magnetic material and enclosed in a protective envelope. The principal advantages of floppy disks over cassette tapes are greatly increased speed of data retrieval, and random access to data.

Connecting the Disk Drive

The disk drive connects to the 64 via the serial bus, which uses the 5 pin socket on the back of the computer. A suitable cable is supplied with the disk drive. Always switch on the disk drive before the 64, to avoid damage to the computer. Never switch the drive on or off with a disk in place.

HOW THE DISK DRIVE OPERATES

The 1541 disk drive unit contains both mechanical and electronic parts. The mechanics are used to hold a disk firmly in place when the door is closed and to rotate the disk at approximately 300 rpm. A special motor called a stepper motor controls the movement of a read/write head similar to the one used in the cassette unit. The stepper motor can place the head in contact with the disk, and move it to and fro along the slot cut in the disk envelope. Because the disk is rotating, this arrangement means that any portion of the disk can be reached by the head. The head can read data from and write it to the disk.

The 1541 contains its own microprocessor, a 6502, which is very similar to the 6510 in the 64. The processor has some RAM memory and a program known as a *disk operating system* stored in ROM. The 2k of RAM is arranged into 256 byte buffers, and any data transfers between computer and disk drive occur via these buffers. There are also some chips to control the interface between the disk drive and the 64.

The Disk Operating System

The disk operating system or DOS is contained in 16k of ROM in the disk drive. This controls all the mechanical functions of the drive and the interface with the computer. One advantage of having the DOS resident within the disk drive is that for many operations the transmission of the appropriate command to the DOS is all the computer needs to do. The DOS carries out the operation leaving the computer free for other tasks.

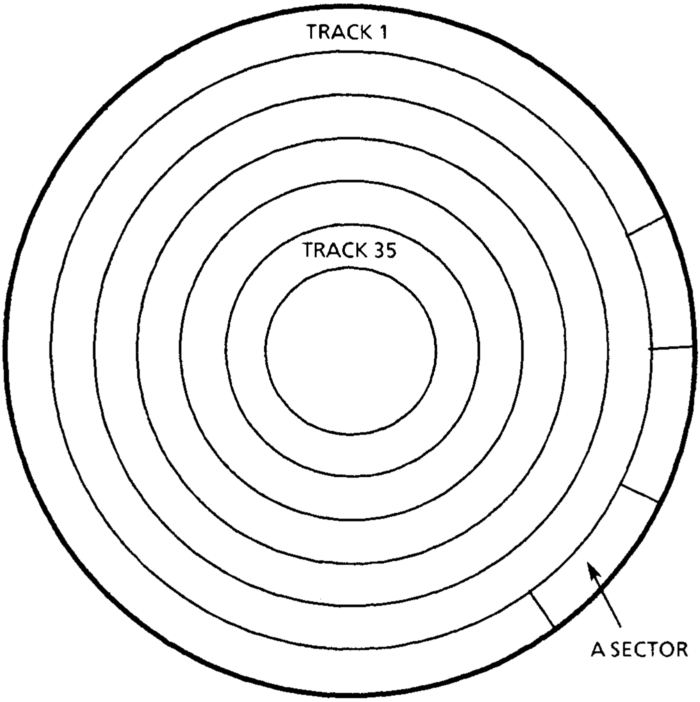
Data Storage on Floppy Disks

Data is stored on a disk in a number of concentric tracks each of which is divided into a number of sectors. The diagram on the next page shows how the tracks are arranged.

A disk for use in a 1541 drive has 35 tracks, track 1 being outermost. Each track is divided into a number of sectors which can store a *block* of data – 256 bytes. The exact number of sectors per track varies, since the tracks are of unequal length – the outermost tracks have 21 sectors while track 35 has only 17 sectors.

TRACK NUMBER	NUMBER OF SECTORS
1 - 17	21
18 - 24	20
25 - 30	18
31 - 35	17

Sector Distribution by Track



The Arrangement of Tracks and Sectors

In addition to the data bytes, each sector contains information used by the DOS for various housekeeping functions and to indicate on which track and sector the next block of data for a file is located.

Care of Floppy Disks

Floppy disks are rather more delicate than cassette tapes, and to prevent damage great care must be taken when handling and storing them. By following the guidelines in this section you will minimise the risk of losing your valuable programs or data.

a) Always return the disk to its envelope immediately after use, and keep disks in a sturdy box designed for the purpose.

b) Do not store disks near magnetic fields (such as those generated by the TV), since stray magnetic fields can destroy data.

c) Keep disks away from heat or sunlight.

d) Never write on the disk's plastic jacket – fill in sticky labels before applying them to the disk.

e) Do not touch the surface of the disk or try to clean it.

f) Remove disks from the drive before switching it on or off.

With the exception of the danger from magnetic fields, the rules for looking after floppy disks are similar to those for audio records. However, the penalties for not observing them are rather more severe!

Formatting Blank Disks

The 1541 disk drive uses standard 5.25 inch floppy disks. Ask your dealer for *single sided single density* disks.

Before you can use a new disk it must be prepared, by a process called *formatting* to enable the DOS to store data on it. The formatting operation defines the tracks and sectors of the disk, and is carried out by the DOS.

To pass instructions to the drive, you must open a channel, in the same way as you would to send data to the cassette unit or any other peripheral device. The channel number 15 is reserved for communications with the disk unit.

To format the disk type:

```
OPEN 15,8,15

PRINT#15,"NEW:TEST DISK,A1":CLOSE 15
```

These two commands will open the disk command channel and send the NEW command to the disk drive. The NEW command, in common with all other DOS commands, can be abbreviated to its first letter, so:

```
PRINT#15,"N:TEST DISK,A1":CLOSE 15
```

would work equally well.

The disk drive will start to make a noise and the cursor will reappear leaving the 64 free for use while the DOS program formats the disk. The process involves the clearing of any old data from the disk and the setting up of the 683 blocks for data storage. To keep track of what is saved on the disk, some of these blocks are devoted to a *directory*, which holds the disk name (in this case TEST DISK, but you could use any name of up to 16 characters) and a list of all the files on the disk. Each block on the disk is labelled with the disk identifier (in this case A1, but any two characters are allowed) which enables the DOS to uniquely identify the disk.

The formatting operation takes about 80 seconds, after which disk activity will cease. You can now examine the directory of the disk by typing:

```
LOAD "$",8

LIST
```

You will see the disk name and identifier in inverse video, and the words '664 blocks free'. The characters '2A' indicate which version of the DOS program was used to format the disk: this is the same for all 1541 drives.

Because nothing has been stored on the disk yet, the maximum number of blocks is available for use (664*256 bytes = 169984 bytes of storage), and there are no entries in the directory.

Clearing the Directory

Once a disk has been formatted, there is no need to do so again. If you have a disk containing programs you no longer require (be certain!!), you can clear the directory by using the NEW command but omitting the disk ID:

```
OPEN 15,8,15

PRINT#15,"NEW:RECLAIMED DISK"
```

This will not clear the disk, but will change the directory header and make all the sectors available for use so that you can write over any existing data.

THE BLOCK AVAILABLITY MAP (BAM)

The DOS needs to know which of the sectors of a disk have not yet been used, so that maximum use can be made of the space available. To keep track of what's left, track 18 sector 0 is reserved for a map of the available sectors of each track on the disk. This map is known as the *block availablity map* or *BAM*.

The arrangement of data in the BAM is shown in the diagram below:

BYTE	CONTENTS	USE
0, 1	18, 01	Pointer to Track & Sector of first Directory Block
2	65	ASCII 'A' indicates disk format
3	0	Not used
4 - 143	BAM	Bit map of blocks used in tracks 1 - 35

The BAM Format

The first four bytes of the BAM are used by the DOS as storage for information about the disk, the remainder is given over to a bit map of the 683 blocks on a disk. Each bit in a byte of the BAM represents a block of data on the disk. If the bit is set then the block is available for use; if not then that block has been used. The BAM is updated every time a program is saved or when a data file is closed so that the DOS has available an up to date picture of the storage space left on the disk.

Loading and Saving Programs

The most obvious and immediate use of the 1541 drive is as a means of storing programs. The BASIC commands **LOAD** and **SAVE** are used in the same way as they are with the cassette unit, except that you must specify the disk drive as the device you wish to use by adding the device number 8 to a command. For example to load a program from disk, type:

```
LOAD "MUTO ATTACK",8
```

The BASIC program MUTO ATTACK will be loaded from the disk into memory.

To save a BASIC program on disk:

```
SAVE "KILLER DAISIES",8
```

You can save a revised version of a program so that it replaces the old one with the same name on the disk. To do this insert an @ in the command like this:

```
SAVE "@:BEAN SHOOT",8
```

Machine code programs or blocks of data can also be loaded from disk, by adding a secondary address of 1 to the **LOAD** command:

```
LOAD "TANK BATTLE",8,1
```

will load the file into the area of memory from which it was saved.

SAVE always saves a BASIC program, so to save machine code programs without a monitor or assembler involves modifying the zero page locations used to mark the beginning and end of a BASIC program in memory. The locations concerned are:

43 Low byte of start address of BASIC program
44 High byte of start address of BASIC program

45 Low byte of start of BASIC variables
46 High byte of start of BASIC variables

In normal use, the **SAVE** command stores the contents of memory between the addresses pointed to by the contents of these locations. To use **SAVE** to save data from another area of memory, these two pointers must be loaded with the start and end address of that area. For example to save a machine code program stored from $C000 to $C100, you would type:

```
POKE 44,192:POKE 43,0

POKE 46,193:POKE 45,0
```

followed by:

```
SAVE "MACHINE CODE",8
```

These operations will occur much more rapidly than with the cassette unit. Because of the random access nature of disk drives, several programs can be stored on the same disk without increasing the time taken to load a program.

As with cassettes, it is possible to prevent overwriting of data. The small notch in one edge of the disk is called a *write protect notch,* and to prevent overwriting this must be covered with one of the small sticky tabs supplied with the disk.

The disk drive is a very flexible device, and is useful for more than storage of programs. To exploit it fully you'll need to know more about how it works.

DISK HOUSEKEEPING OPERATIONS

The DOS recognises a group of commands known as housekeeping commands. As the name suggests these commands allow you to maintain the state of the disk by renaming files, deleting files etc.

To use these commands you must **OPEN** a channel to channel 15 of the disk drive (use the secondary address 15). For example:

```
OPEN 15, 8, 15
```

The commands are then sent to the disk drive using the **PRINT#** command. Remember to **CLOSE** the channel after using it.

The first of the commands is a method of reading the error status of the drive – an error is indicated by the red LED flashing on the front of the drive.

The following program reads the error status of the drive from channel 15:

```
10    OPEN 15,8,15
20    INPUT#15,A$,B$,C$,D$
30    PRINT "ERROR CODE "A$
40    PRINT "ERROR TYPE "B$
50    PRINT "     TRACK "C$
60    PRINT "    SECTOR "D$
```

A list of DOS errors and their meanings is given in Appendix 6.

This is a very cumbersome way of obtaining error information, and a more convenient method is described in the section on the DOS support program on page 150.

Initialise

The INITIALISE command (abbreviated to I) returns the disk drive to its normal state by copying the BAM into disk drive RAM. All files are closed and error conditions cancelled. The syntax is:

```
   PRINT#15,"I"
```

You will find this command useful when experimenting with some of the more complex disk operations in the next chapter!

Validate

A VALIDATE operation will 'tidy up' and force the DOS to re-organise the files on the disk and thus make optimum use of the available disk space and free any partially used blocks. The command is abbreviated to V, and its format is:

```
   PRINT#15,"V"
```

Scratch

You can delete files from the disk with the SCRATCH (abbreviated to S) command - the format is:

```
   PRINT#15,"S:FILE NAME"
```

You can delete a group of files using the 'wild card' facility, for example:

```
   PRINT#15,"S:TEST*"
```

would delete all files on the disk whose file names began with the characters TEST. Be careful when using this facility, or you may delete files you want to keep.

Copy

The COPY (C) command allows you to duplicate a file under a different name on the disk. The syntax is:

```
   PRINT#15,"C:NEW NAME=OLD NAME"
```

Another function of the COPY command is to combine files – this technique is discussed in the section on sequential files in the next chapter.

Rename

It is possible to alter the directory entry for a file using the RENAME (R) command:

```
PRINT#15,"R:NEW ENTRY=OLD ENTRY"
```

THE DOS SUPPORT PROGRAM

One of the programs on the demonstration disk supplied with the 1541 disk drive is a short machine code program which supports the DOS resident in the disk drive. Known as "DOS 5.1", its function is to simplify disk handling commands.

The machine code is loaded into the 64 by a short BASIC program called "C-64 WEDGE", which is also on the disk. This simply loads DOS 5.1 and initialises it with a **SYS** call to its start address.

Since DOS 5.1 is located in the 4k block of memory from $C000 it has no effect on the amount of memory available for BASIC programs.

Under DOS 5.1 the @ and > keys are used to issue commands to the disk drive, in exactly the same way as **PRINT#** is used to issue commands via channel 15.

Directory

For example to obtain a directory of a disk, use the command:

```
    @$
```

The directory will be loaded directly into screen memory and not as a program, so any program in memory will not be overwritten. You can slow down the scrolling of a long directory by holding down the CTRL key, and use the space bar to stop and start the scrolling.

DOS 5.1 allows you to search the directory for a specific program as follows:

```
    @$ PROGRAM NAME
```

If the program is on the disk its name will appear on the screen, otherwise the directory will appear blank.

You can search for and display the names of programs in a specified group by adding an asterisk (*) to the program name:

```
    @$ PROG*
```

This will display the names of any programs in the directory beginning with the characters PROG.

Similarly the command

```
    @$ ????NAME
```

will search for and display the names of any programs whose last four characters are NAME, regardless of the preceding four characters.

Error Status

When a disk error is indicated by the flashing red LED, the status can be read by typing:

```
    @ (or >)
```

without the need to run a program.

This will result in a display of the error code and type (as given in Appendix 6), and the track and sector on which the error occurred, in the following format:

XX, ERROR MESSAGE, TT, SS

where

XX is the error code number

TT is the track number

SS is the sector number

Loading BASIC Programs

To load a BASIC program under DOS 5.1, type:

```
    /PROGRAM NAME
```

and press RETURN.

Quotation marks are optional.

Loading Machine Code Programs

To load a machine code program, type:

```
    %MACH PROG
```

and press RETURN. This has the same effect as the statement:

```
    LOAD "MACH PROG",8,1
```

and loads the machine code program or data into memory, starting at the location from which it was saved, rather than at 2048 as with a normal **LOAD** or / command. Again DOS 5.1 does not require quotation marks around the program name.

Auto Loading BASIC Programs

DOS 5.1 allows you to load a BASIC program so that it will run automatically, by typing:

```
    ↑PROGRAM NAME
```

Saving BASIC Programs

You can save a BASIC program by typing:

```
    ←PROGRAM NAME
```

All the other operations described earlier in this chapter which involve issuing commands to the disk drive via channel 15 can be duplicated with the @ or > keys. For example, to format a disk, type:

```
@N:DISK NAME,ID
```

Quitting DOS 5.1

If you want to stop using DOS 5.1, it can be deactivated with the command:

```
@Q
```

This will return the 64 to its normal state, but will not delete DOS 5.1 from memory, and it can be reactivated by typing:

```
SYS 52224
```

DOS 5.1 makes the life of the 1541 disk user much easier, but it's not very convenient to have to load it into the computer from the demonstration disk every time you use the machine. It is a good idea to copy it onto your own disks. To do this first load DOS 5.1 into the 64 from the demonstration disk, then type:

```
POKE 44,204:POKE 43,0

POKE 46,207:POKE 45,89
```

Now insert your disk and type:

```
SAVE "DOS 5.1",8
```

The machine code will be saved on your disk for your future use, because the four POKEs 'tricked' the 64 into thinking "DOS 5.1" was a BASIC program – the technique is described earlier in this chapter. To reset the pointers afterwards type:

```
NEW
```

CHAPTER 25

ADVANCED DISK OPERATIONS

The disk drive is useful for more than program storage and this chapter covers the various types of data file available with the 1541 drive, and the commands needed to make use of them. A section on the use of machine code with the disk drive is included.

SEQUENTIAL DATA FILES

Most applications for computers require some means of storing and retrieving data. In a simple system, a cassette unit can be used for storage, but this has the limitation that to access any item of data on a tape, the computer must first read through all those that precede it - the data is stored sequentially, one byte after another, along the tape. This method of data storage is acceptable for small amounts of data, but only a very keen enthusiast could make much serious use of such an arrangement for an application such as a database.

Sequential files can also be stored on floppy disks in exactly the same way as with cassettes, but are much faster in use.

As with cassettes, disk sequential files are created with the **OPEN** command. The syntax of the **OPEN** statement when creating a sequential file is:

```
    OPEN LFn,Dev,SA,"NAME,S,Type"
```

The logical file number, LFn, can be any number between 0 and 255, and is used to reference that file throughout the program. A device number, Dev, of 8 specifies the disk drive, and the Secondary Address, SA, may be any number between 2 and 14. Filenames may contain up to sixteen characters, and the character 'S' indicates a sequential file.

The type parameter specifies whether the file is to be read (R) or written (W).

If no type parameter is included, a new file is created ready for a write operation. When the **OPEN** command is executed, the DOS checks to see if the file already exists and if so the file is opened ready for a read or write operation, as specified by the type parameter. If the file doesn't exist and a write operation is specified, a new file is created. Any other use of **OPEN**, such as trying to open a non-existent file for a read operation, will result in an error.

Reading and Writing Sequential Files

The **PRINT#** command is used to write data to a sequential file. Data items can be separated by carriage returns (CHR$(13)), commas or semi-colons. The rules are exactly the same as those for printing to the screen and with cassette files and need not be covered here.

The **GET#** and **INPUT#** commands are used to read data from a sequential file and again their use is exactly the same as with cassette files.

Closing Sequential Files

A file must be closed after use. This will complete the transfer of data from the buffer onto the disk, and free the channel for other uses. If the file is not closed, some data may be lost. The command to close a file is:

```
CLOSE LFn
```

The next program shows how to create a sequential file on a disk, and how data can be written to and read from it:

```
1      REM SEQUENTIAL DEMO
10     OPEN 6,8,6,"SEQUENTIAL FILE,S,
       W":REM OPEN SEQUENTIAL FILE
       FOR A WRITE OPERATION
20     FOR X=0 TO 25:REM WRITE DATA
30     PRINT#6,CHR$(65+X);
40     NEXT X
50     CLOSE 6
99     REM READ IT BACK!
100    OPEN 6,8,6,"SEQUENTIAL FILE,S,
       R":REM OPEN FOR A READ
       OPERATION
110    FOR X=0 TO 25
120    INPUT#6,A$
130    PRINT A$;
140    NEXT X
150    CLOSE 6
```

You can re-open a sequential file to enable it to be updated by inserting a @ symbol in the command like this:

```
OPEN 3,8,10,"@:UPDATED FILE,S,W"
```

Concatenating Sequential Files

You can add the contents of up to four sequential files together making one large file, with the COPY command. For example the following statement would add the contents of the file called STOCK to the file called PRICES and create a new file called

INVENTORY which would contain all the information.

```
PRINT#15, "C:INVENTORY=0:STOCK,
0:PRICES"
```

Sequential files are useful for storing such things as character sets or sprite data, in which no single item of data is to be retrieved alone. The word processor program which we used to write this book stores its text as a sequential file. Because data is stored one item after another, sequential files make very efficient use of the disk space, but are inflexible when it comes to more sophisticated applications.

There is another type of data file you can use with the 1541 drive which overcomes some of the limitations of sequential files and allows you to access data from any point on the disk without having to read all that precedes it. This is known as the *relative file,* and its use results in a large time advantage over sequential files, at the expense of less efficient use of disk space and more complex programming.

RELATIVE DATA FILES

Relative files allow the programmer ready access to any item of data on the disk by structuring the data into records. The DOS keeps a record of the tracks and sectors used by a relative file by establishing *side sectors* – a list of pointers to the start of each record within a relative file. This procedure is handled completely by the DOS, thereby simplifying the programming task.

Each record in a relative file may contain up to 254 characters – the arrangement is shown in the table on the next page.

BYTE	CONTENTS
0, 1	Pointer to T & S of next data block
2-255	254 bytes of data. Empty records have FF as first character, all the rest are null. Partially filled records are padded with nulls.

Relative File Data Block Format

Each side sector is contained in a single data block and can store pointers, in the form of track and sector numbers, for up to 120 records.

BYTE	CONTENTS
0, 1	Pointer to T & S of next side sector block
2	Side sector Number
3	Record Length
4, 5	Track & sector of side sector number 0
6, 7	Track & sector of side sector number 1
8, 9	Track & sector of side sector number 2
10, 11	Track & sector of side sector number 3
12, 13	Track & sector of side sector number 4
14, 15	Track & sector of side sector number 5
16-255	Track & sector pointers to 120 data blocks

Relative File Side Sector Block Format

Each relative file can have up to 6 side sectors, and so may comprise up to 720 records - more than the capacity of a disk!

Creating a Relative File

When a relative file is created, a side sector is created for that file, containing the data given in the table above, and the first record of that file is set up. The BASIC **OPEN** command is used to create a relative file:

```
OPEN LFn,Dev,Chan,"NAME,L,"+ CHR$
(Rec.Length)
```

An example of this command in use is:

```
   OPEN 2,8,2,"DATABASE,L,"+CHR$(50)
```

where file number 2 has been used to create a relative file called DATABASE, having records which are 50 characters long.

Once a file has been created it can be accessed in the usual way, without specifying file type:

```
   OPEN 2,8,2,"DATABASE"
```

Reading and Writing Relative Files

To read or write data in a relative file, the number of the required record must be specified using the POSITION command. Like other DOS commands, POSITION (abbreviated as P) is sent to the drive via channel 15:

```
PRINT#15,"P"CHR$(C)CHR$(Low)CHR$(High)
```

Where C is a channel number and Low and High are the low and high bytes of the record number (two bytes are needed because a record number may be greater than 255).

The POSITION command positions the *file pointer* to the specified record before a read or write operation. If you position the pointer to a record which hasn't been created, the error code 50 is created, which should not be regarded as an error, rather a warning that no **INPUT#** or **GET#** operation should be attempted. It is quite in order to ignore the message if you intend to write data into the record, since the act of writing will create the record. Data is written to a relative file in the same way as to a sequential file - using **PRINT#** after setting the file pointer to the required position.

If it is envisaged that a large number of records are to be created, it is advisable to create the last record at the start of the program. This will force the DOS to create all the intermediate records, and any extra side sectors as required. Each record so created will contain 255 as the first character, indicating that is unused. By creating these records, all subsequent operations with the relative file will be greatly speeded up.

The use of the POSITION command with relative files is illustrated in the following program:

```
10    OPEN 15,8,15
20    OPEN 2,8,2,"RELATIVE
      TEST,L,"+CHR$(100)
30    PRINT#15,"P"CHR$(2)CHR$(100)
      CHR$(0)
40    INPUT#15,A,B$,C,D
50    PRINT "ERROR"A;B$
60    PRINT#2,"THIS IS THE 100TH
      RECORD"
70    CLOSE 15:CLOSE 2
80    END
```

How the Program Works

The program creates a relative file with a record length of 100 characters, and uses the POSITION command to place the file pointer at record 100. A check of the error channel gives the error message ERROR 50 - RECORD NOT PRESENT, but as explained above, this is interpreted as a warning not to read data. Line 60 ignores the error and writes some data into record 100. This also causes records 1 to 99 to be created with 255 as the first character, and the two channels are closed in line 70.

The next program will allow you to read the contents of the relative file just created, by using POSITION to move the file pointer to record 100 and read the contents.

```
99    REM READ IN 100 TH RECORD
100   OPEN 15,8,15
110   OPEN 2,8,2,"RELATIVE TEST"
120   PRINT#15,"P"CHR$(2)CHR$(100)
      CHR$(0)
130   INPUT#2,D$
140   PRINT "THE CONTENTS OF RECORD
      100 ARE:"D$
150   CLOSE15:CLOSE2
```

If you modify the program to read any other record, a 'STRING TOO LONG' error will be generated when the program attempts to read the data. This occurs because each empty record contains a CHR$(255) followed by 99 bytes of CHR$(0) and no terminator character, like a carriage return or comma. The 80 character limit on the BASIC INPUT# command is therefore exceeded and an error occurs.

You can overcome this problem by using GET# to check that the first character of a record isn't 255 before reading the contents of that record. To do this, a further facility of the POSITION command is used – the ability to place the file pointer at any position within a record. This is achieved by adding a further parameter to the P command specifying the position within a record:

```
PRINT#15,"P"CHR$(2)CHR$(15)CHR$(0)CHR$
(10)
```

This command would place the file pointer at the 10th byte of record 15 in the relative file accessed via channel 2.

If you change the following lines in the last program, you will be able to use this technique to test each record in the file:

```
115   INPUT"WHICH RECORD";R
120   PRINT#15,"P"CHR$(2)CHR$(R)
      CHR$(0)CHR$(0)
125   GET#2,T$:IF T$=CHR$(255) THEN
      PRINT "EMPTY RECORD":GOTO 150
```

If you run the modified program using any record number between 0 and 99 you will discover empty records. If you try record 100, the DOS will become confused about the contents of the record because the **GET#** in line 125 has moved several bytes of data into the buffer, so you will get peculiar results! To prevent this occuring in more serious applications you must reset the file pointer after checking for an empty record if you then want to use **INPUT#**.

The ability to position the file pointer to any point within a record means that a record can be divided into fields, allowing more efficient use of the disk space and leading to a more flexible way of creating databases. To make it work, you must keep track of the length of each field in order to be able to read and write into each field. The example given in the 1541 manual of a mailing list shows the principles behind the technique (obvious errors aside!). The program later in this chapter is a more comprehensive example of the use of relative files together with indexing sequential files.

RANDOM DATA FILES

Random data files provide a means of specifying the track and sector of the disk upon which your data is stored. They are harder to program than relative files and offer no advantage over them in most applications. They do offer greater control over what goes where on the disk, but are less efficient in their use of disk space than relative files.

To use random files two channels must be opened to the disk drive, one for sending commands (channel 15) and another for sending data to one of the 256 byte RAM buffers in the disk drive.

Opening a Random File

The syntax of the **OPEN** command for a random file is shown below:

```
   OPEN LFn,Dev,Channel No,"#"
```

An example of its use is:

```
   OPEN 5,8,5,"#"
```

The # symbol can optionally be followed by a number to specify in which of the buffers you want to store the data, but this is not normally necessary. The DOS selects the next available buffer for you, and the only reason for choosing a specific one is for machine code applications where some code is to be stored in the buffer, and you need to know at what address it starts.

Reading and Writing Random Files

Having **OPEN**ed a random file, another command is sent via channel 15 to instruct the DOS to either read the contents of a specified track and sector into the selected buffer, or to write the contents of the buffer to a specified track and sector of the disk.

The two commands which do this are BLOCK-READ and BLOCK-WRITE (abbreviated as B-R and B-W).

BLOCK-READ

The syntax of the BLOCK-READ command is:

```
PRINT#15,"B-R:"Chan;Drv;Tk;Sec
```

Note that the parameters are separated by semi-colons (;) and *not* commas as described in some versions of the 1541 disk drive manual. The drive number,Drv, is 0 for a single drive unit like the 1541 and can be omitted.

The following program demonstrates the use of the BLOCK-READ command to display the contents of any track and sector.

```
5      REM BLOCK READ DEMO
10     OPEN 15,8,15
20     OPEN 5,8,5,"#"
30     PRINT"{CLS}": INPUT "WHICH
       TRACK,SECTOR?";T,S
40     IF(T>18 AND S>20)OR (T<25 AND
       S>18)OR(T<31 AND S>17)OR(T<36
       AND S>17) THEN 30
50     PRINT#15,"B-R:"5;0;T;S
60     FOR Z=0 TO 255
70     GET#5,D$
80     IF ST=0 THEN A$=A$+D$:NEXT Z
90     PRINT "{CLS}TRACK "T" SECTOR"
       S:PRINT
100    PRINT A$
110    CLOSE 5:CLOSE 15:END
```

This program will work for any block, but to see something meaningful, try using it to examine the directory (track 18 sector 1).

How the Program Works

Lines 10 and 20 open channel 15 for commands and channel 5 to the buffer.

Line 30 sets the variables T and S to the chosen track and sector and line 40 checks that such a combination of track and sector exists on the disk. If not, another input is requested.

Line 50 sends the Block Read (B-R) command, specifying the buffer associated with channel 5, drive 0 and the track and sector numbers which specify the data block to be read.

The loop in lines 60 to 80 continues to read data one byte at at time from the buffer for as long as the system status word, ST, is zero. When the end of the data in the block is reached ST is set to 64 and control jumps to line 90, which displays the contents of track T, sector S.

BLOCK-WRITE

The BLOCK-WRITE command performs the opposite action to the B-R command and allows you to write data to any block on the disk. Its syntax is:

```
   PRINT#15,"B-W:"Chan;Drv;Tk;Sec
```

Again the parameters are separated by semicolons, and the drive number is zero for a single drive.

The following program writes the letters of the alphabet onto track 1; sector 1 of the disk - you could check this by using the B–R program above.

```
10 OPEN 15,8,15
20 OPEN 5,8,5,"#"
30 FOR I=0 TO 25
40 PRINT#5,CHR$(I+65)
50 NEXT I
60 PRINT#15,"B-W:"5;0;1;1
70 CLOSE 15:CLOSE 5
```

Notice that the data is written into the buffer via channel 5, and when this operation is complete the contents of the buffer are written to the block specified in the B–W command.

In these examples, we have not specified which buffer is to be used for B–R and B–W operations, leaving the choice up to the DOS. This is perfectly acceptable under normal circumstances, but in very complex programs you may want to know which buffer has been selected for a particular operation. You can find out by issuing a **GET#** command immediately after opening the channel for data (in our case channel 5). The byte returned is the number of the buffer selected by the DOS. You can only interrogate the disk drive for buffer information *before* any read or write operations are carried out with that buffer.

You can probably imagine that writing data haphazardly all over the disk in this way isn't very useful - you can easily lose track of what data is where and if you inadvertently overwrite another file or the directory you can ruin the disk. For random files to co-exist safely with program and other files on a disk, care must be taken to check the BAM for a free block *before* writing data to the disk, and to update the BAM after writing the data.

The command which allows you to do this is BLOCK-ALLOCATE.

BLOCK-ALLOCATE

The syntax of BLOCK-ALLOCATE (abbreviated as B-A) is:

```
   PRINT#15,"B-A:"Drv;Tk;Sec
```

If you precede an attempt to select a block for use in a random file by a B-A command, the DOS will check the BAM to see if that block is available. If it isn't, an error is returned via channel 15, along with the numbers of the next available track and sector. The next program shows B-A in operation.

```
10    PRINT "{CLS}": OPEN 15,8,15
20    T=18:S=1
30    PRINT#15,"B-A:"0;T;S
40    INPUT#15,A,B$,C,D
50    IF A<>65 THEN 80
60    PRINT "TRACK"T" SECTOR"S" IS
      NOT AVAILABLE": PRINT
70    PRINT "THE NEXT FREE BLOCK IS
      AT TRACK "C: PRINT TAB(26)
      "SECTOR"D:END
80    PRINT "TRACK"T" SECTOR"S" MAY
      BE USED"
```

How the Program Works

The track and sector for allocation are set to 18 and 1 which is the start of the directory, and consequently not available. Line 30 attempts to allocate this block, and line 40 reads the results of the attempt from channel 15. If, as in this example, the DOS finds that the requested block is unavailable (i.e. its entry in the BAM is set to 1) then an error code of 65 is returned and the numbers of the next available track and sector are supplied. If the block is available, the B-A command updates the BAM and a write operation can be performed. The results of the B-A attempt are displayed by lines 60 to 80.

In practice, this test would be carried out before any attempt to write data.

BLOCK-FREE

BLOCK FREE (B-F) is a corresponding command which allows you to release a block for use. No data is destroyed but the BAM entry for the specified block is cleared allowing new data to overwrite what is already there.

The syntax of the B-F command is:

```
PRINT#15,"B-F:"Drv;Tk;Sec
```

To make use of random files, you need to keep a record of the blocks allocated to a random file. The best way of doing this is by maintaining a 'directory' in a sequential file. Each random file would have a corresponding sequential file containing information about which blocks have been used.

The following short program stores data in a random file and keeps an index of the track and sectors as they are used.

```
10    T=1:S=1
20    OPEN 15,8,15
30    OPEN 5,8,5,"#"
40    OPEN 4,8,4,"@0:INDEX,S,W"
50    INPUT"NUMBER OF ENTRIES";N
55    PRINT#4,N
60    FOR X=1 TO N
70    PRINT"{CLS}ENTER DATA FOR
      ENTRY NUMBER";X
```

```
80     INPUT D$
90     PRINT#5,D$
100    PRINT#15,"B-A:"0;T;S
110    INPUT#15,A,B$,C,D
120    IF A=65 THEN T=C:S=D:GOTO 100
125    PRINT#15,"B-W:"5;0;T;S
130    PRINT#4,T","S
140    NEXT X
150    PRINT"BYE!!"
160    CLOSE 4:CLOSE 5:CLOSE 15
170    END
```

How the Program Works

Lines 20 to 40 open the three channels, channel 4 being used for the sequential file which scratches any previous file called "INDEX", and line 50 inputs the number of entries to be made to the file. This number is stored as the first character in the sequential file "INDEX". The loop in lines 60 to 140 inputs file data from the keyboard, allocates a block with the B–A command, writes the data to the block and adds the track and sector number of the block to the sequential file.

Reading data back from such random files involves using the data in the sequential file to find the track and sector at which any data item is located, and using the B–R command to access it. This is demonstrated in the next program:

```
200    OPEN 15,8,15
210    OPEN 5,8,5,"#"
220    OPEN 4,8,4,"INDEX,S,R"
230    INPUT#4,N
240    INPUT "WHICH RECORD TO
       RETRIEVE";NR
250    IF NR>N THEN PRINT"NO SUCH
       RECORD !":GOTO 240
260    FOR X=1 TO NR
270    INPUT#4,T,S
```

```
280   NEXT X
290   PRINT#15,"B-R:"5;0;T;S
300   INPUT#5,A$
310   PRINT A$
330   CLOSE 4:CLOSE 5:CLOSE 15
340   END
```

If you run these two programs, and then examine the directory of the disk, you'll see that there is no directory entry for the random file, but that the number of blocks free is reduced by the number of entries you specified in the first program. After you have experimented you can free the blocks taken up by the random files using the VALIDATE command.

The Buffer Pointer

One drawback of the technique in the programs above is that it is inefficient - an entire block would be used to store only one character. You can increase the efficiency of the method by storing more than one data item in a block. To make this possible, the DOS keeps a count of the number of characters written to the buffer during a random file operation - it is kown as the *buffer pointer*. Each time you create a random file data block, the buffer pointer is stored on the disk with the data.

You can use the buffer pointer to divide blocks into fields and so make more efficient use of the disk space.

The buffer pointer can be set to any position within a block with the BUFFER-POINTER command (abbreviated as B-P), its syntax is:

```
PRINT#15,"B-P:"Chan;Position
```

This program shows how you can use the B-P command to subdivide a block into fields.

```
1      REM B-P WRITE
10     PRINT"{CLS}":OPEN 15,8,15
20     OPEN 5,8,5,"#"
30     OPEN 4,8,4,"B-P INDEX,S,W"
40     FOR P=1 TO 220 STEP 20
45     READ D$
50     PRINT#15,"B-P:"5;P
60     PRINT#5,D$","P
70     NEXT P
80     T=1:S=1
90     PRINT#15,"B-A:"0;T;S
100    INPUT#15,A,B$,C,D
110    IF A=65 THEN T=C:S=D:GOTO 90
120    PRINT#4,T","S
130    PRINT#15,"B-W:"5;0;T;S
140    CLOSE4:CLOSE5:CLOSE15
150    END
160    DATA ZERO,ONE,TWO,THREE,FOUR,
       FIVE,SIX,SEVEN,EIGHT,NINE,TEN
```

How the Program Works

Three channels are opened to the disk and the loop between lines 40 and 70 creates each field of the record in the buffer, the buffer pointer being incremented by 20 for each item in line 50. When all eleven data items are in the buffer in positions specified by line 50, a disk block is allocated in the usual way, the track and sector recorded in the indexing sequential file and the data written to the disk by line 130.

The next program allows you to inspect the random file just created, by using the buffer pointer to select a single data item from the buffer.

```
1      REM B-P READ DEMO
10     PRINT"{CLS}"
```

```
20    OPEN 15,8,15
30    OPEN 5,8,5,"#"
40    OPEN 4,8,4,"B-P INDEX,S,R"
50    INPUT"FIELD TO VIEW (0-10)";R
60    IF R<0 OR R>10 THEN 30
70    BP=R*20+1
80    INPUT#4,T,S
90    PRINT#15,"B-R:"5;0;T;S
100   PRINT#15,"B-P:"5;BP
110   INPUT#5,A$,N
120   PRINT"{CLS}FIELD"R"CONTAINS"A$
130   PRINT "AND STARTS AT BYTE"N
140   CLOSE4:CLOSE5:CLOSE15
150   END
```

How the Program Works

The logic of the program is the reverse of that in the B-P read example. The buffer pointer position is calculated from the entered field number, the track and sector of the block are read from the sequential file by line 80, and the buffer filled with the contents of that block by line 90. The buffer pointer is set to the required data item by line 100 and the contents of the field read from the buffer by line 110.

There are two further commands which are variations of the BLOCK-READ and BLOCK-WRITE commands covered in this section. Called USER 1 and USER 2 these commands allow you to deal with the contents of a complete block regardless of the buffer pointer.

USER 1 and USER 2 (abbreviated as U1 or UA and U2 or UB) are two of a set of commands used in machine code applications which are discussed later in this chapter.

The following program is an example of the use of relative file handling techniques, combined with sequential files.

HOME BASE

```
1      REM *************
2      REM *           *
3      REM * HOME BASE *
4      REM *           *
5      REM *************
6      REM
7      REM REQUIRES 1541 DISK DRIVE
8      REM
9      PRINT"{CLS}"
10     GOSUB 10000
20     GOSUB 240
40     PRINT TAB(13);"{RVS} HOME BASE
       {ROF}{CD}{CD}{CD}{CD}"
50     PRINT TAB(9)"{RVS}1{ROF}
       CREATE NEW FILE{CD}"
60     PRINT TAB(9)"{RVS}2{ROF}
       ENTER RECORD{CD}"
70     PRINT TAB(9)"{RVS}3{ROF}
       SEARCH FOR RECORD{CD}"
80     PRINT TAB(9)"{RVS}4{ROF}
       EXIT{CD}{CD}{CD}{CD}{CD}"
100    PRINT L$
110    PRINT TAB(10);"{DKGRY}ENTER
       CHOICE {RVS}1{ROF} TO
       {RVS}4{ROF}{GRN}";
120    GET A$:IF A$="" THEN 120
130    IF A$<"1" OR A$>"4" THEN 120
135    M$=BK$:GOSUB 300
140    ON ASC(A$)-48 GOSUB 1000,4000,
       5000,6000
150    GOTO 20
199    REM READ ERROR CHANNEL
200    OPEN 15,8,15
210    INPUT#15,A$,B$,C$,D$
220    CLOSE 15
```

```
230   RETURN
239   REM CREATE SCREEN LAYOUT
240   PRINT"{GRN}{CLS}{ROF}CHR$
      (169){14 SPACES}CHR$(223)
      {RVS}{24 SPACES}"
250   PRINT"{16 SPACES}{{RVS}{24
      SPACES}{ROF}";
260   PRINTTAB(39);CHR$(223)
270   RETURN
299   REM DISPLAY M$ ON STATUS LINE
300   PRINT"{HOM}{ 23 * CD}";SPC((
      40-LEN(M$))/2);M$;"{GRN}":IF
      NB=1 THEN 330
310   FOR N=1 TO NU:PRINT"{CU}";:
      NEXT N
320   PRINT BK$"{CU}"
330   NU=0:RETURN
349   REM DRAW HORIZ LINE
350   PRINT"{HOM}{21 * CD}";L$;
      "{HOM}{CD}{CD}"
360   RETURN
993   REM
994   REM **********************
995   REM *                    *
996   REM * CREATE A NEW FILE  *
997   REM *                    *
998   REM **********************
999   REM
1000  GOSUB 240
1010  PRINT TAB(10);"{RVS} CREATE A
      NEW FILE {ROF}{CD}{CD}
      {CD}{CD}"
1020  INPUT" FILE NAME (<15 CHARS)";
      FT$
1030  IF LEN(FT$)>14 THEN 1000
1040  IF FT$ = "" THEN RETURN
1050  PRINT:PRINT:INPUT" HOW MANY
      FIELDS/RECORD (<11)";NF
1060  IF NF=0 OR NF>10 THEN
      PRINT"{CU}{CU}{CU}":GOTO 1050
1080  FOR Z=1 TO NF
```

```
1090  GOSUB 240:GOSUB350:PRINT
      TAB(10);"{RVS} CREATE A NEW
      FILE{ROF}{CD}{CD}{CD}{CD}"
1100  PRINT "ENTER NAME OF
      FIELD"Z;:INPUT N$(Z):
      PRINT:PRINT
1105  IF N$(Z)="" THEN RETURN
1110  PRINT"ENTER LENGTH OF
      FIELD"Z;:INPUT FL(Z)
1120  RL=RL+FL(Z)
1130  IF RL>254THEN RL=RL-FL(Z):M$=
      "{RED}RECORD LENGTH EXCEEDED":
      NU=12:GOSUB300:GOTO1110
1140  NEXT Z
1150  GOSUB 240
1160  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
1170  PRINT TAB(10);"{CD}{CD}{RVS}
      CREATE A NEW FILE {ROF}{CD}
      {CD}{CD}"
1180  PRINT"   #"TAB(6);"NAME";
      TAB(25);"LENGTH{CD}"
1190  FOR Z=1 TO NF
1200  PRINTTAB(2);Z;TAB(6);N$(Z);
      TAB(25);FL(Z)
1210  NEXT Z:GOSUB 350
1220  M$="{DKGRY}   {ROF}PRESS
      {RVS}Y{ROF} TO CONTINUE
      {RVS}N{ROF} TO REJECT":
      NU=5:GOSUB 300
1230  GET YN$:IF YN$="" THEN 1230
1240  IF YN$="N" THEN RETURN
1250  OPEN 2,8,2,FT$+",L,"+CHR$(RL)
1255  CLOSE 2
1260  OPEN 4,8,4,"@0:INDEX,S,W"
1270  PRINT#4,FT$
1275  FR=1
1280  PRINT#4,FR
1290  PRINT#4,NF
1300  PRINT#4,RL
1310  FOR Z=1 TO NF
```

```
1320  PRINT#4,N$(Z)",","FL(Z)","
1330  NEXT Z
1340  CLOSE 4
1350  RETURN
3993  REM
3994  REM ********************
3995  REM *                  *
3996  REM * ENTER NEW RECORD *
3997  REM *                  *
3998  REM ********************
3999  REM
4000  IF LEN(FT$)<>0 THEN 4100
4010  OPEN 4,8,4,"INDEX,S,R"
4020  GOSUB 200
4030  IF B$="OK" THEN CLOSE 4:GOTO
      4060
4040  NU=5: M$="{RED}"+B$:GOSUB 300
      :FOR Q=0 TO 1000:NEXT Q
4050  CLOSE 4:RETURN
4060  OPEN 4,8,4,"INDEX,S,R":
      INPUT#4,FT$,FR,NF,RL
4070  FOR Z=1 TO NF
4080  INPUT#4,N$(Z),FL(Z):NEXT Z
4090  CLOSE 4
4100  FOR Z=1 TO NF:GOSUB 240:GOSUB
      350
4110  PRINT "{HOM}{CD}{CR}{DKGRY}"
      FT$"{GRN}"
4120  PRINT
      TAB(11);"{CD}{CD}{RVS}ENTER A
      RECORD {ROF}{CD}{CD}{CD}{CD}"
4130  M$="MAX FIELD LENGTH ="+STR$
      (FL(Z)):NU=16:GOSUB 300
4140  PRINT N$(Z);
4150  INPUT R$(Z)
4160  IF LEN(R$(Z))<FL(Z) THEN 4220
4170  NB=1:M$="{RED}FIELD LENGTH
      EXCEEDED":GOSUB 300:NB=0
4180  FOR T=0 TO 400:NEXT T:GOTO
      4110
4220  NEXT Z
```

```
4230  FOR Z=1 TO NF:GOSUB 240:GOSUB
      350
4240  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
4260  PRINT TAB(11);"{CD}{CD}{RVS}
      ENTER A RECORD {ROF}{CD}{CD}
      {CD}{CD}"
4270  FOR Z=1 TO NF
4280  PRINT N$(Z)" "R$(Z)
4290  NEXT Z
4300  NB=1:M$="{DKGRY}PRESS {RVS}Y
      {ROF} TO ACCEPT {RVS}N{ROF} TO
      REJECT":GOSUB 300:NB=0
4310  GET YN$:IF YN$="" THEN 4310
4320  IF YN$="N" THEN 4510
4330  IF YN$<>"Y" THEN 4310
4340  OPEN 2,8,2,FT$
4350  OPEN 15,8,15:PO=1
4355  FOR Z=1 TO NF
4360  HI=INT(FR/256):LO=FR-HI*256
4370  PO=PO+FL(Z-1)
4380  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO)
4390  PRINT#2,R$(Z)
4400  NEXT Z:FR=FR+1
4410  CLOSE 2:CLOSE 15
4420  OPEN 4,8,4,"@0:INDEX,S,W"
4430  PRINT#4,FT$
4440  PRINT#4,FR
4450  PRINT#4,NF
4460  PRINT#4,RL
4470  FOR Z=1 TO NF
4480  PRINT#4,N$(Z)","FL(Z)","
4490  NEXT Z
4500  CLOSE 4
4510  T$="":FOR Z=1 TO NF:R$(Z)="":
      NEXT Z:RETURN
4993  REM
4994  REM **********************
4995  REM *                    *
4996  REM * SEARCH FOR RECORD *
```

```
4997  REM *                    *
4998  REM **********************
4999  REM
5000  SE$="":Y=0:PO=0:IF LEN(FT$)<>0
      THEN 5100
5010  OPEN 4,8,4,"INDEX,S,R"
5020  GOSUB 200
5030  IF B$="OK" THEN CLOSE 4:GOTO
      5060
5040  NU=5: M$="{RED}"+B$:GOSUB 300
      :FOR Q=0 TO 1000:NEXT Q
5050  CLOSE 4:RETURN
5060  OPEN 4,8,4,"INDEX,S,R":
      INPUT#4,FT$,FR,NF,RL
5070  FOR Z=1 TO NF
5080  INPUT#4,N$(Z),FL(Z):NEXT Z
5090  CLOSE 4
5100  GOSUB 240:GOSUB 350
5110  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
5120  PRINT TAB(10);"{CD}{CD}{RVS}
      SEARCH FOR RECORD {ROF}{CD}
      {CD}{CD}{CD}"
5130  FOR Z=1 TO NF
5140  PRINT "{RVS}"CHR$(Z+64)"{ROF}
      ";N$(Z)
5150  NEXT Z
5160  M$="{DKGRY}   SELECT FIELD FOR
      SEARCH {RVS}A{ROF} TO "+
      "{RVS}"+CHR$(NF+64)+"{ROF}"
5170  NB=1:GOSUB 300:NB=0
5180  GET A$:IF A$="" THEN 5180
5185  IF A$<"A" OR A$>CHR$(NF+64)
      THEN 5180
5190  F=ASC(A$)-64:GOSUB 240:GOSUB
      350:T$=""
5200  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
5210  PRINT TAB(10);"{CD}{CD}{RVS}
      SEARCH FOR RECORD {ROF}{CD}
      {CD}{CD}{CD}"
```

```
5220  M$=BK$:GOSUB 300: M$="{DKGRY}
      ENTER CONTENTS OF FIELD FOR
      SEARCH":NU=16:GOSUB 300
5225  PRINT N$(F);" ";
5230  INPUT SE$
5260  M$=BK$:GOSUB 300:M$="SEARCHING
      FOR RECORD":NB=1:GOSUB 300:
      NB=0
5300  OPEN 2,8,2,FT$
5310  OPEN 15,8,15
5315  FOR Q=0TOF-1:PO=PO+FL(Q):
      NEXT:PO=PO+1
5320  FOR Z=1 TO FR-1
5330  HI=INT(Z/256):LO=Z-HI*256
5350  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO)
5360  INPUT#2,T$
5399  REM WILD CARD SEARCH
5400  IF RIGHT$(SE$,1)<>"*" THEN
      5410
5405  IF LEFT$(T$,LEN(SE$)-1)=LEFT$
      (SE$,LEN(SE$)-1)THEN FO(Y)=Z:
      Y=Y+1:GOTO 5415
5410  IF T$=SE$ THEN FO(Y)=Z:Y=Y+1
5415  NEXT Z
5420  M$=BK$:GOSUB 300:M$=STR$(Y)+"
      RECORDS WERE FOUND":NB=1:GOSUB
      300:NB=0
5430  FOR T=0 TO 500:NEXT T
5440  IF Y=0 THEN GOTO 5790
5499  REM DISPLAY FOUND RECORDS
5500  FOR Z=0 TO Y-1:PO=1
5505  GOSUB 240:GOSUB 350
5510  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
5511  PRINTTAB(10);"{CD}{CD}{RVS}
      SEARCH FOR RECORD {ROF}{CD}
      {CD}{CD}{CD}"
5515  F=FO(Z):HI=INT(F/256):LO=F-
      HI*256
5520  FOR Q=1 TO NF
```

```
5530  PO=PO+FL(Q-1)
5540  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO)
5550  INPUT#2,R$(Q-1)
5560  PRINT BK$"{CR}{CU}{RVS}";
      CHR$(Q+64)"{ROF} "N$(Q);"
      ";R$(Q-1)
5565  NEXT Q
5570  IF Y=1 THEN 5700
5580  IF Z=Y-1 THEN Z=-1
5600  M$=BK$:GOSUB 300:NB=1:
      M$="{DKGRY}    SHOW NEXT
      RECORD {RVS}Y{ROF} OR
      {RVS}N{ROF}?":GOSUB 300:NB=0
5610  GET YN$:IF YN$="" THEN 5610
5620  IF YN$="N" THEN AR=Z:Z=Y-
      1:GOTO 5700
5630  IF YN$<>"Y" THEN 5610
5640  NEXT Z
5700  M$=BK$:GOSUB300:NB=1:M$=
      "{DKGRY}      {RVS}A{ROF}MEND
      {RVS}DEL{ROF}ETE OR {RVS}
      RETURN{ROF}?":GOSUB 300:NB=0
5710  GET YN$:IF YN$="" THEN 5710
5720  IF YN$="A" THEN 5800
5730  IF YN$=CHR$(13) THEN 5780
5740  IF YN$<>CHR$(20) THEN 5710
5750  M$=BK$:GOSUB300:M$="{DKGRY}
       {RVS}  ARE YOU SURE ?  {ROF}"
      :NB=1:GOSUB300:NB=0
5755  GET YN$:IF YN$="" THEN 5755
5760  IF YN$<>"Y" THEN 5700
5770  PO=1:FOR J=1 TO NF:FOR K=1 TO
      FL(J)-1:DE$=DE$+" ":NEXT K:
      PO=PO+FL(J-1)
5775  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO):PRINT#2,DE$:D
      E$="":NEXT J
5780  Y=0:PO=0:CLOSE2:CLOSE15:RETURN
5799  REM AMEND RECORD
```

```
5800  PRINT"{HOM}{CD}{CD}{CD}{CD}
      "BK$"{CR}{CU}"TAB(14)"{RVS}AME
      ND RECORD{ROF}"
5805  M$=BK$:GOSUB 300:M$="{DKGRY}
      {RVS}A{ROF}MEND WHICH FIELD?"
      :NB=1:GOSUB 300:NB=0
5810  GET YN$:IF YN$="" THEN 5810
5820  IF YN$<"A" OR YN$> CHR$(NF+64)
      THEN 5810
5830  GOSUB 240:GOSUB 350:PRINT"
      {HOM}{CD}{CR}{DKGRY}"FT$"{GRN}
      "
5840  PRINT TAB(14);"{CD}{CD}{RVS}
      AMEND RECORD {ROF}{CD}{CD}"
5845  Q=ASC(YN$)-64
5850  PRINT N$(Q);SPC(2);R$(Q-1)
5860  M$=BK$:GOSUB 300:M$="MAX FIELD
      LENGTH ="+STR$(FL(Q)):NB=1:
      GOSUB 300:NB=0
5870  PRINT"{HOM}{10 * CD}"N$(Q);
      SPC(2);:INPUT R$(Q-1)
5880  GOSUB 240:GOSUB 350
5890  PRINT "{HOM}{CD}{CR}{DKGRY}
      "FT$"{GRN}"
5900  PRINT TAB(14);"{CD}{CD}{RVS}
      AMEND RECORD {ROF}{CD}{CD}"
5910  FOR Y=1 TO NF
5920  PRINT N$(Y);SPC(1);R$(Y-1):
      NEXT Y
5930  M$="{RVS}{DKGRY}Y{ROF} TO
      ACCEPT OR {RVS}N{ROF} TO
      REJECT":NB=1:GOSUB 300:NB=0
5940  GET YN$:IF YN$="" THEN 5940
5950  IF YN$="N" THEN RETURN
5960  IF YN$<>"Y" THEN 5940
5970  PO=1:FOR Y=0 TO NF-1:PO=PO+
      FL(Y)
5980  PRINT#15,"P"CHR$(2)CHR$(LO)
      CHR$(HI)CHR$(PO)
5990  PRINT#2,R$(Y):NEXT Y:CLOSE2:
      CLOSE15
```

```
5995  RETURN
6000  PRINT"{CLS}BYE!!":END
9993  REM
9994  REM *******************
9995  REM *                 *
9996  REM * SET UP ROUTINES *
9997  REM *                 *
9998  REM *******************
9999  REM
10000 POKE 53280,5:POKE 53281,15
10010 FOR Z=0 TO 39:L$=L$+CHR$(192):
      NEXT Z
10020 FOR Z=0 TO 38:BK$=BK$+
      CHR$(32):NEXT Z
10100 RETURN
```

How to Use HOME BASE

After loading the program, remove the program disk from the drive and insert the file disk. The database can extend over an entire diskette, so it is best to reserve a disk for each file you wish to keep. If you want to create a new file, insert a blank, formatted disk.

Main Menu

The main menu comprises four options:

1 CREATE NEW FILE

2 ENTER A RECORD

3 SEARCH FOR RECORD

4 EXIT

At the foot of the screen is the status line – this is used throughout the program to request inputs (dark grey text), display error messages (red text) and give information about the current operation (green text). When the program is run the status line will prompt for a choice from the main menu – you will not be able to enter or search for a record until a file has been created on the disk.

1 Create New File

The program will prompt you to enter the name of the file and the number of fields it is to contain. For each field you must enter a name and the maximum number of characters. The total number of characters in a file may not exceed 254.

The file will be created and control will return to the main menu – the file name will be displayed in the top left hand corner of the screen. You can now enter data into the file.

2 Enter a Record

Each field in a record is filled in turn – the status line will display the maximum number of characters allowed in each field. If this number is exceeded an error message will be displayed and you will be asked to re-enter the data. After all the fields have been filled in this way the entire record is redisplayed, at which point you can reject it or accept it. If you accept, the data is stored on the disk, and in both cases the main menu is redisplayed.

3 Search For a Record

The names of the fields will be displayed and you will be asked for the name and contents of the field to be used in the search. A 'wild card' facility is incorporated, so that entering a number of characters followed by an asterisk (*) will locate every record in which the specified field contains those characters. The disk will be searched for the specified field contents and if more than one record is found, you may display them all by responding 'Y' to the SHOW NEXT FIELD prompt. If you press 'N', or only one record was found, you will be given the option to amend or delete the record or return to the main menu.

Amending a Record

The field to be amended is selected by reference letter and the new data entered. The amended record will be displayed for you to accept or reject. Pressing 'Y' will save the amended record and the main menu will be redisplayed.

Deleting a Record

If you opt to delete a record and press 'Y' in response to the ARE YOU SURE? prompt, the record will be deleted and the main menu redisplayed.

How the Program Works

HOME BASE uses a single relative file to store data, and a sequential file called INDEX which contains information about the names and sizes of the fields in a record, length of a record and the number of the next free record. The arrangement of data in the INDEX file is as follows:

FT$	Name of relative file
FR	Number of the next free record
NF	Number of fields per record
RL	Length of a record
N$(1)	Name of field 1
FL(1)	Length of field 1
N$(2)	Name of field 2
FL(2)	Length of field 2
:	:
etc	etc

INDEX is created at the start of the program when a relative file is set up, and updated each time a new record is added to the relative file.

Lines 40 – 150 display the main menu and input a selection from it.

A number of subroutines in lines 200 – 360 perform frequently used operations such as maintaining the status line, reading the error channel, etc.

Lines 1000 – 1350 create a new relative file, named FT$, containing NF fields. The name and length of each field is entered, and if the format is accepted, a relative file having records of length RL is created, and the INDEX file set up.

Lines 4000 – 4510 handle the entry of a new record. If no file is currently loaded, the disk is checked and data from the INDEX file is loaded. If no INDEX file is found on the disk, an error message is generated and the main menu redisplayed. The data for each field is entered and stored in array R$( ). If the new record is accepted, it is stored on the disk at record FR, the INDEX file is updated and the main menu redisplayed.

Lines 5000 – 6000 provide the search facility. The disk is checked for the existence of an INDEX file if no file is currently loaded, and data is read in from INDEX. The field names contained in array N$( ) are displayed and the field for the search, F, is selected from them. The contents of the field for the search, SE$, are input and the appropriate field of each record on the disk is compared with SE$. If the comparison is successful the record number of that record is stored in array FO( ).

If the last character of SE$ is an asterisk (*) the comparison only covers the number of characters in SE$, providing a 'wild card' search.

At the end of the search, the number of records displayed is Y. If no records were found a message to this effect is displayed and the main menu is redisplayed. If only one record was found, the record is displayed and control passes to line 5700, where amend and delete options are offered. If more than one record was found, pressing 'Y' in response to the 'SHOW NEXT RECORD' prompt will allow each record to be displayed in turn, continuing to cycle until 'N' is pressed, at which point control passes to line 5700.

Lines 5750 - 5799 delete the displayed record, by printing blank strings to the appropriate fields on the disk.

Lines 5800 - 5995 allow amendment of a field in the record and if the amendment is accepted, rewrite the entire record to the disk and redisplay the main menu.

Variable Use

FT$	File name
NF	Number of fields/record
N$( )	Array of field names
FL( )	Array of field lengths
RL	Record length
L$	Horizontal line
M$	Message for status line
BK$	Blank line
FR	Next free record number

R$(I) Contents of field I

Improvements to the Program

In order to make this program as generally applicable as possible (and to allow room for the other chapters in the book!), a number of functions are omitted which would make it much better for specific jobs.

For example, the free format of the display could be replaced by a fixed card format to suit your application.

The restriction of 80 characters per field is due to the BASIC **INPUT** command and could be overcome using **GET** and some form of checking routine.

Additionally, some form of hardcopy option might be useful.

When a record is deleted, it can not be re-used and disk space is wasted. One way round this would be to keep another sequential file of deleted record numbers . When a new record is created this would be checked and if there are any entries, the record number of a deleted record would be allocated to the new record.

The program is designed for just one relative file per disk, but if you used yet another sequential file as a directory there's no reason why you shouldn't keep more than one relative file per disk.

With the information provided it would be a useful and instructive exercise to modify and improve HOME BASE to suit your requirements.

MACHINE CODE AND THE 1541 DRIVE

The DOS recognises a group of commands designed to allow you to create machine code programs to run in the disk drive RAM, possibly modifying the operation of the DOS.

To use these commands requires a detailed knowledge of the DOS program and the architecture of the disk drive – information which is not freely available from the manufacturers. However the commands are briefly covered here should you find need to use them.

BLOCK-EXECUTE (B-E)

This command allows you to load machine code routines from the disk into disk drive RAM and execute them. It is similar to the B–R command except that after loading the code, the disk drive's microprocessor begins to execute it. The program must end in an RTS (ReTurn from Subroutine) instruction. The format of the command is:

```
PRINT#15,"B-E:"Ch;Dr;T;S
```

where a block of data is read from track T sector S on drive Dr into the channel Ch buffer, and execution commences at byte 0 of that buffer.

MEMORY WRITE (M-W)

M–W allows you to send data comprising machine code programs from the 64, via channel 15, into disk drive RAM. The format is:

```
PRINT#15,"M-W:"CHR$(Lo)CHR$(Hi)CHR$(N)
CHR$(A)CHR$(B)....etc
```

where a program consisting of bytes A, B etc. up to N characters is sent to RAM starting at address (256*Hi + Lo). Up to 34 bytes may be sent at a time.

MEMORY READ (M-R)

The M–R command provides a means for reading data from disk drive ROM or RAM, one byte at a time, via the error channel into the 64. The format is:

```
PRINT#15,"M-R:"CHR$(Lo)CHR$(Hi)
```

The contents of the location specified by 256*Hi + Lo can be read from channel 15 using **GET#**.

MEMORY EXECUTE (M-E)

Allows you to execute machine code programs in the disk drive memory from the address specified, until an RTS is encountered. The format is:

```
PRINT#15,"M-E:"CHR$(Lo)CHR$(Hi)
```

USER (U)

The USER command makes it possible to link to machine code programs by using a jump table set up in disk drive memory. The command is followed by an ASCII character which forms an index to the table. The characters 1 to 9 or A to J can be used.

The U1 and U2 commands perform the B–R and B–W operations mentioned earlier, but ignore the buffer pointer to operate on an entire data block. The remaining eight point to the locations given in the table opposite, which must be set up to contain the start address of the machine code programs you wish to execute.

USER	OPERATION
U1 or UA	B-R command
U2 or UB	B-W command
U3 or UC	Jump to $0500
U4 or UD	Jump to $0503
U5 or UE	Jump to $0506
U6 or UF	Jump to $0509
U7 or UG	Jump to $050C
U8 or UH	Jump to $050F
U9 or UI	Jump to $FFFA
U; or UJ	Power up Vector

The USER Command Jump Table

The fonnat of the USER command command is :

```
PRINT#15,"UN:"Ch;Dr;T;S
```

USING OTHER DISK DRIVES WITH THE 64

The 1540 Disk Drive

It is possible to use the 1540 disk drive (designed for the VIC 20) with the 64 with one change. The screen must be turned off during the loading of a program with the command:

```
   POKE 53265,11
```

When the program is loaded, turn the screen back on with the command:

```
  POKE 53265,27
```

Twin 1541 Drives

Many disk based applications are greatly enhanced by having two disk drive units. Although it is possible to link 1541 drives together and change their device number as described in the manual, it seems that some errors in the ROMs cause drives used in this way to 'hang up' at random intervals for no apparent reason. We have tried using two drives in this way for making backup disks, and were plagued with such problems, so be warned!

IEEE Drives

A major disadvantage of the 1541 drive is that data transfers between it and the 64 use a serial data bus. This means that data is transmitted one bit at a time, rather than all 8 bits in a byte being transmitted at once, as in a parallel system. For this reason the 1541 is very slow by disk drive standards as most disk drives adopt the faster (and more expensive) parallel technique.

Another problem with it is that only a relatively small amount of data can be stored on a disk because of the way in which it is formatted.

If you have an application which requires greater storage capacity or more rapid data retrieval, you can use the 64 with some of Commodore's larger (and more expensive) drives.

To do so will involve considerable expense, not only for a dual drive unit like the 8050, but also for an interface adaptor to provide the 64 with the necessary IEEE (parallel) interface.

Copying Tape Software to Disk

A major headache in upgrading from tape to disk is that your collection of programs is still on tape. Where once you would accept a long delay in loading, you soon become spoiled by the speed of disk drives, and need to transfer your software to disk. In a few cases this is simply a matter of loading the program into the 64, and saving it onto disk. However most commercial software is protected aginst copying - unfortunately this means that it is also protected against legitimate copying!

One of the most popular ways of protecting 64 tapes is to save a program in several blocks, and have a short loader program as the first program on the tape. The loader is often written in machine code, to protect against the casual pirate, but is usually quite simple - using kernal routines - to load the program from the tape. The most straightforward way to transfer such a program to disk is to find out how many parts the program is saved in and write a short BASIC loader to do the job of the machine code program on the tape. You can then load each block of program from the tape and save it to disk in the normal way, and use your loader to load the blocks back from the disk when required.

There are other ways of protecting software, and it can be quite satisfying to 'crack' this sort of problem - provided, of course, that you don't sell the results of your efforts!

CHAPTER 26

THE MPS801 PRINTER

A printer is a useful addition to any microcomputer system, providing program listings and hard copy of text and graphics generated by the computer.

There are several different types of printer and many variations of each type are available for the Commodore 64.

The MPS801 printer (formerly the 1525)is the most popular choice for ·the home user since it is both cheap and versatile - both character and graphic information can be obtained.

How the Printer Works

The MPS801 is a *dot matrix printer* which uses a set of small pins arranged in a matrix to strike the ribbon and make small dots on the paper. Each dot corresponds to one pin.

The printer receives data from the 64 on the serial bus and its microprocessor interprets the data - sorting printing characters from control characters. For each printing character the processor causes the hammer to strike the appropriate pins to create the dot pattern for that character, moves the print head along the carriage and prints the next character. At the end of a line, the print head returns to the lefthand side and the paper is scrolled ready for the next line of data.

Connecting the Printer

Before connecting the MPS801 to the 64, turn off the computer and printer. Plug one end of the serial interface cable into the 64 and the other into the rear of the printer. If you have a 1541 disk drive connect the disk drive to the 64 and the printer to the spare socket on the drive. In cases where more than one peripheral is connected to the bus, problems may arise in addressing either device. This normally requires you to switch off all peripherals, then switch them on again before they can be used. This is due to a fault in the operating system, about which little can be done.

Testing

The MPS801 is provided with a test facility – insert some paper and move the three position switch at the rear of the printer to 'T'. The printer will print the complete 64 character set and continue to do so until you switch off or move the switch away from the 'T' position.

Using the Printer

Like other peripherals, communication between the printer and the 64 is achieved by opening a channel to the device. The channel is opened using the BASIC **OPEN** command:

```
    OPEN LFn,Dn,SA
```

where:

LFn is the logical file number (any number from 0 to 255) which is used to reference the channel.

Dn is the device number which is either 4 or 5 depending upon the position of the three position switch at the rear of the printer.

SA is the secondary address, which acts like the CBM and SHIFT keys on the 64 by toggling between upper and lower case mode, and upper case and graphics mode printouts.

To select upper case and graphics mode the secondary address is set to 0. If no secondary address is specified in the OPEN command, it defaults to 0. A secondary address of 7 selects upper and lower case printouts.

Once the channel is opened, characters are transmitted to the printer using the **PRINT#** command as in the following example:

```
10     OPEN 4,4
20     FOR Z=1 TO 26
30     PRINT Z,CHR$(Z+64)
40     PRINT#4,Z,CHR$(Z+64)
50     NEXT Z
60     CLOSE 4
70     END
```

Notice that what is displayed on the screen is mimicked by the printer, and that data can be formatted on the printer in the same way as it can on the screen. A comma moves the print head into the next 'column' before printing and a carriage return is issued at the end of each **PRINT#** command. The carriage return can be suppressed by adding a semicolon at the end of the **PRINT#** command. So, if you change line 20 in the previous program to:

```
20     FOR Z=33 TO 112:PRINT CHR$(Z);
       :NEXT Z
```

and delete lines 30, 40 and 50, the full 80 column capability of the printer will be demonstrated.

As with **PRINT**ing to the display, the **TAB** and **SPC** commands work on the printer.

Another way of obtaining copy on the printer is to use the **CMD** command.

The **CMD** command transfers the output from the screen to the specified channel. Its syntax is:

```
CMD LFn
```

where the logical file number must be the same as the one specified in the **OPEN** command.

After issuing the **CMD** command, all data normally output to the 64's screen is directed to the printer. This is the way to obtain program listings:

```
OPEN 4,4

CMD 4

LIST
```

After a listing has been obtained, the output from the 64 is still directed to the printer, and to return to normal you must type:

```
PRINT# LFn

CLOSE LFn
```

This will clear any data remaining in the printer buffer and close the channel, returning output to the TV screen.

It is possible to use the **OPEN**, **PRINT#**, **CLOSE** and **CMD** commands to output data to the printer under program control as this program shows:

```
10    REM PRINTING UNDER PROG
      CONTROL
20    PRINT"{CLS}"
30    INPUT"NAME ";N$
40    OPEN 4,4
50    PRINT#4,"BIG BROTHER IS
      WATCHING YOU "N$"!!!"
60    CLOSE 4
70    END
```

PRINTING MODES

The MPS801 operates in various printing modes selected by control characters in the data sent to the printer. The table below shows the print modes and the control characters used to select them.

The characters in the above table are sent to the printer in **PRINT#** commands and interpreted by the printer as control characters. The mode so selected will continue to be the printing mode until a further control character is detected by the printer.

Cursor Up and Cursor Down

The characters 'cursor up' (CHR$(145)) and 'cursor down' (CHR$(17)) select which of the two 128 character sets is to be used for printing. The two fonts are the same as those selected on the 64 by holding down the SHIFT key and pressing the CBM key. The following program illustrates the difference:

```
10    REM CRSR UP/CRSR DOWN
20    FOR Z=65 TO 90
30    A$=A$+CHR$(Z)
```

CODE	FUNCTION
145	Cursor up mode
17	Cursor down mode
8	Graphics mode
16	Tab print head
18	Reverse on
146	Reverse off
14	Double width mode
10	Line feed
13	Carriage return
27, 16	Specify dot address
26	Repeat graphic data

Printer Control Codes

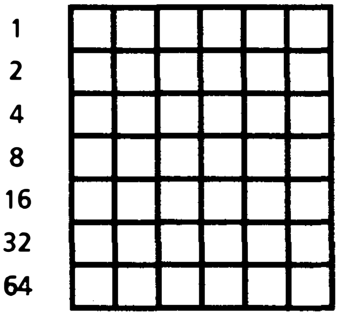
```
40     NEXT Z
50     OPEN 4,4:PRINT#4,A$
60     CLOSE 4
70     OPEN 4,4,7
80     PRINT#4,A$
90     CLOSE 4
100    END
```

NOTE: A secondary address of 7 must be specified in the **OPEN** command for cursor down mode to work.

Graphics Mode

Graphics mode (CHR$(8)) allows the printing of user defined characters.

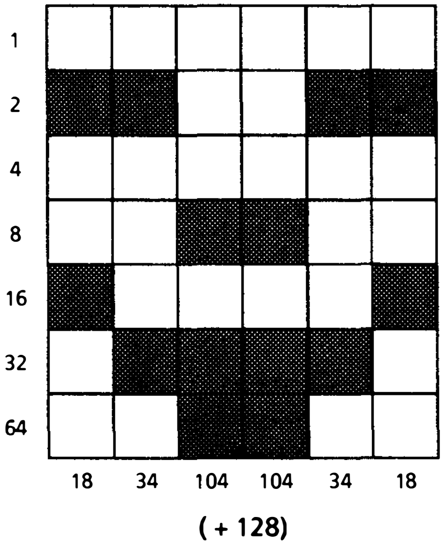
Characters are designed on a 6 by 7 grid like this:



User Defined Character Grid

Notice that the grid is not the same as the one used for defining characters on the 64!

Characters are created by placing dots on the grid where a dot should appear on the paper as in the diagram below:



A User Defined Character

To obtain the **DATA** statement defining the character take one column of the grid at a time and add together the values of the rows in that column containing a dot. Add 128 to the result and repeat the operation for each of the columns.

The data for the character is sent to the printer as a string, following a CHR$(8). The next program will print the character on the grid opposite:

```
10     REM USER DEFINED CHAR PROG
20     FOR Z=0 TO 5
30     READ D
40     A$=A$+CHR$(D)
50     NEXT Z
60     OPEN 4,4
70     PRINT#4,CHR$(8)A$
80     CLOSE 4
90     END
100    DATA 146,162,232,232,162,146
```

You can use the graphics facility to draw patterns on the printer as demonstrated by this program:

```
10     OPEN 4,4
20     FOR N=0 TO 30
30     FOR R=0 TO 6
40     D$=CHR$((2↑R)+128)
50     PRINT#4,CHR$(8)D$
60     NEXT R
70     FOR R=6 TO 0 STEP -1
80     D$=CHR$((2↑R)+128)
90     PRINT#4,CHR$(8)D$;
100    NEXT R
110    NEXT N
120    CLOSE 4
```

Tab Print Head

You can move the printer head under program control to any position using CHR$(16). A two digit number following the CHR$(16) specifies the position between 0 and 79. The next program shows this control character in use.

```
10     REM PRINT HEAD DEMO
20     OPEN 4,4
30     FOR I=10 TO 70 STEP 5
40     P$=STR$(I)
50     H$=MID$(P$,2,1)
60     L$=RIGHT$(P$,1)
70     PRINT#4,CHR$(16)H$L$"POS"P$
80     NEXT
90     CLOSE 4
```

Reverse On / Reverse Off

Reverse text can be printed by preceding the text by CHR$(18). Turn off reverse text with CHR$(146).

```
10     OPEN 4,4
20     A$="REVERSED TEXT"
30     FOR I=1 TO LEN(A$)
40     R=1-R
50     IF R=0 THEN R$=CHR$(146):GOTO
       70
60     R$=CHR$(18)
70     PRINT#4,R$;MID$(A$,I,1);
80     NEXT I
90     CLOSE 4
100    END
```

Double Width Characters

You can highlight important information by printing the text using double width characters. Double width characters have the same dot pattern as standard characters but being twice as wide, only half as many will fit on a line. To print double width characters, precede the text with CHR$(14).

```
10    REM DOUBLE WIDTH CHARACTERS
20    OPEN 4,4
30    PRINT#4,"80 COLUMNS OF TEXT
      THIS SIZE"
40    PRINT#4,CHR$(14)"OR 40 COLUMNS
      OF THIS SIZE"
50    CLOSE 4:END
```

Line Feeds and Carriage Returns

You can control the printer to some extent by sending line feed characters (CHR$(10)) and Carriage return characters (CHR$(13)). These will allow you to create blank lines in your printouts:

```
10    OPEN 4,4
20    PRINT#4,"LINE 1"
30    FOR Z=0 TO 9
40    PRINT#4,CHR$(10)
50    NEXT Z
60    PRINT#4,"LINE 12"
```

Specify Dot Address

The span of the print head can be divided up into 480 columns each one dot wide. CHR$(27) CHR$(16) allows you to select one of these positions to commence printing data. Since there are 480 possible positions the dot address is specified in two bytes which follow the CHR$(27)CHR$(16) like this:

```
PRINT#4,CHR$(27)CHR$(16)CHR$(1)
CHR$(44)
```

specifies dot address 1*256+44=300 and moves the print head to that position.

Repeat Graphic Data

CHR$(26) allows you to repeat a byte of data in graphics mode a specified number of times. For example, if you wanted to draw a thick line across the paper, you would need to repeat a single vertical line many times.

The following program shows how you might do this, using each pin of the print head to create a vertical line character like this: |

```
10      OPEN 4,4
20      PRINT#4,CHR$(8)CHR$(26)CHR$
        (100)CHR$(255)
30      CLOSE 4
```

The single character CHR$(255) is repeated at each dot address across the paper, for 100 times – specified by CHR$(100) – in this example. You could repeat for any number of times between 0 and 255, but to cover the entire width of the paper would require two commands.

You can use CHR$(26) to underline headings as the next program shows:

```
10      OPEN 4,4:PRINT#4,CHR$(15)
20      PRINT#4,TAB(30)"A CENTRED
        HEADING"
30      PRINT#4,TAB(29)CHR$(8)CHR$
        (26)CHR$(114)CHR$(129)
40      CLOSE 4:END
```

With a different choice of character you could make the underlining more bold.

Screen Dump

The printer manual contains a program to output the contents of a standard screen to the printer, but it will not print reverse video characters. The following program rectifies that omission and performs the same operation in rather less space.

```
60000 OPEN 4,4:PRINT#4,CHR$(15)
60010 FOR A=1024 TO 2023 STEP 40
60020 FOR X=0 TO 39
60040 D=PEEK(X+A)
60050 IF D>127 THEN D=D-128:
      R$=CHR$(18):O$=CHR$(146)
60060 P=D-((D<32 OR D>95)*64)-((D>63
      AND D<96)*32)
60070 P$=P$+R$+CHR$(P)+O$
60080 R$="":O$=""
60090 NEXT X
60100 PRINT#4,P$
60110 P$="":NEXT A
60120 CLOSE 4:END
```

In order to obtain hardcopy of screens containing upper and lower case characters, line 60000 must be changed to :

```
60000 OPEN 4,4,7:PRINT#4,CHR$(145)
```

Using the Printer with Machine Code

There are occasions when you might need to control the printer from within a machine code program - for example in a word processing package where you want to dump the contents of an area of memory onto the printer.

This can be achieved quite easily with the use of several of the kernal routines. A full listing of these is given in Appendix 11 and an explanation of them in Chapter 18.

The following program will output the contents of a small block of memory to the printer. The last character in the block is CHR$(13) - a carriage return which causes the printer buffer to be emptied – ensuring that all the data is printed.

```
 10 !*****************
 20 !* PRINTER DEMO *
 30 !*****************
 40 !
 50 ORG $C000                 start address
 60 !
 70          LDA #0
 80          JSR SETNAM 0 = no filename
 90          LDA #4     LFn = 4
100          TAX        Device No. = 4
110          LDY #255   No Secondary Add
120          JSR SETLFS Set up Logical File
130          JSR OPEN   Open file
140          LDX #4     LFn = 4
150          JSR CHKOUT Open channel for
                        output
160          LDX #0     Initialise counter
170 OP       LDA TAB,X  Get character from
                        table
180          JSR CHROUT Output to printer
190          INX             Increment counter
200          CPX #34    Last item in table?
210          BNE OP     No, then get next
                        character
220          JSR CLALL  All done so close all
                        channels
230          RTS        Back to BASIC
240 TAB      BYT 80,82,73,78,84,69,
             68,32,70,82,79,77,32,77,
             65,67,72,73,78,69
250          BYT 32,67,79,68,69,32,
             80,82,79,71,82,65,77,13
270 SETNAM = 65469
280 SETLFS = 65466
290 OPEN   = 65472
300 CHKOUT = 65481
310 CHROUT = 65490
```

```
320 CLALL  = 65511
```

The program will output the data in the table to the printer - displaying the message 'PRINTED FROM MACHINE CODE PROGRAM'. The data in the table could contain control characters to change the output from the printer.

The program could be easily modified to dump the contents of any area of memory to the printer by increasing the size of the loop labelled OP.

The following is a BASIC program to load the machine code. The code is relocatable - it can be placed anywhere in memory. To change the start address you must change the value of S in line 10.

```
1      REM LOADER FOR PRINTER M/C
10     S=49152
20     FOR Z=S TO S+71
30     READ D:POKE Z,D
40     NEXT Z
50     END
20000 DATA 169,0,32,189,255,169,4,
       170,160,255,32,186,255,32,192,
       255
20010 DATA 162,4,32,201,255,162,0,
       189,38,192,32,210,255,232,224,
       34
20020 DATA 208,245,32,231,255,96,80,
       82,73,78,84,69,68,32,70,82
20030 DATA 79,77,32,77,65,67,72,73,
       78,69,32,67,79,68,69,32
20040 DATA 80,82,79,71,82,65,77,13
```

THE COMMODORE 64 OMNIBUS

APPENDICES

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, 18

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 TOKENS & ABBREVIATIONS

COMMAND	ABBREVIATION	DEC. TOKEN	HEX TOKEN
ABS	aB	182	B6
AND	aN	175	AF
ASC	aS	198	C6
ATN	aT	193	C1
CHR$	cH	199	C7
CLOSE	clO	160	A0
CLR	cL	156	9C
CMD	cM	157	9D
CONT	cO	154	9A
COS	---	190	BE
DATA	dA	131	83
DEF	dE	150	96
DIM	dI	134	86
END	eN	128	80
EXP	eX	189	BD
FN	---	165	A5
FOR	fO	129	81
FRE	fR	184	B8
GET	gE	161	A1
GET#	---	---	---
GOSUB	goS	141	8D
GOTO	gO	137	89
IF	---	139	8B

INPUT	---	133	85
INPUT#	iN	132	84
INT	---	181	B5
LEFT$	leF	200	C8
LEN	---	195	C3
LET	lE	136	88
LIST	lI	155	9B
LOAD	lO	147	93
LOG	---	188	BC
MID$	mI	202	CA
NEW	---	162	A2
NEXT	nE	130	82
NOT	nO	168	A8
ON	---	145	91
OPEN	oP	159	9F
OR	---	176	B0
PEEK	pE	194	C1
POKE	pO	151	97
POS	---	185	B9
PRINT	?	153	99
PRINT#	pR	152	98
READ	rE	135	87
REM	---	143	8F
RESTORE	reS	140	8C
RETURN	reT	142	8E
RIGHT$	rI	201	C9
RND	rN	187	BB
RUN	rU	138	8A
SAVE	sA	148	94
SGN	sG	180	B4
SIN	sI	191	BF

SPC	sP	166	A6
SQR	sQ	186	BA
STATUS	ST	---	---
STEP	stE	169	A9
STOP	sT	144	90
STR$	stR	196	C4
SYS	sY	158	9E
TAB	tA	163	A3
TAN	---	192	C0
THEN	tH	167	A7
TIME	TI	---	---
TIME$	TI$	---	---
TO	---	164	A4
USR	uS	183	B7
VAL	vA	197	C5
VERIFY	vE	149	95
WAIT	wA	146	92
+	---	170	AA
−	---	171	AB
*	---	172	AC
/	---	173	AD
=	---	178	B2

NOTE: --- indicates either no abbreviation or a non–tokenised keyword

APPENDIX 4

SUMMARY OF DOS COMMANDS

COMMAND	FORMAT
NEW	N: DISK NAME, ID
COPY	C: NEW FILE = 0: OLD FILE
RENAME	R: NEW NAME = OLD NAME
SCRATCH	S: FILE NAME
INITIALISE	I
VALIDATE	V
BLOCK-READ	"B-R:"Channel;Drive;Track;Block
BLOCK-WRITE	"B-W:"Channel;Drive;Track;Block
BLOCK ALLOCATE	"B-A:"Drive;Track;Block
BLOCKFREE	"B-F:"Drive;Track;Block
BUFFER POINTER	"B-P:"Channel;Position
POSITION	"P"CHR$(Chan)CHR$(Low)CHR$(Hi)CHR$(Pos)
BLOCK-EXECUTE	"B-E:"Channel;Drive;Track;Block
MEMORY-READ	"M-R:"CHR$(Low)CHR$(High)
MEMORY-WRITE	"M-W:"CHR$(Low)CHR$(High)CHR$(no. of Chars)
USER	"Un:"

APPENDIX 5

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 RUNning 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 6

DOS ERROR MESSAGES

20 READ ERROR

The DOS has been unable to read a block of data – either an illegal track and sector have been requested or the disk has been corrupted or has been protected against copying.

21 READ ERROR

The sync byte of the requested track cannot be read – either an unformatted disk or one formatted under another DOS is present, the disk isn't inserted in the drive correctly, or the disk drive unit requires servicing to align the head.

22 READ ERROR

An incorrectly written block has been encountered – an illegal track and sector have been specified.

23 READ ERROR

A checksum error has occured – one or more of the bytes in a block has been read incorrectly, causing the checksum to fail.

24 READ ERROR

The DOS has decoded the data incorrectly – possibly owing to poor electrical connections within the drive or between the drive and the 64.

25 WRITE ERROR

The data block written to disk has been checked against that in DOS memory and been found to be incorrectly written.

26 WRITE PROTECT ON

The DOS has detected the presence of a write protect tab over the write protect notch.

27 READ ERROR

A checksum error in the header for a data block has been detected and the block has not been transferred to DOS memory - possibly due to poor earthing connections.

28 WRITE ERROR

A time-out error has occurred as the DOS tried to locate the sync byte for a data block - caused by bad disk format or hardware failure.

29 DISK ID MISMATCH

The DOS has detected a non initialised disk.

30 SYNTAX ERROR

The command sent to DOS via Channel 15 is illegal.

31 SYNTAX ERROR

The command sent to DOS is invalid.

32 SYNTAX ERROR

The command sent to DOS contains more than 58 characters.

33 SYNTAX ERROR

The file name in a command is invalid.

34 SYNTAX ERROR

Either no file name was sent or the syntax of the command was incorrect causing the DOS to misinterpret the command.

39 SYNTAX ERROR

An unrecognisable command has been sent to DOS.

50 RECORD NOT PRESENT

The file pointer has been positioned at a point after the last record of a relative file on the disk. This doesn't constitute an error if data is to be written (i.e. if a new record is being created).

51 OVERFLOW IN RECORD

An attempt has been made to write too much data to a record in a relative file. Remember that the carriage return terminator counts as one character.

52 FILE TOO LARGE

The file pointer has been positioned to a point where a disk overflow will occur if data is written.

60 WRITE FILE OPEN

An attempt has been made to open a file for reading before it has been closed after a write operation.

61 FILE NOT OPEN

An attempt has been made to access a file that has not been opened.

62 FILE NOT FOUND

An attempt has been made to access a non-existent file.

63 FILE EXISTS

An attempt has been made to save a file under a name which is already in the directory.

64 FILE TYPE MISMATCH

The file type in a DOS command differs from the type given in the directory for that file.

65 NO BLOCK

An attempt has been made to allocate a block which is unavailable according to the BAM. The parameters returned with this message define the track and sector numbers of the next free higher numbered block.

66 ILLEGAL TRACK AND SECTOR

An attempt has been made to access an illegal track and sector.

67 ILLEGAL SYSTEM TRACK OR SECTOR

An attempt has been made to access an illegal system track or sector.

70 NO CHANNEL

The requested channel is not available, or all channels are in use.

71 DIRECTORY ERROR

A discrepancy exists between the BAM count and the directory – possibly caused by overwriting the BAM. Re-initialise the disk to force DOS to re-create the BAM.

72 DISK FULL

Either no blocks are available or the maximum of 144 directory entries has been reached.

73 DOS MISMATCH

An attempt has been made to write to a disk formatted under a non-compatible DOS.

74 DRIVE NOT READY

No disk is present in the drive.

APPENDIX 7

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 8

COLOUR CODES

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

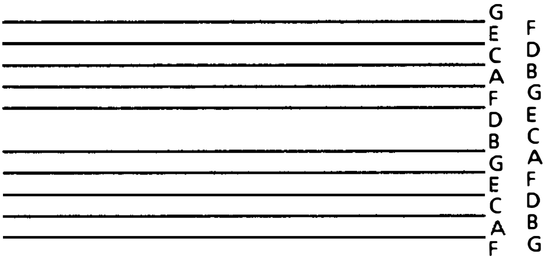
NOTE Only colours 0 to 7 can be used in multi-colour character mode

APPENDIX 9

MUSICAL NOTATION

This appendix will not teach you all about music, but it contains the basic information you need to translate sheet music into 64 programs.

Music is written by positioning symbols which represent the length of notes on a framework (called a stave) representing the pitch.



The lengths of notes are indicated by the note shape:

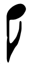
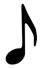
A Semibreve 𝅝 is twice as long as

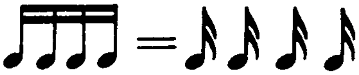
a Minim 𝅗𝅥 which is twice as long as

a Crotchet ♩ which is twice as long as

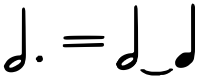
a Quaver ♪ which is twice as long as

a Semiquaver 𝅘𝅥𝅯

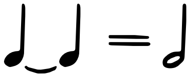
Tails on notes may go up  or down . The feathers on quavers and shorter notes may be joined where they appear in groups:



A dot after a note means that it is made half as long again as a normal note:



The mark ‿ is a tie which means the notes are joined together.



Volume is indicated by markings below the stave.

ff	Very loud
f	Loud
mf	Moderately loud
mp	Moderately soft
p	Soft
pp	Very soft
<	Get louder
>	Get softer

Speed is indicated by markings which may be above or below the stave. Examples are:

Presto	which means Fast
Allegro	Quite fast
Allegro moderato	Moderately fast
Moderato	Medium pace
Andante	Slow
Largo	Very slow

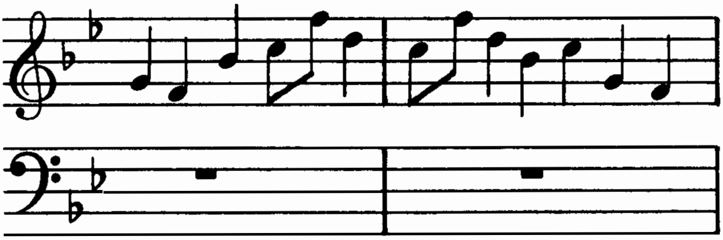
Unfortunately, there are many other Italian words and phrases which may be used. The best thing to do is to adjust your program until the speed sounds right, and not worry too much about what is written on the music.

Two other markings which may appear next to notes are # and b. # (sharp) means that the note should be raised by one semitone and b (flat) means that the note is lowered one semitone.

All other markings which may appear on sheet music (and there are many of them) can be ignored.

Pictures at an Exhibition

Here is the beginning of *Pictures at an Exhibition*, one of the examples used in Chapter 7. The notes shown in this music correspond to the notes represented by the first two items of every group of six in the DATA statements in the program.



Note the 'flat' symbols (b) on the lines of the stave corresponding to the notes B and E. These mean that all B's and E's throughout the music are sharpened.

APPENDIX 10

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 **BInary digiT**, 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 11

KERNAL ROUTINES

ACPTR $FFA5 65445

Inputs a byte from the serial bus into the accumulator. Before use call TALK and TKSA.

CHKIN $FFC6 65478

Selects a previously opened logical file to be an input channel.

CHKOUT $FFC9 65481

Selects a previously opened logical file to be an output channel.

CHRIN $FFCF 65487

Gets a character from a previously opened input channel into the accumulator. If no channel is opened, the keyboard is assumed to be the input device.

CHROUT $FFD2 65490

Sends the character in the accumulator to a previously opened channel. Channel must have been opened by OPEN and CHKOUT routines.

CIOUT $FFA8 65488

Sends the character in the accumulator to a serial bus device. The LISTEN and SECOND routines must have been called to prepare the device for the data.

CINT $FF81 65409

Initialises VIC-II chip and screen editor at the start of a program.

CLALL $FFE7 65511

Closes all open files and resets all I/O chanels.

CLOSE $FFC3 65475

Closes the logical file whose number is held in the accumulator.

CLRCHN $FFCC 65484

Resets all input/output channels, called by CLALL.

GETIN $FFEA 65508

Either loads one character from the keyboard buffer into the accumulator (accumulator contains 0 if buffer is empty), or gets a character from a serial device.

IOBASE $FFF3 65523

Loads the address of the start of memory mapped input/output into the X (low byte) and Y (high byte) registers.

IOINIT $FF84 65412

Initialises all input/output devices and routines.

LISTEN $FFB1 65457

Commands the serial bus device specified by number in the accumulator to listen (be ready to accept data).

LOAD $FFD5 65493

Loads data from a device into memory. If acumulator = 0 then memory is overwritten by new data; a 1 specifies a verify operation, the result of which is returned in ST. SETLFS and SETNAM routines must be used before this routine.

MEMBOT $FF9C 65436

If carry bit is set, returns address of lowest byte of RAM in X and Y registers. If carry bit is cleared before calling, the contents of X and Y specify lowest RAM address.

MEMTOP $FF99 65433

If carry bit is set, returns address of the highest byte of RAM in X and Y. If carry bit is cleared before calling, pointer to highest byte of RAM is set to contents of X and Y registers.

OPEN $FFC0 65472

Opens the logical file set up by SETLFS and SETNAM routines.

PLOT $FFF0 65520

If carry flag is set when called, PLOT loads current cursor position into X and Y registers. If carry flag is cleared before calling, cursor position is specified by contents of X and Y registers.

RAMTAS $FF87 65415

Performs RAM test, sets top and bottom of memory pointers, sets up cassette buffer and initialises screen.

RDTIM $FFDE 65502

Reads the high, middle and low bytes of the system clock into accumulator, X and Y registers respectively.

READST $FFB7 65463

Returns system status word in the accumulator.

RESTOR $FF8A 65418

Restores default values of all system vectors.

SAVE $FFD8 65496

Saves the contents of memory to a device which has previously been setup by SETLFS and SETNAM routines. When called the accumulator must contain an offset to a two byte pointer (stored in page zero) to the start of memory to be saved. The X and Y registers contain the address of the last byte to be saved.

SCNKEY $FF9F 65439

Scans keyboard for key presses. ASCII value of a key is stored in keyboard buffer.

SCREEN $FFED 65517

Returns number of screen columns and rows in X and Y registers.

SECOND $FF93 65427

Sets secondary address for an input/output device in a LISTEN operation.

SETLFS $FFBA 65466

Sets up a logical file, whose number is in the accumulator, device number is in the X register and command is in the Y register.

SETMSG $FF90 65424

Controls the output of error and control messages. If Bit 7 of the accumulator is set, an error message will be output; if Bit 6 is set a control message will be output.

SETNAM $FFBD 65469

Sets up a file name whose length is in the Accumulator. The X and Y registers must contain the start address of the file name in low-high byte order.

SETTIM $FFDB 65499

Sets jiffy clock to the time specified in the accumulator, X and Y registers.

SETTMO $FFA2 65442

Sets timeout flag when an IEEE card is connected.

STOP $FFE1 65505

Tests for STOP key being pressed after calling UDTIM routine. If STOP key was pressed the zero flag will be set.

TALK $FFB4 65460

Commands the serial device whose number is in the accumulator to talk.

TKSA $FF96 65430

Sends a secondary address to a serial device after the TALK routine.

UDTIM $FFEA 65514

Updates the system clock when interrupt routines have been modified.

UNLSN $FFAE 65454

Commands all serial devices to stop receiving data.

UNTALK $FFAB 65451

Commands all serial devices to stop sending data.

VECTOR $FF8D 65421

If carry bit is set when this routine is called, the current contents of the system vector jump addresses are stored in a table whose start address is defined by the X and Y registers. If carry is cleared before calling, the contents of the table pointed to by the X and Y registers are transferred to the system vectors.

APPENDIX 12

THE 6510 INSTRUCTION SET

ADC

Adds the contents of a memory location, or a number, to the accumulator, including the carry bit. Deposits the result in the accumulator.

AND

Performs the logical AND operation between the accumulator and data. Deposits the result in the accumulator.

ASL

Shifts the contents of the accumulator or memory left by one position. Bit 7 moves into the carry flag and Bit 0 is set to zero.

BCC

If the carry flag is clear program branches to current address plus a signed displacement (+127 to −128).

BCS

If the carry flag is set program branches to current address plus a signed displacement (+127 to −128).

BEQ

If the zero flag is set program branches to current address plus a signed displacement (+127 to −128).

BIT

Performs the logical AND operation between the accumulator and memory. If the comparison succeeds the zero flag is set and Bits 6 and 7 of the memory location are copied into the V and N flags.

BMI

If the N flag is set (the result of the last operation was negative) the program branches to the current address plus a signed displacement (+127 to −128).

BNE

If the zero flag is clear the program branches to current address plus a signed displacement (+127 to −128).

BPL

If the N flag is clear the program branches to current address plus a signed displacement (+127 to −128).

BRK

Saves the program counter and status register on the stack and copies the contents of $FFFE and $FFFF into PCLow and PCHigh.

BVC

If the overflow flag is clear the program branches to current address plus a signed displacement (+127 to −128).

BVS

If the overflow flag is set the program branches to current address plus a signed displacement (+127 to −128).

CLC

Clears the carry flag.

CLD

Clears the decimal flag.

CLI

Clears the interrupt mask to enable interrupts.

CLV

Clears the overflow flag.

CMP

Compares the accumulator contents with memory. Sets the Z flag if they are equal or clears it if not. The C flag is set if the contents of the memory location are greater than those of the accumulator.

CPX

Compares the X register with memory data .

CPY

Compares the Y register with memory data .

DEC

Decrements the specified memory location.

DEX

Decrements the X register.

DEY

Decrememts the Y register.

EOR

Performs the Exclusive OR operation between memory and the accumulator, storing the result in the accumulator.

INC

Increments the specified memory location.

INX

Increments the X register.

INY

Increments the Y register.

JMP

Program execution continues at the specified memory location.

JSR

Program execution continues at the subroutine commencing at the sepcified address.

LDA

Loads the accumulator with data.

LDX

Loads the X register with data.

LDY

Loads the Y register with data.

LSR

Shifts the contents of the accumulator or memory location right one position. Bit 7 is set to zero and Bit 0 transferred to the carry flag.

NOP

Performs no operation for two clock cycles.

ORA

Performs the logical OR operation between the accumulator and data.

PHA

Pushes the contents of the accumulator on to the stack.

PHP

Pushes the processor status register onto the stack.

PLA

Pulls the first item of data from the stack and loads it into the accumulator.

PLP

Pulls the first item of data from the stack and loads it into the processor status register.

ROL

Rotates the contents of the specified memory location one position to the left. The carry flag is transferred into Bit 0 and Bit 7 is moved into the carry flag.

ROR

Rotates the contents of the specified memory location one position to the right. The carry flag is transferred into Bit 7 and Bit 0 is moved into the carry flag.

RTI

Retrieves the status register and program counter from the stack and returns from an interrupt routine.

RTS

Restores and increments the program counter after a subroutine call and returns from the subroutine.

SBC

Subtracts data from the accumulator with a borrow and stores the result in the accumulator.

SEC

Sets the carry flag.

SED

Sets decimal mode.

SEI

Sets the interrupt disable mask.

STA

Stores the accumulator contents at a specified memory location.

STX

Stores the X register contents at a specified memory location.

STY

Stores the Y register contents at a specified memory location.

TAX

Transfers the accumulator contents to the X register.

TAY

Transfers the accumulator contents to the Y register.

TSX

Transfers the stack pointer contents to the X register.

TXA

Transfers the X register contents to the accumulator.

TXS

Transfers the X register contents to the stack pointer.

TYA

Transfers the Y register contents to the accumulator.

APPENDIX 13

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 14

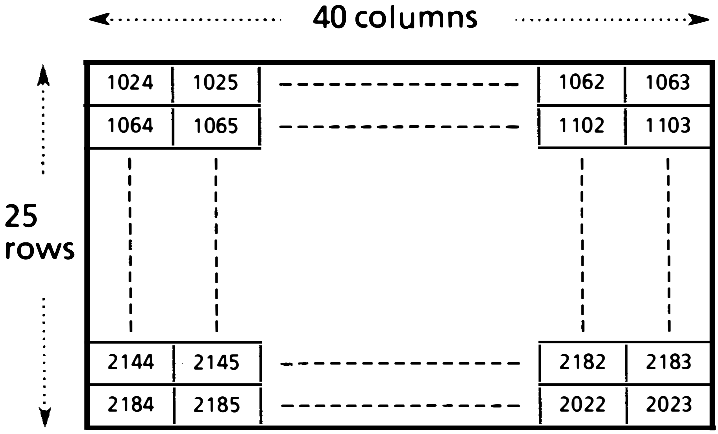
VIC-II REGISTERS

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	BACKGROUND COLOUR 0
53282	BACKGROUND COLOUR 1
53283	BACKGROUND COLOUR 2
53284	BACKGROUND COLOUR 3
53285	SPRITE MULTICOLOUR 0
53286	SPRITE MULTICOLOUR 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

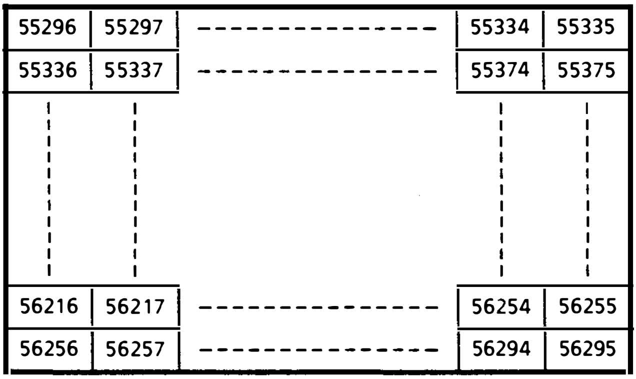
APPENDIX 15

SCREEN MEMORY MAPS

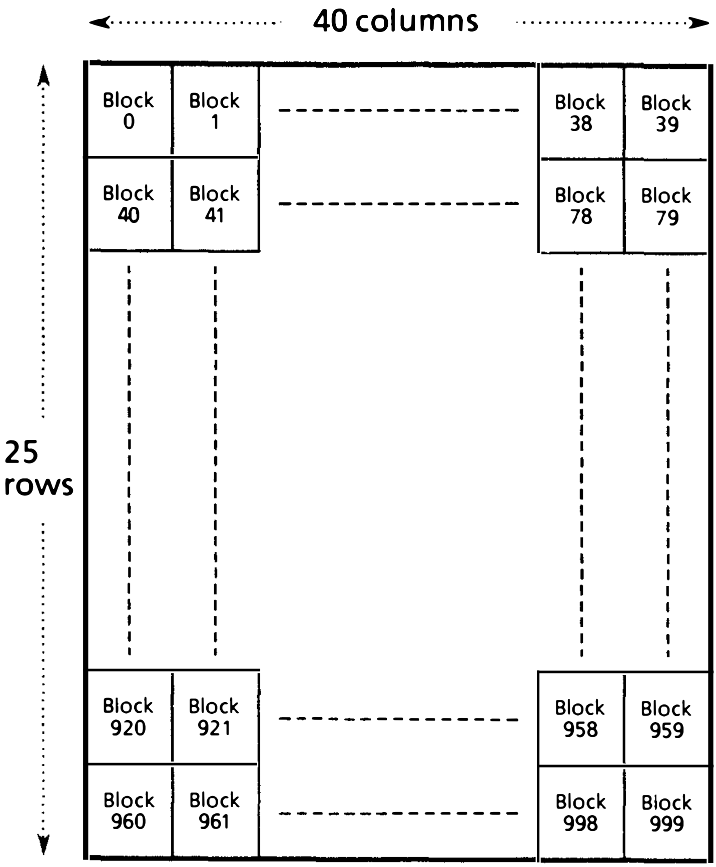


The Character Screen Memory

Each block represents 1 memory location

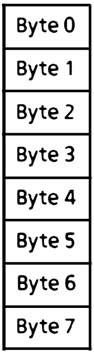


The Colour Memory



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

APPENDIX 16

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	3	#	#	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	π	▒	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 17

MEMORY MAP

Addresses	Contents	
0	Storage for operating system	
1024	Standard screen memory	
2048	Random Access Memory - BASIC programs and variables, Bit Mapped Screen, User Defined Characters, etc.	
40960	BASIC interpreter in ROM	
49152	Unused RAM	
53248	VIC and SID chip registers	Character Definitions in ROM
55296	Screen colour memory	
56320	Input/Output chip registers	
57344	Operating system in ROM	
65535		

APPENDIX 18

GRAPHICS LOADER PROGRAMS

This Appendix contains BASIC listings of loader programs for all the machine code graphics package programs described in Chapters 19 and 20. Refer to these chapters for details of how the programs are used.

How to Use the Loader Programs

The machine code has been split into 5 programs to make it easier for you to enter it in small stages. Because the routines are divided you do not have to type them all in at once - which will reduce the chances of error.

To obtain the machine code for the complete graphics package follow these instructions:

1 Type in the BIT-MAP routine loader, **SAVE** and **RUN** it.

2 Save the machine code by following the instructions at the end of the BIT-MAP routine.

3 Type in the PLOT routine loader, **SAVE** and **RUN** it.

4 Save the machine code by following the instructions at the end of the PLOT routine.

5 Type in the FILL routine loader, **SAVE** and **RUN** it.

6 Save the machine code by following the instructions at the end of the FILL routine.

7 Type in the DRAW routine loader , **SAVE** and **RUN** it.

8 Save the machine code by following the instructions at the end of the DRAW routine.

9 Type in the MIX MODE routine loader , **SAVE** and **RUN** it.

10 Save the machine code by following the instructions at the end of the MIX MODE routine.

11 When all five machine code routines have been saved, you can combine them into a single program by loading each one like this:

```
LOAD "PROG",8,1
```

for a program on disk, or:

```
LOAD "PROG",1,1
```

for a program on tape. You must type **NEW** after loading each program - this will not delete the machine code, it just resets the pointers to the BASIC RAM area.

12 When all five routines have been loaded in this way, you can save them as one program like this:

```
POKE 44,192:POKE 43,0

POKE 46,197;POKE 45,110
```

```
SAVE "GRAPHICS MC",8:REM OR ,1 FOR
TAPE
```

13 To load the graphics package subsequently, use the command:

```
LOAD "GRAPHICS MC",8,1
```

to load from disk, or:

```
LOAD "GRAPHICS MC",1,1
```

to load from tape, and type **NEW** after loading. This will cause the loss of any BASIC program in the 64, so be careful!

BIT-MAP ROUTINE

```
10    REM BIT MAP LOADER
20    FOR X=49152 TO 49331
30    READ D:POKE X,D
40    NEXT X
50    PRINT"BIT MAP CODE LOADED"
60    END
10000 DATA 169,1,141,177,2,208,5,
      169,0,141,177,2,169,96,133,
      252,169
10010 DATA 0,133,251,162,64,32,154,
      192,169,92,133,252,169,0,133,
      251,173
10020 DATA 174,2,10,10,10,10,13,175,
      2,162,8,32,154,192,173,177,2
10030 DATA 208,16,169,216,133,252,
      169,0,133,251,162,8,173,176,
      2,32,154
10040 DATA 192,173,2,221,9,3,141,2,
      221,173,0,221,41,252,9,2,141
10050 DATA 0,221,173,17,208,9,32,
      141,17,208,169,120,141,24,208,
      173,177
```

```
10060 DATA 2,208,8,173,22,208,9,16,
      141,22,208,96,173,2,221,9,3
10070 DATA 141,2,221,173,0,221,41,
      252,9,3,141,0,221,173,17,
      208,41
10080 DATA 223,141,17,208,173,22,
      208,41,239,141,22,208,169,21,
      141,24,208
10090 DATA 96,160,127,145,251,136,
      16,251,72,24,165,251,105,128,
      133,251,169
10100 DATA 0,101,252,133,252,104,
      202,208,231,96
```

Saving the Bit-Map Routine machine code

After runnning the loader program type in the following commands to save the machine code:

```
POKE 44,192:POKE 43,0

POKE 46,192:POKE 45,180

SAVE "BIT MAP":REM ADD ,8 FOR DISK
DRIVE
```

PLOT ROUTINE

```
10    REM PLOT LOADER
20    FOR X=49336 TO 49593
30    READ D:POKE X,D
40    NEXT X
50    PRINT"PLOT CODE LOADED"
60    END
11000 DATA 173,169,2,74,74,74,141,
      186,193,173,168,2,74,173,167,
      2,106
11010 DATA 74,174,177,2,240,1,74,
      141,187,193,173,169,2,41,7,
      141,188
```

```
11020 DATA 193,173,186,193,133,253,
      169,0,133,254,162,6,32,175,
      193,202,208
11030 DATA 250,165,254,133,252,165,
      253,133,251,32,175,193,32,175,
      193,24,165
11040 DATA 253,101,251,133,251,165,
      254,101,252,133,252,169,0,133,
      254,173,187
11050 DATA 193,133,253,32,175,193,
      32,175,193,32,175,193,24,165,
      253,101,251
11060 DATA 133,251,165,254,101,252,
      133,252,24,173,188,193,101,
      251,133,251,169
11070 DATA 0,101,252,133,252,24,169,
      0,101,251,133,251,169,96,101,
      252,133
11080 DATA 252,173,177,2,240,48,173,
      167,2,41,7,141,189,193,56,
      169,7
11090 DATA 237,189,193,141,189,193,
      24,169,1,174,189,193,240,4,10,
      202,208
11100 DATA 252,160,0,174,173,2,240,
      5,17,251,145,251,96,73,255,
      49,251
11110 DATA 145,251,96,173,167,2,41,
      3,141,189,193,56,169,3,237,
      189,193
11120 DATA 10,141,189,193,160,0,173,
      173,2,41,3,174,189,193,240,
      4,10
11130 DATA 202,208,252,141,190,193,
      169,252,174,189,193,240,5,56,
      42,202,208
11140 DATA 252,49,251,13,190,193,
      145,251,96,169,0,6,254,6,253,
      101,254
11150 DATA 133,254,96
```

Saving the Plot Routine machine code

After runnning the loader program, type in the following commands to save the machine code:

```
POKE 44,192:POKE 43,184

POKE 46,193:POKE 45,188

SAVE "PLOT":REM ADD ,8 FOR DISK
DRIVE
```

FILL ROUTINE

```
10    REM FILL LOADER
20    FOR X=50402 TO 50466
30    READ D:POKE X,D
40    NEXT X
50    PRINT "FILL CODE LOADED"
60    END
13000 DATA 173,169,2,141,35,197,32,
      184,192,24,173,169,2,105,1,
      141,169
13010 DATA 2,173,172,2,205,169,2,
      176,236,173,35,197,141,169,2,
      24,173
13020 DATA 167,2,105,1,141,167,2,
      173,168,2,105,0,141,168,2,
      56,173
13030 DATA 171,2,237,168,2,173,170,
      2,237,167,2,176,198,96
```

Saving the Fill Routine Machine Code

After runnning the loader program, type in the following commands to save the machine code:

```
POKE 44,196:POKE 43,226
POKE 46,197:POKE 45,35

SAVE "FILL":REM ADD ,8 FOR DISK
DRIVE
```

DRAW ROUTINE

```
10    REM DRAW LOADER
20    FOR X=49608 TO 50372
30    READ D:POKE X,D
40    NEXT X
50    PRINT"DRAW CODE LOADED"
60    END
12000 DATA 173,170,2,141,197,196,
      173,171,2,141,198,196,173,169,
      2,141,199
12010 DATA 196,169,0,141,224,196,56,
      173,172,2,237,169,2,176,15,
      173,224
12020 DATA 196,73,1,141,224,196,56,
      173,169,2,237,172,2,141,202,
      196,56
12030 DATA 173,170,2,237,167,2,141,
      200,196,173,171,2,237,168,2,
      176,24
12040 DATA 173,224,196,73,1,141,224,
      196,56,173,167,2,237,170,2,
      141,200
12050 DATA 196,173,168,2,237,171,2,
      141,201,196,208,10,173,200,
      196,208,5
12060 DATA 169,1,141,224,196,173,
      202,196,208,5,169,1,141,224,
      196,173,201
12070 DATA 196,208,11,173,200,196,
      205,202,196,176,3,76,10,195,
      56,173,170
12080 DATA 2,237,167,2,173,171,2,
      237,168,2,176,3,32,160,196,
      173,202
```

```
12090 DATA 196,141,205,196,169,0,
      141,206,196,173,200,196,141,
      203,196,173,201
12100 DATA 196,141,204,196,32,51,
      196,173,167,2,141,208,196,173,
      168,2,141
12110 DATA 209,196,32,200,195,141,
      212,196,173,223,196,141,213,
      196,174,224,196
12120 DATA 240,21,24,173,212,196,
      109,169,2,141,212,196,173,213,
      196,105,0
12130 DATA 141,213,196,76,188,194,
      173,169,2,237,212,196,141,212,
      196,169,0
12140 DATA 237,213,196,141,213,196,
      173,168,2,141,209,196,173,167,
      2,141,208
12150 DATA 196,32,200,195,174,224,
      196,208,10,24,109,212,196,141,
      169,2,76
12160 DATA 228,194,56,173,212,196,
      237,222,196,141,169,2,32,184,
      192,24,173
12170 DATA 167,2,105,1,141,167,2,
      173,168,2,105,0,141,168,2,56,
      173
12180 DATA 170,2,237,167,2,173,171,
      2,237,168,2,176,181,76,181,
      195,56
12190 DATA 173,172,2,205,169,2,176,
      3,32,160,196,173,200,196,141,
      205,196
12200 DATA 173,201,196,141,206,196,
      173,202,196,141,203,196,169,0,
      141,204,196
12210 DATA 32,51,196,173,169,2,141,
      208,196,169,0,141,209,196,32,
      200,195
```

```
12220 DATA 141,212,196,174,224,196,
      240,18,24,109,167,2,141,212,
      196,169,0
12230 DATA 109,168,2,141,213,196,76,
      105,195,173,167,2,237,212,196,
      141,212
12240 DATA 196,173,168,2,233,0,141,
      213,196,173,169,2,141,208,196,
      169,0
12250 DATA 141,209,196,32,200,195,
      174,224,196,208,19,24,109,212,
      196,141,167
12260 DATA 2,173,213,196,109,223,
      196,141,168,2,76,162,195,56,
      173,212,196
12270 DATA 237,222,196,141,167,2,
      173,213,196,237,223,196,141,
      168,2,32,184
12280 DATA 192,24,173,169,2,105,1,
      141,169,2,205,172,2,144,182,
      240,180
12290 DATA 173,198,196,141,168,2,
      173,197,196,141,167,2,173,199,
      196,141,169
12300 DATA 2,96,173,214,196,141,217,
      196,173,215,196,141,218,196,
      173,216,196
12310 DATA 141,219,196,169,0,141,
      220,196,141,221,196,141,222,
      196,141,223,196
12320 DATA 141,210,196,141,211,196,
      160,24,78,219,196,110,218,196,
      110,217,196
12330 DATA 144,37,24,173,208,196,
      109,220,196,141,220,196,173,
      209,196,109,221
12340 DATA 196,141,221,196,173,210,
      196,109,222,196,141,222,196,
      173,211,196,109
```

```
12350 DATA 223,196,141,223,196,14,
      208,196,46,209,196,46,210,196,
      46,211,196
12360 DATA 136,208,193,173,222,196,
      96,169,0,141,216,196,141,215,
      196,141,214
12370 DATA 196,141,207,196,160,24,
      56,173,206,196,237,203,196,
      141,206,196,173
12380 DATA 207,196,237,204,196,141,
      207,196,8,46,214,196,46,215,
      196,46,216
12390 DATA 196,14,205,196,46,206,
      196,46,207,196,40,144,21,173,
      206,196,237
12400 DATA 203,196,141,206,196,173,
      207,196,237,204,196,141,207,
      196,184,80,18
12410 DATA 173,206,196,109,203,196,
      141,206,196,173,207,196,109,
      204,196,141,207
12420 DATA 196,136,208,192,46,214,
      196,46,215,196,46,216,196,96,
      173,170,2
12430 DATA 172,167,2,141,167,2,140,
      170,2,173,171,2,172,168,2,141,
      168
12440 DATA 2,140,171,2,173,172,2,
      172,169,2,141,169,2,140,172,
      2,96
```

Saving the DRAW routine machine code

After runnning the loader program, type in the following commands to save the machine code:

```
POKE 44,193:POKE 43,200

POKE 46,196:POKE 45,197
```

```
SAVE "DRAW":REM ADD ,8 FOR DISK
DRIVE
```

MIXED MODE LOADER

```
10    REM MIX MODE LOADER
20    FOR X=50468 TO 50541
30    READ D:POKE X,D
40    NEXT X
50    PRINT "MIX MODE CODE LOADED"
60    END
14000 DATA 120,169,67,141,20,3,169,
      197,141,21,3,169,1,141,26,208,
      169
14010 DATA 0,141,18,208,173,17,208,
      41,127,141,17,208,88,96,173,
      25,208
14020 DATA 41,1,240,33,141,25,208,
      173,18,208,208,11,32,69,192,
      169,200
14030 DATA 141,18,208,76,101,197,32,
      114,192,169,0,141,18,208,104,
      168,104
14040 DATA 170,104,64,76,49,234
```

Saving the MIXED MODE routine machine code

After runnning the loader program, type in the following commands to save the machine code:

```
POKE 44,197:POKE 43,36

POKE 46,197:POKE 45,110

SAVE "MIX MODE":REM ADD ,8 FOR DISK
DRIVE
```

SAVING THE GRAPHICS PACKAGE

After creating the machine code for each of the five routines comprising the graphics package and saving them individually, you can load them all into the 64 using the commands:

```
LOAD"PROG",8,1
```

to load from disk, or

```
LOAD"PROG",1,1
```

to load from tape. You must type **NEW** after loading each program. With all the routines in the machine you can save them together as one program by typing:

```
POKE 44,192:POKE 43,0

POKE 46,197:POKE 45,110

SAVE "GRAPHICS MC",8
```

DATAMAKER

This program will convert the contents of any area of memory into a series of **DATA** statements which are appended to the DATAMAKER program. To use the program enter the line number at which you want the **DATA** to start, and the start and end address of the area of memory you want to convert into **DATA** statements. When the program has finished, delete lines 5 to 500 and save the **DATA** statements for use in your program.

```
5      REM DATAMAKER
10     INPUT "FIRST DATA LINE
       NUMBER";LN
```

```
20     INPUT "START ADDRESS OF
       CODE";S
30     INPUT "END ADDRESS OF CODE";E
100    F=S+16:IF F>E THEN F=E
110    PRINT"{CLS}"LN"DATA";
120    FOR I=S TO F
130    C = PEEK(I)
140    C$ = MID$(STR$(C),2)
150    CK = CK+C
160    PRINT C$;
170    IF I<F THEN PRINT",";
200    NEXT
210    PRINT
300    S=F
310    IF S<E THEN PRINT"LN="LN+10":
       S="S+1":E="E":GOTO100"
320    IF S=E THEN PRINT"LIST"
330    POKE 198,3
340    POKE 631,19
350    POKE 632,13:POKE 633,13
500    END
```

INDEX

SUPER BASIC

SUPER BASIC is an extension to the standard BASIC 2.0 of the Commodore 64, providing 36 extra BASIC commands which give you full control over the Graphics, Sprites and Sound capabilities of your 64, and add many of the features found in more modern versions of BASIC.

SUPER BASIC COMMAND SUMMARY

AT	Moves cursor for PRINT@ & INPUT@ operations
AUTO	Automatically generates line numbers
DEEK	16 bit PEEK
DEL	Deletes blocks of program lines
DOKE	16 bit POKE
HIMEM	Sets the limit of memory for BASIC programs
OLD	Restores accidentally NEWed programs
PACK	Removes all REMs and surplus spaces
PAUSE	Provides a timed delay in jiffies
POP	Removes one address from the RETURN stack
RENUM	Renumbers programs including all branches
RESET	Resets the 64

BLOCK	Fills blocks on hi-res screen with specified colour
CHIRES	Clears hi-res screen and sets hi-res mode
CMULTI	Clears hi-res screen & sets multicolour mode
DRAW	Draws lines on hi-res screen in both modes
FRAME	Changes frame (border) colour
HIRES	Returns to previously set hi-res mode
INK	Changes foreground (ink) colour
PAPER	Changes background (paper) colour
PLOT	Plots or unplots a point on either hi-res display
SETCOL	Sets hi-res colours
TEXT	Returns to the text display

FILTER	Sets up the filtering on SID chip
GATE	Controls gateing of SID voices
MUSIC	Plays musical notes from 8 octaves on any voice
QUIET	Inhibits sound output and resets SID
SHAPE	Attack, Decay, Sustain & Release for each voice
SOUND	Sets frequency & waveform for each voice

VOLUME	Sets volume for SID chip
SPRCOL	Set up sprite colour registers
SPRITE	Enables sprites
SPRMOV	Positions sprites on screen
SPROFF	Disables sprites
SPRSET	Defines sprites
SPRSIZ	Defines sprite sizes

Also included in the SUPER BASIC package is a comprehensive booklet describing how the new commands are used and some example programs. If you would like to order a copy of SUPER BASIC please cut out or copy the form below and send it with your remittance to the address given;

Century Software

To: Mail Order Dept.,
Tiptree Book Services Ltd,
Church Road, Tiptree,
Colchester, Essex CO5 0SR

Please send me ____ copies of **SUPER BASIC** by Peter Lupton and Frazer Robinson, at £9.95 each.

ISBN 0 7126 06661

I enclose my cheque/postal order for £ ____
(cheques should be made payable to *Tiptree Book Services Ltd*).

Please debit my Access/Barclaycard account
(delete as appropriate).

Account number □□□□□□□□□□□□□□□□

Name ____________________

Address ____________________

____

____

Signed ____________________

(Home address please if paying by credit card)

Please allow 28 days for delivery.

***The Commodore 64 Omnibus* is the one book every C64 programmer needs. It is both an introductory handbook and an advanced user's guide to this powerful computer with its superb sound and graphics facilities.**

Pt. I: 'This book is a model of the way in which such books ought to be written...[the authors] have done a thoroughly workmanlike job in guiding the first time user along the first hesitant paths towards programming...The authors have done very well." COMPUTER TALK

Pt. II: 'Let us suppose you are an enthusiastic user of the 64, who is proficient in Basic and would like to go further. You have examined Commodore's *Programmer's Reference Guide* but find parts of it too technical. You would particularly like to experiment with sound and Hi-res graphics, in Basic and Assembler, to understand the Basic Interpreter and to add a few extra commands to Basic. The trouble is that you have been unable to find a suitable book to assist you.

'Look no further! This superb volume is the very thing you need...

'This is indeed a book to dip into at random and explore in depth. It is packed with information presented clearly and logically, with several helpful appendices...Highly recommended!'

COMMODORE HORIZONS

ISBN 0-7126-0676-9

9 780712 606769

£9.95
ISBN 0 7126 0676 9
Century Communications Ltd