

Getting **MORE** from your **commodore** **64**

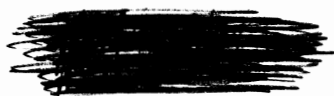


Mark Harrison

 **Sigma Technical Press**

GETTING MORE FROM YOUR COMMODORE 64

Mark Harrison



 Sigma Technical Press



Copyright © 1983 Mark Harrison

All Rights Reserved

No part of this book may be reproduced or transmitted by any means without the prior permission of the publisher. The only exceptions are for the purposes of review, or as provided for by the Copyright (Photocopying) Act or in order to enter the programs herein onto a computer for the sole use of the purchaser of this book.

Acknowledgement

The Commodore symbol and "Commodore 64" are proprietary to Commodore Business Machines Ltd. Appendices A and B are reproduced by kind permission of Commodore.

ISBN 0 905104 59 5

Published by:

SIGMA TECHNICAL PRESS,
5 Alton Road,
Wilmslow,
Cheshire
UK.

Typeset and Production by:

Designed Publications Ltd.
DESIGNED PUBLICATIONS LTD.
8-10 Trafford Road,
Alderley Edge,
Cheshire.

Distributed by:

JOHN WILEY & SONS LIMITED,
Baffins Lane, Chichester,
West Sussex, England.

Index of Programs

1.	Magic Carpet	38
2.	Multiplication	39
3.	Fibonacci	40
4.	Cubes	43
5.	Simultaneous Equations	47
6.	Highest Common Factor	53
7.	Dice	55
8.	E	56
9.	Quadratic Equations	59
10.	Prime Numbers	60
11.	Triangle	64
12.	Character Set	69
13.	Mary Whitehouse	71
14.	Word Search	72
15.	Data Base	74
16.	Manarag	75
17.	Calendar	80
18.	Reaction Timer	83
19.	Digital Clock	84
20.	Bubble Sort	89
21.	Insertion Sort	90
22.	Shell Sort	91
23.	Quick Sort	92
24.	Alphabetical Sorting	93
25.	User Defined Graphics	125
26.	Control	133
27.	Snake	134
28.	Surround	136
29.	Circle	144
30.	Ellipse	146
31.	Sine Wave	146
32.	Epicyclic	147
33.	Spirals	148
34.	Etcher Sketcher	148
35.	Three Dimensional Graphics	151
36.	String Pattern	152
37.	Vertical Scrolling	156
38.	Sprite Demonstration	163
39.	Sprite 3D Priorities	166
40.	Altered Images	167

41.	Grand Prix	170
42.	Sprite Editor	173
43.	Fur Elise	187
44.	Exploring SID	190
45.	Ogonek	195
46.	Disk Error Routine	215
47.	Histogram	221
48.	Copy	223
49.	Telephone Pad	227 & 228
50.	Address Book	233
51.	Birthday Data Base	247
52.	Linked List	258
53.	Tree Sort	269
54.	Animals	272

Contents

1.		
	A Beginner's Guide to Computers	1
	The History of Computers	1
	The Principle Components of a Complete System	3
	Types of Software and their Execution	6
2.	Introducing the Commodore 64	9
3.	Operating your Commodore 64	11
	Switch On	11
	The Keyboard	13
	The Structure of BASIC	15
4.	Computer Arithmetic	19
5.	The Design and Development of Computer Programs	24
	Flowcharts	25
6.	Commodore 64 BASIC - Statements/Commands	29
	System Commands	30
	Other System Commands	31
	Assignment Statements	31
	Input and Output Statements	34
	Other I/O Statements	37
	Control Statements	37
	More Advanced Control Statements	41
	IF ... THEN	44
	GOSUB ... RETURN	45
	Other Statements	48
7.	The Commodore 64 Functions	52
	Simple Numeric Functions	53
	Trigonometric Functions	60
	User Defined Functions	65

8. The Commodore 64 Character Set and String Handling	68
Character Codes	68
String Handling Functions	70
9. Control Characters	77
10. The Time Machine	83
11. Techniques for Sorting	86
Bubble Sort	88
Insertion Sort	89
Quick Sort	91
Alphabetical Sorting	93
12. The Coding of Information for Computer Systems	95
The Binary Number System	95
Hexadecimal Notation	100
13. Computer Logic	103
The Logic Operators	104
Masking	109
14. The Commodore 64's Memory Layout	111
RAM and ROM Layout	111
Memory Management	112
The Commodore 64's Operating System	112
Screen Memory	113
Colour Memory	113
The VIC-II Chip	113
The SID Chip	113
15. Character Display Mode	115
More on the VIC-II	115
Screen Memory	117
The Character ROM	118
User Defined Characters	121
Multicolour-Mode	126
16. Peek and Poke Graphics	130
Extended Background Colour Mode	137

17. High Resolution Graphics	139
Three Dimensional Plotting	150
Multicolour-Mode	154
Smooth Scrolling	155
18. Sprites	157
Sprite Formats	157
Switching On Sprites	159
Sprite Colour	159
Positioning Sprites	160
Further Sprite Facilities -	
Expanded Sprites	165
Animation Using Sprites	167
Collision Detection	170
Designing Sprites	173
Multicolour Sprites	175
The Raster Register	176
Interrupts	177
19. Sound and the 64	179
Characteristics of Sound Waves	179
Controlling the Sound Features	181
Volume	182
Waveform	183
Frequency	184
ADSR	185
More Advanced Sound Facilities	193
Filtering	193
Vibrato	194
20. Basic Input/Output	197
Peripherals :	197
Logical Files	197
Primary Addresses: Device Numbers	198
Secondary Address	199
OPENing and CLOSEing Logical Files	200
Communicating with a Peripheral	201
Data Files	203
21. Program Storage	206
The C2N Datasette Unit	206
Disk Drives	

The VIC-1541 Single Disk Drive	210
Storing Programs on Floppy Disk	211
Opening a Command Channel	211
Initialise Disk	212
Loading the Disk Directory	212
The Disk Maintenance Commands	213
Errors with Disk Handling	214
Filename Matching	215
22. The VIC-1515 Graphic Printer	217
The CMD Command	218
Listing a Program	218
More Advanced Control	218
Double Width Mode	219
Graphic Mode	219
Cursor Up/Down	220
Reverse Character Mode	220
Controlled Printing	220
The COPY Statement	223
23. Data Storage on Cassette	224
Cassette Data Files	225
Writing a Data File	226
Reading a Data File	228
24. Data Storage on Disk	230
Sequential Data Files	230
Setting Up a Sequential Data File	231
Listing a Sequential File	231
Searching for a Record	231
Deleting a Record	232
Amending a Record	232
Adding a Record	232
Random Access Data Files	240
Advanced Disk Control	240
Communication Channels	242
Advanced Disk Control Commands	243
Block Allocation and Hashing	246
25. Dynamic Data Structures	254
The Forward Linked List	255
More Advanced Lists	261
Circular List	261

Double Linked Lists	262
Stacks and Queues	264
Graphs	265
Trees	272
Heuristic Programming	275
 26. Machine Code Programming	
6510 Microprocessor Assembler Language	
Instruction Set	277
 Appendix A: ASCII and CHR\$ Codes	
Appendix B: PEEK / POKE Codes	
Appendix C: Screen Control Characters	
Appendix D: MOS 6566 Video Interface Controller	
Appendix E: MOS 6581 Sound Interface Device	
Appendix F: Abbreviations for BASIC Keywords	
Appendix G: Software Available for the Commodore 64	



CHAPTER 1

A BEGINNERS GUIDE TO COMPUTERS

With the Commodore 64, you can enter the fascinating world of personal computing and, like many thousands of other people, you will probably become addicted to the subject. However - be warned, an addiction to computing means spending much of your time happily tapping away at the keyboard of your computer. If you are new to computing, do not be overwhelmed by the initial sight of what looks like a bewildering array of shiny hardware with a complex instruction manual; possibly the hardest part of personal computing is deciding to spend your hard earned money on a machine - the rest is easy. Even if you have no previous computing experience, it should only take just a couple of hours to do some useful things. Like the majority of hobbies, computing skills will only develop with continual practice and by reading around the subject. Always try any ideas or queries you have on the computer and see what actually happens; also be confident that you know the reasons why any result occur.

Before studying the particular capabilities of your Commodore 64 it is both important and interesting to understand the basic ideas and components of a general computer system. You will soon realize that a computer costing less than a hi-fi system has similar features to others costing several thousand pounds.

THE HISTORY OF COMPUTERS

Since one of the major uses of the human brain is the processing of information, it is obvious that man from early times would have tried to automate this. Aids for counting and storing numbers are evident, such as pebble counters, marks made on sticks and abacuses, from early Chinese, Greek and Egyptian civilizations. A useful development was the discovery of some mathematical functions called logarithms which reduced the

problems of multiplication and division to the easier ones of addition and subtraction.

Tables of logarithm values and also, slide rules, which work on a similar principle were both commonly used right up into the 1970s when their use was superseded by the introduction of cheap pocket calculators.

The first serious ideas for automatic computation were conceived by several mathematicians independently in the early nineteenth century. These people, however, had the problem that their ideas were over one hundred years ahead of the necessary technology that they would require. Some of the best ideas were introduced by Charles Babbage who decided, while involved in the tedious process of checking astronomical tables that methods in automatic computation were required. One of Babbage's developments was based on the principles used in automatic looms which were able to weave complicated patterns by sensing, mechanically, the formations of punched holes in control cards. Babbage saw this idea as being suitable for storing other types of information. He came up with several ideas for mechanical machines which kept a record of data and allowed mathematical operations to be carried out on them.

In the next one hundred years, several electromechanical machines were designed and constructed but it was not until 1946 that the first electronic computer was produced at the University of Pennsylvania. This device was called ENIAC (Electronic Numerical Integrator and Calculator) and contained over 18000 thermionic valves and diodes which occupied 140 square metres of floor space. A mathematician, John Von Neumann, who worked on the ENIAC project devised the principles of computer systems which are still in use today. In the previous electromechanical machines, the instructions - calculations and the like - that were carried out on the data were read in as required; thus the speed was limited by the rate at which the instructions could be read. Von Neumann decided that the instructions should be coded and stored in the computer along with the data; this enabled calculations to be carried out electronically at speeds which were many times faster than those previously achieved.

Von Neumann's basic ideas included the main principle of present-day computing: input of data; processing; followed by output of results. He was also responsible for the "architecture" of computer circuitry that has survived to this day: memory for storage of data and processing instructions, and the use of a Central Processing Unit that steps through the instructions, sequentially. In fact, such a design is still called a Von Neumann machine.

Today the ENIAC machine could be miniaturized onto wafers of silicon

less than 3mm square, containing thousands of microcircuits in the form of integrated circuits. It is, however the same ideas of John Von Neumann, i.e. Input, Storage, Processing and Output that are used in today's computers.

THE PRINCIPLE COMPONENTS OF A COMPUTER SYSTEM

Briefly, a computer may be described simply as an electronic device that utilizes a stored list of given instructions. This stored list of instructions that is continually mentioned is called a program, the person who writes the instructions is called a programmer and the act of writing the instructions is called programming. Note that it is standard to use the American spelling of 'program' rather than the English version 'programme'.

Sometimes the instructions are referred to as software while the actual physical components of the system are called hardware. When computers were first produced, the cost of the software was relatively cheap in comparison to the expensive hardware. Nowadays, this has reversed because the hardware has become cheap and reliable due to mass production and miniaturization while the labour costs of programmers who are developing more complicated software have increased.

We shall now look in detail at the hardware common to any computer system.

Recapping from the previous section, a computer is able to receive, memorize, manipulate and return information. The specific components that are required in such a system are shown in Fig 1.1.

An analogy to a computer system is the human being who uses five senses to receive information, limbs and voice to return information and a brain to memorize and process the information. Although a human can undertake much more complicated tasks than a computer, the computer has the ability to process information many times faster and it also never forgets what it has been told. It is vital to realize from the beginning that a computer is relatively stupid in so far as it can never think for itself - it is only capable of doing what has actually been specified in a given program.

When a computer makes an error it will always be the fault of the programmer unless, of course, there is a fault or "bug" in the software supplied with your machine. This is always a convenient excuse!

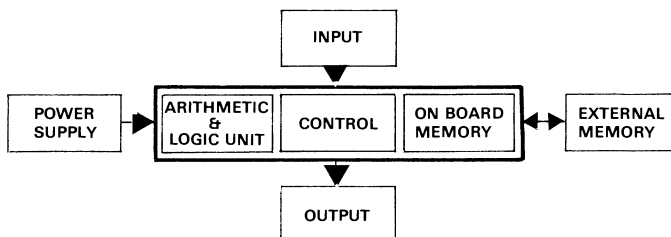


Figure 1.1 The Principle Components of a Computer System

It probably comes as no great surprise to the reader that a computer is useless unless it receives some information that it can understand. The transfer of such information is undertaken through the means of an input device, examples of which include a keyboard, a punch card reader, magnetic tape reader etc.

Similarly it is pointless in letting a computer process this information if we cannot access the results from the machine; thus information may be transferred from the computer through output devices, examples of which include a visual display unit (television screen), punched paper tape, a line printer etc. Devices that are capable of both input and output are called Input/Output (or I/O) devices.

Since a computer could not operate on information that it cannot remember, there is an actual hardware component whose function is to store all the data held in the system. This unit is called the memory. The best analogy to a computer's memory is the visualization of a long sequence of storage boxes or cells each identified by a unique number commencing with zero and incremented by one each time, (imagine the lockers at a railway station or the valuables boxes in the vaults at banks). The label of each storage box or cell in the memory is called an address. In each box or cell a number may be stored. In the case of most personal computers, including the Commodore 64, this number must be within the range of 0 to 255 inclusive.

The standard Commodore 64 contains two different types of memory. These are read only memory (ROM) and random access memory (RAM), the difference being:

ROM	RAM
is memory that is preprogrammed when manufactured with permanent data. The contents cannot be changed by the computer used.	is general purpose memory in which data may be stored and then retrieved when required. RAM loses its contents when the power is switched off.

When a number is retrieved from either ROM or RAM the contents of the appropriate cell remain unaltered. When a number is stored in a cell within RAM, the previous contents are overwritten and lost for ever. You can not, however, store new data in a ROM. As an example consider the following two sections of memory, one ROM and the other RAM and see what happens when an attempt is made at storing a value in a certain cell:

ROM

Unsuccessfully trying to store 69

in address 1030.

RAM

Successfully trying to store 69 in

address 17223.

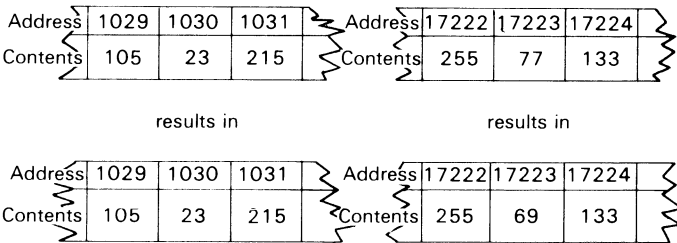


Figure 1.2 Storing Values in RAM and ROM

In addition to numbers, it is possible to store characters and program instructions by representing them as numerical values; these values are known as character codes and instruction codes.

Apart from storing a user’s program and its associated data, a section of the computers memory is reserved for system software. System software is a program that is stored permanently in the computer and is used for controlling all the operations of the computer system. This software must be permanent and not affected either by switching the power off or by changing the program instructions, so a computer manufacturer always stores system software in ROM.

One final comment about memory - for the time being - is on quantities of storage. When referring to a size of memory the abbreviation k is frequently chosen to represent 1024 cells of memory. 1024 is 2 raised to the power of 10. For example, if we read about an 8k program it will

require 8×1024 cells of memory. If the quantity of memory supplied with the computer is insufficient it is possible to connect (or interface) additional memory which is called external memory.

Referring back to figure 1.1 you will see there is a component in a computer system called the arithmetic and logic unit. Its job is to evaluate any mathematical expression involving addition, subtraction, multiplication or division (arithmetic) and also to test if numbers are positive, negative or equal to zero (logic). While the need for the arithmetic function is fairly obvious, the logic function is required so that the computer can make decisions and know what actions to perform next.

The remaining component in the computer system is the control section which organizes the operation of the computer actions; control keeps track of what instructions are being obeyed and undertakes the operations specified by each instruction. One aspect of this section is that, normally, it causes each instruction to be processed in turn which is, you'll remember, what Johnny Von Neumann decided!

Having taken a quick browse through the basic hardware that all computer systems contain, we shall now examine the different types of software and see how a computer copes with it. When the instructions in a program are obeyed, the program is said to have been executed.

TYPES OF SOFTWARE AND THEIR EXECUTION

Earlier it was pointed out that the cells in the computer's memory could store programs by representing them with unique numerical values; for example the code to add two values together on a particular computer might be 35. Other codes would be used for other operations. Since a program would contain more than one instruction for it to be of any practical use, such a program might look like this:

35 27 133 211 23 39 5 221 100.....etc....

Obviously this program is meaningless to anyone who has not been told what instructions each value represents. This type of program is known as a machine code program and although it may seem tedious and difficult to use, computer programmers in the days of the first computers had no option but this type of programming. It soon became obvious to the programmers that it would be more efficient for them to use codes that bore some resemblance to the instructions (such as "ADD") that they

refer to. At this stage it is important for the reader to realize that a computer can only ever understand programs that are in the machine code format; thus if a program is written with instructions referred to by different codes it must at some stage be translated into the equivalent machine code program. The translation is accomplished by yet another computer program - ultimately, written in machine code.

The next stage in the advancement of software was the introduction of assembly programs. These replaced the machine code instructions like 35 or 169 by abbreviations for the instruction that the code represents, such as for example LD for Load, ADD for Addition, SUB for Subtraction etc. It takes no imagination to realize that an assembly program would be easier to write, read and comprehend than a machine code program.

These abbreviations, which are commonly referred to as mnemonics, can then be converted into their corresponding machine code values by a section of software called an assembler program and then the machine code version can be used as before. So long as machine code works, the computer doesn't care where it came from!

A further advance in software since the late 1950s has been the development of several sophisticated programming languages known as high level languages.

There are many different ones, the most common being FORTRAN, ALGOL, COBOL, Pascal, CORAL and BASIC. The choice of which language to use depends usually on the application. For example, a business application would probably use COBOL, a scientific application might use FORTRAN and an application that required results to be produced within strict time limits could well use CORAL. Unlike assembler programs, the codes in high level languages do not correspond one to one with the machine code values but, instead, allow the coding to resemble the nature of the problem to be solved. For example, requiring a message "CORRECT" to be output if the answer to a question is 99 might in a high level language be written as:

```
INPUT answer;  
IF answer = 99  
THEN OUTPUT "CORRECT";  
STOP;
```

It should be evident at this stage that a high level language, when compared to the other types of software, is by far the easiest for readability. As with assembly programs, for a computer to understand the code of a high level language it must first be translated into its machine code equivalent by a special software program. It might be reassuring at

this stage to know that you are totally isolated from this translation process - you do not need to know how this process works - just that it does. All we'll say on this subject is that there are two methods used for translating high level languages into machine code versions. The first method uses software called a compiler which converts the whole high level language program into machine code once and for all and then it is the machine code version that is used every time. The second method, which is used by the Commodore 64 uses software called an interpreter which converts each small part of the high level language program into machine code as it is needed. Interpreters are slower than compilers, but generally easier to use on a personal computer.

That concludes the initial chapter in this book and by now you should have started to obtain a grip on the basic principles of what a computer is and how it functions. In the next chapter we shall take a look at the Commodore 64 taking notice of its particular components and seeing how it resembles the general computer system as just described. Building on this general introduction, the reader will then be taken on a programming course leading up to come of the most advanced techniques that may be used on the Commodore 64.

CHAPTER 2

INTRODUCING THE COMMODORE 64

Having studied the general basic concepts of a computer system we shall now take our first general look at the Commodore 64 system. The standard Commodore 64 looks extremely elegant and resembles a previous Commodore computer known as the VIC-20. It is supplied in a grey casing approximately 400 x 200 x 80mm and contains a brown 66 moving-key keyboard with the layout of the keys in the standard 'qwerty' typewriter arrangement.

The Commodore 64's central processing unit (CPU) is a 6510A and is responsible for controlling all the operations of the computer. This is a relatively low cost processor suitable for small systems and is a development on the 6502 used in previous Commodore systems. The Commodore 64's memory comprises of 64K of RAM, from which the name of the machine arises, with an additional 20K of ROM containing software for the operating system, the BASIC interpreter and the character set.

To display information, the Commodore 64 requires either an ordinary domestic colour or black and white television set, although on the latter the colours will naturally appear as shades of grey. A lead is supplied which carries the TV signal from the video socket found at the rear of the Commodore 64 to the aerial socket of the television; reception is received at approximately channel 36 on U.H.F. When switched on, the display screen will consist of 25 rows of 40 columns with the central area appearing dark blue and the screen border and characters displayed being a lighter blue. The graphic features of the Commodore 64, although complex to use are some of the best available. It is possible to program using a 320 by 200 dot (or pixel) high resolution display and to define high resolution shapes called sprites that can be made to move, expand, contract, pass in front of or behind each other and to detect collisions.

At the rear, and the right hand side of the Commodore 64 is a multitude of

connectors which are used for interfacing to other components. The Commodore 64 when launched had the advantage over other new systems that a proven range of printers, disc drives, joysticks etc. had already been developed for the VIC-20; such peripherals could also be used on the Commodore 64. The sockets at the side are labelled 'control port 1' and 'control port 2' and are used for connecting to joysticks or light pens. The edge connectors at the back are used for interfacing to the cassette interface and other I/O cartridges. Of the two DIN sockets that are present, one carries signals for audio and video output and the other, which is called the serial port is used for connection to the VIC printer or VIC disc drive. The cassette recorder used for storing programs on normal cassette tape is the C2N Datasette which is also used by the VIC-20. The data transfer system uses an advanced recording method which is not compatible with other computer systems. You can not easily use a normal domestic cassette recorder.

An excellent feature of the Commodore 64 are the sound generation facilities which use three independently controlled oscillators. Although the complex sounds produced are emitted from the loudspeaker of the television set, the signals are also present at the DIN socket mentioned previously and so they may be recorded, amplified or played through an external speaker providing the appropriate equipment is available. Commodore claim that their machine can match the sounds produced by some of today's most sophisticated electronic synthesizers.

The standard programming language used by the Commodore 64 is Commodore's Version 2 BASIC which is used on both their VIC-20s and their later PETs; this does, however, lack user friendly commands for controlling the extensive graphic and sound facilities. Tests show that although the Commodore 64 is slightly slower at executing BASIC programs than the VIC-20 it is very good when compared to its major competitors. Since Commodore has retained the same programming language it should be reasonably simple to convert VIC-20 software to run on the Commodore 64.

Our next step is to switch on our Commodore 64 and start writing our first programs.

CHAPTER 3

OPERATING YOUR COMMODORE 64

SWITCH ON

To switch your Commodore 64 on and start it working, there are two connections which have to be made:

- 1) Connect the power supply provided to the socket marked 'POWER' and switch on, the red L.E.D. (light emitting diode) on the top should light up.
The power supply should be 9 volts at 1.0 A unregulated as supplied by Commodore.
- 2) Connect the television aerial socket to the video socket found at the rear of the computer with the cable provided. The television should then be tuned to receive the best reception which is close to channel 36. When the screen clears to a steady picture the message

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

should appear. This tells us which version of BASIC our computer is using and how much memory we have available for a users program. Your Commodore 64 is now ready and waiting to be given an instruction.

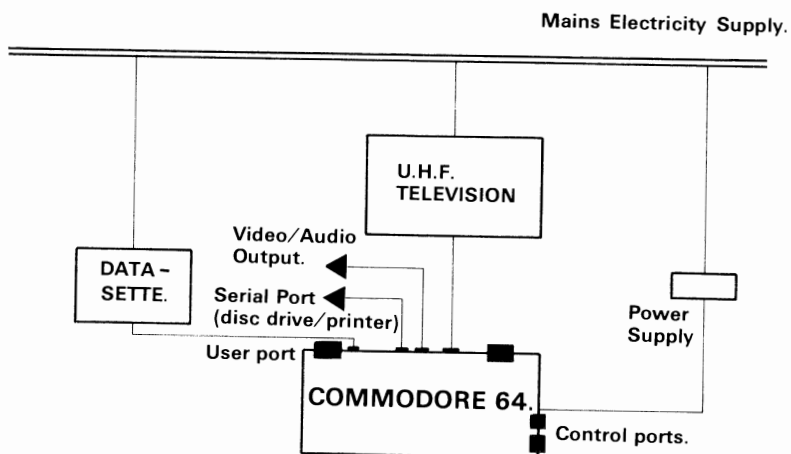


Figure 3.1 The Commodore 64 Computer System



Figure 3.2 The Commodore 64 Keyboard

The Keyboard

Now that we have our computer set up, the next stage is to familiarize ourselves with the keyboard which may be slightly different from those you have seen before. The keyboard of the Commodore 64 works very similar to that of a standard typewriter except that some keys have special functions or additional symbols associated to them; you should only require about an hour's practice before the use of the keyboard becomes second nature.

There are in fact two modes that the keyboard can be in. At switch on, the keyboard is in Upper Case/Graphic Mode in which case everything is automatically typed in capitals. The graphic characters situated on the front of the keys are accessed by pressing the key with either **⌘** for the left hand character or **SHIFT** for the right hand character.

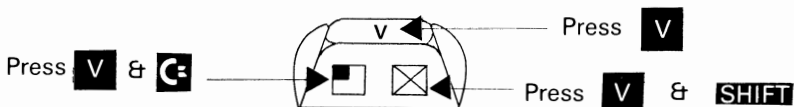


Figure 3.3

By pressing **⌘** and **SHIFT** simultaneously the keyboard switches from Upper Case/Graphic Mode to Upper/Lower Case Mode. In this case the keys return lower case characters when pressed individually, upper case characters when pressed with **SHIFT** and as before, the graphic character at the front left of the key when pressed with **⌘**.

Pressing **⌘** and **SHIFT** again will switch back to Upper Case/Graphic Mode.

You will probably have noticed a flashing square on the display screen, this is known as the cursor and it indicates the position on the screen where the next character you type will be displayed. We shall learn later how to control the position of the cursor so that we can choose where on the screen to display our information.

The keyboard has the facility for the user to 'type ahead' by storing up to ten key presses. The characters corresponding to the keys are stored in a section of memory called a buffer and kept in the order of 'first in, first out'; such a structure is called a queue. When control needs to know what key has been pressed it looks in the buffer, reads the first key present and then deletes it, which makes room for future keys. This idea enables very rapid typing to be undertaken without losing keyboard input.

We shall now quickly examine the keys with specific functions; the keys will be mentioned again in more depth when appropriate.

RETURN

- tells the computer to accept all the information that is present on the line indicated by the cursor. It means "I've finished typing my input".

RUN STOP

- stops execution of a BASIC program.

RESTORE

- when pressed simultaneously with **RUN STOP** resets the computer to its 'switch on' state.

CTRL

- when pressed simultaneously with a
 - i) digit key '1' to '8' turns the colour of future displayed text to the colour on the front of the key.
 - ii) digit key '9' or '0' turns on/off reverse video.
 - iii) certain other keys will have specific uses by the computer, other remaining keys may have functions defined in a program.

SHIFT

- i) in Upper Cases/Graphic Mode accesses right hand graphic symbols.
 - ii) in Upper/Lower Case Mode accesses the upper case characters.
 - iii) other uses with certain specific function keys.
- when pressed simultaneously with **ESC**
 - i) **SHIFT** alternates from one keyboard mode to another.
 - ii) graphic character key returns symbol situated on the front left.
 - iii) digit key '1' to '8' accesses alternative set of colours.

CLR HOME

- resets the cursor position to the top left hand corner of the screen; if pressed simultaneously with **SHIFT** the screen is cleared also.



- used for moving the cursor position and the position of data to the immediate right of the

cursor. As we shall soon, these keys are very important when editing programs.

We shall now take a look at the Commodore 64's BASIC programming language.

The Structure of BASIC

The programming language used by the Commodore 64 is just one of the many versions of the BASIC language (Beginners All-purpose Symbolic Instruction Code) that are now available. This simple language was designed originally for beginners to computing but has since become one of the most popular computer languages. Although efforts were initially made to standardize the statements in BASIC, there has recently been a tendency for computer manufacturers to produce versions of BASIC with statements that are unique to their machine. As a result there are some statements available on other computers that are absent from the Commodore 64 and vice-versa.

A BASIC program is made up of lines, each containing one or more instructions, preceded by a line number, which in the case of the Commodore 64 lies between 1 and 63999. The program statements are stored in the order of their line numbers and this is the order in which they are obeyed or executed. Thus, if a line is required to be inserted for execution between two lines already in a program, the new line must commence with a number that is between the two numbers of the existing lines in the program.

It is possible to have more than one statement associated with a certain line number. This is achieved by entering the particular statements after the line number in the order of their required execution and by separating them by the colon ':' symbol.

Entering a Program Line

To show how a program is entered into the Commodore 64 we will consider the following example which consists of one simple statement:

```
10 PRINT "CBM"
```

When executed, the statement 'PRINT' will tell the computer to display all the characters specified within the quotation marks, i.e. in this case 'CBM'.

Enter the program into your Commodore 64 by pressing the following keys. At this point the computer is said to be in command or direct mode.

First type the line number of the statement, i.e. 10

1 0

For clarity a space is inserted after the line number.

SPACE BAR

The BASIC keyword PRINT is typed.

P R I N T

For clarity again, a space is inserted after the keyword.

SPACE BAR

The data to be displayed is given

SHIFT & 2 C B M SHIFT & 2

And finally, to accept the program line, press

RETURN

Now the computer contains a program consisting of a single line. To execute it type RUN.

R U N (followed by RETURN)

It is important that no other characters are on the same line as the RUN statement.

The program is then executed and the Commodore 64 goes into program mode.

When the instructions contained in line 10 have been obeyed the letter 'CBM' should appear at the next line on the screen and the message 'READY' is displayed. The 'READY' message is known as a report and in this case tells us that the program was executed without any problems and no errors were detected. Other reports will be displayed when errors occur and give clues to what the problems are; often the line number where the error occurred is also displayed. After a report has been displayed, the Commodore 64 will have returned to command mode and will wait for further instructions.

Errors

When a program fails, the error which caused the problem can be either a syntax error or a logical error. A syntax error occurs when the grammar of a

statement is wrong, for example if a required comma is missing. A logical error occurs when the programmer has made an error in his ideas of the program logic, e.g. attempting to divide by zero. Once an error has been located in a program we need a method of changing the incorrect statement to its required form; this is known as editing.

Editing Your Programs

Even the most experienced readers make errors in their programs and require to make some amendments. The most simple method is to re-type the faulty line using the same line number as before and this, when entered, will over write the original line. Thus, a very simple method for removing unwanted lines from a program is to type just the line number followed by **RETURN**. Similarly, additional lines may be inserted by typing the line number followed by the statement and **RETURN**, and control will take the line and automatically sort out the position of the new line.

There is however a more convenient method. The Commodore 64 contains what is called a screen editor which basically means that alterations can be made directly to a program line displayed on the screen by moving the cursor to where the change is required and typing the amendment on top. Pressing **RETURN** will read the whole line and replace any original version.

To use the screen editor type LIST (followed by **RETURN** of course) and the program currently stored will be displayed on the screen.

Pressing  or  with or without **SHIFT** will move the cursor about the screen in the direction shown on the key. If the key is continually pressed it will repeat itself.

When the cursor has overlayed the position requiring an alteration, the change can be made. Pressing any character key will now overwrite what was originally displayed. Also, characters may be deleted from the screen

by  and room can be made by the inserting of spaces with **SHIFT**

& 

When a change has been made press **RETURN** and the new line will be accepted by the computer. Once all the alterations have been made type LIST again and check that the new program is correct. Make sure that no 'stray' characters are on the same line as LIST or the error report '? SYNTAX ERROR' will be given. A safe procedure is to type **SHIFT** and

 which clears the screen and allows LIST to be typed on a fresh line.

If a line number is overwritten with a different line number, a new line identical to the original is created but the first line is left untouched. Thus if several similar lines have to be entered it may be quicker to type it in for one line and then edit the line numbers.

Note that the computer can only read a maximum of two screen lines at a time. Thus if you try to enter a line consisting of more than eighty characters the extra will be lost.

Having looked at both computer systems in general and your own Commodore 64, it is now time to get down to BASICS!!

CHAPTER 4

COMPUTER ARITHMETIC

While everybody realizes that a computer can handle very complicated arithmetic expressions it is a common misconception for people to think that this is a simple task - in fact it probably is one of the hardest!! There are two main reasons for this, the first being that a computer is expected to be able to cope with a vast range of numbers and the second that an individual address in the computer memory is restricted to hold a whole number between 0 and 255 inclusive. It is, however, comforting to know that when a high level language is used, a programmer need not have any knowledge of the complicated methods used to evaluate his arithmetic expressions as this is undertaken automatically by a section of software stored in the system software.

Those readers who have little experience in mathematics have probably never given a great deal of thought to the many notations in which numbers may be written. For example, the first three different types of numbers we ever learned about were whole numbers, fractions and decimals. In a similar fashion, there are three particular notations that we shall now consider all of which are commonly used in computing. You will recognize the first two notations although the names given to them might be new.

1) **INTEGER** - Integers, commonly called whole numbers, are numbers which are a sequence of digits which do not involve a decimal point or any fractional component. They may be either positive or negative. e.g. -156, -22, -1, 0, 1, 2, 69. In Commodore 64 BASIC, integers are restricted to the range ± 32767

2) **REAL** - Real numbers are a sequence of decimal digits with a single decimal point either at the end, the beginning or between two digits. The digits to the left of the decimal point are weighted in positive powers of 10 and form the integer component of the number. Likewise the digits to the right of the decimal point are weighted in negative powers of 10 and form the fractional component of the number. Real numbers may be positive or

negative. e.g. -2.3, 0.4, 21.025, .125, -0.275

To illustrate this idea:

		617.105
Negative powers of 10 {	$5 \times 10^{-3} = 5 \times 0.001 = 0.005$	
	$0 \times 10^{-2} = 0 \times 0.01 = 0.00$	
	$1 \times 10^{-1} = 1 \times 0.1 = 0.1$	
Positive powers of 10 {	$7 \times 10^0 = 7 \times 1 = 7.0$	
	$1 \times 10^1 = 1 \times 10 = 10.0$	
	$6 \times 10^2 = 6 \times 100 = 600.0$	
		617.105

3) EXPONENTIAL - Exponential numbers are written in a notation which is suitable for either very large or very small numbers. They are written using a real number followed by the symbol 'E' followed by an integer number. 'E' means "times 10 to the power of".

Thus yEx means 'y times 10 to the power of x'.

e.g.

$$1.234\text{E}-2 = 1.234 \times 10^{-2} = 0.01234$$

$$1.234\text{E} 0 = 1.234 \times 10^0 = 1.234$$

$$1.234\text{E}+2 = 1.234 \times 10^2 = 123.4$$

The effect of the integer following the 'E' is to indicate how many places the decimal point has to be shifted. A positive value means shift to the right and a negative value means shift to the left.

e.g. 1E10 means 1 followed by ten '0's i.e. 10,000,000,000

Using this scientific notation, the largest number (the number furthest away from zero) which the Commodore 64 is capable of storing is approximately 1.7×10^{38} and the smallest number, (the number nearest to zero) is approximately 2.9×10^{-39} . Numbers are stored to an accuracy of between nine and ten digits.

Having discovered the different types of numbers and notations that are used by computers, we shall take a look at how use them in arithmetic. Firstly we shall study how our data is referred to in the computer's memory during the execution of a program. For this we use what are called variables which, incidentally, have the same function in BASIC as in any other high level language. Following this we shall look at the numbers and variables written in mathematical expressions and see how the computer interprets them.

Variables

All pieces of data that are stored in the Commodore 64's memory, as a

result of typing a BASIC program are labelled by symbolic names. Such a name refers to the data stored at one particular location. If we consider a statement in which a numerical value is assigned to, for example, a name 'MAX', then the computer would reserve a section in its memory referenced by the name 'MAX' and the value would be stored there. From then on, any reference to the variable 'MAX' would be, in fact, referring to the value stored in the section of memory that was specifically reserved for 'MAX'. It is because this data may be changed that the names are called 'variables'.

On the Commodore 64 there are certain rules for choosing the names of variables. Variables can have names of any length and may consist of both alphabetic and numeric characters providing the name starts with a letter.

However, in Commodore BASIC only the first two characters are recognized. Thus 'MAXIMUM' and 'MAGNITUDE' refer to the same variable. Variables that are restricted to contain integers have similar names but are followed by the symbol % e.g. A%. Strings of characters can be stored in variables which have names followed by the symbol \$ e.g. A\$. Variable names must not contain a sequence of characters that are reserved BASIC keywords.

Valid variable names:

- numerical variable A
- integer variable A1%
- numerical variable MAX
- string variable A\$
- string variable PT\$

Invalid variable names:

- numerical variable 2A (does not start with letter)
- integer variable 4% (does not start with letter)
- string variable A.B (contains invalid character)
- string variable TOP\$ (contains keyword TO)

Note that A, A\$ and A% are unique variables.

Algebraic Expressions

When we are considering mathematical problems of the form

“what is $5 + x/3 - y$?”

we are talking about the evaluation of algebraic expressions. An algebraic expression is a list of variables names known as operands, linked together

by any of the following operators:

+ ADDITION - SUBTRACTION * MULTIPLICATION / DIVISION ↑ RAISED TO THE POWER OF - NEGATION

For example

$A * B / C$

$A \uparrow B$

$-A + B + C - 2.5$

$A + B * C * 3.0$

It is important to recognize the simple difference between subtraction and negation. While subtraction works on two operands e.g. $A - B$ Negation involves just one operand e.g. $-A$

Brackets may also be included in algebraic expressions in which case the algebraic expressions most deeply nested by brackets are evaluated first e.g.

$(A + B + C) / 3$
 $(A + B) / (C + D)$

Sometimes if we choose to omit the brackets we may not achieve the result that we require. For example, consider the expression $A + B * C$

You might think that the computer could interpret this in two ways:

- 1) C multiplied by the result of A plus B
- 2) A added to the result of B multiplied by C

Thus it is important that some rules are specified giving the order that operands are operated on. The usual method is to give each operator a value known as its hierarchy or priority. The computer first considers all the operators with the highest priority and then evaluates them from left to right. It then considers the operators with the second highest priority and evaluates them from left to right. This continues until the whole expression has been evaluated.

The priority values for the operators used by the Commodore 64 are shown in Figure 4.1

Operator	Priority
↑	9
- (negate)	8
*	7
/	7
+	6
- (subtract)	6

Figure 4.1 Operator Priorities for the Commodore 64

E.g. Calculate $-4+3↑2*6-4*9/12$

Taking operators of priority 9: $-4+\underbrace{3↑2}*6-4*9/12$

Taking operators of priority 8: $\underbrace{-4}+9*6-4*9/12$

Taking operators of priority 7: $-4+\underbrace{9*6}-\underbrace{4*9/12}$

Taking operators of priority 6: $-4+\underbrace{+54}-3$

Gives the final answer 47

CHAPTER 5

THE DESIGN AND DEVELOPMENT OF COMPUTER PROGRAMS

We have now reached a suitable point to examine the various stages in the development of a computer program. A beginner to computing should show more concern in learning the skills in program design than worrying about mastering all the features of a particular computer language. Even a professional computer programmer would not get straight down to the coding level and expect to produce a program that functioned perfectly correctly and contained no logical errors.

Program development may be split up into four distinct stages:

The first stage is for the programmer to recognise the problem to be solved and ensure that it has been properly defined. This may seem rather trivial but it is a common occurrence for a professional programmer to find that the client is not totally sure of what is required. The most obvious method of tackling a complex problem is to split it up into smaller sub-sections which may be considered independently as separate problems. This also enables errors and problems with the complex problem to be easily located.

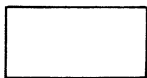
In the second stage the programmer has to recognise the factors in the problem that are liable to vary (i.e. the variables) and then find the mathematical relationships that occur. By using his skill the programmer can analyse this information to produce a series of rules that solve the problem; such a method is called an algorithm. Programming skills can ideally be learned by studying good quality software written by experienced programmers in books and magazines. Some examples of algorithms and programming techniques will be given in this book.

Providing enough thought goes into these first two stages the programmer should find the third stage reasonably simple. This involves turning the algorithm into code using a computer language that the computer understands. It is essential that the programmer makes the program unambiguous. For example, it is no use telling the computer to perform some operation on a piece of data if the computer does not know what this data is, or telling the computer to continue at a certain point in the program if that point does not exist. The next few chapters of this book details the particular features of the language used by the Commodore 64 and demonstrates how algorithms may be coded efficiently.

The final stage is for the programmer to test the program to ensure that it functions as it was required to do. Through testing, a programmer may detect errors but unless the program is tested using all feasible input data and with all possible paths in the program flow, the absence of errors is not guaranteed. Taking, for example, just the variability in numerical input, the range of data is so vast that total testing is virtually impossible. Thus a programmer should choose selected input data that will take different paths in the program flow. Incidentally, any errors that are found during testing are often referred to as bugs and the task of locating and solving these errors is called debugging.

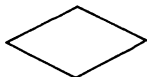
Flowcharts

The end result of the design of an algorithm will be a series of instructions that are obeyed in a specific order, this normally being in the sequence they are written. It is, however, possible for the algorithms to include decisions and jumps which causes the execution of the algorithm to proceed with different instructions. In order to cope with the program flow in complex algorithms that contain numerous jumps and decisions, some programmers like to draw diagrams called flowcharts. Flowcharts contain the instructions of the algorithm written within symbols of differing shapes and are linked together by flowlines or arrows to indicate the sequence of operations. Types of symbols drawn are shown below, their use differs depending on the type of instruction it represents.



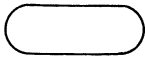
PROCESS

- any process function causing a change in information or location of information.



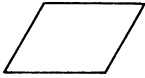
DECISION

- a switching operation that causes one of a number of alternative paths to be taken.



TERMINATION POINT

- a point in the flowchart where the execution starts, ends or is delayed.



INPUT/
OUTPUT

- an I/O function where information is made available for processing (input) or where processed information is returned (output).



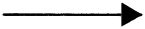
PREDEFINED PROCESS

- where a number of operations have been specified elsewhere.



CONNECTOR

- where flow exits to another part of the chart, for example when a flowchart cannot be completed on one side of paper. An identification number is written inside the two corresponding connectors.



FLOWLINE

- direction of program flow.

For example, the following flowchart shows the stages in the process of making a mug of coffee.

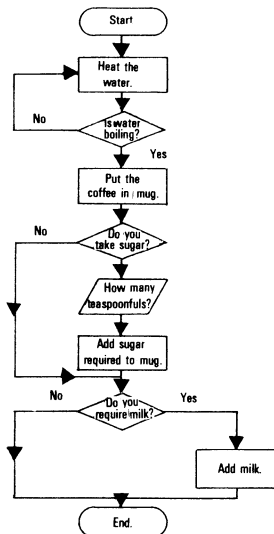


Figure 5.1

To demonstrate the development of an algorithm we shall take as an example the problem of finding the square root of a number i.e. finding a value that when multiplied by itself gives the original number. There is in fact a command on your Commodore 64 that returns the square root of a number but the algorithm shown below is used by the majority of computers and calculators for finding square roots. The algorithm starts by making a guess at the result and then proceeds into a loop which continually arrives at a more accurate result by using the value previously obtained. The more accurate value is obtained by finding the average of the result obtained plus the value that, when multiplied by the result obtained, gives the number whose square root is required. If this seems rather complicated look at the formula below where x is the number whose square root is required, r_1 is the nearest root so far obtained and r_2 is the new root.

$$\text{Then } r_2 = \frac{1}{2} \left(r_1 + \frac{x}{r_1} \right)$$

For example, if we require the square root of 25 and we think the answer is about 4.5 then a more accurate value is

$$\frac{1}{2} \left(4.5 + \frac{25}{4.5} \right) = 5.03$$

Continually repeating this process will eventually return the value 5. The algorithm should terminate when the root obtained is within an acceptable error limit; this is when $(r_2 * r_2) - x$ is less than some small value i.e. when the difference between the root squared and the original number is negligible.

It is important that the algorithm takes into account that square roots cannot exist for negative numbers.

To some readers it may not be apparent why the algorithm works; an explanation has been omitted as it involves mathematics which is unimportant to the Commodore 64 user. It is however common practice in commercial programming for the algorithm to be devised by a different person to the one that actually codes the program; it is sufficient for the programmer just to know that the algorithm works. Therefore, you may now be in the position of many a commercial programmer.

Our algorithm for finding square roots can now be drawn in flowchart form as in Figure 5.2. From this the problem can be coded using a language that the computer to be used can understand.

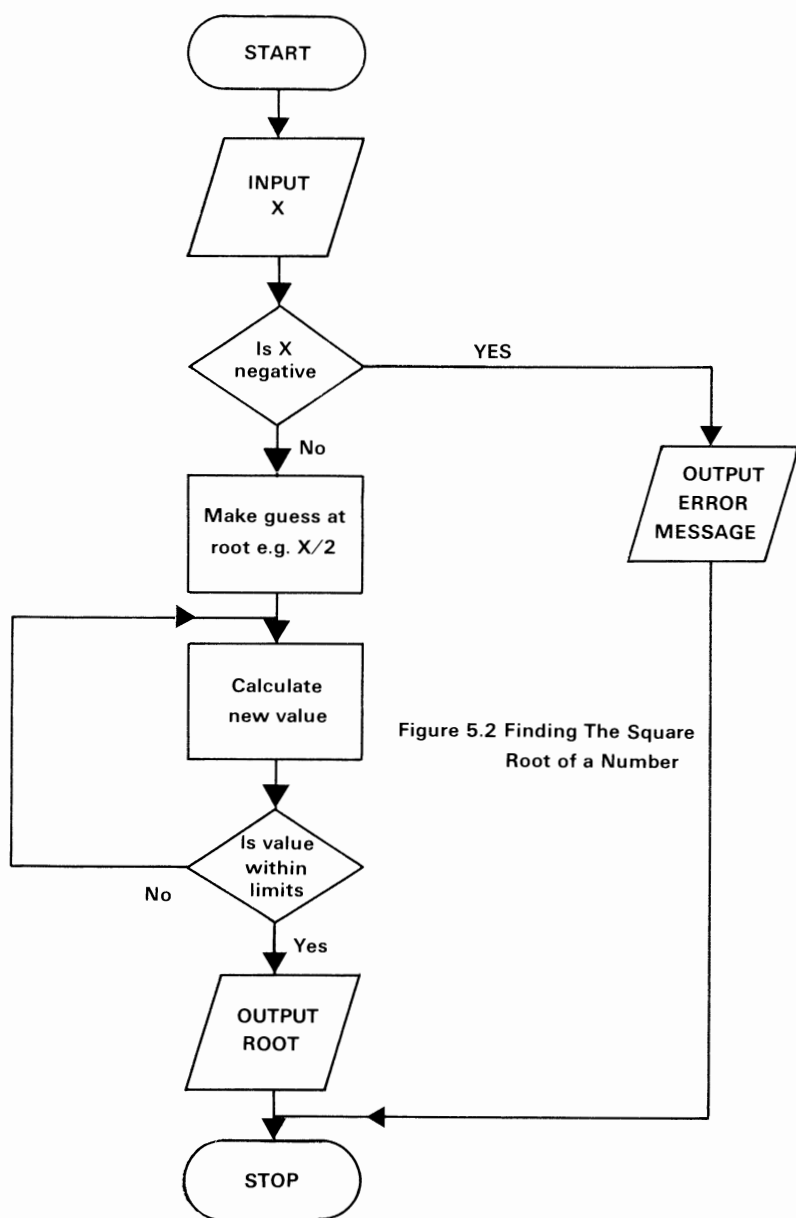


Figure 5.2 Finding The Square Root of a Number

CHAPTER 6

Commodore 64 BASIC - Statements/Commands

As we saw earlier, a BASIC program is a list of numbered lines each containing one or more statements. A statement can do one of two things. It may either just describe some item of information or it may tell the computer to undertake a specific operation. The next section of the book deals with the following statements which are available on the Commodore 64:

The Commodore 64 Statements

CLOSE	GOTO	PRINT
CLR	IF....THEN	PRINT#
CMD	INPUT	READ
CONT	INPUT#	REM
DATA	LET	RESTORE
DEF FN	LIST	RETURN
DIM	LOAD	RUN
END	NEW	SAVE
FOR...TO...STEP	NEXT	STOP
GET	ON	SYS
GET#	OPEN	VERIFY
GOSUB	POKE	WAIT

This list of statements can be subdivided into the following categories:-

1) System Commands - these are unique from the other four groups in that they are not usually used within a program. They are used in the operation of the computer system and are accessed by typing the required statement without a preceding line number.

2) Assignment Statements - these are used to allocate either numeric

values or a sequence of characters to specific sections or locations in the computer's memory.

3) Input/Output Statements - these are used to allow information to be entered into (input) and obtained from (output) the computer.

4) Control Statements - these are used to control the sequence in which the program statements are executed.

5) Other Statements - these comprise the repertoire of Commodore 64 BASIC excluding groups 1 to 4.

Syntax Notation Used

For each BASIC keyword that is introduced, the syntax will be stated by using the following notation:

- i) BASIC keywords are written in capitals.
- ii) variable data is written within angle brackets '< >'
- iii) optional data is written within square brackets '['']'
- iv) data items which may be repeated are followed by ...

Note that all the programs that are given have been spaced out for clarity; experienced programmers may wish to compress them so that they use up less memory by, for example, leaving out spaces, semicolons etc.

SYSTEM COMMANDS

The following statements are 'system commands' - their use is of greater importance when entered without a preceding line so that they are obeyed immediately. They may be used in a program but this use is very limited.

RUN - The RUN command will clear all existing variables and execute the current program stored. A line number may be specified in which case the execution will commence from that line.

syntax: RUN [**<line number>**]

CONT - The CONTinue command will restart the execution of a program which has terminated from the position where it had stopped. In the case that the program had been stopped due to an error, CONT will cause the error report 'CAN'T CONTINUE' to be given.

syntax: CONT

NEW - The NEW command will delete all existing variables and any programs from the computer.

syntax: NEW

LIST - The LIST command enables the program currently stored in the computer to be displayed on the screen. When the program contains more information than can be displayed, the screen will scroll upwards; the scrolling may be slowed down by pressing down **CTRL**. The LISTing may be terminated by **RUN**
STOP

syntax: LIST [**<first line>**] - [**<last line>**]

It is possible to restrict the LISTing to display just a sub-section of the program stored; this is done by specifying the first line required followed by '-' and then the last line required. If the first line is omitted, control will list from the first line present, likewise if the last line is omitted control will list until the last line.

OTHER SYSTEM COMMANDS

The three remaining commands, LOAD, SAVE and VERIFY are used for communicating with either the cassette interface or disc drives; we shall meet them again later and study them in more depth.

ASSIGNMENT STATEMENTS

LET - The LET statement is used to assign either a numerical value to a numerical variable or a sequence of characters to a string variable. It is important to remember that the original contents of the variable will be overwritten and lost for ever.

syntax: LET **<variable>** = **<expression>**

Note that the omission of the word LET, which would use valuable memory space, is perfectly valid.

examples:-

(i) 10 LET A = 99

A value of 99 is assigned to variable 'A'

(ii) 10 LET MZ = 1.25E-3

A value of 0.00125 is assigned to variable 'MZ'

(iii) 10 COUNT = COUNT+1

A value of 1 is added to the value stored in variable 'COUNT' and the result is assigned back to variable 'COUNT'. Remember that in Commodore 64 BASIC variables are only significant to two character positions. If the variable 'COUNT' did not previously exist the computer will assume that it had a value zero.

(iv) 10 LET C\$ = "COMMODORE 64"

The sequence of characters "COMMODORE 64" is assigned to variable C\$.

(v) 10 N\$ = "KEY" + "BOARD"

The sequence of characters "KEYBOARD" is assigned to the variable N\$. When strings are added together in this fashion it is called concatenation. But, remember that strings may not be subtracted, multiplied, divided or raised to a power.

Note that, before being used, numeric variables can be assumed to be zero, and string variables to be null.

READ/DATA - The READ statement is another method used for assigning values to variables and is particularly useful if an identical long list of data is required every time a program is executed. The READ statement is followed by a list of variables. The values or characters that are assigned to these variables are written in a matching list preceded by the DATA statement. Any number of such DATA statements are allowed and these may appear anywhere in the program. Before the program is executed, the computer searches for all the DATA statements and then remembers them as if they were all joined together in one long line. During program execution, when the computer comes across a READ statement followed by a list of variables, it takes as many items from the start of the DATA statement as are required and then assigns them in turn to the listed variables. Any subsequent READ statements will start from the position in the DATA statement where the preceding READ statement left off.

syntax: READ <variable> [,<variable>] ...

DATA <data item> [,<data item>] ...

For example

(i) 10 DATA 1,1,2,5,8,13
 15 READ F1,F2,F3,F4,F5,F6,F7

The values 1,1,2,3,5,8 and 13 are assigned to variables F1,F2,F3,F4,F5,F6 and F7 respectively.

(ii) 10 DATA A,B,C
 15 READ A\$,B\$,C\$

The characters A, B and C are assigned to variables A\$, B\$ and C\$ respectively. If the characters to be assigned include punctuation symbols, cursor control characters or spaces then the string should be enclosed within quotation marks (".....")

It is very important that the DATA statement contains the correct type of expression for the type of variable that the READ statement requires. Consider for example:-

10 DATA 20, "THE AVENUE"
 15 READ A\$,A

This would cause an error report "?SYNTAX ERROR" to be displayed because the second variable expects the item of information in the DATA statement to be numeric. Note, however, that the string variable can accept the digits 20 to be characters rather than numbers.

One final consideration with the READ and DATA statements is what happens when the items in the DATA statements are all used up and a READ statement still has variables that have not been assigned. Try the following example:-

10 DATA 1,2,3
 15 READ a,b,c,d

Here we require four items of information but have only specified three items in the DATA statement. Such an event will cause the error report "?OUT OF DATA" to be displayed.

RESTORE - By using the RESTORE statement we have the ability to re-

READ the information stated in the DATA statements. On execution, RESTORE will set the computer to recommence READING from the first item of information in the first DATA statement.

syntax: RESTORE

example:-

```
5 DATA 1,2,3,4
10 READ A1, A2, A3, A4
15 RESTORE
20 READ B1, B2, B3, B4
```

The values 1,2,3 and 4 are first assigned to variables A1, A2, A3 and A4 respectively. The identical data is then assigned to variables B1, B2, B3 and B4 respectively.

This concludes the section on assignment statements that are available on the Commodore 64, although we shall see later that values may also be assigned to variables using certain 'input' statements. In the next section of statements we shall learn how to enter information into the computer and how to access the processed results by using the input and output statements.

INPUT & OUTPUT STATEMENTS

PRINT - The PRINT statement which is used to display information on the television screen is probably the most used I/O statement. It is followed by the print items which are either variables, arithmetic expressions, strings of characters or control characters, which as we shall see later affects the position on the screen and the colour of the information that is displayed on the screen.

syntax: PRINT [<print item>] [<,or;> <print item>] ...

examples:-

(i) 10 PRINT A

The value of variable 'A' is displayed.

(ii) 10 PRINT A\$

The value of variable 'A\$' is displayed.

(iii) 10 PRINT "AVERAGE"

The sequence of characters 'AVERAGE' is displayed.

(iv) 10 PRINT 2*5+4

The expression '2*5+4' is evaluated and displayed.

(v) 10 PRINT

The absence of any print items causes a blank line to be displayed.

It is possible to include several print items within one PRINT statement by separating them with either a comma, semicolon or space. Their corresponding effect is:-

(i) , comma - The television screen used for output may be visualised as consisting of several eighty character lines, each line consisting of eight print zones of ten characters. A comma separating two print items causes the second item to be displayed at the first position in the next print zone.

(ii) ; semicolon - A semicolon separating two print items causes the second item to be printed immediately after the preceding item.

(iii) space - A space separating two string constants or variables will have the same effect as a semicolon. Care should, however, be taken when used between a string and a numeric item as this will cause the output of data to cease.

It should be noted that during the PRINTing of numerical data, all positive values are preceded by a space and all values are followed by an additional space.

example:-

(vi) 10 PRINT "AVERAGE";(x+y+z)/3

The word AVERAGE is displayed followed by the result of the expression.

(vii) 10 PRINT A,B

The values of variables A and B are displayed at the start of print zone 1 and 2 respectively.

(viii) 10 PRINT 1;2;3;4

The digits 1,2,3 and 4 are displayed '1234'. What would happen if the semicolons were exchanged for commas? - try it and see.

We shall later learn how to use control characters within strings to control both the colour and the position on the screen of the information PRINTed.

INPUT - The INPUT statement enables us to enter either numerical data or strings of characters and then assigns them to a specified variable. When

the INPUT statement is encountered, the execution of the program halts until all the information that the computer requires has been entered through the keyboard. The computer indicates that it requires data by displaying a '?' symbol - the data can then be typed in and is entered by pressing RETURN.

syntax: INPUT ["?" < message > "?" ;] < variable list >

If more than one variable is required then they should be separated by commas.

example:-

(i) 10 INPUT X

The program is halted and a request made for numerical data. When the data has been entered the value will be assigned to variable X. If X already exists, its previous value is overwritten and so lost.

(ii) 10 INPUT A,B,C

Three items are required by the program to be entered; these may be entered on one line separated by commas or on separate lines. In the latter case the '??' message is given at the start of each line to show that more data is expected. On the other hand, if too much data is entered the error report '?EXTRA IGNORED' is given, and the additional data made redundant.

(iii) 10 INPUT "PROMPT ";Q\$

The message 'PROMPT' is displayed and control expects a string of characters to be entered. Strings that contain the 'comma' character require the text to be entered within quotation marks.

If invalid data is entered, such as character text instead of numerical data, the error report '?REDO FROM START' is given and the '?' prompt reappears. Remember that numerical data can be accepted when string data is required since the digits may be accepted as characters.

It is important to remember, especially when entering text, that when **RETURN** is pressed only two screen lines are read - thus, at the most eighty characters may be INPUT at a time.

GET - An excellent feature of the Commodore 64 is the ability to use the GET statement to sense which keys if any have just been pressed during the execution of the program.

syntax: GET <variable list>

As keys are pressed during program mode, they are stored in a section of memory called the keyboard buffer. A maximum of ten keys can be stored at any one time.

When the GET statement is executed, characters are read from the keyboard buffer and assigned to the variables in the variable list; as a key is read it is removed from the buffer, releasing space for more keys to be stored. If a numeric variable is specified and non numeric data is read from the keyboard buffer, the error report 'SYNTAX ERROR' is displayed. The solution is to read all data as strings and then, if necessary, convert the characters to digits.

The input of information using GET has the advantage over INPUT in that it does not have to be followed by RETURN; but remember, unlike INPUT, it will not wait for you to press a key.

example:-

(i) 5 GET Q\$

The first key present in the keyboard buffer is assigned to Q\$.

(ii) 10 GET A\$, B\$, C\$

Three keys are read from the keyboard buffer and assigned to A\$, B\$ and C\$.

OTHER I/O STATEMENTS

Discussion of the six remaining I/O statements, i.e. OPEN, CLOSE, CMD, PRINT#, GET# and INPUT# will be left until later where they will be featured in detail. Their use is to enable the Commodore 64 to communicate with external devices such as the cassette interface, disc drives, printer etc.

Probably some of the most vital statements in a computer's repertoire are those that allow us to control the order that the instructions in a program are carried out. We shall now examine the 'control statements' present on the Commodore 64.

CONTROL STATEMENTS

GOTO - By using the GOTO statement it is possible to alter the usual sequence in which the numbered statements in a program are obeyed.

syntax: GOTO <line number>
or
GO TO <line number>

When the GOTO statement is executed, control jumps to the line whose number is specified. If this line is absent from the program, the error report 'UNDEF'D STATEMENT' is given.

examples:-

(i) 10 GOTO 500

Control jumps to line 500.

(ii) 10 GOTO 10

Here we have a continual loop which will go on for ever. The only way of stopping this is to press **RUN**
STOP

While in command mode, it is possible to execute a program by omitting a line number and pressing:- GOTO n where n is the line number from where you wish to commence. This method differs from the system command RUN in that GOTO does not clear the variables before execution.

MAGIC CARPET

We shall now take a short break from learning about the available BASIC keywords and demonstrate some of the statements and system commands that we have seen. Our first example is a simple three line program which can produce some fascinating patterns on the television screen.

Program 1 - MAGIC CARPET

```
10 REM *****
15 REM * MAGIC CARPET *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 INPUT G$
45 PRINT G$;
50 GOTO 45
```

After typing in the program and executing it by using RUN, enter between three and nine characters (you may use numbers and letters but the graphic characters shown at the front of the keys usually produce more interesting results). The string of characters is assigned to variable G\$ which is then continually printed on the screen. When the screen has become full press **RUN STOP** to stop the program.

Entering four or eight characters will cause vertical patterns to be displayed while entering any other combination causes diagonal patterns.

MULTIPLICATION

Program 2 will display the complete multiplication table for any entered value. When the screen becomes full, the display will scroll up one line as each result is given. To stop the program press **RUN STOP**

Program 2 – MULTIPLICATION

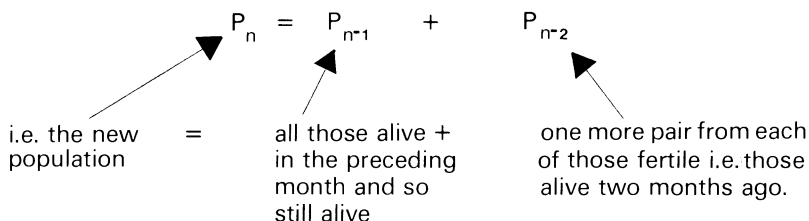
```
10 REM *****
15 REM * MULTIPLICATION *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 INPUT "ENTER VALUE " /NUMBER
45 COUNT=0
50 COUNT=COUNT+1
55 RESULT=NUMBER*COUNT
60 PRINT NUMBER;"*";COUNT;"=";RESULT
65 GOTO 50
```

THE FIBONACCI SEQUENCE

Program 3 calculates a mathematical sequence of numbers known as the 'Fibonacci Numbers'. In the thirteenth century, an Italian mathematician Leonardo Fibonacci studied the vast explosion in the population of rabbits. He considered how the population grew, starting with just one pair of rabbits and assumed that it took a pair of newborn rabbits one month before they became fertile and could reproduce. From then on, they

would breed an additional pair each month after. It is assumed that no deaths occur in the region of time considered.

Analysing the problem it can be seen that if the population of month n is P_n , then:



It is also known that $P_1 = 1$ and $P_2 = 1$

The values of P_n give the 1,1,2,3,5,8,13,21,34,55 and this is known as the Fibonacci sequence.

Program 3 - FIBONACCI SEQUENCE

```

10 REM *****
15 REM *   FIBONACCI   *
20 REM *=====*
25 REM * MRH   1/7/1983 *
30 REM *****
35 REM
40 PRINT "THE FIBONACCI SEQUENCE."
45 P1=1
50 PRINT "MONTH 1 ";P1;"RABBIT "
55 P2=1
60 PRINT "MONTH 2 ";P2;"RABBIT "
65 M=2
70 P=P1+P2
75 M=M+1
80 PRINT "MONTH";M;P;" RABBITS"
85 P1=P2: P2=P
90 GOTO 70

```

MORE ADVANCED CONTROL STATEMENTS

FOR....NEXT

The FOR and NEXT statements are used when we require a section of program to be repeated a specified number of times. When the program flow causes a section of code to be repeated like this - it is called a loop.

```
syntax : FOR <variable> = initial value TO <terminating value>
          [STEP <increment>]
          <section of code to be looped>
          NEXT [<variable list >] , .....
```

The looping is controlled by assigning an initial value to a numerical variable, known as the control variable, and then executing the section of code until the statement NEXT is found. Control then jumps back to the corresponding FOR statement and the control variable is incremented by the value specified after STEP.

If the STEP value is positive then the looping will continue until the control variable is greater than the terminating value in which case control continues after the NEXT statement. Similarly if the STEP value is negative then the looping will continue until the control variable is less than the terminating value.

examples:-

(i)

```
10 FOR J = 1 TO 20 STEP 1
15 PRINT J
20 NEXT J
```

The numbers 1 to 20 are displayed.

(ii)

```
10 FOR J = 20 TO 1 STEP -1
15 PRINT J
20 NEXT J
```

The numbers 20 down to 1 are displayed.

If the STEP value is +1 then the FOR statement may be simplified to

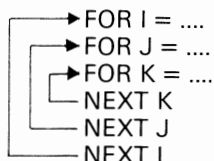
FOR <variable> = X TO Y
and a STEP value of +1 is assumed by default.

FOR loops may be nested inside each other up to a maximum of nine levels providing they do not cross each other.

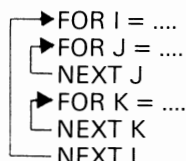
These are perfect examples:-



```
→FOR I = ....  
NEXT I
```

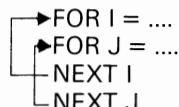


```
→FOR I = ....  
  →FOR J = ....  
    →FOR K = ....  
      NEXT K  
    NEXT J  
  NEXT I
```



```
→FOR I = ....  
  →FOR J = ....  
    →FOR K = ....  
      NEXT K  
    NEXT J  
  NEXT I
```

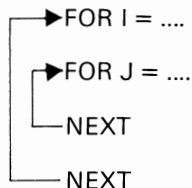
This one, however,
would cause all
sorts of problems
and so should be
avoided.



```
→FOR I = ....  
  →FOR J = ....  
    NEXT I  
  NEXT J
```

If the control variable in a NEXT statement is omitted, the control variable with a FOR statement at the corresponding level is assumed.

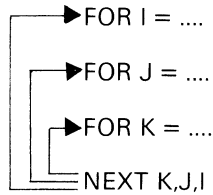
example:-



```
→FOR I = ....  
  →FOR J = ....  
    NEXT  
  NEXT
```

It is possible to use a single NEXT statement to increment several control variables; the names of the variables should be separated by commas and listed in the order of the innermost nested loop first to the outermost nested loop.

example:-



CUBES

To illustrate FOR NEXT loops, Program 4 calculates the cubes (i.e. raised to the power of 3) of the first 20 integers; it does however use a very strange method to do so. The results can be checked with the $\uparrow$ operator. To demonstrate how the program works, write down the first few odd numbers:

1,	3,	5,	7,	9,	11,	13,	15,	17,	19,	
1	8		27			64				

Notice that

- 1 raised to the power of 3 is the sum of the first one number.
- 2 raised to the power of 3 is the sum of the next two numbers.
- 3 raised to the power of 3 is the sum of the next three numbers.

and so on.

Program 4 - CUBES

```
10 REM *****
15 REM *      CUBES      *
20 REM *=====*
25 REM * MRH   1/7/1983 *
30 REM *****
35 REM
40 PRINT " CUBES."
45 M=1
50 FOR X=1 TO 20
```

```
55 T=0
60 FOR Y=1 TO X
65 T=T+M
70 M=M+2
75 NEXT Y
80 PRINT X;"↑ 3 =" ;T
85 NEXT X
```

Both the GOTO and the FOR....NEXT control statements execute a chunk of BASIC unconditionally. Here is a more powerful statement.

IF...THEN...

The IF statement enables the computer to become 'intelligent' and able to test conditions on which depending on results, different actions may be undertaken.

syntax : IF condition or THEN <line number> or GOTO <line number>	THEN <statements>

IF the <condition> is found to be 'true' then the following statement (or statements, separated by :) are executed, otherwise control continues the program from the next present line. If the condition is 'true' and a line number is stated, control will jump to that particular line.

The condition statement consists of either two algebraic expression or two string expressions separated by one of the following symbols:-

SYMBOL	MEANING
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal to

examples:-

- (i) $A + B = C/2$
Is A plus B equal to one half of C?
- (ii) $A <> 69$
Is A not equal to 69?
- (iii) $X > 3 * Y$
Is X greater than 3 times Y?
- (iv) $Q\$ = \text{"YES"}$
Is Q\$ the same as the string "YES"?

Note - String expressions must never be compared with algebraic expressions.

When comparing strings, a string is found to be less than another if it comes first in alphabetical order.

When lower case characters are available, lower case comes before upper case characters.

Digit characters come before all alphabetic characters.

examples:-

"A" < "B"	"a" < "A"
"AA" < "AZ"	"5" < "FIVE"
"A□" < "AA"	"6" < "six"
"□A" < "A□"	"six" < "SEVEN"
"1" < "g"	"Photo" < "Photos"

We shall use this information later to write an alphabetic sorting program.

GOSUB ...RETURN

The GOSUB and RETURN statements enable us to re-use portions of BASIC several times. Consider a program where a section of code is required to be repeated several times; instead of continually reproducing identical statements they may be written once and called for when required. Such a section of code is called a subroutine.

syntax: GOSUB line number

RETURN

To implement a subroutine we use a GOSUB <line number> where the line number of the subroutine is specified. At the end of a subroutine RETURN must appear: this tells control to return to the line immediately after the line which called the subroutine.

example:-

```
10 PRINT "MAIN PROGRAM"
15 GOSUB 100
20 PRINT "MAIN PROGRAM"
25 GOSUB 100
30 PRINT "MAIN PROGRAM"
35 GOSUB 100
40 GOTO 10
100 PRINT "SUBROUTINE CALLED"
105 RETURN
```

} Main Program

} Subroutine

In this simple example the subroutine starting at line 100 is called three times. Note that it is very important never to let the program run through a subroutine by accident. In the example show, a GOTO statement has been inserted at line 40 to avoid this.

A subroutine may also call another subroutine. A subroutine that calls itself is said to be recursive.

SIMULTANEOUS EQUATIONS

Program 5 is able to solve the pair of simultaneous equations:-

$$\begin{aligned} aX + bY &= c \\ dX + eY &= f \end{aligned}$$

where a,b,c,d,e and f are known constants and X and Y are required to be found.

It can be proved mathematically that the solutions are

$$X = \frac{(f * b - c * e)}{(b * d - a * e)} \quad \text{and} \quad Y = \frac{(c * d - a * f)}{(b * d - a * e)}$$

These results can easily be written as a program but care has to be taken when the situation $(b * d - a * e) = 0$ arises since a number may not be divided by zero.

Program 5 - SIMULTANEOUS EQUATIONS

```

10 REM *****
15 REM * SIMULT. EQNS. *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT "SIMULTANEOUS EQUATIONS."
45 PRINT
50 INPUT "EQUATION ONE (A,B,C) :";A,B,C
55 INPUT "EQUATION TWO (D,E,F) :";D,E,F
60 PRINT
65 PRINT A;"X +";B;"Y =" ;C
70 PRINT D;"X +";E;"Y =" ;F
75 G=B*D-A*E
80 IF G=0 THEN PRINT "NO SOLUTION EXISTS": GOTO 100
85 PRINT
90 PRINT "SOLUTION"
95 PRINT "X=";(F*B-C*E)/G,"Y=";(C*D-A*F)/G
100 PRINT
105 INPUT "STOP ? (Y/N) ";Q$
110 IF Q$="Y" THEN END
115 PRINT
120 GOTO 40

```

ON

The ON statement is used to branch to different positions in a program depending on the value of an algebraic expression.

syntax : ON expression GOTO <line number> [,<line number>]
or
GOSUB

The algebraic expression is evaluated and control is passed to the line number whose position in the list is the integer component of the expression.

example:-

10 ON X GOTO 50, 60, 70

If X = 1 then control continues at line 50

If X = 2 then control continues at line 60

If X = 3 then control continues at line 70

If the value of the expression is zero or greater than the number of items in the list, control will proceed from the next line after the ON statement. If the value of the expression is negative the error report ' ? ILLEGAL QUANTITY ' is given.

And finally:

OTHER STATEMENTS

DIM

It is often required to reserve a block of cells in the memory which can be referred to by a name (referring to the block) and a number (referring to the particular cell in the block); such a block is called an array. Control is instructed to reserve blocks of a particular length by the DIM (dimension) statement.

syntax : DIM variable (<subscript list>) , [<variable>] (<subscript list>)]

Consider, for example

10 DIM A(8)

This instructs the computer to reserve an array called 'A' consisting of nine elements.

A(0)	A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)

Each cell can store one number. When the block is declared each cell is initialized to zero. We may refer to any cell by the variable name followed by the cell number in parenthesis. For example cell 5 is referred to as A(5). The term in brackets is called the subscript. The same cell could be referred to by A(variable) if the variable had been set equal to 5. Also A(algebraic expression) is valid such as A(2*X+6) but the result of the

expression must lie in the range specified by the DIM statement or the error report '? BAD SUBSCRIPT' will be given.

It is also possible to set up arrays with more than one subscript.

example:

```
10 DIM B(3,4)
```

B(1,1)	B(1,2)	B(1,3)	B(1,4)
B(2,1)	B(2,2)	B(2,3)	B(2,4)
B(3,1)	B(3,2)	B(3,3)	B(3,4)

Such an array is called a two dimensional array; remember however that the cells are physically stored sequentially in the computer's memory.

The same idea works for an n dimensional array.

The DIM statement must only be used once for each variable otherwise an error report '? REDIM'D ARRAY' is given. If an array is referenced in a program without it ever being dimensioned then it is assumed to have been dimensioned to length 11 by default.

If the array variable is followed by either a % or \$ sign, then, as usual the variables are either integers or strings respectively.

DEF FN

An explanation of functions and DEF FN, which is a user-defined function will be delayed until later when a complete chapter will specialize on this topic.

CLR

The CLR statement releases all the memory that had been reserved during the execution of a program. The actual program code is left untouched but all the variables, arrays, subroutine and FOR statement addresses, files etc. are deleted.

syntax : CLR

END

The END statement ceases the execution of a program and displays the

'READY' message. Execution can be resumed from the terminating point by CONT.

syntax : END

REM

REMark statements are not executed by the computer but are used to clarify sections of a program to a reader.

syntax : RED comment

The comments may contain any characters as control will ignore everything following REM to the end of the line.

POKE

The POKE statement enables us to store specific values directly into memory cells in the computer.

syntax : POKE <numeric 1> , <numeric 2>

The statements enable two algebraic expressions to be evaluated on which the result of the second expression is stored at the address given by the first expression. The value stored must be within the range 0 to 255 or the error report '? ILLEGAL QUANTITY' is given. We shall see later that this is a very powerful statement.

Try this next example on your computer.

example:

```
10 FOR I = 0 TO 15
15 REM DELAY
20 FOR J = 1 TO 500:NEXT J
25 POKE 53281,I
30 NEXT I
```

Here we are changing the value stored at address 53281. By executing the program you will see what the data stored at this particular location tells the computer system. Using the same technique, what does location 53280 hold?

STOP

The STOP statement ceases the execution of a program and displays the report 'BREAK IN LINE XXXXX' where the line number of the STOP statement is displayed. Execution may be resumed from the terminating

point by CONT.

syntax : STOP

SYS

On the Commodore 64 it is possible to execute a machine code program (see Chapter 1) directly instead of using the BASIC interpreter. When the SYS statement is executed, control will execute the machine code program from the address specified.

syntax : SYS address

WAIT

The WAIT statement is used to temporarily halt the execution of a program until the contents at a specified address are altered. This use is most important in I/O operations.

syntax : WAIT address , mask 1 , mask 2

To understand the use of WAIT, the reader will require a knowledge of logical operations which are not explained until later. For reference, WAIT will cause program execution to temporarily halt until the contents of the specified address exclusively-ORed with mask 2 (if present) and ANDed with mask 1 become zero.

CHAPTER 7

THE COMMODORE 64 FUNCTIONS

Briefly a function can be described as a rule which relates one set of values to another. The value in the first set is known as the argument and the value in the second set is the result. The Commodore 64 has the following set of functions available:

ABS
AND
ASC
ATN
CHR\$
COS
EXP
FN
FRE
INT
LEFT\$

LEN
LOG
MID\$
NOT
OR
PEEK
POS
RIGHT\$
RND
SGN
SIN

SPC
SQR
STATUS
STR\$
TAB
TAN
TIME
TIME\$
USR
VAL

Each one of these functions can be placed in one of the following categories:-

(i) Simple numeric and mathematical functions - includes random number generator and specialized functions which have numerous applications in mathematics, science and engineering.

(2) System functions - used for finding out information about the state the computer system is in.

(3) String functions - used for manipulating characters and text.

(4) Logical functions - used to link conditional statements like those found in the IF statement. We shall also see how to use them to manipulate the patterns that are made when numbers are stored in the computers memory.

SIMPLE NUMERIC FUNCTIONS

INT - The INT function returns the integer component of the argument by rounding down.

syntax: INT <numeric>

example:-

(i) 10 PRINT INT(-5.6), INT(0), INT (5.6)
This will cause -6, 0 and 5 to be displayed.

HIGHEST COMMON FACTOR

Program 6 demonstrates the use of the INT function by finding the highest common factor of two positive integers i.e. it finds the largest number that can be divided into both number exactly. Try first to follow the algorithm on paper by drawing three columns labelled A, B and C with an initial value in each. Follow the program and record the value of each variable in the corresponding column as it changes.

Program 6 - HIGHEST COMMON FACTOR

```
10 REM *****
15 REM *   H.C.F.   *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 PRINT :PRINT " HIGHEST COMMON FACTOR " :PRINT
45 INPUT " ENTER FIRST NUMBER ";A
50 INPUT " ENTER SECOND NUMBER ";B
55 IF A<1 OR B<1 THEN 40
```



```

60 C=B :B=A-B*INT(A/B) :A=C
65 IF B<>0 THEN 55
70 PRINT :PRINT " H.C.F. IS ":C
75 PRINT :PRINT " PRESS SPACE BAR TO CONTINUE"
80 GET K$ :IF K$=" " THEN 40
85 GOTO 80

```

RND - The RND function is a RaNDom number generator which is extremely useful for certain games. The generator is not truly random but a very long sequence of numbers; this, however, is more than adequate for most purposes. The position in the sequence is dependent on a value stored in the memory called the seed. If the seed is reset to the same number each time RND is called, then the same random number will be returned. If the seed is set to an unknown value at the beginning of the program then the sequence of generated numbers will commence at an unknown position. This method of creating numbers is known as a pseudo random number generator.

syntax: RND (< numeric >)

On the Commodore 64, RND returns a value between 0 and 1.

If the argument of RND is positive the number returned will be the next number in the sequence, while negative values re-seeds the generator. However, if the argument is zero then the seed will be 'randomised' to the value of the hardware clock which we shall meet later.

example:-

(i) 10 LET A = RND(0)

The seed is set to the value of the clock and a random number between 0 and 1 is assigned to A.

DICE DISTRIBUTION

Program 7 represents a dice being rolled 100 times. This is achieved by obtaining a random integer between 1 and 6 by using $\text{INT}(\text{RND}(0)*6+1)$. The program also calculates the average and variance (a mathematical value for how wide the distribution is).

Program 7 - DICE

```
10 REM *****
15 REM *      DICE      *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 PRINT :PRINT " DICE DISTRIBUTION " :PRINT
45 S=0 :SS=0
50 FOR X=1 TO 100
55 D=INT(RND(0)*6+1)
60 S=S+D :SS=SS+D*D
65 PRINT D;" ";
70 NEXT X
75 A=S/100
80 PRINT :PRINT " AVERAGE " ;A
85 V=SS/100-A*A
90 PRINT :PRINT " VARIANCE " ;V
95 PRINT :PRINT " PRESS SPACE BAR TO REPEAT"
100 GET K$ :IF K$=" " THEN 40
105 GOTO 100
```

MATHEMATICAL FUNCTIONS

The Commodore 64 has a number of functions available that are commonly used in mathematical and scientific applications; some readers may however find them far too specific for their needs.

(i) EXP - EXP(x) is equal to a constant 'e' raised to the power of x where e is defined by mathematicians as 2.718281828. To explain how such a curious number was chosen - consider what happens to $(1 + 1/n)^n$ as n gets very large (mathematicians would say as n tends to infinity and write it as $n \rightarrow \infty$). By executing Program 8 we see that $(1 + 1/n)^n$ tends to this number e as defined above as n tends to infinity. Note that the Commodore 64 may introduce inaccuracies as n becomes very large.

i.e. written mathematically

$$\lim_{n \rightarrow \infty} (1 + 1/n)^n = e$$

It turns out a more general result is obtained from the expression $(1 + x/n)^n$

This is found to tend to e^x as n tends to infinity.

$$\lim_{x \rightarrow \infty} (1 + x/n)^n = e^x$$

syntax: EXP (<numeric>)

Program 8 - E.

```
10 REM *****
15 REM * EXPONENTIAL E *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT " EXPONENTIAL E " :PRINT
45 FOR X=0 TO 5
50 N=10↑X
55 E=(1+1/N)↑N
60 PRINT N;TAB(10);E
65 NEXT X
```

A graph of e^x plotted against x would look like

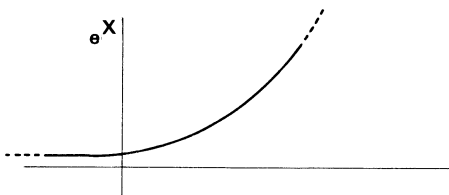


Figure 7.1

(ii) LOG - The LOG function produces the natural logarithm of a number. The logarithm of a number is the power needed to produce that number by raising the base to the power.

syntax: LOG (<numeric>)

example:-

$$\text{LOG} (25) = 3.21887583$$

$$(\text{because } e^{3.21887583} = 25 = 25)$$

From natural logarithms it is possible to find the logarithm in any base using the following relationship

$$\text{Log}_s (x) = \frac{\text{LOG} (x)}{\text{LOG} (s)}$$

If s is 10 then the resulting logarithms are called common logarithms.

example:-

$$\text{Log}_{10} (100) = \frac{\text{LOG} (100)}{\text{LOG} (10)} = \frac{4.60517}{2.30258} = 2$$

(iii) SQR - The SQR function returns the square root of a number, i.e. if the argument is positive the function returns the number that when multiplied by itself is equal to the argument. If the argument is negative the error report 'ILLEGAL QUANTITY' is given.

$$\text{SQR} (x) = \begin{cases} \sqrt{x} & \text{if } x \text{ is greater than or equal to } 0, \text{ but} \\ & \text{error if } x \text{ is less than } 0 \end{cases}$$

syntax: SQR (<numeric>)

iv) ABS - The ABS function returns the absolute value of a number, this has the same magnitude as the argument but with a positive sign.

$$\text{ABS} (x) = \begin{cases} x & \text{if } x \text{ is greater than or equal to } 0 \\ -x & \text{if } x \text{ is less than } 0 \end{cases}$$

syntax: ABS (< numeric >)

(v) SGN - The SGN function returns the signum of a number, this is +1 if the argument is positive, -1 if the argument is negative or 0 if the argument is zero.

$$\text{SGN}(x) = \begin{cases} +1 & \text{if } x \text{ is greater than } 0 \\ 0 & \text{if } x \text{ is zero} \\ -1 & \text{if } x \text{ is less than zero} \end{cases}$$

syntax : SGN (< numeric >)

QUADRATIC EQUATIONS

Program 9 illustrates the square root function, SQR, in use. A quadratic equation is one of the form:

$$ax^2 + bx + c = 0$$

where a,b and c are known constants and x is required to be found. It can be proved mathematically that the solutions are

$$x = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}$$

If $b^2 - 4 * a * c$ is greater than zero then the expression within the square root can be evaluated and two solutions of x can be found. These are known as real solutions.

If $b^2 - 4 * a * c$ is equal to zero then the square root part does not exist and there is only one solution of x. This is also a real solution.

If $b^2 - 4 * a * c$ is less than zero then the square root of the expression does not exist. Mathematicians say that these solutions are complex and that the real part is $-b/(2*a)$ and the imaginary part is $\sqrt{-(b^2 - 4 * a * c)} / (2 * a)$ which are expressions which can be evaluated. Note that complex solutions are of no importance to the operation of the Commodore 64.

Program 9 will solve a set of quadratic equations.

Program 9 - QUADRATIC EQUATIONS

```
10 REM *****
15 REM * QUADRATIC EQNS *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT :PRINT " QUADRATIC EQUATIONS " :PRINT
45 INPUT " ENTER FIRST COEFFICIENT " :A
50 INPUT " ENTER SECOND COEFFICIENT " :B
55 INPUT " ENTER THIRD COEFFICIENT " :C
60 PRINT :PRINT A;"X^2 +" :B;"X +" :C :PRINT
65 D=B*B-4*A*C
70 IF D<0 THEN 95
75 PRINT " REAL ROOTS " :PRINT
80 PRINT " 1. X= " :(-B+SQR(D))/(2*A)
85 PRINT " 2. X= " :(-B-SQR(D))/(2*A)
90 GOTO 110
95 PRINT " COMPLEX ROOTS " :PRINT
100 PRINT " REAL PART   = " : -B/(2*A)
105 PRINT " COMPLEX PART = " :SQR(-D)/(2*A)
110 PRINT :PRINT " PRESS SPACE BAR TO REPEAT "
115 GET K$ :IF K$=" " THEN 40
120 GOTO 115
```

PRIME NUMBERS

A prime number is an integer that is only divisible exactly by 1 and itself. Program 10 prints out the first 100 prime numbers by assuming that the first two prime numbers are 2 and 3 and that all other prime numbers are odd.

The program tests to see if the odd number x is prime by dividing it by all the prime numbers less than $\sqrt{x}$ that have already been found; if in each case the remainder is non zero the number x is also prime. There is one point here that needs further explanation - why only divide by prime numbers previously obtained that are less than $\sqrt{x}$? Non prime numbers are not used since these would have at least two smaller prime numbers as factors. Numbers greater than $\sqrt{x}$ are also not used since if they were a factor then it means a factor less than $\sqrt{x}$ must also exist.

As each prime number is obtained it is stored in the array P(100) so that it can be used in testing for future numbers. Because the Commodore 64's arithmetic is only accurate to $9 \frac{1}{2}$ significant figures, a number is assumed to be a factor if it has a remainder of less than $1E-10$.

Program 10 - PRIME NUMBERS

```
10 REM *****
15 REM * PRIME NUMBERS *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT " THE FIRST 100 PRIME NUMBERS : " :PRINT
45 PRINT " PRIME NUMBER 1";TAB(18);"IS 2"
50 PRINT " PRIME NUMBER 2";TAB(18);"IS 3"
55 DIM P(100)
60 P(1)=2 :P(2)=3 :X=5 :N=3
65 FOR I=2 TO N
70 IF X/P(I)-INT(X/P(I)) < 1E-10 THEN 100
75 IF SQR(X) < P(I) THEN 85
80 NEXT I
85 P(N)=X
90 PRINT " PRIME NUMBER";N;TAB(18);"IS";P(N)
95 N=N+1
100 X=X+2
105 IF N<101 THEN 65
```

MORE MATHEMATICAL FUNCTIONS - TRIGONOMETRIC FUNCTIONS

The Commodore 64 has four trigonometric functions available, SIN (sine), COS (cosine), TAN (tangent) and ATN (arctangent).

Trigonometric functions are all about angles and this is where we encounter our first problem. The majority of people measure angles in degrees, where a circle is divided into 360 degrees, a right angle is 90 degrees and a straight line is 180 degrees. Mathematicians, however

prefer to measure angles with a larger unit called the radian where 1 radian is the angle subtended by an arc of unit length on a circle of unit radius. The Commodore 64 uses radians so that if you require your angles in degrees they will have to be converted.

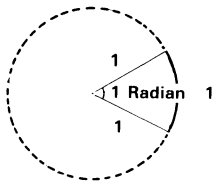


Figure 7.2

By definition there are two Pi radians in a circle where Pi is approximately 3.14159265358 and can be written, as on the Commodore 64, as π . Thus 1 radian is

$$\frac{360}{2\pi}$$

degrees which is about 57.3 degrees. To convert radians into degrees, multiply by $\frac{180}{\pi}$

$$\text{e.g. } \frac{\pi}{3} \text{ radians} = \frac{\pi}{3} * \frac{180}{\pi} = 60 \text{ degrees}$$

To convert degrees into radians multiply by $\frac{\pi}{180}$

$$\text{e.g. } 45 \text{ degrees} = 45 * \frac{\pi}{180} = \frac{\pi}{4} \text{ radians}$$

Pi is accessible on the Commodore 64 direct from the keyboard using



We shall now look at the four functions. Consider the following right angle triangle with an angle of X radians and the sides labelled opposite, adjacent and hypotenuse.

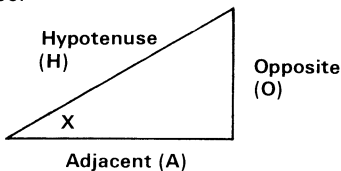


Figure 7.3

Sine, cosine and tangent are defined as the ratio of lengths of certain sides to other sides.

SINE - The sine of angle X is defined as the ratio of the length of the opposite side to the length of the hypotenuse in any right angled triangle with an angle of X radians.

$$\text{SIN}(X) = \frac{O}{H}$$

A graph of $\text{SIN}(X)$ plotted against X would look like:

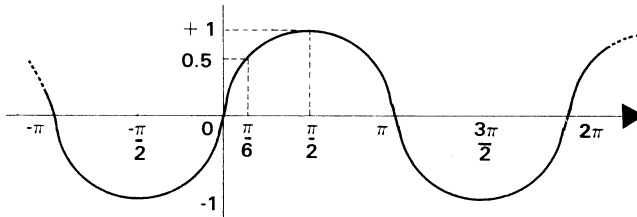


Figure 7.4 The Sine Wave

syntax: $\text{SIN} (< \text{numeric} >)$

COS - The cosine of angle X is defined as the ratio of the length of the adjacent side to the length of the hypotenuse in any right angled triangle with an angle X radians.

$$\text{COS}(X) = \frac{A}{H}$$

A graph of $\text{COS}(X)$ plotted against X would look like:

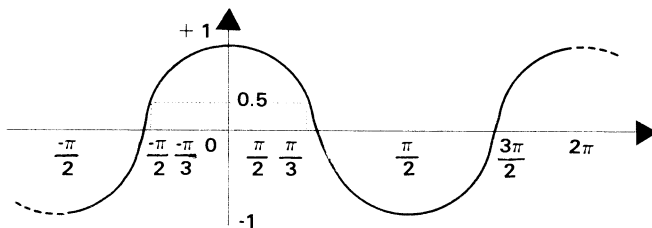


Figure 7.5 The Cosine Wave

syntax : COS (< numeric >)

TAN - The tangent of angle X is defined as the ratio of the length of the opposite side to the length of the adjacent side in any right angled triangle with an angle of X radians.

$$\text{TAN}(X) = \frac{O}{A}$$

A graph of TAN(X) plotted against X would look like:

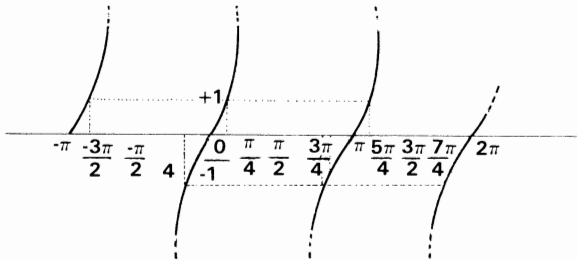


Figure 7.6 The Tangent Lines

syntax : TAN (< numeric >)

The remaining trigonometric function is:-

ATN - The arctangent of y turns the angle whose tangent is y.

syntax : ATN (< numeric >)

The arcsine and arccosine values, which are inverse sine and cosine functions respectively can be found using the following formulae

$$\text{Arcsine}(y) = \text{ATN} \left(\frac{y}{\text{SQR}(1 - y^2)} \right)$$

$$\text{Arccosine}(y) = \text{ATN} \left(\frac{y}{\text{SQR}(1 - y^2)} \right) + \frac{\pi}{2}$$

Note that in both cases y is always in the range $-1 \leq y \leq +1$

Program 11 calculates the angles of triangles where the lengths of all three sides are known. Consider the triangle below where lengths A, B and C are known.

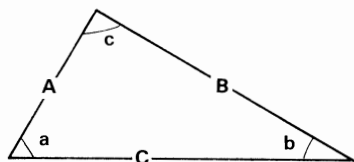


Figure 7.7

The following relationship is known to exist:

Rearranging the equation gives: $C^2 = A^2 + B^2 - 2.A.B. \cos(c)$

$$c = \arccos \left(\frac{A^2 + B^2 - C^2}{2 * A * B} \right)$$

Similar relationships exist for the remaining two angles. Program 11 allows the lengths of the three sides to be entered and then calculates the angles in degrees. Note that the sum of the three angles should be 180 degrees.

Program 11 - TRIANGLE

```

10 REM *****
15 REM *   TRIANGLE   *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT :PRINT " TRIANGLE ": PRINT
45 INPUT " ENTER SIDE ONE ";A
50 INPUT " ENTER SIDE TWO ";B
55 INPUT " ENTER SIDE THREE";C :PRINT
60 PRINT
65 X=(B*B+C*C-A*A)/(2*B*C)
70 AA=(PI/2-ATN(X/SQR(1-X*X)))*180/PI
75 X=(A*A-B*B+C*C)/(2*A*C)

```

```

80 BB=(π/2-ATN(X/SQR(1-X*X)))*180/π
95 X=(A*A+B*B-C*C)/(2*A*B)
90 CC=(π/2-ATN(X/SQR(1-X*X)))*180/π
95 PRINT " ANGLE OPPOSITE SIDE ONE ";AA
100 PRINT " ANGLE OPPOSITE SIDE TWO ";BB
105 PRINT " ANGLE OPPOSITE SIDE THREE";CC
110 PRINT
115 PRINT " SUM OF ANGLES ";AA+BB+CC
120 PRINT :PRINT" PRESS SPACE BAR TO REPEAT"
125 GET K$: IF K$=" " THEN 40
130 GOTO 125

```

USER DEFINED FUNCTIONS

In Commodore 64 BASIC it is possible to define your own functions by using the statement DEF FN.

syntax : DEF FN name (< dummy variable >) = <expression>

It is followed by a variable name - a single letter - which distinguishes it from any other function. After this comes a dummy variable which has no effect on any existing variable with the same name. This variable is called the argument of the function. Finally the function is defined by an expression which contains the dummy variable.

The defined function may then be called and used in the same fashion as any other numeric function on the Commodore 64 by the function FN.

syntax : FN <name> (< numeric >)

FN is followed by the name referring to the function that is required and the numeric value or "argument" of the function. If a user defined function is called before it has been defined the error report '? UNDEF'D FUNCTION' is given.

When a FN is called, the argument is substituted for the dummy variable in the DEF FN, all other variables mentioned remain the same. The value of the expression is then evaluated and returned.

examples:-

1) 10 DEF FN R(X) = INT(X + 1/2)

FN R(X) will return the value of X rounded to the nearest integer.

2) 10 DEF FN F(X) = X - INT(X)

If X is positive FN F(X) will return the fractional component of X.

3) 10 DEF FN A(X) = (A + B + C)/3

Provided that A, B and C have already been assigned values, FN A(X) will return their average value. Note that X is unused and so the function can be called with any argument.

4) 10 DEF FN D(X) = INT(RND(0)*6+1)

FN D(X) will return a random integer between 1 and 6 inclusive.

SYSTEM FUNCTIONS

FRE - It is possible to find how much RAM is available for your program and its variables at any stage by the function FRE. The argument of the function has no effect.

syntax : FRE (< dummy >)

Unfortunately, Commodore have written the system software so that if the result of FRE is negative, then 65535 has to be added to obtain the real value.

examples PRINT 65535 + FRE (0)
 or PRINT FRE (0)

STATUS - Returns a STATUS code for the last I/O operation. This will be explained in more depth later.

syntax : ST [ATUS]

PEEK - The PEEK function enables us to directly access memory cells in the computer and is often used in conjunction with the POKE statement; while these instructions may at first seem curious, understanding them comprehensively opens the way for some advanced computing techniques.

syntax : PEEK (< numeric >)

The function returns the value stored at the address specified.

example:-

```
10 PRINT PEEK (16678)
```

This displays the value stored at address 16678 in the computer's memory.

When we have started to study the Commodore 64's memory layout we will find the POKE and PEEK instructions invaluable.

We shall meet the remaining functions available at a later stage in the book.

CHAPTER 8

THE COMMODORE 64 CHARACTER SET & STRING HANDLING.

CHARACTER CODES.

The Commodore 64's character set can be visualized as its own unique alphabet consisting of 256 items which include alphabetic and numeric characters, pixel graphics, punctuation marks, mathematical symbols, control characters etc. Each item is distinguished by its own character code which is a unique number between 0 and 255.

The Commodore 64 has two functions that enable the user to convert from items in the character set to their character codes and vice versa.

ASC - When ASC is applied to a string it will return the character code of the first character in the string.

syntax: ASC (<string>)

example

i) 10 X = ASC ("ABCDE")

This assigns a value of 65 to 'X' because the character code of "A" is 65

CHR\$ - When CHR\$ is applied to a number in the range of 0 to 255 it will return the character whose code is that number.

syntax : CHR\$ (< string >)

example:-

(ii) 10 X\$ = CHR\$ (65)

This assigns the character "A" to 'X\$'

The complete set of Commodore 64 character codes is given in Appendix A.

Program 12 displays the Commodore 64's character codes with the exception of the control characters, which as we will see later, would affect the display.

Program 12 - CHARACTER SET

```
10 REM *****
15 REM * CHARACTER SET *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT " CHARACTER SET " :PRINT
45 FOR X=33 TO 129
50 PRINT CHR$(X);
55 NEXT X
60 FOR X=161 TO 255
65 PRINT CHR$(X);
70 NEXT X
```

Three more Commodore 64 string functions are:-

LEN - When applied to a string it returns the number of characters in that string.

syntax : LEN (< string >)

examples:-

LEN ("COMMODORE") would return 9

LEN ("") would return 0

STR\$ - When applied to a number it returns the number as a string of characters in the form that it would be printed.

syntax : STR\$ (< numeric >)

examples:-

STR\$ (12.01)	would return " 12.01"
STR\$ (-12.01)	would return "-12.01"
STR\$ (100.000)	would return " 100"
STR\$ (1000000000)	would return " 1E+09"

VAL - When applied to a string it returns the number in print format that exists starting at the left most character. Any unrecognized characters after the number are ignored. If the first character is unrecognized as a number then a value of 0 is returned.

syntax: VAL (<string>)

examples

VAL ("100") would return 100
VAL ("R101") would return 0

STRING HANDLING FUNCTIONS

Commodore 64 BASIC has three string handling functions which are invaluable for manipulating text and characters. These are LEFT\$, RIGHT\$ and MID\$ and are used for extracting a sub-string or slice from another string.

LEFT\$(A\$,N) and RIGHT\$(A\$,N) extract N characters from the left-hand end and right-hand end of string A\$ respectively.

example:-

if A\$ = "ABCDEFGHIK"

then LEFT\$(A\$,3) would return "ABC"

and RIGHT\$(A\$,4) would return "GHIJ"

syntax: LEFT\$ (<string>, <numeric>)
RIGHT\$ (< string > , <numeric >)

A more powerful statement is MID\$ which can be used to obtain slices from within the string, unlike LEFT\$ and RIGHT\$ which are limited to starting from one end of the string.

syntax: MID\$ (<string>, <numeric 1> [, >numeric 2>])

MID\$(A\$,N,M) will extract M characters starting at the Nth character

from string A\$. If the last argument M is omitted then all succeeding characters from the Nth are obtained.

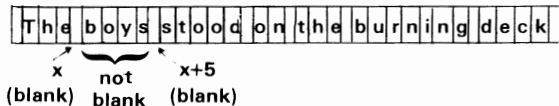
MARY WHITEHOUSE

Program 13 illustrates the use of slicing strings with the MID\$ function. A sentence of English can be entered and then displayed with all the four letter words blotted out.

A string containing at the most 80 characters is set up such that the INPUT statement only fills the inner characters, leaving the first and last blank.

A pointer X is set up to move along each character in turn. The computer first checks that character X is a space then that the next four characters are all characters and, finally, that the next character is another space. If it is found that all these conditions are true then a four letter word is present and so this subsection must be replaced by four stars. If it is found that not all the conditions are true then X is incremented by 1 and the next set of six characters is considered.

example:-



Program 13 - MARY WHITEHOUSE

```
10 REM *****
15 REM * MARY WHITEHOUSE *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT :PRINT " THE MARY WHITEHOUSE PROGRAM "
45 PRINT :PRINT " ENTER STATEMENT : "
50 INPUT A$ :A$=" "+A$+" "
55 L=LEN(A$)
60 FOR X=1 TO L-5
65 IF MID$(A$,X,1) <> " " THEN 95
70 FOR Y=1 TO 4
```

```

75 IF MID$(A$,X+Y,1) = " " THEN 95
80 NEXT Y
85 IF MID$(A$,X+5,1) <> " " THEN 95
90 A$=LEFT$(A$,X)+"*****"+RIGHT$(A$,L-X-4)
95 NEXT X
100 PRINT :PRINT A$
105 PRINT :PRINT " PRESS SPACE BAR TO REPEAT "
110 GET K$ : IF K$=" " THEN 40
115 GOTO 110

```

WORD SEARCHES

We can use these string functions to identify specific words or characters in a sentence. Program 14 illustrates this - it allows a sentence and a keyword to be entered and then searches to see if the keyword is in the sentence.

The program searches for spaces which, when found, indicate that the characters from the previous space make up one complete word. Now that a word has been located it can be compared to see if it is the required keyword.

Program 14 - WORD SEARCH

```

10 REM *****
15 REM * WORD SEARCH *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT :PRINT " WORD SEARCH "
45 PRINT :PRINT " ENTER KEYWORD : "
50 INPUT W$
55 PRINT :PRINT " ENTER SENTENCE : "
60 INPUT S$ :S$=S$+" "
65 L=LEN(S$) :P=1
70 FOR X=1 TO L
75 IF MID$(S$,X,1) <> " " THEN 95
80 IF MID$(S$,P,X-P) <> W$ THEN P=X+1 :GOTO 95
85 PRINT :PRINT " KEYWORD IS PRESENT"

```

```

90 GOTO 105
95 NEXT X
100 PRINT :PRINT " KEYWORD IS NOT PRESENT"
105 PRINT :PRINT " PRESS: "
110 PRINT "          SPACE BAR TO REPEAT "
115 PRINT "          ANY KEY FOR NEW KEYWORD"
120 GET K$: IF K#="" THEN 120
125 IF K#=" " THEN 55
130 GOTO 40

```

How would you modify this program to count the number of times the keyword occurs?

The previous example is limited by the INPUT statement which restricts the size of an entered sentence to a maximum of eighty characters. The next example demonstrates again the word searching technique to search through a series of items contained in DATA statements for a specific keyword. In addition to locating the keyword, program 15 also returns a word that is related to the keyword.

The program is devised for a software house who have a vast number of staff each with varying skills and experience with different machines and languages. The staff scheduler required a means of keeping a record of the skills of each employee and needed a method of obtaining all the names in the company who had experience with a specified item. So, he needed some form of simple database.

The names of each employee are stored in DATA statements followed by their skills. To distinguish between a name and a skill, the name is preceded by the symbol 'E'. The program does a word search on each item; if a word is preceded by 'E' then the employee's name is temporarily stored while his/her skills are examined. If a required skill is found then the employee's name can be displayed. The program then continues and searches through the remaining items until the termination symbol '%' is located.

This is probably one of the simplest forms of database programs that exists and is only limited by the memory size of the computer. It has the advantage that the staff scheduler can easily add to his records as the skills of his staff expand by amending the DATA statements. From this example you can change the DATA statements to set up a database with something more relevant to your own needs.

Program 15 - DATABASE

```

10 REM *****
15 REM *   DATABASE   *
20 REM *=====*
25 REM * MRH   1/7/1983 *
30 REM *****
35 REM
40 PRINT :PRINT " SHARKY'S SOFTWARE SKILLS LTD "
45 RESTORE :N$="" :S$=""
50 PRINT :INPUT " ENTER SKILL REQUIRED ";S$
55 PRINT
60 READ W$
65 IF MID$(W$,1,1)="%" THEN 85
70 IF MID$(W$,1,1)="£" THEN N$=MID$(W$,2) :GOTO 60
75 IF W$=S$ THEN PRINT W$;"      ";N$
80 GOTO 60
85 PRINT :PRINT " PRESS SPACE BAR TO CONTINUE "
90 GET K$ :IF K$=" " THEN 40
95 GOTO 90
1000 REM DATA ITEMS
1005 REM =====
1010 REM
1015 DATA £BROWN R,      FORTRAN 77, ADA, VAX
1020 DATA £LEES J,      CORAL 66, COBOL, ARGUS, TANDEM
1025 DATA £ROBERTS R,    BASIC, COBOL, PDP II ,
        ARMY SYSTEMS
1030 DATA £SINCLAIR D,    ALGOL 68, COBOL, BURROUGHS
1035 DATA £STEVENSON F,    PASCAL, UNIX, VAX, MICROS
1040 DATA £THOMPSON B,    POLICE SYSTEMS
1045 DATA £YATES D,      BASIC, Z80, 8064
2000 DATA %

```

The final example of this chapter illustrates how we can manipulate text. A word is chosen at random from a list contained in DATA statements and then the letters are jumbled up to produce an anagram. The anagram is then displayed for a player to try and seek the original word. By entering '?' the player can request the scrambled word to be displayed. The list of words given can be expanded on if required by adding them in additional DATA statements.

Program 16 - MANARAG!

```

10 REM *****
15 REM *   MANARAG   *
20 REM *****
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 PRINT :PRINT , "ANAGRAM"
45 PRINT :PRINT " FIND THE SCRAMBLE WORD:"
50 A$="" :G$="" :RESTORE
55 N=80 :REM THE NUMBER OF WORDS
60 FOR J=1 TO (RND(0)*N+1)
65 READ W$
70 NEXT J
75 L=LEN(W$) :WW$=W$
80 FOR J=1 TO L
85 N=INT(1+RND(0)*L)
90 L$=MID$(W$,N,1) :IF L$="*" THEN 85
95 W$=LEFT$(W$,N-1)+"*"+MID$(W$,N+1)
100 A$=A$+L$
105 NEXT J
110 IF A$=WW$ THEN 60
115 PRINT :PRINT , ,A$
120 PRINT :INPUT " ENTER GUESS      ";G$
125 IF G$="" THEN 120
130 IF MID$(G$,1,1)="?" THEN PRINT :PRINT , "ANSWER " ,
    WW$ :GOTO 145
135 IF WW$ <> G$ THEN PRINT :PRINT " TRY AGAIN " :GOTO
    120
140 PRINT :PRINT , "WELL DONE "
145 PRINT :PRINT " PRESS SPACE BAR TO REPEAT "
150 GET K$ :IF K$= " " THEN 40
155 GOTO 150
1000 REM                               WORD LIST
1005 REM                               =====
1010 DATA HYDROGEN, MONOPOLY, LOOPHOLE, CHIMNEY, PACK, CHOR
    AL, CHRONIC, DUET, ELLIPSE
1015 DATA FASCISM, HONEY, GRUEL, MALARIA, RATIONAL, RATIO, S
    COLD, CRY, COAX, LIBEL, QUIZ
1020 DATA EGOTISM, SAINT, BAPTIZE, ABACUS, AEON, SKULL, TRIP

```

```

      OD, DELAY, FLINT, KNUCKLE
1025 DATA BERET, SLIT, MEMORY, SERVICE, OMEGA, SOCIABLE, NEG
      LECT, HOBBY, RAQUET, SLASH
1030 DATA RACKET, WARRANT, COLUMN, CAVALIER, GROSS, DECEIVE
      , TRAITOR, SCAB, SWINE, WOE
      1035 DATA WHIRL, ACIDIC, POWER, AROUSE, INNINGS, JEWEL
      , GRUDGE, CARNIVAL, GIGGLE, BIKINI
1040 DATA JUGGLE, FOXY, KHAKI, ACQUIT, JUBILEE, ENIGMA, LOCH
      , POULTRY, ORACLE, POTTERY
1045 DATA ORTHODOX, INTERNAL, SQUEEZE, LACQUER, IRONY, FLEX
      IBLE, HORRIFIC, COMPUTE, TUT
1050 REM          <<  ETC.  >>

```

CHAPTER 9

CONTROL CHARACTERS

By now you will be well aware that in command mode you can move the cursor about the screen with the cursor control keys  &  or reposition it at the top left hand corner with . It was also

mentioned briefly that you could alter the colour of the text displayed by pressing one of the digit keys with either  or . Each of these keys produce control characters. Commodore BASIC has a unique and powerful facility which enables such control characters to be programmed into PRINT statements and so can be used whilst in program mode. When programmed into a string each control character is represented by a unique graphic symbol and is displayed in the program listing. Probably one of the hardest things for a beginner to Commodore BASIC is to be able to associate which control key is associated with which particular symbol in a listing. All the Commodore 64 control symbols are listed in Appendix C.

When in command mode and entering a program the control keys will usually act as normal. If, however, a set of quotation marks are entered then the keyboard changes into quote mode; from then onwards the control keys have no effect except to return their appropriate control character symbol. A second set of quotation marks or  will switch the keyboard back into normal mode. So that additional control characters

can be edited into a line,   also switches the keyboard

into quote mode. Once all the spaces that were INSerTed have been filled, the keyboard automatically returns to normal mode.

Note that if quote mode is switched on by pressing quotes then the DELete key will remove the character to the right of the cursor; if however

it is switched on by INSerTing characters then the DELEte key returns its own control character symbol until all the inserted spaces are filled. As a result you cannot DELEte characters if inserted spaces are still present in the string.

By referring to Appendix C you will see there are five control characters that cannot be accessed directly from the keyboard. These are shifted RETURN (acts as a carriage return and linefeed), switch to upper/lower case mode, switch to upper case/graphic mode, enable keyboard mode switch and disable keyboard mode switch. These control characters can be entered into a string by leaving a space for them and then editing the line after RETURN has been pressed. The characters are then entered by

pressing **CTRL** RVS ON which switches the text to reverse video and then followed with the corresponding graphic symbol which relates to the required control key.

When using cursor control characters remember that whenever a character is displayed on the screen the cursor position moves one position to the right.

The semicolon can be used to concatenate strings containing control characters in the same fashion as with normal characters - the next example which displays a message with a border surrounding it demonstrates this technique.

```
10 REM BORDER
15 PRINT " " :
20 FOR I=1 TO 3 :PRINT " " :NEXT I
25 FOR I=1 TO 20 :PRINT " "; :NEXT I
30 FOR I=1 TO 10 :PRINT " "; :NEXT I
35 FOR I=1 TO 20 :PRINT " "; :NEXT I
40 FOR I=1 TO 10 :PRINT " "; :NEXT I
45 PRINT " GETTING MORE FROM ";
50 PRINT "THE";
55 PRINT "COMMODORE 64 "
```

The semicolons are included to concatenate the sequential printing of the characters as the FOR...NEXT loops are executed.

The Commodore 64 also has two functions which can be used in conjunction with control characters in PRINT statements to reposition the cursor.

SPC - The SPC function is used to output a fixed number of SPaCes to the screen, or as we shall see later to other peripherals.

syntax : SPC (< numeric >)

The argument of the function is limited to the range 0 to 255.

TAB - The TAB function is used to move the cursor a fixed number of column positions relative to the start of the current print line.

syntax : TAB (< numeric >)

The argument of the function is limited to the range 0 to 255. If the current print position has already past the required column the print position remains unaltered.

These two functions are illustrated in program 17, CALENDAR.

This displays the calendar for any particular month and year. The original calendar was devised by Julius Caesar, but he fixed the year to be too long by eleven minutes. This was corrected with the introduction of the Gregorian calendar in Italy in 1582, although it was not introduced into England until 1752. The program will not give correct answers before this date.

The program uses a formula in a subroutine at line 200 to calculate the day of the week on which a particular month commences. To use the program, enter the month as a number from 1 to 12, followed by the year using all 4 digits.

Program 17 - CALENDAR

```

10 REM *****
15 REM *   CALENDAR   *
20 REM *****
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 PRINT "☐☐ CALENDAR"
45 INPUT "☐ ENTER MONTH";M
50 IF M<1 OR M>12 THEN 15
55 INPUT "☐ ENTER YEAR ";Y
60 PRINT "☐ MONTH";M;SPC(3);" YEAR";Y;TAB(40);
   "☐ CORRECT"
65 GET K$:IF K$="N" THEN 40
70 IF K$<>"Y" AND K$<>" " THEN 65
75 PRINT "☐☐ MONTH";M;SPC(3);" YEAR";Y
80 PRINT "☐☐ SUN MON TUE WED THU FRI SAT ☐"
85 GOSUB 200
90 DD=DD:M=M+1
95 IF M>12 THEN M=1:Y=Y+1
100 GOSUB 200
105 IF DD<D THEN DD=DD+7
110 N=28+DD-D
115 P=4*D
120 FOR X=1 TO N
125 PRINT TAB(P);X;
130 P=P+4:IF P>=26 THEN P=0:PRINT
135 NEXT X
140 PRINT:PRINT "☐☐ PRESS SPACE BAR TO CONTINUE "
145 GET K$:IF K$=" " THEN 40
150 GOTO 145
200 MM=M-2:YY=Y
205 IF MM>0 THEN GOTO 215
210 MM=MM+12:YY=YY-1
215 C=INT(Y/100):YY=YY-100*C
220 DD=1+INT(2.6*MM-0.19)+YY+INT(YY/4)+INT(C/4)-2*C
225 DD=DD-7*INT(DD/7)
230 RETURN

```

MORE ON CONTROL CHARACTERS

By referring to Appendix A you will notice that the control keys mentioned have their own ASCII character code - thus we could include control keys in our program by using

```
PRINT CHR$(X)
where X is the relevant character code
```

For example, two control characters we met earlier were the enable/disable keyboard mode switch. Also, these character codes are not directly accessible from the keyboard and so it is usually easier to obtain them by this means.

i.e. Type: `PRINT CHR$(8) RETURN`

Now try to obtain upper/lower case keyboard mode by pressing **SHIFT** & **C** simultaneously.

You should find that it cannot be accessed.

Similarly `PRINT CHR$(9)` will enable the keyboard mode switch.

ANIMATED EFFECTS

With the aid of control characters and `FOR...NEXT` loops we can display a sequence of characters to produce some simple animation. The basic idea is to construct a string which includes the characters to be displayed along with some cursor control characters to move the characters from its previous print position and some blank characters to overwrite the previous display. By controlling the program execution with the `FOR...NEXT` loop we can print the string several times in succession to produce some interesting animated effects.

example: Horizontal Motion

```
10 REM HORIZONTAL MOTION
15 PRINT "XXXXXXXX"
20 FOR J=1 TO 39
25 PRINT " >||";
30 FOR K=1 TO 40 : NEXT K
```

```

35 NEXT J
40 FOR J=1 TO 39
45 PRINT "  ████";
50 FOR K=1 TO 40 : NEXT K
55 NEXT J
60 GOTO 20

```

We could use a similar technique to achieve vertical movement. The next example combines both horizontal and vertical movement to produce a diagonal motion.

example: Diagonal Motion

```

10 REM  DIAGONAL MOTION
15 PRINT "███"
20 FOR J=1 TO 22
25 PRINT " █";CHR$(186);"██ █"CHR$(186);"████";
30 FOR K=1 TO 40 : NEXT K
35 NEXT J
40 GOTO 15

```

We shall see later how to draw more realistic figures with a higher resolution and then to use similar techniques for a more acceptable display.

CHAPTER 10

THE TIME MACHINE

The Commodore 64 has two internal counters which are steadily incremented automatically with time. With this facility we have the scope to enable us to add timing to our programs for such uses as a clock, stopwatch or for timing moves in games. We can also time the lengths of processes to compare how efficient different programming methods are.

The timers are accessed by the names `TIME` and `TIME$`; these are reserved keywords and so cannot be used as variables. Note that the timers are not incremented during tape I/O.

`TIME` - The `TIME` function accesses an internal timer known as a jiffy clock which is initialised to zero at switch on and from then onwards is incremented by 60 units every second.

syntax : `TI [ME]`

We can use `TIME` to time to an accuracy of 1/60th of second. To achieve this we need to know the initial and final values of `TIME`. By dividing the difference by 60 the time in seconds is calculated. Sometimes, to obtain a true value, a fraction of a second should be subtracted to allow for the processing time in reading the jiffy clock.

Program 18 demonstrates `TIME` by testing the speed of your reactions.

Program 18 - REACTION TIMER

```
10 REM *****
15 REM * REACTION TIMER *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
```

```

40 PRINT "P" PRESS SPACE BAR TO COMMENCE"
45 GET K$: IF K$<>" " THEN 45
50 FOR I=0 TO RND(0) STEP 0.005
55 PRINT "Q"
60 GET K$: IF K$<>" " AND K$<>" " THEN PRINT "XXXXXXXXXX
      CHEAT!! " :END
65 NEXT I
70 PRINT "XXXXXXXXXXXXX" BANG !! "
75 T1=TIME
80 GET K$: IF K$="" OR K$=" " THEN 80
85 T2=TIME-T1-0.14
90 PRINT "XXXXXXXX TIME TAKEN IS ";
      INT(T2*100/60)/100
95 GOTO 40

```

TIMES - The **TIMES** function works like a normal six character digital clock consisting of hours, minutes and seconds. The system software uses the jiffy clock to regularly update **TIMES** but unlike **TIME**, **TIMES** can be assigned an initial value - just as you set a clock.

```
syntax : TI [ME] $
```

Program 19 demonstrates `TIMES` by converting the six characters that make up `TIMES` into a more acceptable clock display. Note how we access the hours, minutes and seconds from `TIMES` by taking a slice.

Program 19 - DIGITAL CLOCK

```

10 REM *****
15 REM * DIGITAL CLOCK *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT "ENTER TIME."
45 INPUT "ENTER HOURS: "; H$ : H=VAL(H$)
50 IF H<0 OR H>23 THEN 45
55 INPUT "ENTER MINUTES: "; M$ : M=VAL(M$)
60 IF M<0 OR M>59 THEN 55

```

```

65 IF LEN(H$)=1 THEN H$="0"+H$
70 IF LEN(M$)=1 THEN M$="0"+M$
75 PRINT "PRESS SPACE BAR TO COMMENCE "
80 GET K$: IF K$<>" " THEN 80
85 T$=H$+M$+"00"
90 IF LEN(T$)<6 THEN 40
95 TIME$=T$
100 PRINT "TIME "
105 PRINT " "
110 FOR I=1 TO 8 :PRINT " "; :NEXT I
115 H$=MID$(TIME$,1,2)
120 M$=MID$(TIME$,3,2)
125 S$=MID$(TIME$,5,2) :F=VAL(S$)
130 IF F=2*INT(F/2) THEN 140
135 PRINT H$;" ";M$;" ";S$ :GOTO 105
140 PRINT H$;" ";M$;" ";S$ :GOTO 105

```

When we have studied sound you could amend this program to chime at a specified time or at every hour.

CHAPTER 11

TECHNIQUES FOR SORTING

One of the earliest points made in this book was that one of the few advantages that a computer has over the human brain is its ability to undertake long, laborious but relatively simple tasks in a short time. One such task that we shall examine is the mundane job of sorting data comprised of numeric or alphabetic items into some sequence - usually into increasing or alphabetic order. There have been many algorithms devised for doing such operations although their efficiency is usually proportional to their complexity.

We will restrict ourselves to looking at the following four algorithms.

METHOD	COMPLEXITY	EFFICIENCY	
		Small Quantities of data	Large Quantities of data
Bubble sort	Very simple	Excellent	Poor
Insertion sort	Very simple	Excellent	Average
Shell sort	Average	Good	Good
Quick sort	Complex	Good	Very good

Figure 11.1

For each method, it is usual to store the data in either a numeric or string array. The size of the array is limited only by the available RAM. But, one factor which should always be considered when deciding upon which algorithm to use is the storage requirement; some methods require a

great deal of memory in addition to the array containing the items of data.

Each of the algorithms mentioned above will now be explained in detail with a program routine to illustrate it.

The main code below can be used to set up random data and to call each sort program. In order that the programs can be compared the main code will also give the processing time required.

MAIN CODE FOR SORTING

```
10 REM *****
15 REM * MAIN CODE FOR *
20 REM *   SORTING   *
25 REM *=====*
30 REM * MRH   1/7/1983 *
35 REM *****
45 CLR
50 PRINT "10 SORTING TECHNIQUES."
55 INPUT "10 NUMBER OF ITEMS TO BE SORTED ";N
60 IF N<5 OR N>1000 THEN 55
65 N=INT(N) :DIM D(N)
70 FOR I=1 TO N
75 D(I)=INT(RND(0)*2500)
80 NEXT I
85 PRINT "10 ENTER:10"
90 PRINT " 1) BUBBLE SORT"
95 PRINT " 2) INSERTION SORT"
100 PRINT " 3) SHELL SORT"
105 PRINT " 4) QUICK SORT"
110 INPUT "10 OPTION ";K$ :K=VAL(K$)
115 IF K<1 OR K>4 THEN 110
120 TIME$="000000"
125 ON K GOSUB 1000,2000,3000,4000
130 PRINT "10 DATA SORTED"
135 H$=MID$(TIME$,1,2)
140 M$=MID$(TIME$,3,2)
145 S$=MID$(TIME$,5,2)
150 PRINT "10 TIME TAKEN "
155 PRINT " ";H$;" HOURS ";M$;" MINUTES ";S$;" SECONDS"
160 PRINT "10 DO YOU WANT TO SEE DATA (Y/N)"
165 GET K$ :IF K$="" THEN 165
```

```

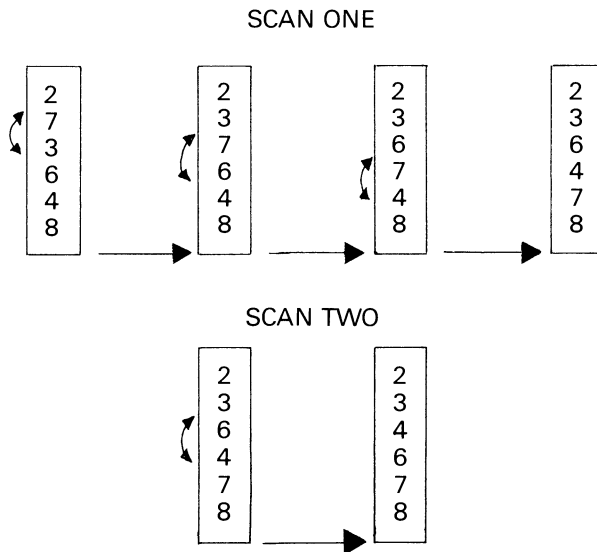
170 IF K$<>"Y" THEN 45
175 FOR I=1 TO N
180 PRINT D(I)
185 NEXT I
190 GET K$: IF K$="" THEN 190
195 GOTO 45

```

BUBBLE SORT

The first method of sorting is called a 'bubble sort' because the lowest values can be thought of as floating upwards to one end of the array and the highest values to the bottom. The array is continually scanned and two neighbouring items are swapped if the first has a higher value than the other; if no exchanges are made then the list is in order.

example



SCAN THREE

2
3
4
6
7
8

Figure 11.2

No exchanges are made on scan 3 which implies that the list is ordered.

Program 20 - BUBBLE SORT

```
1000 REM    BUBBLE SORT
1005 REM    =====
1010 :
1015 FOR J=1 TO N-1
1020 F=0
1025 FOR I=1 TO N-J
1030 IF D(I) <= D(I+1) THEN 1045
1035 F=1
1040 T=D(I+1) : D(I+1)=D(I) : D(I)=T
1045 NEXT I
1050 IF F=0 THEN 1060
1055 NEXT J
1060 RETURN
```

INSERTION SORT

The 'Insertion Sort' method enables a list of items to be ordered with a single scan of the array. When an item is found to be out of order, its correct position is found and then by moving certain other items one place along the array it can be correctly positioned.

example

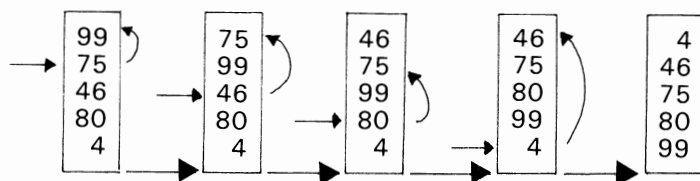


Figure 11.3

Program 21 - INSERTION SORT

```
2000 REM   INSERTION SORT
2005 REM   =====
2010 :
2015 FOR J=2 TO N
2020 T=D(J)
2025 FOR I=J-1 TO 1 STEP -1
2030 IF D(I) <= T THEN 2045
2035 D(I+1)=D(I)
2040 NEXT I
2045 D(I+1)=T
2050 NEXT J
2055 RETURN
```

SHELL SORT

Whereas the 'bubble sort' only compares adjacent data items in the array, a shell sort makes initial comparisons between items that are far apart on the assumption that if elements are far apart and are required to be swapped it is more efficient to do it as soon as possible. The separation between data elements is called the 'sort interval'. Initially, the sort interval is set to the number of elements and then on each scan the interval is reduced by one half until the final scan is equivalent to the 'bubble sort'.

Program 22 - SHELL SORT

```
3000 REM SHELL SORT
3005 REM =====
3010 :
3015 SI=N
3020 IF SI < 1 THEN 3070
3025 SI=INT(SI/2)
3030 F=0
3035 FOR I=1 TO N-SI
3040 IF D(I) <= D(I+SI) THEN 3055
3045 F=1
3050 T=D(I+SI) : D(I+SI)=D(I) : D(I)=T
3055 NEXT I
3060 IF F=0 THEN 3020
3065 GOTO 3030
3070 RETURN
```

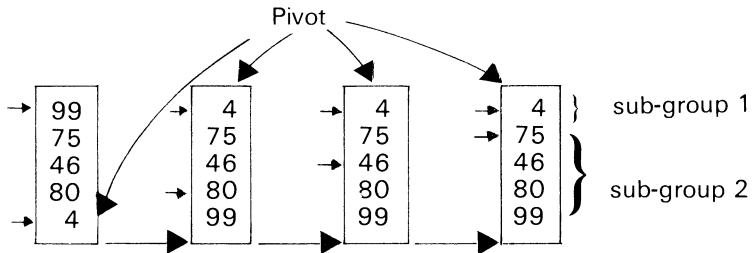
QUICK SORT

And finally, the 'quick sort' algorithm. It is one of the more complex sorting algorithms but generally gives the fastest results. The general idea is that the items are split up (partitioned) into sub-groups which in turn are partitioned further and so on until the sub-groups are small enough that a form of 'bubble sort' can be applied on the group efficiently. However, while the sub-groups are being sorted the program must remember the position of the remaining unsorted data items. This is done by storing the start and end position of the array of the items that remain unsorted. These positions are stored in a second array so that the first positions stored are the first positions retrieved; such a structure is called a stack. The sort is complete when the stack becomes empty.

Two pointers are used in the quick sort, one indicates the first and the other the last item in the array. One of the two pointers will always point to the element in the array which was initially at the first position in the array; this pointer is called the pivot. The two items that are currently pointed to are swapped if the one nearer the start of the array is greater than the other. The second pointer is then moved one position along the array towards the pivot and the process is repeated. This continues until the two

pointers meet, in which case the array is partitioned about the pivot into sub-groups and the process is then repeated on each new sub-group. The positions are stored in the array S(N,2). If, with vast quantities of data a 'stack overflow' message appears, then the dimension of this array must be increased.

example



These two sub-groups are then sorted.

Figure 11.4

Program 23 - QUICK SORT

```

4000 REM QUICK SORT
4005 REM =====
4010 :
4015 DIM S(40,2)
4020 PS=1
4025 S(1,1)=1 : S(1,2)=N
4030 IF PS=0 THEN 4110
4035 II=S(PS,1) : JJ=S(PS,2) : PS=PS-1
4040 P=0 : I=II : J=JJ
4045 IF D(I) < D(J) THEN 4060
4050 P=1-P
4055 T=D(I) : D(I)=D(J) : D(J)=T
4060 IF P=0 THEN I=I+1
4065 IF P=1 THEN J=J-1
4070 IF I < J THEN 4045
4075 IF I >= JJ THEN 4095
4080 PS=PS+1

```

```

4085 IF PS > 40 THEN PRINT"STACK OVERFLOW" :END
4090 S(PS,1)=I+1 :S(PS,2)=JJ
4095 JJ=I-1
4100 IF JJ > II THEN 4040
4105 GOTO 4030
4110 RETURN

```

ALPHABETICAL SORTING

Our final sort program uses the insertion sort algorithm to put a list of text strings into alphabetical order. Remember from earlier that when strings are compared, a string is found to be less than another if it comes first in alphabetical order (i.e. as in a telephone directory).

Program 24 - ALPHABETICAL SORTING

```

10 REM *****
15 REM * ALPHABETICAL SORTING *
20 REM *=====*
25 REM *      MRH  1/7/1983      *
30 REM *****
35 REM
40 PRINT "ALPHABETICAL SORTING "
45 INPUT "ENTER NUMBER OF STRINGS ";N
50 DIM A$(N)
55 PRINT "ENTER EACH STRING "
60 FOR J=1 TO N
65 INPUT A$(J)
70 NEXT J
75 FOR J=2 TO N
80 T$=A$(J)
85 FOR K=J-1 TO 1 STEP -1
90 IF A$(K)<T$ THEN 105
95 A$(K+1)=A$(K)
100 NEXT K
105 A$(K+1)=T$
110 NEXT J
115 PRINT "SORTED LIST "

```



```
120 FOR J=1 TO N
125 PRINT A$(J)
130 NEXT J
135 PRINT "␣ PRESS SPACE BAR TO REPEAT "
140 GET K$ :IF K$=" " THEN 40
145 CLR :GOTO 140
```

One other sort algorithm called the 'tree sort' algorithm will be met later when we study Dynamic Data Structures.

CHAPTER 12

THE CODING OF INFORMATION FOR COMPUTER SYSTEMS

THE BINARY NUMBER SYSTEM

So far, we have taken it for granted that a computer is able to store data in its memory but we have yet to study the form in which the data is stored.

Although in chapter one the memory in a computer system was compared with that of the human brain this comparison leaves a lot to be desired. The brain can memorize all types of information including letters, numbers and words. In comparison, the computer's memory is very simple; the nature of electronic components being either switched on or off reduces its capabilities to being able to recognize just two states; these are usually written for convenience as '0' and '1'. Just as in English where words and sentences are built up by using more than one letter, and numbers consist of several digits, computer expressions are represented by sequences of '0's and '1's. Such patterns of '0's and '1's are called binary numbers.

When numbers are written in the normal decimal or base 10 representation the digit furthest to the right gives the number of units, the digit to its left gives the number of tens (the base of the number system), the next digit to the left gives the number of hundreds (the base squared), and so on.

Binary numbers use a base of 2; the digit furthest to the right gives the number of units, the digit to its left gives the number of twos (the base), the next digit gives the numbers of fours (the base squared), the next digit gives the number of eights (the base cubed), and so on.

Conventionally the number 345 in the familiar decimal numbering system that we use means

$$\begin{array}{rcl}
 & 3 & 4 & 5 & & \text{(decimal)} \\
 & | & | & | & & \\
 & \text{---} & \text{---} & \text{---} & & \\
 & 5 \times 10^0 = 5 \times 1 = 5 & & & & \\
 & 4 \times 10^1 = 4 \times 10 = 40 & & & & \\
 & 3 \times 10^2 = 3 \times 100 = 300 & & & & \\
 & & & & & \underline{345} \text{ (decimal)}
 \end{array}$$

Likewise a binary number such as 11001 is equivalent to

$$\begin{array}{rcl}
 & 1 & 1 & 0 & 0 & 1 & & \text{(binary)} \\
 & | & | & | & | & | & & \\
 & \text{---} & \text{---} & \text{---} & \text{---} & \text{---} & & \\
 & 1 \times 2^0 = 1 \times 1 = 1 & & & & & & \\
 & 0 \times 2^1 = 0 \times 2 = 0 & & & & & & \\
 & 0 \times 2^2 = 0 \times 4 = 0 & & & & & & \\
 & 1 \times 2^3 = 1 \times 8 = 8 & & & & & & \\
 & 1 \times 2^4 = 1 \times 16 = 16 & & & & & & \\
 & & & & & & & \underline{25} \\
 & & & & & & & \text{(decimal)}
 \end{array}$$

When counting a decimal a '1' is carried over into the next column whenever a '9' is reached, i.e. after 9 comes 10, after 29 comes 30 and after 99 comes 100

Similarly since in binary only the digits '0' and '1' are available, when counting a '1' is carried over whenever a '1' is reached, i.e. after 0 comes 1, after 1 comes 10, after 10 comes 11, after 11 comes 100. (See Figure 12.3 for further examples).

It can be seen that the binary numbering system works on the same principle as the decimal numbering system but since more digits are required to represent a number in binary than in decimal, it would be more cumbersome for us to use.

It is now possible to explain why in chapter one it was stated that a number stored at a particular address in the computer memory had to be within the range 0 to 255. This is because a storage location can contain only eight binary digits; thus the range is from 00000000(0 decimal) to 11111111 (255 decimal). When referring to binary numbers in computers an individual digit is called a bit and a group of eight bits at an address is called a byte.

Binary numbers may be combined by addition, subtraction, multiplication and division.

Binary addition uses the following five rules.

$$\begin{aligned}
 0 + 0 &= 0 \\
 0 + 1 &= 1 \\
 1 + 0 &= 1 \\
 1 + 1 &= 0 \text{ carry } 1 \\
 \text{carry} + 1 + 1 &= 1 \text{ carry } 1
 \end{aligned}$$

e.g.

$ \begin{array}{r} 1\ 1\ 1 \\ +\ 0\ 1\ 1 \\ \hline 1\ 0\ 1\ 0 \end{array} $	Taking the 1st column, Taking the 2nd column, Taking the 3rd column, Taking the 4th column,	$1 + 1 = 0 \text{ carry } 1$ $1 + 1 + \text{carry} = 1 \text{ carry } 1$ $1 + \text{carry} = 0 \text{ carry } 1$ $\text{carry} = 1$

$$\begin{array}{r}
 1\ 1\ 1 \\
 \swarrow \swarrow \swarrow
 \end{array}$$

Carry
Indicators

Binary subtraction uses the following four rules.

$$\begin{aligned}
 0 - 0 &= 0 \\
 1 - 0 &= 1 \\
 1 - 1 &= 0 \\
 0 - 1 &= 1 \quad (\text{found by borrowing from next higher digit})
 \end{aligned}$$

e.g.

$ \begin{array}{r} 0\ 1 \\ \cancel{1}\ 0 \\ 0\ 1 \\ \hline 0\ 1 \end{array} $	Taking the 1st column, 0-1 not possible borrow from next highest digit $10 - 1 = 1$ Taking the 2nd column, $0 - 0 = 0$

e.g.

$ \begin{array}{r} 0\ 1\ 1 \\ \cancel{1}\ \cancel{0}\ 0 \\ -\ 0\ 0\ 1 \\ \hline 0\ 1\ 1 \end{array} $	Borrow indicators

e.g.

$$\begin{array}{r}
 \begin{array}{c} \text{0} \quad \text{1} \\ \text{1} \quad \text{0} \quad \text{1} \\ - \quad \text{0} \quad \text{1} \quad \text{1} \\ \hline \text{0} \quad \text{1} \quad \text{0} \end{array}
 \end{array}$$

Borrow indicators

One complication when dealing with binary numbers is the handling of negative numbers. Consider the problem of subtracting 92 from 20 using binary numbers:

DECIMAL	BINARY
20	00010100
- 92	- 01011100
- 72	10111000

The above binary result is obtained only if we do not worry where the 'borrow' for the left hand bit came from. But, 10111000 in binary is equal to 184 in decimal, so that -72 decimal seems to have the same binary representation as 184. How can we explain this? To do so, we must find a way of representing negative numbers. These can be visualized as numbers in the opposite direction to positive numbers, i.e. if an electric meter reading 0000 were wound back one unit it would read 9999. To obtain -72 it should be wound back 72 units, like this:

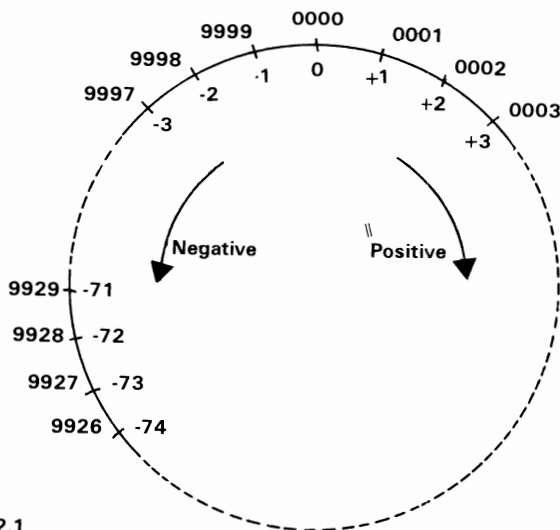
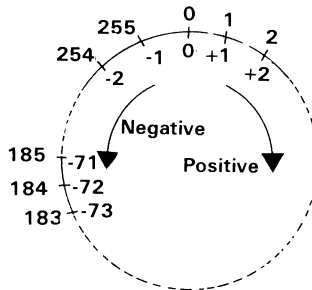


Figure 12.1

Thus -72 can be represented on our meter as 9928 as long as we remember that it is really a negative number.

Using a similar method for binary numbers, it is convenient to represent negative numbers in a form that is known as two's complement. In a binary number containing x bits, this simply means that $(-n)$ is stored as $(2^x - n)$.

For example a computer using 8 bits would store -72 (decimal) as $(2^8 - 72) = 184$



A binary number can be converted into two's complement form simply by converting all the '1's into '0's and the '0's into '1's and then adding 1.

e.g. Negate 72:

72 in binary is	0 1 0 0 1 0 0 0
	▼
Convert digits	1 0 1 1 0 1 1 1
	▼
Add 1	1 0 1 1 1 0 0 0

So -72 (decimal) is represented by 1 0 1 1 1 0 0 0

e.g. Negate (-72): (-72) in decimal is 1 0 1 1 1 0 0 0

	▼
Convert digits	0 1 0 0 0 1 1 1
	▼
Add 1	0 1 0 0 1 0 0 0

So -(-72) (decimal) is represented by 01001000, which as we would expect is 72 (decimal).

If this is a valid form of -72 we would expect $72 + (-72)$ to be zero.

$$\begin{array}{r}
 01001000 \\
 + 10111000 \\
 \hline
 100000000
 \end{array}$$

Ignoring the first digit, the last eight digits show this to be so. The additional bit on the extreme left is known as the sign bit. If the sign bit is '0' the number is positive and if the sign bit is '1' the number is negative. i.e. in two's complement form, when the two numbers add up to zero the sign bit is irrelevant. The use of the sign bit simply enables the computer to distinguish between positive and negative numbers. As examples, we would expect $172 + (-25)$ to be positive and $117 + (-159)$ to be negative.

sign bit (0 for normal form, 1 for two's complement form)

$$\begin{array}{rcl}
 172 & \boxed{0} & 10101100 \\
 - 25 & + \boxed{1} & 11100111 \\
 \hline
 147 & \boxed{0} & 10010011 \\
 & \uparrow &
 \end{array}
 \quad \begin{array}{l}
 \text{(25 in two's complement form,} \\
 \text{thus the sign bit is set)}
 \end{array}$$

positive, thus result is in normal form.

$$\begin{array}{rcl}
 117 & \boxed{0} & 01110101 \\
 - 159 & \boxed{1} & 01100001 \\
 \hline
 - 42 & \boxed{1} & 11010110 \\
 & \uparrow &
 \end{array}
 \quad \begin{array}{l}
 \text{(159 in two's complement form,} \\
 \text{thus the sign bit is set)}
 \end{array}$$

negative, thus result is in two's complement form.

HEXADECIMAL NOTATION

Unlike computers, we find these long strings of '0's and '1's difficult to memorize and awkward to work with, so some simpler notations have been developed to help us. One such system uses a numbering system with a base of sixteen and is known as the hexadecimal numbering system. This requires sixteen digits but instead of designing six new symbols, the first letters of the alphabet are used. The sixteen digits are:

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

It is important in this context not to interpret the new symbols as letters

but as digits that are capable of the mathematical operations such as addition, subtraction etc.

As an example, consider the hexadecimal number 2FA:

$$\begin{array}{rcl}
 \begin{array}{ccc} 2 & F & A \end{array} & \text{(hex)} & \\
 \begin{array}{l} | \\ | \\ | \end{array} & \begin{array}{l} A \times 16^0 = A \times 1 = 10 \text{ (decimal)} \times 1 = 10 \\ F \times 16^1 = F \times 16 = 15 \text{ (decimal)} \times 16 = 240 \\ 2 \times 16^2 = 2 \times 256 = 2 \text{ (decimal)} \times 256 = 512 \end{array} & \\
 \hline
 & & = 512 \\
 & & \underline{240} \\
 & & 762 \\
 & & \text{(decimal)}
 \end{array}$$

There is a very simple method for converting binary numbers into its hexadecimal equivalent. This involves splitting up the binary bits into groups of four bits, starting from the right, then each digit of the hexadecimal number is represented by the value of the four bits.

For example	1	0	1	1	1	1	0	1	0
Split bits into groups of four	1	0	1	1	1	1	0	1	0
Calculate value of each group					(15)			(10)	
Convert to hexadecimal					F			A	
	2								

Thus 2FA (hex) and 1011111010 (binary) are equivalent.

As a byte contains eight bits, its value may be expressed with just two hexadecimal digits; thus, with the possible exception of decimal numbers the hexadecimal numbering system is often used in computing in preference to any other numbering systems.

For example

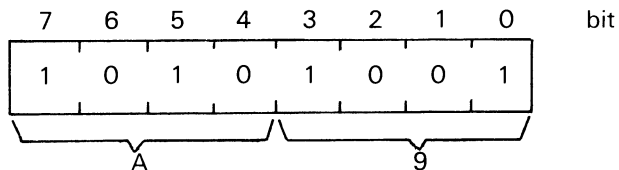


Figure 12.2

i.e. The first hexadecimal digit represents the value of the four most significant bits and the second hexadecimal digit represents the value of the four least significant bits. So, the hex form of this number is A9.

EXAMPLES OF DECIMAL, BINARY AND HEXADECIMAL NUMBERS

Decimal	Binary	Hex. Dec.	Decimal	Binary	Hex.Dec.
-15	111110001	F1	+ 1	1	1
-14	111110010	F2	+ 2	10	2
-13	111110011	F3	+ 3	11	3
-12	111110100	F4	+ 4	100	4
-11	111110101	F5	+ 5	101	5
-10	111110110	F6	+ 6	110	6
- 9	111110111	F7	+ 7	111	7
- 8	111111000	F8	+ 8	1000	8
- 7	111111001	F9	+ 9	1001	9
- 6	111111010	FA	+10	1010	A
- 5	111111011	FB	+11	1011	B
- 4	111111100	FC	+12	1100	C
- 3	111111101	FD	+13	1101	D
- 2	111111110	FE	+14	1110	E
- 1	111111111	FF	+15	1111	F
0	000000000	0	+16	10000	10

Figure 12.3

When working with decimal and hexadecimal numbers you should remember that although the computer appears to understand them through its I/O devices it still stores and operates on them in their expanded binary form.

If, having read this chapter several times, you still do not feel happy about binary and hexadecimal numbers - don't worry! These strange notations are not very important in learning computer programming but, as your skills develop, a full understanding of these numbers will be vital to be able to use the advanced techniques that are available - especially in machine code programming.

CHAPTER 13

COMPUTER LOGIC

THINKING LOGICALLY

If we considered the normal everyday things which we do, we would find that the majority of the tasks require numerous decisions to be made. In order for a computer system to simulate some of the mental processes of a human brain, it must also be able to make logical decisions. Although a computer can only think in terms of '0's and '1's it is still possible for the computer to solve complicated tasks involving decisions by reducing the problem to one of '0's and '1's. This is binary logic and is often referred to as Boolean Algebra by mathematicians.

This logic can be explained most clearly by considering an example

e.g.

<<If my car was serviced more than 12 months ago (A)
and I have driven at least 6000 miles since the last oil change (B)
then I shall change the oil. (C)>>

There are three sections to this statement which have been labelled A,B and C. A and B refer to the two questions I have to ask myself and C refers to the action I shall take. A,B and C are sometimes called binary variables and take the values '1' if the statement is true and '0' if the statement is false. Analysing the statement reveals there are four possible combinations for A and B which are shown below.

- (1) Car has been serviced within 12 months.
Less than 6000 miles have been driven.
- (2) Car has been serviced within 12 months.
More than 6000 miles have been driven.
- (3) Car has not been serviced within 12 months.
Less than 6000 miles have been driven.
- (4) Car has not been serviced within 12 months.
More than 6000 miles have been driven.

Since I only change the oil if statement A and statement B are true then I only change it for the fourth option, i.e. if the car has not been serviced within 12 months and more than 6000 miles have been driven since the last oil change.

These results can be written in tabular form using '1' for true and '0' for false. Such a table is called a truth table, an example of which is shown below:

	A	B	C
(1)	0	0	0
(2)	0	1	0
(3)	1	0	0
(4)	1	1	1

Figure 13.1 The AND truth table

If however the original statement was amended to read

<<If my car was serviced more than 12 months ago (A)
or I have driven at least 6000 miles since the last oil change (B)
then I shall change the oil (C)>>

Then I would change the oil if either (or both) statement A or statement B was true. The truth table would be

	A	B	C
(1)	0	0	0
(2)	0	1	1
(3)	1	0	1
(4)	1	1	1

Figure 13.2 The OR truth table

THE LOGIC OPERATORS

We have just seen that it is possible to combine conditional statements in the English language by linking up the statements with either AND or OR. It

is also possible to combine statements in computer languages, such as Commodore 64 BASIC by using two operators which also are called AND and OR. These operators, which are very important in decision statements, are used between conditional statements as shown.

- (condition 1) AND (condition 2)

AND returns a result TRUE if and only if both conditions are TRUE. Otherwise the result is FALSE.
- (condition 1) OR (condition 2)

OR returns a result FALSE if and only if both conditions are FALSE. Otherwise the result is TRUE.

These results are summarized in the following tables.

AND	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

Figure 13.3

OR	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

Figure 13.4

There is one other logical operator called NOT that the computer will understand and is used before a conditional statement

- NOT (condition 1)

NOT returns a result TRUE if the condition is FALSE and a result FALSE if the condition is TRUE.

NOT	
FALSE	FALSE
TRUE	TRUE

Figure 13.5

It is also possible to combine more than two conditional statements using more than one operator.

For example

NOT (condition 1) AND (condition 2) OR (condition 3) AND (condition 4)

However just like the numerical operators, logical operators are processed in a fixed order and not simply from left to right. The priority values of each operator on the are:

OPERATOR	PRIORITY
NOT	4
AND	3
OR	2

Figure 13.6

i.e. the NOTs are processed first, then and ANDs and finally the ORs. As with numerical expressions, the order that logical expressions are calculated may be changed by the insertion of brackets, since expressions within brackets are calculated first. Relational operators such as =, <, >, >= etc have a priority of 5.

One other logical operator that is occasionally used but is not available on the Commodore 64 is EX-OR (exclusive or) which operates on two operands. This gives the following results.

EXOR	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	FALSE

(condition 1) EXOR (condition 2)

Figure 13.7

EXOR returns a result TRUE if one and only one of the conditions are TRUE. Otherwise the result is FALSE.

We shall now see how these powerful operators may be used in programs.

The most useful application of logical operators is to combine several conditions into one IF statement.

Example:

10 IF (A>2 AND A<B) OR (A>6 AND A<C) THEN statement

The statement that follows will be executed
if A is greater than 2 but less than B
or A is greater than 6 but less than C

By combining the conditions like this into one statement we would save the use of several IF statements testing each individual condition.

As one might expect, things can get a little more complicated if more advanced techniques are required. To start with, the computer does not actually use the logical values of true and false but represents them internally as '-1' (11111111 in two's complement) and '0' (00000000) respectively.

For an example try:

```
PRINT 5 = 6
& PRINT 2 = 2
```

and see what is displayed.

The AND, OR and NOT operators work as shown in their truth tables previously given but when applied to true/false logical values, they work on all eight bits.

e.g.

true AND true = true	
11111111	(true)
11111111	(true)
11111111	(true)

true OR false = true	
11111111	(true)
00000000	(false)
11111111	(true)

taking each column
1 AND 1 = 1

taking each column
1 OR 0 = 1

```

false AND true = false
00000000      (false)
11111111      (true)
-----
00000000      (false)

```

```

NOT false = true
00000000 (false)
-----
11111111 (true)

```

taking each column
0 AND 1 = 0

taking each column
NOT 0 = 1

You can see that these examples conform with the basic rules of logic that we have just learnt.

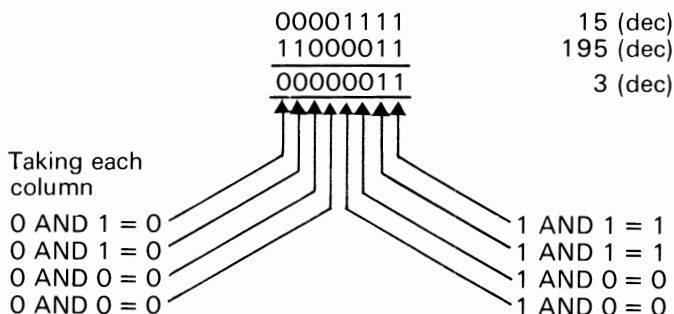
The fun starts when we forget about true and false and consider the use of AND, OR and NOT with numbers other than '0' and '1'. For example try:

PRINT 15 AND 195

The answer is 3. How can we explain this?

To do so we must convert 15 and 195 into binary numbers and then use our AND truth table on each column.

i.e.



Likewise if the OR operator is used between two numbers, each column is calculated using the OR truth table. The NOT operator converts all the '0's in the binary form to '1's and vice versa.

e.g.

15 OR 195

NOT 195

00001111 15 (dec)
11000011 195 (dec)

11001111 207 (dec)

11000011 195 (dec)

00111100 60 (dec)

Sometimes the results will be in two's complement form.

MASKING

With some of the advanced features of the Commodore 64 we are going to need the ability to set and determine the value of specific bits within the computer's memory. Some microprocessors have facilities for checking and altering the individual bits in a byte; unfortunately this is not so with the Commodore 64, so we are going to have to rely on a technique known as masking which uses the logical operators AND and OR.

Our requirements will be:-

1) to be able to select a single bit X

this is done by ANDing the byte with another containing just a single set bit - the result is that every bit except X is set to zero.

e.g.

AND 1 1 0 1 X 1 0 1
 0 0 0 0 1 0 0 0

 0 0 0 0 X 0 0 0

 ↑
 X AND 1 = X

So, if the result in this example were 00001000 (decimal 8) then bit 4 must have been set to 1.

2) to be able to unset a single bit X (i.e. change a 1 to a 0)

this is done by ANDing the byte with another which has all bits set except one corresponding to X - the result is that only the required bit is altered

e.g.

$$\begin{array}{r}
 \text{AND} \quad \begin{array}{cccccccc} 1 & 1 & 0 & 0 & 0 & 1 & X & 0 \end{array} \\
 \hline
 \begin{array}{cccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{array} \\
 \hline
 \begin{array}{cccccccc} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{array} \\
 \hline
 \end{array}$$

↑
X AND 0 = 0

3) to be able to set a single bit X

this is done by ORing the byte with another which has all bits unset except one corresponding to X - the result is that only the required bit is altered

e.g.

$$\begin{array}{r}
 \text{OR} \quad \begin{array}{cccccccc} Y & Y & Y & Y & X & Y & Y & Y \end{array} \\
 \hline
 \begin{array}{cccccccc} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array} \\
 \hline
 \begin{array}{cccccccc} Y & Y & Y & Y & 1 & Y & Y & Y \end{array} \\
 \hline
 \end{array}$$

↑
X OR 1 = 1

Before proceeding further, it is important that you understand the principles of this chapter as the techniques mentioned will be commonly used in the next section of the book.

CHAPTER 14

THE COMMODORE 64's MEMORY LAYOUT

RAM and ROM Layout

The Commodore 64's memory, like that of any other computer can be visualised as a long sequence of cells, each being given a unique identification label called its address and capable of storing a number within a specific range. The 65535 locations in the Commodore 64's memory can be subdivided into several distinct sections each of which has a specific use as show in Figure 14.1.

ADDRESS (DEC)	ADDRESS (HEX)	DESCRIPTION
0	0000	Used by 6510 microprocessor
1	0001	
2	0002	Used by operating system
1023	03FF	
1024	0400	SCREEN MEMORY
2039	07F7	
2040	03F8	SPRITE POINTERS
2047	07FF	
2048	0800	Available for a BASIC program.
40959	9FFF	
40960	A000	
49151	BFFF	8K COMMODORE BASIC INTERPRETER
49152	C000	Available for special programs.
53247	CFFF	
53248	D000	VIDEO INTERFACE CONTROLLER (VIC) REGISTERS.
53294	D02E	
54272	D400	SOUND INTERFACE DEVICE (SID) REGISTERS
55295	D7FF	
55296	D800	COLOUR MEMORY
56319	DBFF	
56320	DC00	I/O REGISTERS
57343	DFFF	
57344	E000	8K COMMODORE KERNAL OPERATING SYSTEM
65535	FFFF	

Figure 14.1 The Commodore 64's Memory Layout

There are certain names and terms used in Figure 14.1 that have not yet been introduced; their use and importance to the system will now be explained.

MEMORY MANAGEMENT

The Commodore 64 has 64K of RAM and 20K of ROM; however, with only 64K addresses available, careful management of the different memory sections is required. This control is undertaken by the system software which overlays (i.e. uses the same range of addresses) the ROM sections on top of part of RAM; the system can tell when certain blocks of ROM are switched in by inspecting specific bits at address 1 - this location is called the 6510 input/output port. When ROM is switched in, we can still write to the underlying RAM but when reading from an overlaid region, the values from the ROM are returned.

We will use this fact - as shown later - to switch off the ROM which contains the information on the shapes of the symbols in the standard character set in order that we can create new character definitions in RAM.

We shall now take a look at the different sections of the Commodore 64's memory. We will then use some of the advanced techniques that require a detailed knowledge of the memory layout.

THE COMMODORE 64's OPERATING SYSTEM

There are three sections of ROM containing permanent programs in the Commodore 64; these three program modules are collectively known as the operating system.

The three individual components are

1. The Kernal - controls the fundamental computer system such as the input/output of data, program processing, peripherals, etc.
2. The BASIC Interpreter - accepts English-like BASIC statements and converts them into machine code for processing.
3. The Screen Editor - controls the output of information to the display screen, the editing of BASIC text and the input of data.

SCREEN MEMORY

The screen memory is a section of RAM which stores the information that is currently displayed on the screen. It consists of 1000 bytes, each byte representing the character stored at one of 1000 character positions on the screen. Initially, at switch on, the screen memory commences at address 1024 (0400 [hex]) although, as we shall see later, it is possible to reposition it to other areas of RAM.

COLOUR MEMORY

The colour memory is a section of RAM and contains 1000 bytes, each of which has a one-to-one relationship with a screen memory byte (see above). The byte values specify the colour of the corresponding character position but the exact use of the individual bits in a colour byte depends on which graphic mode is in use. The colour memory commences at address 55296 (D800 [hex]) and cannot be repositioned elsewhere.

THE VIC-II CHIP

All of the excellent graphic facilities available on the Commodore 64 are controlled by a piece of hardware called the MOS 6567 Video Interface Controller or as we shall call it, the VIC-II. This component gives us several graphic modes which include the normal 25 x 40 character display, a 300 x 200 high resolution dot display and sprites which are high resolution mobile objects. The VIC-II also controls the colours of the display.

The VIC-II uses 47 bytes of RAM known as control registers to record all the important information about the screen; most of the graphic facilities are obtained by POKEing values directly into these locations. A complete memory map of the VIC-II's control registers which run from address 53248 (D000 [hex]) onwards can be found in Appendix D.

THE SID CHIP

The sound and music synthesizer effects of the Commodore 64 are

similarly controlled by an item of hardware, in this case the MOS 6581 Sound Interface Device or SID. The effects available are obtained by controlling three independent oscillators and, as a result, some very complex sounds can be created. There are 29 control registers situated from address 54272 (D400 [hex]) that are used directly with POKE to control the SID; a complete memory map of these registers is given in Appendix E.

CHAPTER 15

CHARACTER DISPLAY MODE

We shall now delve further into the graphic facilities of the 64 and the different display modes that are available, which has been previously mentioned, are controlled by the VIC-II chip.

MORE ON THE VIC-II

Although the Commodore 64 has 64K of addressable locations, the VIC-II is restricted to accessing only 16K blocks or banks at a time. By dividing the 64K into four distinct banks it is possible to alter which block is being looked at and so, all the 64K can be accessed. It is very important that at all times you are aware of which bank the VIC-II is using.

Address (dec)	Address (hex)	Bank	CIA Bit Pattern	Value of Bit Pattern
0000 to 16383	0000 to 3FFF	0	11	3
16384 to 32767	4000 to 7FFF	1	10	2
32768 to 49151	0000 to BFFF	2	01	1
49152 to 65535	C000 to FFFF	3	00	0

Figure 15.1 Banks of Memory used by the VIC-II Chip

When the Commodore 64 is initially switched on the VIC-II points to bank 0.

To change which bank is being accessed by the VIC-II we have to change some bits, known as bank select bits which are located in an I/O device called the 6526 Complex Interface Adapter Chip #2 (or CIA #2). The bits are located in the following registers:

1. 56578 (DD02 (hex)) - Set bits 0 and 1, which enable a change to be made.

2. 56576 (DD00 (hex)) - Set bits 0 and 1 to the bit pattern given in Figure 15.1 that corresponds to the bank required.

To be able to set/unset bits you must understand masking which was explained in Chapter 13 on Logical Operators.

For example, to set the last two bits at address 56578 the value already there must be ORed with the value 3.

$$\begin{array}{r}
 \text{i.e.} \\
 \begin{array}{r}
 \text{X X X X X X X X} \\
 \text{OR } 0 0 0 0 0 0 1 1 \quad (3) \\
 \hline
 \text{X X X X X X } \boxed{1 1} \\
 \hline
 \end{array}
 \end{array}$$

Likewise to set the last two bits at address 56576 to one of the bit patterns given in Figure 15.1, the value at 56576 must first be ANDed with 252 which turns off the first two bits and the new value obtained can be ORed with the bit pattern required.

e.g. to set the VIC-II to point to bank 2,

$$\begin{array}{r}
 \begin{array}{r}
 \text{X X X X X X X X} \\
 \text{AND } 1 1 1 1 1 1 0 0 \quad (252) \\
 \hline
 \text{X X X X X X 0 0} \\
 \hline
 \end{array} \\
 \begin{array}{r}
 \text{X X X X X X X X} \\
 \text{OR } 0 0 0 0 0 0 0 1 \quad (1) \\
 \hline
 \text{X X X X X X } \boxed{0 1} \\
 \hline
 \end{array}
 \end{array}$$

In BASIC, changing the bank that the VIC-II points at involves:

Changing bank accessed by the VIC-II:

POKE 56578, PEEK (56578) OR 3
&
POKE 56576, (PEEK (56576) AND 252) OR B
where B is the bit pattern given in Figure 15.1

SCREEN MEMORY

One of the things that the VIC-II controls is a section of RAM which contains the information that is currently displayed. One byte is used to store the character at each character position so, since there are 25 rows of 40 columns, 1000 bytes are required; these are arranged in the memory sequentially-one row after another. We can move the position of these 1000 bytes to one of 16 positions within the 16K of RAM that the VIC-II is pointing to. This is done by setting the upper 4 bits of address 53272 (D018 (hex)) to a bit pattern that corresponds to the required location.

i.e.

To set screen memory position:

POKE 53272, (PEEK (53272) AND 15) OR B
where B is the value of the
bit pattern given in
Figure 15.2

The addresses available are given in Figure 15.2 but remember that they are relative to the first byte in the bank that the VIC-II is currently pointing to.

Relative to Bank		Bit Pattern Of 53272	Value of B
Address (dec)	Address (hex)		
00000	0000	0000XXXX	0
01024	0400	0001XXXX	16
02048	0800	0010XXXX	32
03072	C000	0011XXXX	48
04096	1000	0100XXXX	64
05120	1400	0101XXXX	80
06144	1800	0110XXXX	96
07618	1C00	0111XXXX	112
08192	2000	1000XXXX	128
09216	2400	1001XXXX	144
10240	2800	1010XXXX	160
11264	2C00	1011XXXX	176
12288	3000	1100XXXX	192
1331	3400	1101XXXX	208
14336	3800	1110XXXX	224
15360	3C00	1111XXXX	240

Figure 15.2 Positions for Screen Memory

When the Commodore 64 is switched on, the screen position defaults to the locations commencing at address 1024 (0400 (hex)).

We shall see later how to display characters on the screen by POKEing directly into the screen memory. Likewise we can find which character is at a certain position by PEEKing the corresponding character position.

THE CHARACTER ROM

In the Commodore 64 there is a 4K ROM character generator; that is, it contains the information on the appearance of the characters used by the computer. The binary representation of every eight bytes within this ROM contains the pattern for each character; every bit set represents a pixel dot set on an 8x8 grid that makes up the character.

e.g. The character 

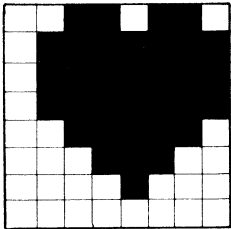
Address	Value	Binary	Character
53912	54	00110110	
53913	127	01111111	
53914	127	01111111	
53915	127	01111111	
53916	62	00111110	
53917	28	00011100	
53918	8	00001000	
53919	0	00000000	

Figure 15.3

The location of the character generator ROM in the Commodore 64's memory is

Character Set	Address	
	(dec)	(hex)
Upper Case/Graphic Characters	53248–55295	D000–D7FF
Upper/Lower Case Characters	55296–57343	D800–DFFF

Figure 15.4 Location of character generator ROM

There is, however, the complication that the addresses used by this ROM are also used by the I/O registers. So, how does the operating system cope with a section of memory being assigned two tasks? When we are trying to access one of these locations how do we know if we are referring to an I/O register or part of the character ROM?

What happens is that the addresses normally refer to the I/O registers which have their values stored in RAM. Whenever the system requires character formation data the VIC-II accesses an image of the ROM which is overlayed on the above addresses; such an image does not affect the underlying values stored in the RAM. This important concept basically means that the section of memory concerned has two parallel set of independent locations, with one being accessed directly and the other through the VIC-II.

It is possible to switch in the ROM memory accessed by the VIC-II so that it can be read directly; this enables us to obtain the character formation data for such uses as enlarged characters or user defined graphics. To do so we must follow these five steps:

- 1) Disable interrupts - an interrupt is a message from an I/O device to the microprocessor saying that it wants to be serviced.
- 2) Switch out the I/O registers and overlay the addresses with an image of the character generator ROM.
- 3) Access data - if the interrupts were not switched off and an interrupt occurred (such as keyboard input) then the system would probably crash since the I/O registers were unavailable.
- 4) Switch I/O registers back in.
- 5) Enable interrupts.

This is done in BASIC by:

```
Disable interrupts:
      POKE 56334, PEEK (56334) AND 254

Enable interrupts:
      POKE 56334, PEEK (56334) OR 1

Switch out I/O register and overlay
character generator ROM:
      POKE 1, PEEK (1) AND 251

Switch in I/O registers:
      POKE 1, PEEK (1) OR 4
```

An interesting technique is to modify the standard characters. For example, the following program produces enlarged characters.

```
10 REM *****
15 REM *      ENLARGED      *
20 REM *    CHARACTERS    *
25 REM *=====*
30 REM *   MRH 1/7/1983   *
35 REM *****
```

```

40 REM
45 FOR I= 53248 TO 55297 STEP 8
50 POKE 56334,PEEK(56334) AND 254
55 POKE 1,PEEK(1) AND 251
60 PRINT "D"
65 FOR J=0 TO 7
70 D(J)=PEEK(I+J)
75 NEXT J
80 POKE 1,PEEK(1) OR 4
85 POKE 56334,PEEK(56334) OR 1
90 PRINT "ADDRESS CONTENTS PATTERN"
95 FOR J=0 TO 7
100 PRINT I+J; TAB(10); D(J); TAB(18);
105 FOR K=7 TO 0 STEP -1
110 KK%=2^K
115 PRINT CHR$( 832 + (134 AND (D(J) AND KK%)<> 0 ));
120 IF D(J) > KK% THEN D(J)=D(J)-KK%
125 NEXT K
130 PRINT
135 NEXT J
140 PRINT "PRESS SPACE BAR."
145 FOR T=1 TO 300: NEXT T
150 GET A$
155 IF A$="" THEN 150
160 IF A$<>" " THEN STOP
165 NEXT I

```

USER DEFINED CHARACTERS

Although the character formation data is normally stored in ROM, it is possible to load it into RAM and then tell the VIC-II to access it from its new position. This enables us to alter some of the data and create our own graphic symbols such as Greek letters, animals, musical notes, space invaders, aeroplanes, bombs, rockets, explosions, etc; the list is endless. A full character set occupies 256 x 8 or 2K bytes of valuable memory; it is, however, possible to use a smaller sub-set.

The new character is defined by setting eight bytes to represent the 8 x 8 grid of pixel dots that make up a character as in Figure 15.3. Initially, the

character should be drawn on a grid and next to each line, starting with the left hand pixel a '1' should be written for each filled square and a '0' for each blank square. The binary number can then be POKEd into the correct addresses.

e.g. a space invader

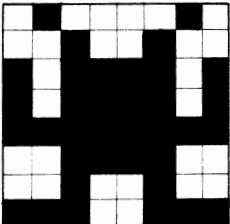
Character	Binary	Decimal
	01000010 00100100 10111101 10111101 11111111 00111100 00100100 11100111	66 36 189 189 255 60 36 231

Figure 15.5

The next stage is to see where we are going to POKE these values.

If you thought the ideas of ROM images were difficult to grasp - then brace yourself, there is more to come; this is all necessary since the VIC-II can only access 16K at a time. In addition to the character ROM image overlaying the I/O registers at the addresses given in Figure 15.4 there is a duplicate image at addresses 4096-8191 (1000-1FFF (hex)) in bank 0 and another at 36864 - 40959 (9000-9FFF (hex)) in bank 2. Again, it is very important at this stage to realise that these images are only apparent to the VIC-II so that the RAM at these locations is unaffected and so is still available by programs and data.

Normally, the VIC-II, while pointing to bank 0 or 2 uses the VIC-II's ROM

images from these locations to create the characters displayed on the screen; note that in the other banks the ROM images are not available and so the characters displayed appear as an irrelevant jumble of dots. We can also tell the VIC-II to forget about images and look directly at other locations in the RAM for the character formation data. As before, if we do not put something at this new position that corresponds to the characters required we shall again get a meaningless display. The new character set can start at one of eight positions in each bank although there are specific locations in certain banks that are used by the system and so should be avoided

e.g.

0 used for system data
 2048 start of users BASIC program
 40960 start of BASIC interpreter
 (see memory layout Figure 14.1)

We can control the location that the VIC-II expects to contain character formation data by setting bits 1 to 3 at address 53272 (D018 (hex)) to one of the values given in Figure 15.6 Note that the addresses given are relative to the first byte in the bank, thus to obtain the true location, the address of the first byte in the bank must be added.

Relative to Bank		Bit Pattern Of 53272	Value of B
Address (dec)	Address (hex)		
00000	0000	XXXX000X	0
02048	0800	XXXX001X	2
04096	1000	XXXX010X	4
06144	1800	XXXX011X	6
08192	2000	XXXX100X	8
10240	2800	XXXX101X	10
12288	3000	XXXX110X	12
14336	3800	XXXX111X	14

Figure 15.6 Positions for character formation data

To do this in BASIC:

Set position for character formation data:

POKE 53272, (PEEK (53272) AND 241) OR B
where B is the value
of the bit pattern
given in Figure 15.6

As a demonstration, in Program 25 we shall downline load the character set from the ROM into the locations commencing at 12288 (3000(hex)). As before we must remember to turn off the interrupts. Before doing so we are going to 'trick' the computer into thinking that the last byte available for BASIC is 12287; this avoids our character generation data being overwritten and corrupted. We do this by setting two pointers which are stored with the system data; each is two bytes and contain the first address unavailable to BASIC.

- 1. Address 51/52 - Points to the highest address available for variables.
- 2. Address 55/56 - Points to the highest address available for the BASIC program.

When the pointers have been amended the memory layout will look like this:

Address (dec)	Address (hex)	Contents
00000 to 01023	0000 to 03FF	SYSTEM DATA
01024 to 02047	0400 to 07FF	SCREEN MEMORY
02048 to 12287	0800 to 2FFF	BASIC PROGRAM & VARIABLES
12288 to 16383	3000 to 3FFF	CHARACTER INFORMATION DATA STORED IN RAM
AS USUAL		

Pointers indicating last bytes available for BASIC program and variables.

Figure 15.7

To set the pointers to an address X:

$X = 12288$

$H = \text{INT}(X/256)$ (High byte)

$L = X - 256 * H$ (Low byte)

POKE 51, L

POKE 52, H

POKE 55, L

POKE 56, H

Everything that has been mentioned in this chapter will now be illustrated in Program 25. Try to recognize the important stages undertaken by the Program.

Program 25 - USER DEFINED GRAPHICS DEMONSTRATION

```
10 REM *****
15 REM * USER DEFINED GRAPHICS *
20 REM * DEMONSTRATION *
25 REM *=====*
30 REM * MRH 1/7/1983 *
35 REM *****
40 REM
45 S=16: B=3: C=12
50 REM SET SCREEN POSITION
55 POKE 53272,(PEEK(53272) AND 15) OR S
60 POKE 56578,PEEK(56578) OR 3
65 REM SET BANK
70 POKE 56576,(PEEK(56576) AND 252) OR B
75 X=C*1024 :H=INT(X/256):L=X-256*H
80 POKE 51,L:POKE 52,H
85 POKE 55,L:POKE 56,H
90 PRINT "***** CHARACTER SET LOADING FROM ROM TO RAM"
95 PRINT CHR$(142) CHR$(8)
100 POKE 56334,PEEK(56334) AND 254
105 POKE 1,PEEK(1) AND 251
110 FOR J= 0 TO 1023
115 POKE J+12288,PEEK(J+53248)
```



```

120 NEXT J
125 POKE 1,PEEK(1) OR 4
130 POKE 56334,PEEK(56334) OR 1
135 PRINT TAB(12);"■ LOADED"
140 REM SET CHARACTER SET POSITION
145 POKE 53272,(PEEK(53272) AND 241) OR C
150 PRINT "■■■■■ PRESS SPACE BAR FOR INVERSE CHARACTERS"
155 GET A$
160 IF A$<>" " THEN 155
165 FOR J= X+8 TO X+216
170 POKE J,255-PEEK(J)
175 NEXT J
180 PRINT TAB(4);"■■■ PRESS SPACE BAR FOR INVASION"
185 GET A$
190 IF A$<>" " THEN 185
195 FOR J=12784 TO 12791
200 READ D
205 POKE J,D
210 NEXT J
215 PRINT "■"
220 FOR J=1 TO 280
225 PRINT "■";CHR$(62);" ";
230 NEXT
235 END
240 DATA 66,36,189,189,255,60,36,231

```

In order that we can change the character formation data it is important to know the order of the characters in the character generation ROM so that we can POKE into the correct offsets. The order of the characters in this ROM is given in appendix B.

MULTICOLOUR-MODE

Up to now we have been restricted to having just two colours at every character position and changing them by using the colour control and reverse video keys. Now we shall look at a new technique which increases the colours available to four per character position whilst reducing our resolution by one half i.e. one character position is built up on a 4 x 8 grid of 'rectangular pixels'; each pixel being set to one of the four possible colours.

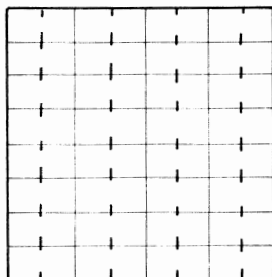


Figure 15.8

Because the resolution has been reduced, it is difficult, though not impossible, to use the multicolour techniques to any great advantage with only single characters. To switch into multicolour mode, bit 4 at address 53270 (D016 (hex)) must be set.

i.e.

Switch on multicolour-mode:

POKE 53270, PEEK (53270) OR 16

Switch off multicolour-mode:

POKE 53270, PEEK (53270) AND 239

As well as setting this flag we have to control multicolour-mode at each character position; so we could, for example, have half the display in multicolour-mode and the other half in normal mode. This is done by setting bit 3 of the corresponding byte in the colour memory to 1 otherwise the character will be displayed in normal mode. When the screen is scrolled upwards this bit will automatically be set for the data appearing at the base of the screen.

The colour of each rectangular pixel in a character is dependent on the two bits of the character formation data as shown below. Of the four possible colours, three are stored at addresses 53281, 53282 and 53283; the fourth is specified as normal by the value stored in the screen memory. The codes for the colours available on the Commodore 64 are given in Figure 15.11.

Bit Pair	Colour	Address
0 0	Background Colour #0	53281 (D021 (hex))
0 1	Background Colour #1	53282 (D022 (hex))
1 0	Background Colour #2	53283 (D023 (hex))
1 1	As normal	Colour memory

Figure 15.9

e.g.  as before would look somewhat like this

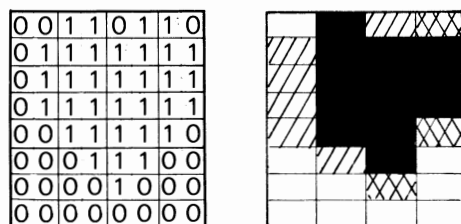


Figure 15.10

It can be seen that this, and similarly most other characters do not resemble their original character; this is one of the reasons why the multicolour-mode technique is not of great importance unless we introduce high resolution displays. The best use of this is to put the character set into RAM and then define your own multicolour shapes using four or more characters positioned next to each other.

One point to notice is the ability to change colours on the display by a single POKE to one of the background colour addresses (53281/3). When this is done all the pixel dot pairs will change instantly. Some excellent effects can be created by this technique. For example, a display could be built up with all four colours set the same so that it would be "invisible". Then, by a couple of POKES to the background colour

addresses, the complete display can be obtained at once. If several pictures are drawn, each using different background colours, it is possible to obtain some animated effects. This is done by displaying one of the pictures at a time by setting all the colours the same except one. The animation is created by alternating the colours so that all the pictures are obtained in succession.

The colours available on the Commodore 64 and their colour codes are listed below.

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

Figure 15.11 The Commodore 64's Spectrum

CHAPTER 16

PEEK AND POKE GRAPHICS

Now we have built up a repertoire of the Commodore 64's BASIC statements and have studied a little graphics we can expect to concentrate on presenting the screen in a more acceptable fashion. Probably every reader of this book will have a friend who, at first sight of your machine, will invite you to switch it on and expect to see graphics like those found in amusement arcades. We shall now take our first step towards such displays.

Up to now we have seen two methods of amending the information displayed on the screen; these are:

1) Clearing the screen with the control character CHR\$(147) and then rebuilding the display; this is done line-by-line, each of which has to be stored and so uses up valuable memory space. This also has the disadvantage that the rebuilding can take a long time.

2) Amending the required section of the screen by repositioning the cursor onto the correct position using the cursor control characters and then printing over the original section.

A third method which was mentioned briefly earlier is to POKE characters directly into the screen memory. This technique also enables one to PEEK addresses in the screen memory to find out what characters are at certain positions on the screen. Another advantage over the previous two methods is that the amended screen is produced much faster.

There are no definite rules for the programmer to use when deciding on whether to use PRINT or POKE to display data; you just have to use your experience to assess which method would be more suitable for the task required. PRINT is both useful and fast for displaying large quantities of data. The problem arises with controlled PRINTing where the data is required to be displayed at a specified position. Whereas the computer

can undertake to PRINT quickly, the time required to calculate the positions and then to set up the cursor can be quite lengthy. This is further emphasised by the fact that Commodore BASIC lacks the command AT used in several other versions of BASIC that enables the user to set up the cursor position by specifying the screen co-ordinates in the PRINT statement. This is one reason why it is difficult to write fast action arcade-type games in BASIC.

However, using the POKE statement enables us to move small quantities of data about the screen at a speed which can give some very acceptable animated effects. With the Commodore 64's screen consisting of 25 rows and 40 columns there are 1000 bytes that can be POKEd. These bytes are ordered one row after another starting with the top left hand corner of the screen; the method of calculating the address of a particular character position is given below.

Character at row R and column C:

$$\text{address} = \text{start of screen memory} + C + 40 * R$$

A further thing to remember is that when you POKE directly into the screen memory, the corresponding position in the colour memory must also be set; if this is omitted the displayed character will be the same colour as the background and so, obviously, will not be visible. Once the displacement from the first byte in the screen memory is calculated, setting the colour memory byte is just a matter of POKEing a colour value into the byte at the same displacement from the start of colour memory.

Character at row R and column C:

$$\text{address} = \text{start of colour memory} + C + 40 * R$$

One important point that must always be remembered is the codes POKEd into the screen memory are not the same as the ASCII CHR\$ codes. The PEEK/POKE codes are numbers which give the position of the characters in the character generator ROM.

Using these PEEK/POKE codes rather than the normal ASCII codes enables the system to quickly access information on the format of the data to be displayed. The PEEK/POKE codes are listed in Appendix B. Because there are two different codes for characters, careful management of a program is required.

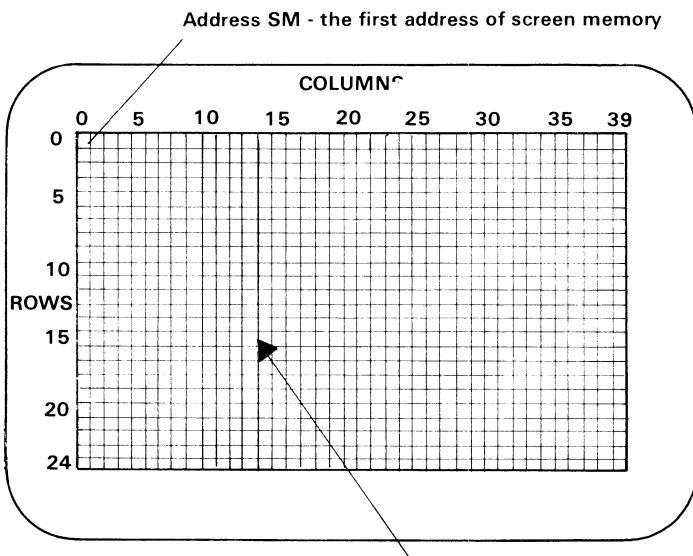


Figure 16.1
The Commodore 64 Screen Display

Characters X, row R column C.

Once the address of a character's position is known it is a simple task to reposition it onto one of its neighbouring positions. Rather than calculating a new column and row, the new location can be calculated by either adding -1 or +1 to move left or right, -40 or +40 to move up or down or a combination of these to move diagonal.

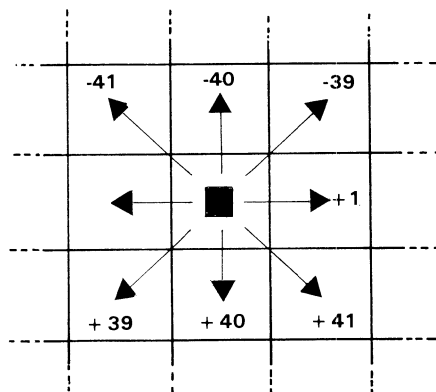
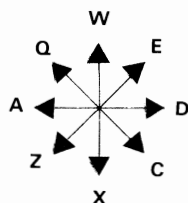


Figure 16.2

It is important to POKE a space into the original position if you don't want a trail to be left.

An example that utilizes these techniques in Program 26, which demonstrates how we can control the movement of a character - a small ball - on the screen. The program also ensures that the character does not leave the screen (or we shall start POKEing undesired values into parts of the RAM other than the screen memory section). This is a simple task; first we set up a border around the screen and, from then on, we can detect the character hitting the border by PEEKing the position that the character is to be moved onto - if it is part of the border we leave the ball stationary. The controls which are used are shown below and are converted into the displacement values given in Figure 16.2 in order that the value of the character position can be amended.



Program 26 - CONTROL

```

10 REM *****
15 REM *    CONTROL    *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 FOR J=0 TO 39
45 SM=1024 :CM=55296 :PC=500
50 PRINT "J"
55 INPUT "HIDDEN LINE (Y/N) ";F$
60 PRINT "J"
65 FOR J=0 TO 39
70 POKE SM+J,160 :POKE SM+J+960,160
75 POKE CM+J,0 :POKE CM+J+960,0
80 NEXT J
85 FOR J= 0 TO 24
90 POKE SM+40*J,160 :POKE SM+40*J-1,160
95 POKE CM+40*J,0 :POKE CM+40*J-1,0
100 NEXT J
105 POKE PC+SM,81 :POKE PC+CM,7
110 GET A$

```



```

115 IF A$="" THEN 140
120 IF A$="P" THEN 60
125 D= (39 AND A$="Z")+(40 AND A$="X")+
      (41 AND A$="C") :IF D<>0 THEN 140
130 D=-(39 AND A$="E")-(40 AND A$="W")-
      (41 AND A$="Q") :IF D<>0 THEN 140
135 D= ( 1 AND A$="D")-( 1 AND A$="A")
140 IF PEEK(SM+PC+D)=160 THEN 110
145 POKE PC+SM, 32+(55 AND F$="N")
150 PC=PC+D :GOTO 105

```

We shall now take a look at two games that make use of this fast technique for moving characters about the screen.

SNAKE

In the game of 'Snake' a player takes control of a snake like figure on the screen which has to eat food that randomly appears for a limited time. If however the snake moves onto the border or onto itself then the game ends and the score obtained is displayed. When the food appears, a calory value is also displayed; this decreases with time so it is in the snakes interest to eat the food as soon as possible. As the snake eats up the food its length will grow in proportion to the amount obtained; however, a large snake is less controllable and so greater care is required. To move the snake to the left press key 'N' and to the right press key 'M'.

Program 27 - SNAKE

```

10 REM *****
15 REM *      SNAKE      *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 DIM S(1000) :A(1)=40 :A(2)=1 :A(3)=-40 :A(4)=-1
45 SM=1024 :CM=55296 :HS=0
50 SC=0 :T=1521 :H=1525 :L=5 :J=1 :PRINT "J"
55 FOR K=CM TO CM+999 :POKE K,7 :NEXT K
60 FOR K=SM TO SM+39 :POKE K,160 :POKE K+40,160 :POKE

```

```

      K+960,160 :NEXT K
65 FOR K=SM TO SM+960 STEP 40 :POKE K,160 :POKE K-1,16
  0 :NEXT K
70 FOR K=T TO H :POKE K,87 :NEXT K
75 FOR K=1 TO L :S(K)=T+K-1 :NEXT K
80 PH=L+1 :PT=1
85 PRINT "S=SCORE";SC;TAB(12);"LENGTH";L;TAB(24);"HI
  GH SCORE";HS
90 GET K$
95 IF K$="N" THEN J=J+1 :IF J=5 THEN J=1
100 IF K$="M" THEN J=J-1 :IF J=0 THEN J=4
105 H=H+A(J) :SQ=PEEK(H) :V=SQ-48
110 IF SQ=160 THEN 195
115 IF SQ=87 THEN 195
120 IF SQ=32 THEN 130
125 F=F+V :SC=SC+V :CV=0 :L=L+V
130 S(PH)=H :PH=PH+1
135 POKE H,87
140 IF F<>0 THEN F=F-1 :GOTO 150
145 POKE S(PT),32 :PT=PT+1
150 IF PT>1000 THEN PT=1
155 IF PH>1000 THEN PH=1
160 IF RND(0)<0.01 THEN CV=9
165 IF CV<10 THEN CV=0 :IF PEEK(CP)<>87 THEN POKE CP,
  32
170 IF CV<>0 THEN CV=CV-1 :POKE CP,INT(CV/10)+48 :GOTO
  185
175 IF RND(0)<0.8 THEN 185
180 CV=100 :CP=SM+1000*RND(0) :IF PEEK(CP)<>32 THEN 180
185 FOR K=1 TO 50 :NEXT K :REM THIS DELAY CAN BE REDUCED
  TO SPEED UP GAME
190 GOTO 85
195 IF SC>HS THEN HS=SC
200 PRINT "S=SCORE";SC;TAB(12);"LENGTH";L;TAB(24);"
  HIGH SCORE";HS
205 FOR K=1 TO 200 :NEXT K
210 PRINT "XXXXXXXXXXXXPRESS SPACE BAR TO CONTINUE"
215 GET K$ :IF K$=" " THEN 50
220 GOTO 215

```

SURROUND

Our second example of PEEK/POKE graphics is a game in which the player and the computer are each in control of drawing a line. The controls are 'W' (up), 'X' (down), 'A' (left) and 'D' (right). If none of these keys are pressed then the line will continue to be drawn in its current direction. Also, a key that forces the line to return back onto itself will be ignored. The object of the game is to survive for a longer period of time than your Commodore opponent without moving onto either line or the border.

Program 28 - SURROUND

```
10 REM *****
15 REM *   SURROUND   *
20 REM *=====
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 FOR J=1 TO 8 :READ A(J) :NEXT J
45 SM=1024 :CM=55296 :BT=0
50 DATA 40,-40,1,-1,40,-40,1,-1
55 PRINT "J" :FOR K=CM TO CM+999 :POKE K,7 :NEXT K
60 FOR K=SM TO SM+39 :POKE K,160 :POKE K+40,160 :POKE
   K+960,160 :NEXT K
65 FOR K=SM TO SM+960 STEP 40 :POKE K,160 :POKE K-1,1
   60 :NEXT K
70 XM=A(INT(RND(0)*4+1)) :YM=A(INT(RND(0)*4+1)) :X=53
   0 :Y=545 :T=TIME :M=1
75 POKE SM+X,160 :POKE CM+X,3 :POKE SM+Y,160 :POKE CM
   +Y,5
80 FOR K=1 TO 40 :REM  DELAY TO AFFECT SPEED OF GAME
85 PRINT "##### TIME ";INT((TIME-T)/60);TAB(20);
   " BEST TIME ";BT
90 M=0 :GET K$
95 IF K$="D" THEN M=1
100 IF K$="A" THEN M=-1
105 IF K$="W" THEN M=-40
110 IF K$="X" THEN M=40
115 IF M=-XM THEN 125
120 IF M<>0 THEN XM=M
125 X=X+XM
```

```

130 IF PEEK(SM+X)<>32 THEN GOTO 215
135 POKE SM+X,160 :POKE CM+X,3
140 IF RND(0)>0.9 THEN 165
145 FOR J=1 TO 3
150 IF PEEK(SM+Y+J*YM)<>32 THEN 165
155 NEXT J
160 GOTO 200
165 M=INT(RND(0)*4+1)
170 FOR P=1 TO 3
175 FOR Q=M TO M+3
180 YM=A(Q)
185 FOR R=1 TO 4-P
190 IF PEEK(SM+Y+R*YM)<>32 THEN 205
195 NEXT R
200 Y=Y+YM :POKE SM+Y,160 :POKE CM+Y,5 :GOTO 85
205 NEXT Q:NEXT P
210 PRINT "##### YOU WIN !! " :GOTO 220
215 PRINT "##### I WIN !! "
220 T=(TIME-T)/60 :IF T>BT THEN BT=INT(T)
225 FOR T=1 TO 2000 :NEXT T
235 PRINT "##### PRESS SPACE BAR TO CONTINUE "
240 GET K$ :IF K$=" " THEN 55
245 GOTO 240

```

Extended Background Colour Mode

There is one final technique that can be useful with PEEK/POKE graphics that enables us to control the background colour of a character position. The 'extended background colour mode' is switched on by setting bit 6 of address 53265 (D011 (hex)).

i.e.

Switch on 'EBCM'

POKE 53265, PEEK (53265) OR 64

Switch off 'EBCM'

POKE 53265, PEEK (53265) AND 191

This mode does, however, have the drawback that the available character set is reduced to the first 64 characters given in Appendix B. The reason for this is that the screen memory is used slightly differently from normal. The first six bits are used as normal, i.e. represent the PEEK/POKE code of the character displayed; the two most significant bits are used to select one of four possible background characters that are stored at addresses 53281/4. The value stored at these locations will be a value between 0 and 15 each of which represents one of the colours listed in Figure 15.11.

Bit 6/7 Of Screen Memory	PEEK/POKE Code for CHR\$(X)	Background Colour Address
0 0	0 + X	#0 53281 (D021 (hex))
0 1	64 + X	#1 53282 (D022 (hex))
1 0	128 + X	#2 53283 (D023 (hex))
1 1	192 + X	#3 53284 (D024 (hex))

Figure 16.3

CHAPTER 17

HIGH RESOLUTION GRAPHICS

Initially we considered the Commodore 64's display screen as being divided up into 25 lines of 40 characters; we then went on to see how each character position was further divided into an 8 x 8 grid of smaller Pixel dots that could be set on or off. The next display mode we shall look at consists of the display screen being built up of 320 x 200 pixel dots each of which can be controlled to be either on or off. Unfortunately, the Commodore 64's BASIC lacks such commands as PLOT or UNPLOT as found on other versions of BASIC which make the control of individual pixels simple. To make reference to a specific pixel unambiguous in our discussions we shall label them with a two number co-ordinate (r,c) where r is the row and c is the column that the pixel lies in.

e.g.

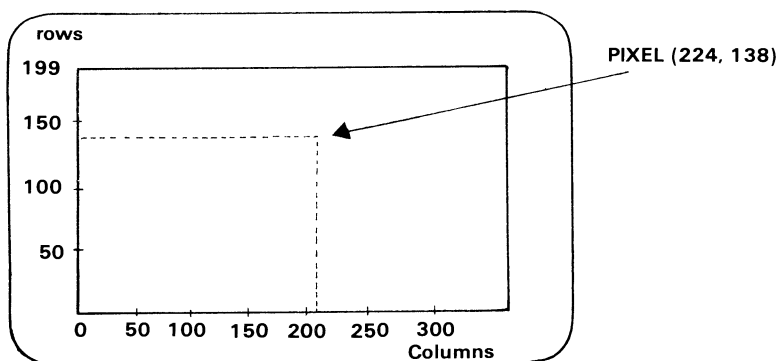


Figure 17.1 The Commodore 64's Pixel Layout

It should be noted that the addresses given in Figure 17.2 are relative to the location from which the character memory commences.

The first stage is to calculate which row and column pixel (r,c) lies in; using a little mathematics this is found to be:

$$\text{row} = 24 - \text{INT} (r/8)$$

and

$$\text{col} = \text{INT} (c/8)$$

i.e. a pixel (r,c) lies in the row $24 - \text{INT} (r/8)$ and column $\text{INT} (c/8)$; considering this character position we find:

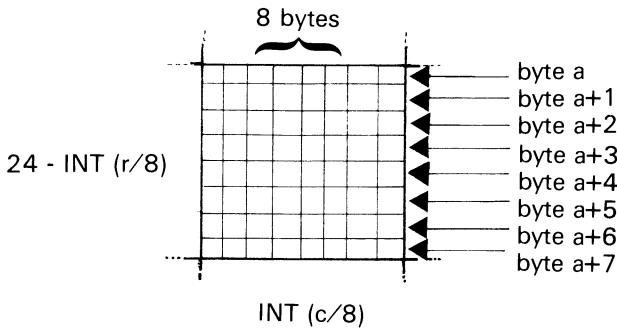


Figure 17.3

The first byte of this character will be:

$$a = \text{row} * 40 + \text{col} * 8$$

Pixel (r,c) will lie in the byte:

$$a + 7 - (r \text{ AND } 7)$$

{ by the masking technique
gives the remainder
when r is divided by 8 }

$$= \text{row} * 40 + \text{col} * 8 + 7 - (\text{r AND } 7)$$

(replacing 'a' from above)

$$= (24 - \text{INT}(\text{r}/8)) * 320 + \text{INT}(\text{c}/8) * 8 + 7 - (\text{r AND } 7)$$

(rearranging)

$$= 7687 - \text{INT}(\text{r}/8) * 320 + \text{INT}(\text{c}/8) * 8 - (\text{r AND } 7)$$

Also the individual bit within this byte is given by:

$$b = 7 - (\text{c AND } 7) \quad \text{(masking again)}$$

It is not too important to remember exactly how these results are arrived at
- but you do need to refer to the results themselves

Location of bit representing pixel (r,c)

$$\text{byte} = 7687 - 320 * \text{INT}(\text{r}/8) + 8 * \text{INT}(\text{c}/8) - (\text{r AND } 7)$$

$$\text{bit} = 7 - (\text{c AND } 7)$$

To set pixel:

POKE byte, PEEK (byte) OR 2^{bit}

To switch on high resolution display mode we must set bit 5 at address 53265 (D011 (hex)).

i.e.

To switch on high resolution graphics:

POKE 53265, PEEK (53265) OR 32

To switch off high resolution graphics:

POKE 53265, PEEK (53265) AND 223

If you have already gone ahead and switched into high resolution mode you will have noticed that the display screen has turned into a multicolour jumble of dots. Clearly, in any program the first thing we must do is clear the section of RAM containing the characters and colours. We have already seen that the character memory commences at the address given by the bits at address 53272 (Figure 15.6). The colours, however, do not come from colour memory as you would expect but from the screen memory (see Figure 15.2). The colours for each character position are given by the upper and lower 4 bits of the corresponding byte in the screen memory. The upper 4 bits give the colour of any pixel that is set and the lower 4 bits give the background colour of the character position; the colour codes are given in Figure 15.11.

We shall now continue and look at some examples using high resolution graphics. All the programs will contain three identical blocks of code and a fourth that is unique to the example. The three common blocks are used to initialise the screen, to plot a pixel (r,c) and to return to character display mode.

line 10	INITIALISATION
line 1000	GRAPHIC PROCEDURE
line 2000	PLOT PIXEL
line 3000	RETURN TO NORMAL MODE

The three common blocks will only be listed for the first example.

When execution has ceased, press any key to return to normal character display mode.

Figure 17.4

Example:

CIRCLES

The co-ordinates of the points on a circle are given by $(a \cos \theta, a \sin \theta)$, where a is the radius of the circle.

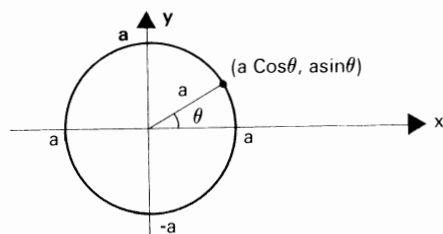


Figure 17.5

If we let θ change from 0 to 2π we obtain all the points on the circle.

Program 29 - CIRCLE

```
10 REM *****
15 REM * HIGH RESOLUTION GRAPHICS *
20 REM * MAIN CODE *
25 REM *=====*
30 REM * MRH 1/7/1983 *
35 REM *****
40 REM
45 REM *** INITIALISATION ***
50 PRINT "C"
55 S=16: B=3: D=8 :SM=1024
60 POKE 53272,(PEEK(53272) AND 15) OR S
65 POKE 56578,PEEK(56578) OR 3
70 POKE 56576,(PEEK(56576) AND 252) OR B
75 X=D*1024 :H=INT(X/256):L=X-256*H
80 POKE 51,L:POKE 52,H
85 POKE 55,L:POKE 56,H
90 FOR J=SM TO SM+999
95 POKE J,1
100 NEXT J
```

```

105 FOR J=X TO X+7999
110 POKE J,0
115 NEXT J
120 POKE 53272,(PEEK(53272) AND 241) OR D
125 POKE 53265,PEEK(53265) OR 32
130 GOSUB 1000
135 GOSUB 3000
140 STOP

```

```

1000 REM *** GRAPHIC PROGRAM ***
1005 REM --- CIRCLE ---
1010 READ RADIUS,XX,YY
1015 PP=2*π
1020 FOR J=0 TO PP STEP PP/40
1025 C=XX+RADIUS*COS(J)
1030 R=YY+RADIUS*SIN(J)
1035 GOSUB 2000
1040 NEXT J
1045 RETURN
1900 DATA 55,160,100

```

```

2000 REM *** PLOTTING ***
2005 BYTE=X+7687-320*INT(R/8)-(R AND 7)+INT(C/8)*8
2010 BIT=7-(C AND 7)
2015 POKE BYTE,PEEK(BYTE) OR 2↑BIT
2020 RETURN
3000 REM *** RETURN TO CHAR. DISPLAY MODE ***
3005 GET Q$
3010 IF Q$="" THEN 3005
3015 POKE 53265,PEEK(53265) AND 223
3020 POKE 53272,(PEEK(53272) AND 240) OR 4
3025 PRINT "J"
3030 END

```

ELLIPSES

The co-ordinates of the points on an ellipse are given by $(a \cos \theta, b \sin \theta)$

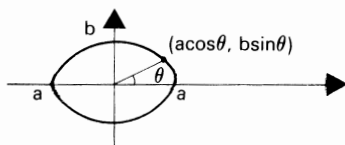


Figure 17.6

If we let θ change from 0 to 2π we obtain all the points on the ellipse.

Program 30 - ELLIPSE

```
1000 REM *** GRAPHIC PROGRAM ***
1005 REM --- ELLIPSE ---
1010 READ AA,BB,XX,YY
1015 PP=2*π
1020 FOR J=0 TO PP STEP PP/40
1025 C=XX+AA*COS(J)
1030 R=YY+BB*SIN(J)
1035 GOSUB 2000
1040 NEXT J
1045 RETURN
1900 DATA 100,55,160,100
```

SINE WAVE

The next example plots a sine wave

Program 31 - SINE WAVE

```
1000 REM *** GRAPHIC PROGRAM ***
1005 REM -- SINE WAVE --
1010 FOR J=0 TO 6*π STEP π/20
1015 C=J*50/π
```

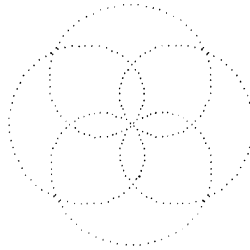
```

1020 R=100+75*SIN(J)
1025 GOSUB 2000
1030 NEXT J
1035 RETURN

```

EPICYCLICS

Program 32 will plot a curve from the family of curves known as epicyclics. Try experimenting by changing the argument in $\text{Cos}(I*2/3)$ to other values, for example $\text{Cos}(I*4/3)$.



Program 32 - EPICYCLICS

```

1000 REM *** GRAPHIC PROGRAM ***
1005 REM      -- EPICYCLIC --
1010 FOR J=0 TO 6*π STEP π/40
1015 C=100+75*COS(J*2/3)*COS(J)
1020 R=100+75*COS(J*2/3)*SIN(J)
1025 GOSUB 2000
1030 NEXT J
1035 RETURN

```

SPIRALS

Program 33 produces a pattern by superimposing several spirals.

Program 33 - SPIRALS

```
1000 REM *** GRAPHIC PROGRAM ***
1005 REM --- SPIRALS ---
1015 LET XX=160 :LET YY=100
1020 FOR ZZ=0 TO 17 STEP 0.1
1025 FOR WW=0 TO 3 STEP 0.5
1030 C=XX+5*ZZ*COS(WW+ZZ)
1035 R=YY+5*ZZ*SIN(WW+ZZ)
1040 GOSUB 2000
1045 NEXT WW
1050 NEXT ZZ
1055 RETURN
```

ETCHER SKETCHER

In our next example, the program allows pictures to be drawn by plotting lines and curves at the control of the user. The direction of a plotted line is chosen by selecting one of the following keys.



By pressing key '9' (RVS ON) we can plot the line in the background colour; this is useful if part of the picture is required to be erased or if the current plot position is to be moved elsewhere without leaving a trailing line. Our common section of code for plotting a pixel has been amended in order that it can either plot or unplot a pixel (r,c).

By pressing key 'O' (RVS OFF) we can return our plotted line to normal colour.

To return to normal character display mode press key 'N'. To clear the screen press key 'P'.

Program 34 - ETCHER SKETCHER

```
1000 REM *** GRAPHIC PROGRAM ***
1005 REM -- ETCHER-SKETCHER --
1015 R=100 :C=160 :F=1
```

```

1020 GOSUB 2000
1025 GET Q$
1030 IF Q$="" THEN 1025
1035 IF C>0 AND (Q$="Q" OR Q$="R" OR Q$="Z") THEN C=C
-1
1040 IF C<319 AND (Q$="E" OR Q$="D" OR Q$="C") THEN C=
C+1
1045 IF R>0 AND (Q$="Z" OR Q$="X" OR Q$="C") THEN R=R
-1
1050 IF R<199 AND (Q$="Q" OR Q$="W" OR Q$="E") THEN R=
R+1
1055 IF Q$="9" THEN F=1
1060 IF Q$="0" THEN F=0
1065 IF Q$="P" THEN 1070
1066 GOTO 1020
1070 RETURN
2000 REM *** PLOTTING ***
2005 BYTE=X+7687-320*INT(R/8)-(R AND 7)+INT(C/8)*8
2010 BIT=7-(C AND 7)
2015 IF F=0 THEN POKE BYTE,PEEK(BYTE) AND (255-2*BIT)
:GOTO 2025
2020 POKE BYTE,PEEK(BYTE) OR 2*BIT
2025 RETURN

```


THREE DIMENSIONAL PLOTTING

The next program demonstrates that it is possible to draw a function of two variables as three dimensional solid surfaces. To see how the program works, consider Figure 17.7. The picture is built up by plotting parallel curved lines at positions on the screen which relate to the value at the plotted position.

If you refer to Figure 17.7, you will see that we are generating one slice at a time, from the front to the back of the 'solid' surface.

To produce the 3-dimensional effect, the program considers the line ST and all lines parallel to it, and calculates the value of the function at the points where the straight lines meet the curved lines. The point is plotted on the screen at a position which takes into account the value of the function and also what curved line is being considered. However to give the impression of a solid surface, the point would only be plotted if there was no surface in front of it, thus a point is only plotted if the y co-ordinate of the plot position is higher than any previous value found on the line being considered.

Try replacing the function line with your own ideas, however, they must be scaled so that the plotting does not leave the screen.

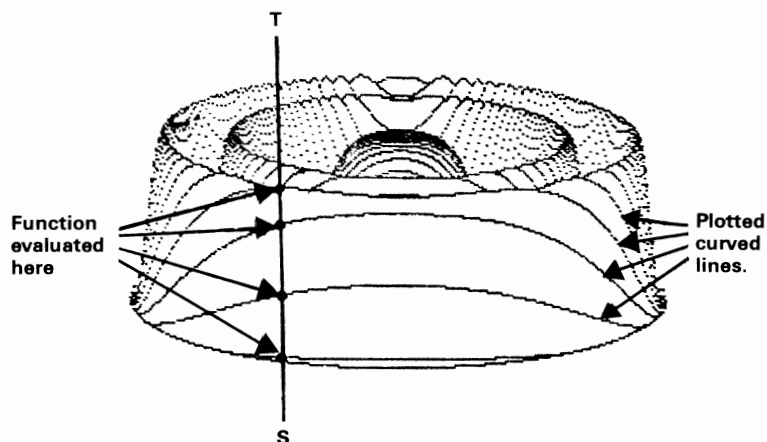


Figure 17.7 Three Dimensional Plotting

Program 35 - THREE DIMENSIONAL PLOTTING

```

10 REM *****
15 REM * 3D GRAPHICS *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 S=16 :B=3 :C=8 :SM=1024
45 POKE 53272,(PEEK(53272) AND 15) OR S
50 POKE 56578,PEEK(56578) OR 3
55 POKE 56576,(PEEK(56576) AND 252) OR B
60 X=C*1024 :H=INT(X/256) :L=X-256*H
65 POKE 51,L :POKE 52,H
70 POKE 55,L :POKE 56,H
75 POKE 53272,(PEEK(53272) AND 241) OR C
80 POKE 53265,PEEK(53265) OR 32
85 FOR J=SM TO SM+999
90 POKE J,1
95 NEXT J
100 FOR J=X TO X+7999
105 POKE J,0
110 NEXT J
115 FOR L=0 TO 127
120 M=SQR(16129-L*L)
125 MAX=-1E9
130 FOR N=-M TO M STEP 4
135 V=25*(SIN(0.001*(L*L+N*N)))
140 P=V+N/4+88
145 IF P<=MAX THEN 165
150 MAX=P
155 R=P :C=160-L :GOSUB 1000
160 C=160+L :GOSUB 1000
165 NEXT N
170 NEXT L
175 GET K$ :IF K$<>" " THEN 175
180 POKE 53265,PEEK(53265) AND 223
185 POKE 53272,(PEEK(53272) AND 240) OR 4
190 STOP
1000 BYTE=X+7687-320*INT(R/8)-(R AND 7)+INT(C/8)*8
1005 BIT=7-(C AND 7)

```

```

1010 POKE BYTE,PEEK(BYTE) OR 21BIT
1015 RETURN

```

Another BASIC command that the Commodore 64 lacks is DRAW (X,Y) which plots a line from the current plot position, say (CC,RR) to a new position X units horizontally to the right (left if negative) and Y units vertically upwards (down if negative).

e.g. DRAW (20,-30)

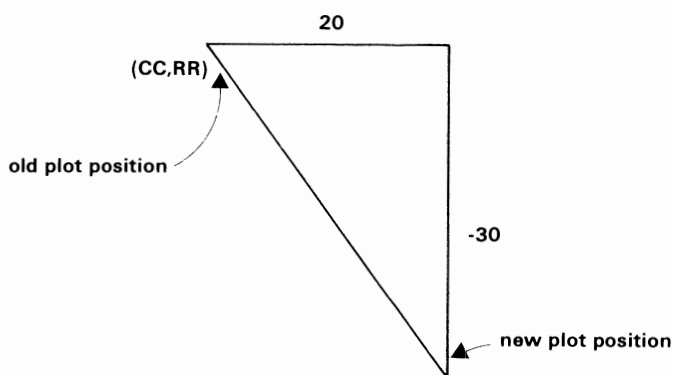


Figure 17.8

The next example uses a common block of code commencing at line 4000 to simulate this command. To use it in your own programs set XX and YY to the horizontal and vertical displacements required and (RR,CC) to the co-ordinate of the current plot position. The new plot position will automatically be assigned to (RR,CC).

Program 36 - STRING PATTERN

```

1000 REM *** GRAPHIC PROGRAM ***
1005 REM -- STRING PATTERN --
1015 FOR P=0 TO 180 STEP 10

```

```

1020 C=70+P : R=10 :GOSUB 2000 : RR=R :CC=C
1025 XX=180-P :YY=P :GOSUB 4000
1030 XX=-P :YY=180-P :GOSUB 4000
1035 XX=P-180 :YY=-P :GOSUB 4000
1040 XX=P :YY=P-180 :GOSUB 4000
1045 NEXT P
1050 RETURN
4000 REM *** DRAW ***
4005 IF ABS(XX)<ABS(YY) THEN 4050
4010 FOR J=CC TO CC+XX STEP SGN(XX)
4015 C=J :IF XX=0 THEN R=RR :GOTO 4025
4020 R=YY/XX*(J-CC)+RR
4025 GOSUB 2000
4030 NEXT J
4035 GOTO 4080
4050 FOR J=RR TO RR+YY STEP SGN(YY)
4055 R=J
4060 IF YY=0 THEN C=CC :GOTO 4070
4065 C=XX/YY*(J-RR)+CC
4070 GOSUB 2000
4075 NEXT J
4080 CC=C :RR=R
4085 RETURN

```

MULTICOLOUR MODE

As with character display mode it is possible to instruct the Commodore 64 to interpret the bits in the character formation data as groups of two bits to give four possible colours. As before this results in a loss of resolution.

The bit patterns representing the following colours are:

Bit Pair	Colour Data		
<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0	Background Colour
0	0		
<table border="1"><tr><td>0</td><td>1</td></tr></table>	0	1	Colour stored in upper 4 bits of screen memory
0	1		
<table border="1"><tr><td>1</td><td>0</td></tr></table>	1	0	Colour stored in lower 4 bits of screen memory
1	0		
<table border="1"><tr><td>1</td><td>1</td></tr></table>	1	1	Colour stored in colour memory
1	1		

Figure 17.9

To switch into multicolour mode from normal high resolution mode we must set bit 4 at address 53270 (D016 (hex)).

i.e.

Switch on multicolour mode:

POKE 53270, PEEK (53270) OR 16

Switch off multicolour mode:

POKE 53270, PEEK (53270) AND 239

SMOOTH SCROLLING EFFECTS

With the aid of the VIC-II we have the facility to scroll the screen one pixel at a time in one of the four directions: up, down, left or right. The VIC-II allows us to position the screen at one of eight horizontal positions and one of eight vertical directions on the display screen, each position being one pixel displaced from the next. To obtain the scrolling effect we position the screen at each horizontal or vertical position (depending on the required direction) which gives a movement of eight pixels. We then have to use some routine to quickly reposition the characters in the screen memory one complete character position in the direction of the scroll and at the same time reposition the screen to the first pixel position. By continually repeating this, a smooth scroll is achieved. To move the characters in the screen memory we require a machine code routine in order that the movement is fast; using a BASIC program with POKes would be far too slow. If we are scrolling vertically upwards we could use the automatic scrolling undertaken by the system software which occurs whenever printing has reached the base of the screen; this, however, is not perfect as it causes a slight flicker.

We can convert the conventional 25 x 40 character screen into a 24 row and/or 38 column display in order that the items that are being smoothly scrolled onto the screen do not suddenly appear. Although fewer columns and rows are visible, the original characters are still present to be printed on - they just are not visible. This is done by unsetting bit 3 at addresses 53270 (D016 (hex)) and 53265 (D011 (hex)).

i.e.

38 column mode:

POKE 53270, PEEK (53270) AND 247

40 column mode:

POKE 53270, PEEK (53270) OR 8

24 row mode:

POKE 53265, PEEK (53265) AND 247

25 row mode:

POKE 53265, PEEK (53265) OR 8

The eight possible screen positions are controlled by the lower 3 bits at addresses 53270 (D016 (hex)) and 53265 (D011 (hex)).

i.e.

X horizontal position

POKE 53270, (PEEK (53270) AND 248) OR X

Y vertical position

POKE 53265, (PEEK (53265) AND 248) OR Y

This is demonstrated in the next program.

Program 37 - VERTICAL SCROLLING

```
10 REM *****
15 REM * VERTICAL SCROLL *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 POKE 53265,PEEK(53265) AND 247
45 PRINT "J"
50 FOR J=1 TO 24 :PRINT :NEXT J
55 POKE 53265,(PEEK(53265) AND 248)+7 :PRINT
60 PRINT "      GETTING MORE FROM YOUR 64 ";
65 FOR J=6 TO 0 STEP -1
70 POKE 53265,(PEEK(53265) AND 248)+J
80 NEXT J
85 GOTO 55
```

CHAPTER 18

SPRITES

At last we have reached the interesting topic of 'sprites'; these are user defined mobile shapes that are controllable in several ways and as a result give the Commodore 64 some of the best graphic facilities that are available on a microcomputer. Having specified the shape and the colours of the sprite along with the position required on the screen, the VIC-II undertakes all the processing needed to display the sprite. Since this operation is controlled by an item of hardware instead of our BASIC program, it is quite feasible for us to write some fast action displays. With the VIC-II we can control up to eight sprites at any one time.

SPRITE FORMATS

If you recall how we produced user defined characters, you'll remember that we defined new characters on an 8 x 8 grid built up of pixel dots; by setting the bits within the eight bytes that represented the character we could select which pixels were required to be on or off. Similarly, a sprite is built up of pixels on a 21 x 24 grid and so requires 63 bytes to record the pixel data.

Each byte will contain the state of 8 of the pixels in the sprite. Thus, we need 63 numbers, (which can be stored in a DATA statement until POKEd into RAM) to represent our sprite. An additional byte is used to indicate the end of the sprite formation data. Each sprite has a byte allocated to it known as the sprite pointer which specifies whereabouts in the RAM and 64 numbers are stored. This sprite pointer has a value between 0 and 255 and when multiplied by 64 gives the location of this data relative to the first byte in the bank currently being accessed by the VIC-II.

e.g. in bank 0, if a sprite pointer contains 190 then the sprite data commences at address $190 * 64 = 12160$

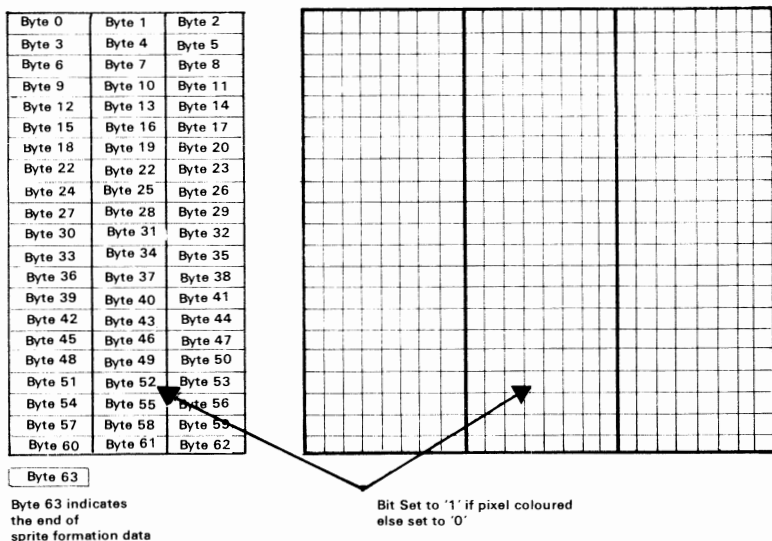


Figure 18.1 The Sprite Grid - 21 x 24 Pixels

Sprite data may not however be situated at a position where character ROM images are present.

The sprite pointer registers are the last eight bytes of the 1K of screen memory; remember that only 1000 bytes are actually used for screen data.

Also, if the screen memory is repositioned then the sprite pointers are moved accordingly.

Normal Sprite Pointer Registers		
Sprite	Address (dec)	Address (hex)
Sprite 0	02040	07F8
Sprite 1	02041	07F9
Sprite 2	02042	07FA
Sprite 3	02043	07FB
Sprite 4	02044	07FC
Sprite 5	02045	07FD
Sprite 6	02046	07FE
Sprite 7	02047	07FF

Figure 18.2 Sprite Pointer Registers

Switching On Sprites

To switch on a sprite, i.e. make it visible, a bit representing the sprite at address 53269 (D015 (hex)) has to be set; this byte is known as the Sprite Enable Register.

7 6 5 4 3 2 1 0 bit

Sprite No.

7	6	5	4	3	2	1	0

 53269

Similarly to switch off a sprite the same bit has to be unset.

i.e.

To switch on sprite no. S

POKE 53269, PEEK (53269) OR 2^{1S}

To switch off sprite no. S

POKE 53269, PEEK (53269) AND (255-2^{1S})

Sprite Colour

A sprite may be any one of the 16 available colours as given in Figure 15.11; for each sprite there is one byte which contains the colour of the pixels that are set. Any unset pixel will take the colour of the background and so appears transparent.

Sprite Colour Registers.		
Sprite	Address (dec)	Address (hex)
Sprite 0 Colour	53287	D027
Sprite 1 Colour	53288	D028
Sprite 2 Colour	53289	D029
Sprite 3 Colour	53290	D02A
Sprite 4 Colour	53291	D02B
Sprite 5 Colour	53292	D02C
Sprite 6 Colour	53293	D02D
Sprite 7 Colour	53294	D02E

Figure 18.3 Sprite Colour Registers

Positioning Sprites

The final thing that we need to know before we can see a sprite demonstration program is how to position and move sprites on the screen. All that we need to do is tell the VIC-II the co-ordinates on the screen where the top left hand corner of the sprite is to be placed.

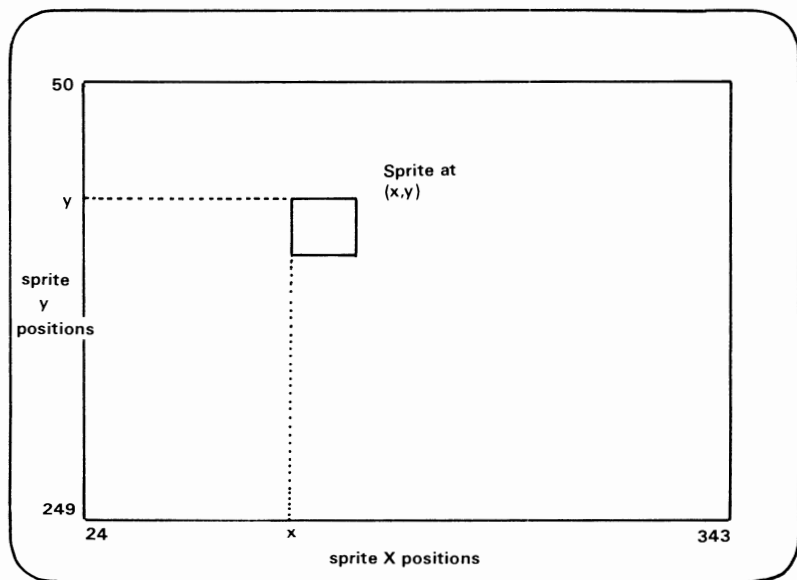


Figure 18.4 Sprite Located at Position (x,y)

Note that the screen co-ordinates are different from those used in the high resolution chapter and that they do not commence at 0. By having screen positions that are just off the visible section of the screen it is possible to move a sprite gradually onto the screen rather than the whole sprite just suddenly appearing at the edge.

Each sprite has an X and a Y register assigned to it; these contain its position co-ordinates which for X, lies in the range 0 to 511, and for Y, lies in the range 0 to 255. This leads to a slight problem for the X co-ordinates since a register is limited to a number in the range 0 to 255; an additional register is therefore used which contains the eight most significant bits for the X registers. Thus if an X co-ordinate is greater than 255 then the corresponding bit in the X m.s.b. register is set and the X register restarts counting from 0.

7 6 5 4 3 2 1 0 bit

Sprite no. 7'6'5'4'3'2'1'0 most significant bits register 53264

The sprite position registers are situated at the following addresses:

Sprite Position Registers		
Sprite	Address (dec)	Address (hex)
Sprite 0 X coord. Sprite 0 Y coord.	53248 53249	D000 D001
Sprite 1 X coord. Sprite 1 Y coord.	53250 53251	D002 D003
Sprite 2 X coord. Sprite 2 Y coord.	53252 53253	D004 D005
Sprite 3 X coord. Sprite 3 Y coord.	53254 53255	D006 D007
Sprite 4 X coord. Sprite 4 Y coord.	53256 53257	D008 D009
Sprite 5 X coord. Sprite 5 Y coord.	53258 53259	D00A D00B
Sprite 6 X coord. Sprite 6 Y coord.	53260 53261	D00C D00D
Sprite 7 X coord. Sprite 7 Y coord.	53262 53263	D00E D00F
X m.s.b. register	53264	D010

Figure 18.5 Sprite Position Registers

The sprite formation data will be stored in DATA statements ready to be POKEd into the RAM commencing at address (192 * 64) or 12288. Program 39 displays and controls the movement of these sprites; to move the rocket up press key 'X' and to move down press key 'Z', to release the bomb press key 'B'. Any other key will cause the rocket to change to a horizontal motion.

When controlling the sprites a test has to be made for the horizontal co-ordinate reaching 255 in order that the most significant bit register is set accordingly. When the sprite has reached the right hand side of the display the sprites X co-ordinate is reset to zero so that the rocket can cross the screen again; when this is done it is advisable to switch off the sprite otherwise the sprite will appear briefly at the centre of the screen. The reason this happens is that when the X co-ordinate is reset we have to POKE into two separate bytes. For the short period of time between the first and second POKE the X co-ordinate will have a value that the VIC-II will interpret as a position near the centre of the screen.

Program 38 - ROCKET SPRITE

```

10 REM *****
15 REM * SPRITE DEMONSTRATION *
20 REM *=====*
25 REM *      MRH 1/7/1983      *
30 REM *****
35 REM
40 S1=192 :C1=7 :S2=193 :C2=7
45 SP=2 :Y=100 :PRINT "□"
50 POKE 2040,S1 :POKE 53287,C1
55 POKE 2041,S2 :POKE 53288,C2
60 FOR J=0 TO 127
65 READ D
70 POKE S1*64+J,D
75 NEXT J
80 H=0 :B=0 :POKE 53248,0 :POKE 53264,0
95 POKE 53269,1
90 FOR X=0 TO 350 STEP SP
95 IF X>255 THEN H=1 :POKE 53264,PEEK(53264) OR 3
100 POKE 53248,X-256*H :POKE 53250,X-256*H
105 GET K$
110 IF K$="" THEN GOTO 120
115 D=(1 AND K$="Z")-(1 AND K$="X")
120 IF Y+D<220 AND Y+D>50 THEN Y=Y+D

```

```

125 POKE 53249,Y
130 IF K$="B" AND B=0 THEN BY=Y: B=1: T=0: POKE 53251
    ,BY :POKE 53269,3
135 IF BY>250 THEN POKE 53269,1 :B=0
140 IF B=1 THEN T=T+0.1 :BY=BY+T :POKE 53251,BY
145 NEXT X
150 POKE 53269,252
155 GOTO 80
1000 REM      ---- [ROCKET DATA] ----
1005 DATA 0,0,0,0,0,0,0,0,0,127,0,0,49,128,0,24,192,
    0,12,96,0
1010 DATA 127,255,224,48,0,56,112,0,15,48,0,56,127,
    255,224,12,96,0,24,192,0
1015 DATA 49,128,0,127,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    ,0,0,0
1020 REM      ---- [BOMB DATA] ----
1025 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1030 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1035 DATA 0,51,0,0,30,0,0,63,0,0,63,0,0,30,0,0,0,0,
    0,0,0,0

```

FURTHER SPRITE FACILITIES

We shall now look at some further features that we can use with our sprite.

Expanded Sprites

Each sprite may be expanded to twice its dimension in a vertical and/or horizontal direction. The sprite still contains the same number of pixels and the resolution does not increase; all that happens is the pixels appear larger. To expand a sprite horizontally we have to set a bit at address 53277 (D017 (hex)). Likewise to expand a sprite vertically we must set a bit at address 53277 (D01D (hex)).

	7	6	5	4	3	2	1	0	bit
Sprite no.	7	6	5	4	3	2	1	0	53271 or 53277

i.e.

Expand sprite no. S horizontally

POKE 53277, PEEK (53277) OR 21S

Contract sprite no. S horizontally

POKE 53277, PEEK (53277) AND (255 - 21S)

Expand sprite no. S vertically:

POKE 53271, PEEK (53271) OR 21S

Contract sprite no. S vertically:

POKE 53271, PEEK (53271) AND (255 - 21S)

For example we can improve the appearance of our rocket sprite in Program 39 by adding the statement, POKE 53277, 1.

SPRITE PRIORITIES

When sprites are moved about the screen there are bound to be times when they cross over each other. So, there need to be fixed rules over when each sprites appear in front of the other. Each sprite is given a priority value which is fixed. Sprite 0 has the highest value and so will appear in front of all other sprites, sprite 1 has the next highest priority and appears in front of all the sprites with the exception of sprite 0. Sprite has the next highest value and so on up to sprite 7 which always appears behind the other sprites. The three dimensional effects that this can cause are illustrated in Program 39 which shows how sprites can appear through other sprites which contain holes.

Program 39 - SPRITE PRIORITIES

```
10 REM *****
15 REM * 3D SPRITES *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 PRINT "THREE DIMENSIONAL EFFECT OF SPRITES"
45 S=192
50 FOR J=0 TO 7
55 READ COL
60 POKE 53287,COL :POKE 2040+J,S
65 POKE 53249+2*J,100+4*J
70 POKE 53248+2*J,100+4*J
75 NEXT J
80 FOR J=0 TO 63
85 READ D :POKE S*64+J,D
90 NEXT J
95 POKE 53281,0 :POKE 53280,0
100 POKE 53277,255 :POKE 53271,255 :POKE 53269,255
105 GET K$ :IF K$="" THEN 105
110 POKE 53269,0
115 END
1000 REM -- COLOUR DATA --
1005 DATA 1,3,4,7,8,10,11,12
1010 REM -- SPRITE DATA --
1015 DATA 255,255,255
1020 DATA 192,0,3,192,0,3,192,0,3,192,0,3,192,0,3,
```

```

192,0,3,192,0,3
1025 DATA 192,0,3,192,0,3,192,0,3,192,0,3,192,0,3,192,0,3,
192,0,3
1030 DATA 192,0,3,192,0,3,192,0,3,192,0,3,192,0,3,192,0,3,
192,0,3
1035 DATA 255,255,255,0

```

Note how we have used one set of sprite formation data for all eight sprites simply by setting all eight sprite pointers to the same position.

ANIMATION USING SPRITES

By rapidly changing the value stored in the sprite pointer it is possible to alter the sprite formation data to give a series of frames and an animated effect. This is demonstrated in program 40 which displays the four frames below which illustrate a bird like figure flapping its wings.

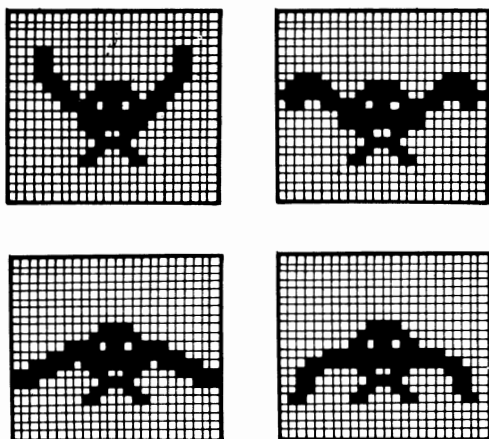


Figure 18.8

Program 40 - ALTERED IMAGES

```

10 REM *****
15 REM *   IMAGES   *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****

```

```

35 REM
40 S=192 :X=140 :Y=120 :PRINT "J"
45 POKE 53281,3 :POKE 53280,3 :POKE 53287,11
50 POKE 53269,1 :POKE 53277,0 :POKE 53271,0
55 FOR J=0 TO 3
60 FOR I=0 TO 63
65 READ D :POKE (S+J)*64+I,D
70 NEXT I :NEXT J
75 FOR J=1 TO 6
80 READ D :P(J)=D
85 NEXT J
90 FOR J=1 TO 6
95 POKE 2040,S+P(J)
100 POKE 53240,X
105 POKE 53249,Y
110 LET X=X+(RND(0)*2-1)
115 LET Y=Y+(RND(0)*2-1)
120 IF X<24 OR X>255 OR Y<50 OR Y>210 THEN X=140 :Y=120
125 NEXT J
130 GOTO 90
1000 REM -- IMAGE ONE --
1005 DATA 0,0,0,0,0,0,0,0,0,0,0,0,24,0,24,24,0,24,24,0,
24,28,0,56
1010 DATA 14,60,112,7,126,224,2,219,192,0,255,0,0,255,
0,0,102,0
1015 DATA 0,60,0,0,102,0,0,195,0,0,0,0,0,0,0,0,0,0,0,0,
0,0
1100 REM -- IMAGE TWO --
1105 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1110 DATA 56,0,28,124,60,62,254,126,127,103,219,230,1,
255,128,1,255,128,0,102,0
1115 DATA 0,60,0,0,102,0,0,195,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0
1200 REM -- IMAGE THREE --
1205 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0
1210 DATA 0,60,0,0,126,0,3,219,192,15,255,240,62,255,
124,248,102,31
1215 DATA 224,60,7,0,102,0,0,195,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0

```


Sprite to display character priorities are dependent on eight bits at address 53275 (D01B (hex)). Each bit is set if the corresponding sprite is required to have a lower priority than the display characters, similarly it is unset if the sprite is to have a higher priority.

	7	6	5	4	3	2	1	0	bit
Sprite no.	7	6	5	4	3	2	1	0	53275

Collision Detection

A very useful facility with sprites is the ability for the VIC-II to detect collisions with other sprites or display characters; the collisions only occur when pixels that are set collide - not just the 21 x 24 grid area. Sprites that are involved in such a collision have their corresponding bit set at address 53278 (D01E (hex)) for a sprite/sprite collision and at address 53279 (D01F (hex)) for a sprite/character collision. Once these addresses have been examined by PEEKing the bits are automatically reset. Using the features in this chapter we can now write fast action arcade type games in BASIC.

Program 41 - GRAND PRIX

```

10 REM *****
15 REM * GRAND PRIX *
20 REM *=====
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 GOSUB 700 :REM INITIALISE
45 GOSUB 600 :REM START
50 IF PEEK(53278)=3 OR PEEK(53279)=1 THEN 500
   :REM COLLISION DETECTED
55 GOSUB 200 :REM CONTROL CAR 1
60 GOSUB 300 :REM CONTROL TRACK
65 GOSUB 400 :REM CONTROL CAR 2
70 SC=SC+1 :GOTO 50
75 :
200 K=PEEK(197) :REM KEY CURRENTLY PRESSED
205 X=X+3*((X>25 AND K=39)-(X<315 AND K=36))
210 H=INT(X/256) :L=X-256*H
215 POKE 53249,Y :POKE 53248,L :POKE 53264,H

```

```

220 RETURN
225 :
300 PRINT SPC(P);
305 ON D+2 GOTO 310,315,320
310 PRINT " / " :GOTO 325
315 PRINT " | " :GOTO 325
320 PRINT " \ "
325 D=INT(RND(0)*3-1) :P=P+D
330 IF P<1 OR P>22 THEN P=P-D :D=0
335 RETURN
340 :
400 IF F=0 THEN XX=P*8+70 :YY=255 :POKE 53269,3 :F=1
      :POKE 53288,INT(RND(0)*4+4)
405 IF F=1 THEN YY=YY-8
410 IF YY<30 THEN F=2
415 IF F=2 AND RND(0)>0.9 THEN F=0
420 POKE 53251,YY :POKE 53250,XX
425 RETURN
430 :
500 POKE 53269,0
505 FOR J=1 TO 4
510 POKE 53281,7
515 FOR T=1 TO 150 :NEXT T
520 POKE 53281,2
525 FOR T=1 TO 150 :NEXT T
530 NEXT J
535 POKE 53281,11
540 IF SC>HS THEN HS=SC
545 PRINT "### SCORE ";SC
550 PRINT "### HIGHEST SCORE ";HS
555 PRINT "### PRESS ANY KEY TO CONTINUE "
560 WAIT 197,191 :REM PAUSE UNTIL KEY IS PRESSED
565 GOTO 45
570 :
600 PRINT "I"
605 FOR J=1 TO 24 :PRINT :NEXT J
610 P=INT(RND(0)*5+8) :D=0 :F=0
615 SC=0 :X=160
620 POKE 53250,0 :POKE 53251,0 :POKE 53249,Y
      :POKE 53248,X
625 POKE 53279,0 :POKE 53278,0 :POKE 53269,1
630 RETURN

```

```

635 :
700 POKE 53269,0
705 S=192 :POKE 2040,S :POKE 2041,S
710 FOR J=0 TO 63
715 READ V :POKE S*64+J,V
720 NEXT J
725 POKE 53277,0 :POKE 53271,0
730 POKE 53276,3 :POKE 53286,0 :POKE 53287,2
735 PRINT "000 GRAND PRIX "
740 INPUT "000 ENTER LEVEL (1-5) ";LV
745 IF LV<1 OR LV>5 THEN 735
750 POKE 53281,11 :POKE 53280,0
755 HS=0 :Y=60+LV*25 :X=160
760 RETURN
780 DATA 0,170,0,12,170,48,14,170,176,14,170,176,12,
      170,48,0,170,0,0,170,0
785 DATA 0,170,0,0,170,0,0,170,0,0,170,0,0,170,0,0,
      170,0,0,40,0,3,40,192
790 DATA 3,170,192,3,170,192,3,40,192,0,40,0,0,40,0,
      0,0,0,0,0,0,0

```

Designing Sprites

The final program of this chapter is a routine which will enable a programmer to quickly produce the data required for sprite formation. All instructions are given in the program.

Program 42 - SPRITE EDITOR

```
10 REM *****
15 REM * SPRITE EDITOR *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 POKE 53281,0 :POKE 53280,0 :POKE 53269,0
45 PRINT "J000 SPRITE EDITOR "
50 PRINT "000 USE SCREEN EDITOR TO DRAW SPRITE WITHIN
   DATA STATEMENTS GIVEN"
55 PRINT "0 ANY CHARACTER WILL BE INTERPRETED AS A
   PIXEL SET"
60 PRINT "0 MOVE CURSOR TO TOP LINE AND THEN PRESS
   RETURN 21 TIMES"
65 PRINT "0 TYPE 'RUN 100' TO CONTINUE
70 PRINT "00000 PRESS ANY KEY TO COMMENCE"
75 WAIT 197,191
80 PRINT "J";
95 POKE 53281,11 :POKE 53280,11
90 LIST 10000-10220
95 END
100 PRINT "J000 SPRITE EDITOR "
105 POKE 53281,0 :POKE 53280,0 :DIM V(62) :S=192
110 FOR I=0 TO 20
115 READ A$
120 FOR J=0 TO 2
125 B$=MID$(A$,8*J+1,8)
130 N=0
135 FOR K=1 TO 8
140 C$=MID$(B$,K,1)
145 N=2*N+(1 AND C$<>" ")
150 NEXT K
155 V(3*I+J)=N
160 NEXT J
```



```

165 NEXT I
170 FOR I=0 TO 20
175 FOR J=0 TO 2
180 PRINT TAB(J*5);V(3*I+J);
185 POKE S*64+I*3+J,V(3*I+J)
190 NEXT J
195 PRINT
200 NEXT I
205 POKE 2040,S
210 POKE 53248,220 :POKE 53249,120
215 POKE 53269,1 :POKE 53287,7
220 END

10000 DATA "*****"
10010 DATA "*****"
10020 DATA "*****"
10030 DATA "*****"
10040 DATA "*****"
10050 DATA "*****"
10060 DATA "*****"
10070 DATA "*****"
10080 DATA "*****"
10090 DATA "*****"
10100 DATA "*****"
10110 DATA "*****"
10120 DATA "*****"
10130 DATA "*****"
10140 DATA "*****"
10150 DATA "*****"
10160 DATA "*****"
10170 DATA "*****"
10180 DATA "*****"
10190 DATA "*****"
10200 DATA "*****"
10210 REM   ENTER DATA STATEMENTS
10220 REM   RUN 100

```

MULTICOLOUR SPRITES

In order that sprites can contain pixels of up to four different colours, any sprite can be switched into multicolour mode. Like multicolour characters, the colours obtained are dependent on bit pairs in the formation data. Again, like multicolour characters, the resolution of a sprite will be reduced by one half from 24 pixels across to 12 double pixels; this loss in resolution is more than compensated by the ability to create some excellent coloured sprites. The colours displayed by the different bit patterns are:

Bit Pair	Colour Displayed		
<table border="1"><tr><td>0</td><td>0</td></tr></table>	0	0	Screen colour
0	0		
<table border="1"><tr><td>0</td><td>1</td></tr></table>	0	1	Sprite multicolour #0 stored at address 53285 (D025 (hex))
0	1		
<table border="1"><tr><td>1</td><td>0</td></tr></table>	1	0	Sprite colour register
1	0		
<table border="1"><tr><td>1</td><td>1</td></tr></table>	1	1	Sprite multicolour #1 stored at address 53286 (D026 (hex))
1	1		

To switch a sprite into multicolour mode the corresponding bit at address 53276 (D01C (hex))

	7	6	5	4	3	2	1	0	bit								
Sprite no.	<table border="1"><tr><td>7</td></tr></table>	7	<table border="1"><tr><td>6</td></tr></table>	6	<table border="1"><tr><td>5</td></tr></table>	5	<table border="1"><tr><td>4</td></tr></table>	4	<table border="1"><tr><td>3</td></tr></table>	3	<table border="1"><tr><td>2</td></tr></table>	2	<table border="1"><tr><td>1</td></tr></table>	1	<table border="1"><tr><td>0</td></tr></table>	0	532 / 6
7																	
6																	
5																	
4																	
3																	
2																	
1																	
0																	

i.e.

Sprite no. S in multicolour mode:

POKE 53276, PEEK (53276) OR 2↑S

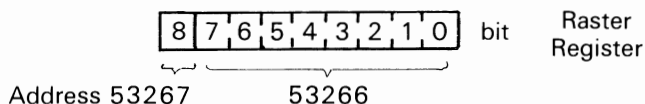
Sprite no. S in normal colour mode:

POKE 53276, PEEK (53276) AND (255- 2↑S)

Note that in collision detection, the sprite multicolour #0 is taken to be transparent.

RASTER REGISTER

When a television screen is scanned it builds up the picture line by line. The line number that is currently being scanned is stored at the raster register which is located at address 53266 (D012 (hex)) and the most significant bit of 53267 (D013 (hex)).



If you ever go on to use machine code on your Commodore 64 you can use this register for some excellent effects; unfortunately, the BASIC interpreter is far too slow to be of any use with such techniques. The effects available basically involve timing changes. For example, the fast execution of machine code programs is liable to cause screen flickering when using sprites. This can be overcome by moving and redisplaying the sprites while the television is scanning part of the display which is not in the visible region, i.e. when the value in the raster register is less than 50 or greater than 249.

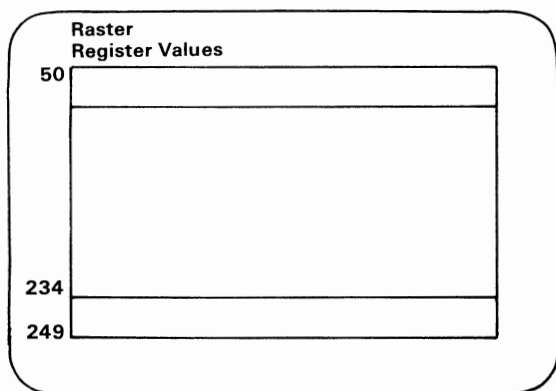


Figure 18.9 Raster Register Values

We could use a similar technique to display certain parts of the screen in high resolution mode and the remaining parts in normal character display mode. This is done by switching from one display mode to another as the raster register reaches certain values. For example, consider a high resolution screen with the top and bottom two lines in normal character display mode. Such a screen is achieved by continually monitoring the raster register; when it reaches 66 the screen is put into high resolution mode and when it reaches 234 it is returned to normal character display mode. Unfortunately this technique is only feasible with machine code to obtain the necessary speed.

INTERRUPTS

Experts on 6510 machine code programming will be able to use the technique of interrupts to detect when the raster register has reached specified values.

Interrupts occur when certain conditions have arisen, the processor then leaves whatever it was doing and undertakes to do another bit of processing and then on completion jumps back to what it was previously doing - all rather similar to a BASIC subroutine. A particular example on the Commodore 64 is the keyboard interrupt; every 1/60th of a second control leaves whatever it was doing and checks to see what key, if any, is being pressed.

Three useful interrupts that can be detected with sprites on the Commodore 64 are when the raster count is equal to a stored value, a sprite/character or sprite/sprite collision occurs.

When one of these interrupts occurs, the corresponding bit of the interrupt status register at address 53273 (D019 (hex)) becomes set; once set the bit must be cleared before another interrupt can be detected.

Bit at 53273	Condition
0	Current raster count = stored raster count
1	Sprite/character collision
2	Sprite/sprite collision

Figure 18.10 Sprite Interrupts

In order for an interrupt to occur when the raster register reaches a required value, the value must first be POKEd into the raster register. The interrupt will then take place as soon as the raster count reaches this value.

One final point, in order to detect these interrupts, the corresponding bit (0, 1 or 2) of the Interrupt Enable Register at address 53274 (D01A hex) must be set. Thus, it is possible to select which interrupts are required.

CHAPTER 19

SOUND AND THE COMMODORE 64

There will, no doubt, be many owners of the Commodore 64 who when choosing a microcomputer were swayed towards their machine because of the excellent sound facilities that it boasts. Our next topic is to investigate these features and see what sounds can be produced. It should be pointed out at this stage that there is only room in this book to scratch the surface of the subject since the topic is so immense.

Commodore have produced a book specialising on the subject called 'Making Music on your Computer' which is worthwhile reading for anyone who really wants to learn more on the subject. Also, anyone whose main interests lie in the subject, are advised to purchase some of the additional software that is available which enables the user to access the sound features more easily.

CHARACTERISTICS OF SOUND WAVES

Our first step is to examine the characteristics of sound waves. We all have plenty of experience with sound in everyday life and know that different sounds can be distinguished - but what makes one sound different from another? Each sound is made by a transfer of energy caused by the disturbance of air from one point, called the source, to the ear. The difference in the sounds are effected by three factors i) pitch ii) quality and iii) volume.

If the source oscillates f times a second, the sound emitted is said to have a frequency of Hz (Hertz). Also if the energy travels a distance of λ from the source in one second then the sound produced has a wavelength of λ .

This can be represented diagrammatically as in Figure 19.1

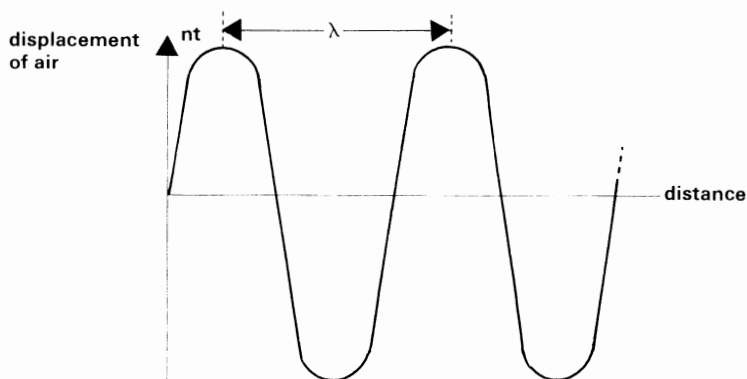


Figure 19.1 Sound Wave

The pitch of a note is dependent on the frequency of the sound wave produced; a high frequency produces a high pitched note and a low frequency produces a low pitched note. The musical interval between two notes is proportional to the ratio of the frequencies rather than the difference. Thus, the difference in pitch to the ear is the same if a note increases from 250 Hz to 500 Hz as from 1000 Hz to 2000 Hz. If one note has twice the frequency of another then the two notes are said to be one octave apart.

A tune which is played on several different musical instruments can still be distinguished even if exactly the same notes are played. This is because the quality of the waveforms from each instruments differ. For example, while a tuning fork would produce a waveform like that in Figure 19.1, a musical instrument would probably look more like that in Figure 19.2.

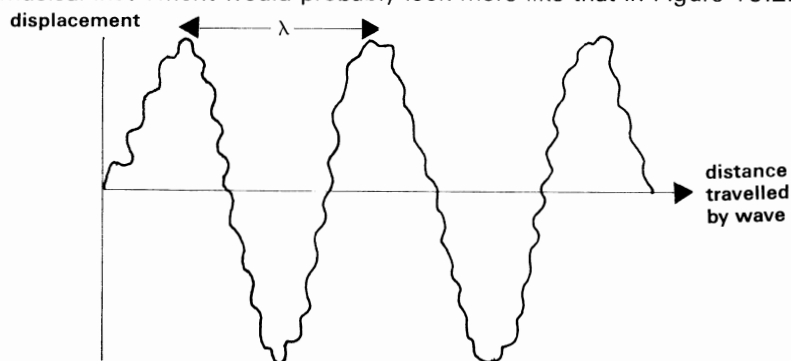


Figure 19.2 Waveform produced from Some Musical Instrument

CONTROLLING THE SOUND FEATURES

Just as with the graphic facilities which were mainly controlled by an item of hardware, i.e. the VIC-II, the sound facilities are controlled by the 'Sound Interface Device' (or SID). The sound output obtained from the SID consists of three voices emitted from three identical oscillators. Each of the voices can emit sounds with one of four different waveforms, i.e. sawtooth, triangular, pulse and white noise as in Figure 19.3 to 19.6. It is the waveforms that gives the sound some of its distinctive properties. It is also possible to 'AND' the waveforms which is like merging the two together which can give us even more flexibility to produce some interesting sounds.

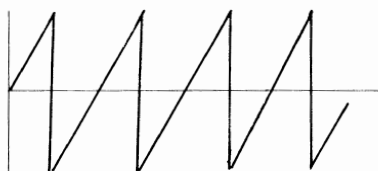


Figure 19.3 Sawtooth Waveform

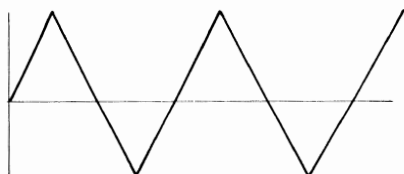


Figure 19.4 Triangular Waveform

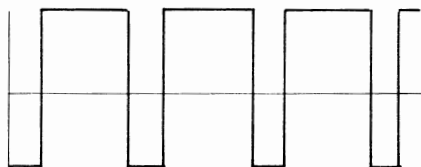


Figure 19.5 Pulse Waveform

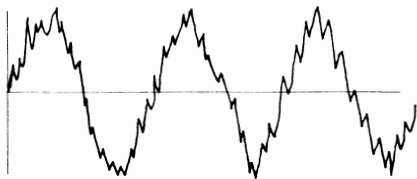


Figure 19.6 White Noise

To make a sound there are four properties of it that we must let the SID know about; these are

- 1) Volume
 - 2) Waveform
 - 3) Frequency
- and 4) ASDR

The SID also needs to know from which voices the sounds are emitted from. The values for each of these properties are given to the SID by POKEing values into the SID control registers at addresses 54272 to 54300 (D400 - D41C (hex)).

Referring to Appendix E (which gives these registers) it can be seen that voice 1 is controlled by registers 54272 - 54278, voice 2 by 54279 to 54285 and voice 3 by 54286 to 54292; the remaining registers have other specific uses. To some users the idea of POKEing values into the SIDs control registers and the lack of any suitable BASIC sound statements may seem rather unfriendly; others however will prefer this direct method of control.

Our first step is to see how to set these properties to obtain a sound.

VOLUME

The volume of the sound is set to a value in the range 0 to 15 by POKEing the value into address 54296 (D418 (hex)). A volume set to 0 will turn off the sound.

To select Volume V (0 - 15):

POKE 54296, V

WAVEFORM

The waveform required is selected by setting one of the following bits plus bit 0 at addresses 54276 (D404 (hex)), 54283 (D40B (hex)) or 54290 (D412 (hex)) depending on the voice required.

Waveform	Bits Set
Triange	0 & 4
Sawtooth	0 & 5
Pulse	0 & 6
White Noise	0 & 7

Figure 19.7

Bit 0 is known as the gate bit, when set the note is sounded, when unset the note is switched off.

To set a triangular waveform in voice V (1-3):

POKE 54276 + (V-1) * 7, 17

To set a sawtooth waveform in voice V (1-3):

POKE 54276 + (V-1) * 7, 33

To set a square waveform in voice V (1-3):

POKE 54276 + (V-1) * 7, 65

To set a white noise waveform in voice V (1-3):

POKE 54276 + (V-1) * 7, 129

With a pulsewave the proportion of the high displacement to the low displacement must be specified.

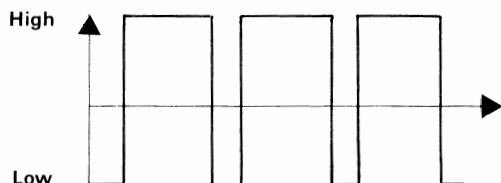


Figure 19.8 Pulse Waveform

If the high displacement is $X\%$ of the total, then a pulse width of $40.95 \times X$ has to be stored. The pulse width is stored using 12 bits with the 8 least significant bits at 54274 (D402 (hex)), 54281 (D409 (hex)), 54288 (D410 (hex)) and the four most significant bits at 54275 (D403 (hex)), 54282 (D40A (hex)), 54289 (D411 (hex)) all depending on which voice is required.

To set a pulse width of $X\%$ in voice V

$$P = X * 40.95$$

$$H = \text{INT} (P/256)$$

$$L = P - 256 * H$$

POKE 54274 + (V-1) * 7, L
POKE 54275 + (V-1) * 7, H

FREQUENCY

The frequency value is a number between 0 and 65535, and is stored using 16 bits at addresses 54272/3 (D400/1 (hex)), 54279/80 (D407/8 (hex)) or 54286/7 (D40E/F (hex)) depending on the voice required; the latter address in each case contains the most significant byte.

The frequency output, in Hertz, is approximately $3/50$ ths of the stored value.

To set frequency value $F(0-65535)$ for voice V

$$HF = \text{INT} (F/256)$$

$$LF = F - 256 * HF$$

POKE 54272 + (V-1) * 7, LF
POKE 54273 + (V-1) * 7, HF

ADSR

There are four more items that must be specified for each voice called attack, decay, sustain and release each of which has a value in the range 0 to 15.

When a note is sounded there are four stages of loudness which are heard; these are:

- 1) Initially the volume increases from zero to its maximum value at a rate, called the attack.
- 2) It then falls back to an average volume at a rate called the decay.
- 3) This average volume is then sounded; the proportion of the peak volume to this average volume is called the sustain level.
- 4) Finally, the volume falls back to zero at a rate called the release.

These parameters are called the ADSR values.

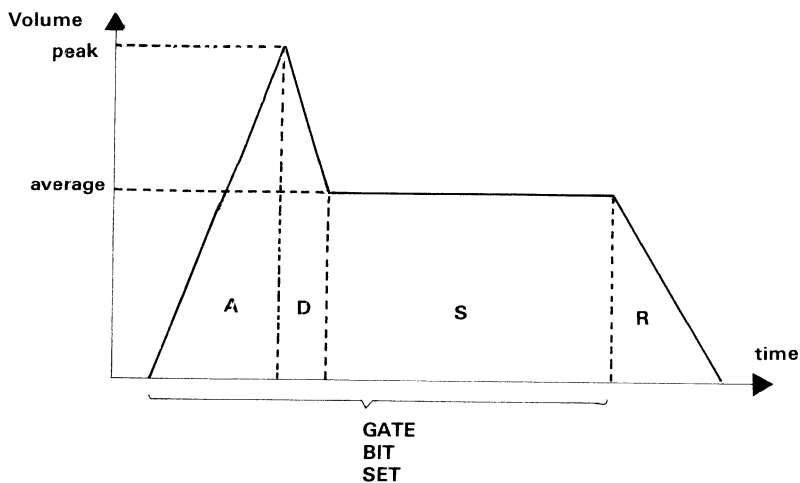


Figure 19.9 The ADSR Envelope

The attack, decay and release rates with their corresponding values that have to be stored are listed in Figure 19.10.

Value	Attack Rate	Decay Rate	Release Rate
0	2ms	6m	6ms
1	8ms	24ms	24ms
2	16ms	48ms	48ms
3	24ms	72ms	72ms
4	38ms	114ms	114ms
5	56ms	168ms	168ms
6	68ms	204ms	204ms
7	80ms	240ms	240ms
8	100ms	300ms	300ms
9	250ms	750ms	750ms
10	500ms	1.5s	1.5s
11	800ms	2.4s	2.4s
12	1s	3 s	3 s
13	3s	9 s	9 s
14	5s	15 s	15 s
15	8s	24 s	24 s

Figure 19.10 Attack, Decay and Release Rate Values

The attack rate is stored in the upper 4 bits and the decay rate in the lower 4 bits at addresses 54277 (D405 (hex)), 54284 (D40C (hex)) or 54291 (D413 (hex)) depending on the voice required. Similarly the sustain level is stored in the upper 4 bits and the release rate in the lower 4 bits at address 54278 (D406 (hex)), 54285 (D40D (hex)) or 54292 (D414 (hex)) again depending on the voice required.

To set attack rate A (0-15), decay rate D (0-15) in voice V:

$\text{POKE } 54277 + (V-1) * 7, A * 16 + D$

To set sustain level S (0-15), release rate R (0-15) in voice V:

$\text{POKE } 54278 + (V-1) * 7, S * 16 + R$

Our first program to utilise the Commodore 64's sound facilities in Program 44 which plays a famous tune 'Fur Elise'. The notes of the tune have their frequencies stored in DATA statements; by replacing these values you can produce your own tunes - Figure 19.11 should assist the musical readers to obtain the notes they require. Remember that the musical interval between two notes is proportional to the ratio of their frequencies and notes that are one octave apart have a frequency ratio of 2; thus the frequency of any note can easily be calculated.

Another improvement that can be made on program 43 for your own tunes is to include a second parameter in the DATA statements that gives the duration of each note.

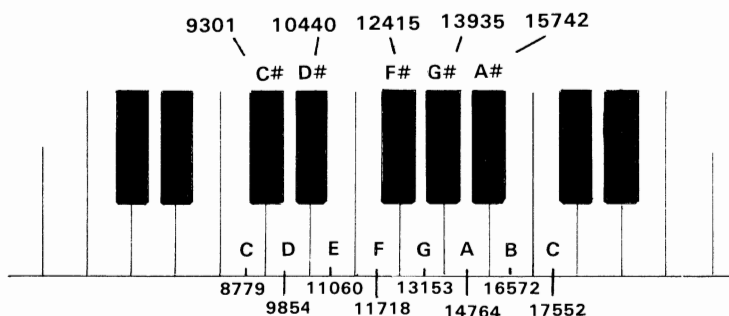


Figure 19.11 Frequency Values

If you ever use more than one voice (on your 64!) it is important to co-ordinate the timing of them. One method with BASIC programming is to use three arrays with each corresponding element containing the three frequencies that are to be sounded simultaneously.

Program 43 - FUR ELISE

```

10 REM *****
15 REM *   FUR ELISE   *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 A=1 :D=8 :S=15 :R=8 :W=15 :DR=115 :W=33 :PRINT "J"
45 POKE 54277,A*16+D
50 POKE 54278,S*16+R
55 POKE 54296,W
56 POKE 54276,W-1
60 DIM N(35)
65 FOR J=0 TO 2

```

```

70 RESTORE
75 FOR I=0 TO 11
80 READ V :N(J*12+I)=V*(2↑J)
85 NEXT I
90 NEXT J
95 DATA 2145,2273,2408,2551,2703,2864,3034,3215,3406,
    3609,3823,4050
100 READ V
105 IF V<0 THEN POKE 54276,W-1 :END
110 F=N(V) :H=INT(F/256) :L=F-256*H
115 POKE 54272,L :POKE 54273,H
120 POKE 54276,W
125 FOR T=1 TO DR :NEXT T
130 POKE 54276,W-1
135 GOTO 100
1000 DATA 16,15,16,15,16,11,14,12,9,9,4,9,11
1005 DATA 4,8,11,12,12,4
1010 DATA 16,15,16,15,16,11,14,12,9,9,4,9,11
1015 DATA 2,12,11,9
1020 DATA 16,15,16,15,16,11,14,12,9,9,4,9,11
1025 DATA 4,8,11,12,12,4
1030 DATA 16,15,16,15,16,11,14,12,9,9,4,9,11
1035 DATA 2,12,11,9,9,11,12,14
1040 DATA 16,15,16,15,16,11,14,12,9,9,4,9,11
1045 DATA 4,16,4,16,16,17,15,16,15,16,15,16,15
1050 DATA 16,15,16,15,16,11,14,12,9,9,4,9,11
1055 DATA 2,12,11,9,9,11,12,14
1060 DATA 9,9,11,12,14
1065 DATA 16,7,17,16,16,14,5,16,14,14,12,4,14,12,11,4,16
1070 DATA 4,16,4,16,16,17,15,16,15,16,15,16,15
1075 DATA 16,15,16,15,16,11,14,12,9,9,4,9,11
1080 DATA 2,12,11
1085 DATA 9,9,12,12,12,12,17,16,16,14,14,22,21,21,19,
    17,16,14,12,10,9,9,7,9,10
1090 DATA 12,14,15,15,16,16,17,9,12,12,14,11,12,19,7,
    18,9
1095 DATA 19,11,19,12,19,14,19,16,19,24,23,21,19,16,
    14,19,17,14,12,19,7,19,9
1100 DATA 19,11,19,12,19,14,19,16,19,24,23,21,19,17,
    16,14,19,17,14,16,17,16,17

```

1105 DATA 16,15,16,11,16,15,16
1110 DATA 11,16,15,15,16
1115 DATA 11,16,15,15,16
1120 DATA 11,16,15,15,16
1125 DATA 11,16,15,15,16,16,15
1130 DATA 16,7,17,16,16,14,5,16,14,14,12,4,14,12,11,
4,16
1135 DATA 4,8,11,12,12,4
1140 DATA 4,16,4,16,16,17,15,16,15,16,15,16,15
1145 DATA 2,12,11,9,9,11,12,14
1150 DATA 2,12,11
1155 DATA 16,7,17,16,16,14,5,16,14,14,12,4,14,12,11,
4,16
1160 DATA 4,16,4,16,16,17,15,16,15,16,15,16,15
1165 DATA 16,7,17,16,16,14,5,16,14,14,12,4,14,12,11,
4,16
1170 DATA 2,12,11,9
1175 DATA -1

Trying to teach someone to be able to use the available sound facilities to the utmost would be like trying to teach the piano or guitar - and from a single chapter in a book a sheer impossibility! Anyone who has ever tried to learn a musical instrument will be well aware that it takes years of practice to obtain such talents - and the Commodore 64 sound machine is no exception. Since the only way for the reader to master these skills is to practice, you can use the next program which enables you to explore how changing the parameters of the SID affects the sounds produced. In order to eliminate any synchronization problems, only one voice has been utilised.

The program uses a small musical keyboard of the digit keys 1 to 9. Your local computer shop may be able to supply you with a larger, real musical keyboard which could be interfaced to the Commodore 64, in which case the program should be amended to accept a larger number of key inputs.

By pressing function key '1' you have the option of changing any of voice 1's control registers.

Program 44 - EXPLORING THE SID

```

10 REM *****
15 REM * EXPLORING SID *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 A=0 :D=8 :S=8 :R=8
45 V=15 :DR=75 :W=17 :W$=" TRIANGLE"
50 FR=3000 :P=5
55 POKE 54277,16*A+D :POKE 54278,16*S+R :POKE 54296,V
60 PRINT "***** EXPLORING SID "
65 PRINT "***** PRESS ANY FUNCTION KEY TO",
    , " CHANGE PARAMETERS "
70 GET K$ :IF K$="" THEN 70
75 K=ASC(K$)-48
80 IF K>84 AND K<93 THEN GOTO 200
85 IF K<1 OR K> 9 THEN 70
90 F=(FR+100*K)*P
95 H=INT(F/256) :L=F-256*H
100 POKE 54273,H :POKE 54272,L
105 POKE 54276,W
110 FOR T=1 TO DR :NEXT T
115 POKE 54276,W-1

```

```

120 GOTO 70
200 PRINT "XXXXXXXX EXPLORING SID "
205 PRINT "XXXXXXXX PRESS THE FOLLOWING KEYS TO",
    , " CHANGE PARAMETERS XXX"
210 PRINT " F1 ... ATTACK ";TAB(18);A
215 PRINT " F2 ... DECAY ";TAB(18);D
220 PRINT " F3 ... SUSTAIN ";TAB(18);S
225 PRINT " F4 ... RELEASE ";TAB(18);R
230 PRINT " F5 ... PITCH ";TAB(18);P
235 PRINT " F6 ... DURATION";TAB(18);DR
240 PRINT " F7 ... VOLUME ";TAB(18);V
245 PRINT " F8 ... WAVEFORM";TAB(18);W$
250 IF W$=" PULSE" THEN PRINT SPC(18);PW;"%"
255 PRINT "XXX SPACE BAR TO RETURN "
260 GET K$ :IF K$="" THEN 260
265 IF K$=" " THEN 55
270 K=ASC(K$)-132 :IF K<1 OR K>8 THEN 260
275 ON K GOSUB 300,400,500,600,350,450,550,650
280 GOTO 200

```

```

300 PRINT "XXXX ATTACK ";A
305 INPUT "XXX ENTER NEW VALUE (0-15)";A
310 IF A<0 OR A>15 THEN 300
315 RETURN
350 PRINT "XXXX DECAY ";D
355 INPUT "XXX ENTER NEW VALUE (0-15)";D
360 IF D<0 OR D>15 THEN 350
365 RETURN
400 PRINT "XXXX SUSTAIN ";S
405 INPUT "XXX ENTER NEW VALUE (0-15)";S
410 IF S<0 OR S>15 THEN 400
415 RETURN
450 PRINT "XXXX RELEASE ";R
455 INPUT "XXX ENTER NEW VALUE (0-15)";R
460 IF R<0 OR R>15 THEN 450
465 RETURN
500 PRINT "XXXX PITCH ";P
505 INPUT "XXX ENTER NEW VALUE (1-10)";P

```

```

510 IF P<1 OR P>10 THEN 500
515 RETURN
550 PRINT "DURATION ";DR
555 INPUT "ENTER NEW VALUE (1-300)";DR
560 IF DR<1 OR DR>300 THEN 550
565 RETURN
600 PRINT "VOLUME ";V
605 INPUT "ENTER NEW VALUE (0-15)";V
610 IF V<0 OR V>15 THEN 600
615 RETURN
650 PRINT "WAVEFORM";TAB(15);W
655 PRINT "PRESS:"
660 PRINT " F1 ... TRIANGLE "
665 PRINT " F3 ... SAWTOOTH "
670 PRINT " F5 ... PULSE "
675 PRINT " F7 ... WHITE NOISE "
680 GET K$: IF K$="" THEN 680
685 K=ASC(K$)
690 IF K=133 THEN W=17 :W$=" TRIANGLE"
695 IF K=134 THEN W=33 :W$=" SAWTOOTH"
700 IF K=135 THEN W=65 :W$=" PULSE"
705 IF K=136 THEN W=129 :W$=" WHITE NOISE"
710 IF K<>135 THEN GOTO 735
715 INPUT "ENTER PULSE WIDTH (0-100)";PW
720 IF PW<0 OR PW>100 THEN 715
725 PP=PW*40.95 :H=INT(PP/256) :L=PP-256*H
730 POKE 54274,L :POKE 54275,H
735 RETURN

```

MORE ADVANCED SOUND FACILITIES

FILTERING

By passing the sounds obtained through filters we can alter the pitch of the notes produced to increase yet further the possible combinations. There are three filters available (as shown in Figures 19.12 - 19.14). Basically, a frequency is specified and depending on the type of filter involved, frequencies above or below this value are cut off.

LOW PASS FILTER

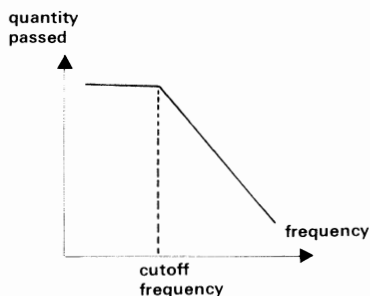


Figure 19.12

BAND PASS FILTER

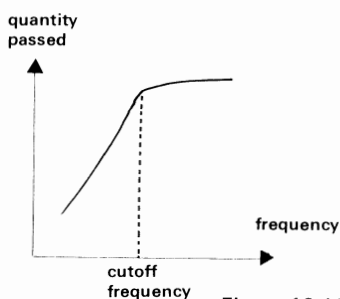
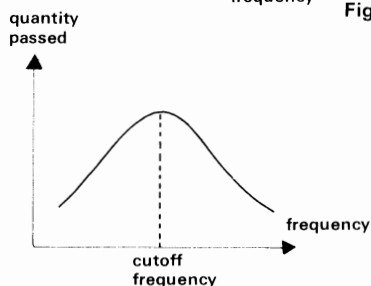


Figure 19.13

HIGH PASS FILTER

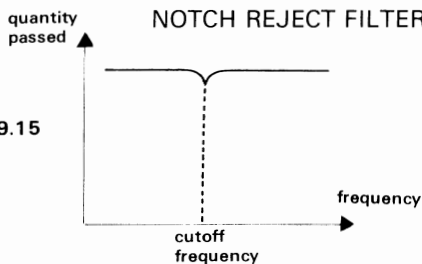
Figure 19.14



By combining a high pass and a low pass filter, a notch reject filter is obtained.

Figure 19.15

NOTCH REJECT FILTER



The filters required are obtained by setting the following bits at address 54296 (D418 (hex)) remember that the bottom four bits at this address contains the volume and so should not be changed unintentionally.

FILTER	BIT
LOW PASS	4
BAND PASS	5
HIGH PASS	6

Figure 19.16

The cut-off frequency is stored in 11 bits; the 3 least significant bits are situated in bits 0-2 at address 54293 (D415 (hex)) and the 8 most significant bits are at address 54294 (D416 (hex)).

VIBRATO

At address 54299 is a random number generator whose values alternate in a fashion dependent on the selected waveform of oscillator 3.

Waveform	Numbers Produced
Sawtooth	Continually incremented from 0 to 255 at a rate proportional to frequency of oscillator 3.
Triangle	Continually incremented from 0 to 255 and then back to 0 again.
Pulse	Oscillates between 0 and 255.
White Noise	Completely random.

Rapid changes about a frequency which causes an effect called vibrato can be obtained by adding the values obtained from this number generator to the frequency values as they are heard. The effects caused will be different for each of the four cases. The output from voice 3 can be eliminated by setting bit 7 at address 54296.

One example below illustrates this by playing a tune called 'Ogonek' with each note being vibrated slightly as it is played.

Program 45 - OGONEK

```

10 REM *****
15 REM *   OGONEK   *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 A=0 : D=8 : S=15 : R=0 : V=15 : DR=6 : W=17 : PRINT "J"
45 POKE 54296,V+128 : POKE 54276,W-1
50 POKE 54278,S*16+R : POKE 54277,A*16+D
55 POKE 54290,17 : POKE 54286,120
60 DIM N(35)
65 FOR J=0 TO 2
70 RESTORE
75 FOR I=0 TO 11
80 READ X : N(J*12+I)=X*(2↑J)
85 NEXT I
90 NEXT J
95 DATA 2145,2273,2408,2551,2703,2864,3034,3215,3406,
      3608,3823,4050
100 READ X
105 IF X<0 THEN POKE 54276,W-1 : END
110 POKE 54276,W
115 FOR T=1 TO DR
120 F=N(X)+PEEK(54299)/2 : REM ADD VIBRATO
125 H=INT(F/256) : L=F-256*H
130 POKE 54272,L : POKE 54273,H
135 NEXT T
140 GOTO 100
1000 DATA 0,4,9,12,16,4,14,2,6,4,14,4,8,1,14,4,11,12,1
      4,4,11,12,9
1005 DATA 0,4,9,12,9,4,0,4,16,17,19,8,17,16,19,8,11,17

```

```

,17,9,14,9,17,9,17,19,21
1010 DATA 9,19,21,16,4,8,11,16,14,11,8,0,12,24,24,24,1
      2,23,21,24,12,16,23,17
1015 DATA 9,14,9,17,9,17,9,1,7,19,21,9,19,21,16,8,12,1
      6,8,12,16,0,8,11,16
1020 DATA 0,4,9,12,16,4,14,2,6,4,14,4,8,1,14,4,11,12,1
      4,4,11,12,9
1025 DATA 12,16,12,9,21
1030 DATA -1

```

CHAPTER 20

BASIC INPUT/OUTPUT

PERIPHERALS

When we refer to the term 'Peripheral' we usually mean some external add-on device such as a printer or disk drive that we can interface to a computer system. With our own Commodore 64 system we can also include such items as the keyboard and screen which, if we require, can be handled in the same fashion as other external devices. However, since these devices are in common use, the designers of the system decided to keep things simple and produced some additional routines such as PRINT, INPUT, etc to handle these devices.

When we want to 'talk' to the Commodore 64's peripherals we do so by communicating with them by means of logical files. The ideas of a logical file is quite common in computing although it can be a difficult concept to grasp initially; fortunately for the beginner, the Commodore 64's system software handles the logical files automatically for the keyboard and screen if PRINT, INPUT, etc are used.

LOGICAL FILES

The best way of visualising these logical files is to imagine a normal office type of file as in Figure 20.1 The Commodore 64 has several BASIC commands available which enable the programmer to put data into the file or to examine its contents. It is also possible to assign a peripheral to a logical file so that these commands are actually sending data to or receiving data from the device.

Up to a maximum of ten logical files may be used simultaneously on the Commodore 64 and each is distinguished by a unique numerical label. This label is limited to the range of 1 to 255; although its choice is up to the programmer we shall see that it is wise to use some value that bears a resemblance to the device or its characteristics in order to keep track of the program logic.

Before we can access a logical file it has to be opened; this seems common sense if we think of our office file. Although the permutations of the statement used to open a logical file are numerous, there are essentially two items that we must always tell the computer; firstly by what label or file number we are going to refer to the logical file and, secondly, information about the peripheral that is to be assigned to the file.

The data about the peripheral can consist of one or two items; these are called primary and secondary addresses.

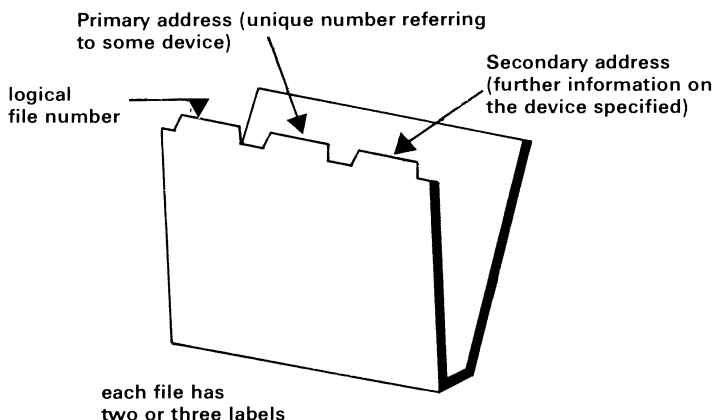


Figure 20.1 Office File Analogy of a Logical File

PRIMARY ADDRESS - DEVICE NUMBERS

If we refer right back to chapter 3 we will recall that at the rear of the Commodore 64 is a DIN socket called the 'Serial Port' which we said was used for interfacing the computer to items such as printers or disk drives.

This connection can, in fact, join several peripherals to the Commodore 64 in a 'daisy chain' type of arrangement with the joining links being known as the serial bus (or IEEE-488 bus).

In order that a computer can send a message down the serial bus to be picked up by just one peripheral, the message must include an identifier for the device it is to be received by. In the Commodore 64 system, the message is sent to a file which itself has been assigned to a single device by means of a unique identifying number called the primary address.

The primary address or device number of each peripheral is 'hard coded' into it at manufacture although, for some peripherals, it may be quite a simple task to amend it. For devices that are connected to the serial port, the device number is restricted to the range 4 to 31.

The standard device numbers of primary addresses for peripherals in the Commodore 64 system are:

Device Number	Peripheral
0	Keyboard
1	Cassette interface
2	Vic Modem - device for translating data into a series of pulses for transmitting through communication devices (e.g. telephones)
3	Display screen
4 (or 5)	Printer
8 (or 10 or 11)	Disk drives

Figure 20.2

For readers who are hardware experts, this leaves plenty of device numbers for their own projects requiring control such as for example, electric thermometers, voltmeters, heating controls, light shows etc.

If a primary address is omitted by the programmer then a value of 1 is assumed by default, thus the Commodore 64 will start communicating with the cassette interface.

Secondary Address

The secondary address is an instruction value sent to a device that utilises a separate processing unit from that used by the central computer system.

Using a separate processing unit enables the peripheral to operate independently from the Commodore 64 and so does not consume the computer's resources. The exact values of secondary addresses depend on the device involved, as we shall see when we study the Commodore peripherals i.e. the cassette interface, printer and disk drives. If a peripheral expects a secondary address to be specified but it is omitted by the programmer then a value of 0 is assumed by default.

OPENING and CLOSING LOGICAL FILES

Having seen what a logical file is used for, let's see how to open such files. Also, since there is a limit to the number of logical files that can be opened at any one time we shall see how to close the logical files when they have been finished with. Unsurprisingly, the BASIC commands for this are OPEN & CLOSE.

Syntax: OPEN <file no.>, [<device>] [,<secondary address>]
[; "<file name>" [,<type>] [,<mode>"]]

It can be seen that the full syntax of the OPEN statement contains more than the file number and the device addresses; the use of the remaining terms will be illustrated later. For the time being it is sufficient to keep things simple.

example

1) 10 OPEN 1,4

This opens up a logical file and assigns it to device 4 (the printer); in this case the file is referred to by the number 1.

Syntax:

CLOSE <file no.>

example

2) 20 CLOSE 1

This CLOSEs the logical file OPENed above.

If we try to access a logical file that had not been opened then the error report 'FILE NOT OPEN' will be given. Similarly if we try to open a logical file that is currently open then the error report 'FILE OPEN' will be given - this error, however, also closes the file involved and so, rather confusingly, it must be re-opened before it can be used.

COMMUNICATING WITH A PERIPHERAL

Once a link has been set up with a peripheral by means of a logical file we can 'talk' to it using the BASIC statements PRINT#, INPUT# and GET# followed by the file number of the link with the logical file concerned.

These statements are much the same as PRINT, INPUT and GET except that they communicate with the screen or keyboard. Using these new commands we can link up any device to the 64.

To send information to the device, PRINT# is used; to receive strings or values from the device, INPUT# is used, and to receive a single byte from the device, GET# is used.

Syntax:

PRINT# <file no,> <data list>

INPUT# <file no,> <variable list>

GET# <file no,> <variable list>

example

To send a message to the printer:

```
10 OPEN 1,4
15 PRINT#1, "MESSAGE"
20 CLOSE 1
```

Where whole program listings are concerned, communication with a peripheral is undertaken by the statements SAVE, LOAD and VERIFY; in these cases the logical files are automatically controlled by the system software but a device number must be specified.

Syntax:

SAVE ("filename")(,<device no>)(,<secondary address>)

LOAD ("filename")(,<device no>)(,<secondary address>)

VERIFY ("filename")(,<device no>)

To transfer a program currently stored in the Commodore 64's memory to a device such as a disk drive or cassette unit, the statement SAVE is used along with a file name of up to 16 characters, the receiving device number and, if required, a secondary address. Program listings are automatically SAVED twice which enables errors on retrieval to be found. A secondary address of 1 tells the computer, when reloading, to put the program at the same place in the memory rather than from the usual location at address 2048, a secondary address of 2 puts an end of tape marker at the end of the program and a secondary address of 3 combines both of these.

To check that a program has been stored correctly, we use the command VERIFY as soon as the program has been SAVED. If a problem has occurred then the error report 'VERIFY ERROR' will be given and the SAVE process should be repeated.

To retrieve a program that has been previously SAVED, LOAD is used, together with the file name of the program and the device number of the peripheral containing the program.

Having communicated with a peripheral it is possible to enquire about the state of the system with the function STATUS which returns an eight bit binary number.

Syntax:

ST [ATUS]

For communications involving the cassette unit and peripherals attached to the serial bus the following bits will be set:

Bit	Output from Tape	Input to Tape	I/O with Serial Bus
0			Write time out
1			Read time out
2	Short block		
3	Long block		
4	Error in output	Mismatch	
5	Check sum error		
6	End of file		End-of-identify
7	End-of-tape		Device not present

Figure 20.3

DATA FILES

As well as programs, data may be stored on disk and cassettes, although there are several limitations with the latter.

There are two ways of storing data, and in each case a collection of related data is called a file. We can have either:

A sequential file:

the data items are stored one after another; to access a single item the file has to be searched through from the beginning until the required item is found.

A random access file:

any data item can be accessed almost equally quickly by specifying some value that references the position of the data in the file.

In any file will be a series of records which contain the data. Records can be sub-divided further to contain several related items; each sub-division is called a "field". An ideal analogy is an office filing cabinet, with each record being represented by a folder in the cabinet and the fields the information represented by the different items of information in the folder.

Thus when a record is accessed from a file, a complete set of usefully related data is obtained.

For example an order record for a car dealer might contain seven fields:

Order No.	Car	Model	Colour	Customer	Price	Date Due

Figure 20.4

Such records could be sorted so that they are in some sort of order, such as alphabetical order of the customers name, or by ascending order number. The field that dictates the order is called the key, and it is up to the programmer to choose a suitable key field.

When dealing with files it is usual to use a 'record pointer' which points to a particular record in the file; the value of the pointer is sometimes called the address of the record. With a random access file it is possible to specify a key and, providing there is a unique key to address relationship, the operating system can calculate the correct address of the record.

SUMMARY

In this chapter we have seen how the Commodore 64 communicates with its peripherals by means of logical files. The important thing to remember is that all files must be OPENed before they can be accessed, CLOSED when finished with and, when OPENed, the file number, the device number (primary address) and optionally an instruction (secondary address) must be specified. To 'talk' to a device, the commands PRINT#,

INPUT# and GET# are used for data files and SAVE, LOAD and VERIFY when program listings are involved. Data is stored in files either sequentially or randomly with disks but, due to the slow speed of a cassette unit, there is no real alternative but to store data sequentially on cassette tape.

We shall now continue and see how to use logical files to communicate with the Commodore peripherals.

CHAPTER 21

PROGRAM STORAGE

By now you should be well aware, probably due to a previous misfortune, that when the power is disconnected from the Commodore 64, the stored program is lost for ever. There is, however, the facility for storing the programs on magnetic media so that they can be retrieved for use at a later date - the process is just like recording a record, which can then be played whenever you want. The Commodore cassette unit or disk drives are required for program storage.

THE C2N DATASSETTE UNIT

The Commodore C2N datassette unit is a device for storing programs and/or data onto normal domestic cassette tape. The unit has been designed for use with all the Commodore computers; it is interfaced to the Commodore 64 by plugging the connector from the unit into the smallest of the expansion ports found at the rear of the computer. The connector will only fit one way round, so do not force it. It is also wise to keep the unit away from the television since magnetic fields can affect the unit and the cassette tape.

It is also very important to make sure that the tape heads of the unit are kept clean at all times. When cassette tape passes over the tape heads, it leaves a small deposit; after a time this will mount up to such an extent that the recordings will be muffled and the Commodore 64 will not be able to decipher the signals. It is also worth using high quality tapes since some poor quality ones will not give a large enough signal for the system to interpret. If the signal is too low the Commodore 64 will search indefinitely for a program or data file. Another fault with poor quality tapes is that they shed their magnet particles more easily. This can cause a more rapid build up of depositions or the tape may lose a particle containing vital information.

STORING PROGRAMS ON CASSETTE TAPE

To save a program on cassette tape:

- 1) Cue up the part of the tape to be used.
- 2) Type `SAVE "name of program"` (RETURN) The prompt 'PRESS RECORD & PLAY ON TAPE' will appear on the screen.
- 3) Press the buttons on the cassette unit and a red L.E.D. on the unit will light up. The screen will clear while the program is transferred to cassette tape. When the action is complete the screen will reappear with the message 'READY' and the motor on the cassette unit will have automatically stopped.

Programs are automatically stored twice in order to detect errors on playback.

Note that for the cassette unit, the device number need not be specified as a primary address of 1 is assumed by default. We can save the program with the name omitted but this would mean that the program could not be recalled by name which makes locating it on the tape a little tricky.

It is possible to ensure that a program has been saved properly before switching off the Commodore 64 by the command `VERIFY`. To do so:

- 1) Cue up the section of tape to a point before where the program commences.
- 2) Type `VERIFY "name of program"` (RETURN) The prompt 'PRESS PLAY ON TAPE' will appear on the screen.
- 3) Press the play button on the cassette unit and the screen will clear as before.
- 4) When the program has been located it will prompt with 'FOUND name of program' on the screen.

On pressing the space bar, the program will be checked against the program stored in the computer's memory. If any error is found the error report '? VERIFY ERROR' will be given. If everything is correct the report 'OK' is returned; as before, the cassette unit's motor will automatically stop. If nothing seems to happen press **RUN STOP** to break control and then reSAVE the program.

To load a program from cassette tape, the same procedure as VERIFY is used except that the command LOAD is used instead. LOAD will delete any old program that may exist before the new program is retrieved. When a name is specified, the computer will search for the first program on the tape with the specified name; if no name is given then the first program on the tape will be loaded.

A quick way of obtaining the command LOAD is to press



DISK DRIVES

The disk drive is now becoming the most common item of hardware for storing information used by a microcomputer. The advantages include high reliability and fast access times. Unfortunately their high cost has led them to be regarded by some hobbyists as expensive luxuries.

In the days of the first computers, information was usually stored on either punched paper tape or cards; this was not ideal, since it was bulky and not reusable. The next medium involved magnetic tape but, as we have seen with our own miniature cassette tape, the access times are very high if the data required is wound a long way from the required position. As the efficiency of computers developed a new medium was soon required that had very fast access times and the disk drive evolved with a "disk pack" as standard. A disk pack contains several magnetic disks stacked on the same spindle with a space between each disk where separate read/write heads could be slotted. The disk pack rotates several thousand times a minute. Because all of the parts of the disk were exposed, the access times were very short; using such disks avoided waiting for the required section of the storage medium to be cued up.

While these disk packs are still in common use today, for microcomputers, the most important development has been the 'floppy disk' which is a single magnetic disk.

There are two common sizes of floppy disk available - the 8 inch and the 5 1/4 inch. The smaller size, often used in microcomputer applications is commonly called the 'mini floppy' or 'diskette'; there are signs that a new 3 1/2 inch 'micro floppy' could become a new standard. The magnetic

disk of the floppy is protected within a stiff square envelope with just a few small openings. The disk drive makes contact with the centre of the disk through one of these openings and rotates the disk within its envelope which remains stationary; the read/write heads then make contact while the magnetic surface of the disk through another opening.

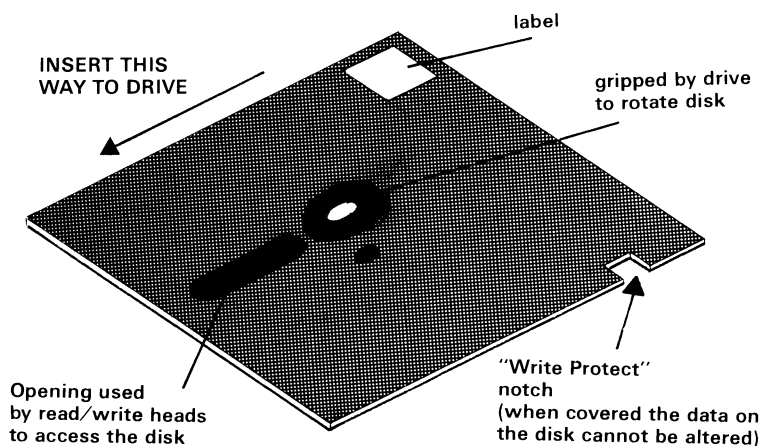


Figure 21.1 The Floppy Disk

Disk drives are controlled by software called the disk operating system (DOS) which is usually included within the disk drive and is processed by a separate microprocessor. This separate 'intelligence' enables the operations required by the user to be kept simple with most of the work done by the DOS; this reduces the processing overheads on the computer's central microprocessor. The DOS is interfaced to the disk drive itself by means of the "disk controller" which is a hardware unit inside the disk drive.

One of the important things that the DOS knows is that the data is organised on the disk in concentric tracks. With the Commodore disk drive, there are 35 tracks available on one side only of the disk. Since each track can store just under 5K bytes of data the total disk can store nearly 175K; the exact value for each track depends on its physical size, which in turn depends on its distance from the centre of the disk. A disk with this type of storage is said to be Single Density, Single Sided.

Instead of storing all the information continuously along one track it is

more efficient to divide each track into sections as shown in Figure 21.2. Such boundaries are put onto the disk by the DOS with a formatting routine. Data is transferred to and from the disk in units of one section at a time.

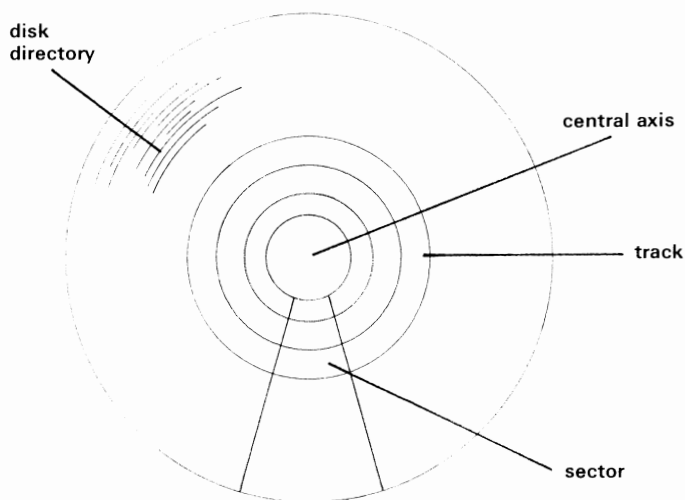


Figure 21.2 The Floppy Disk Format

Another one of the functions of the DOS is to keep track on the identity of the floppy and the files that it contains. To do so, approximately 1K is reserved on the disk to store the disk directory which contains the name of the disk and a list of the files that are present with their size and position on the disk.

Since floppy disks are quite delicate it is important that you handle them with care. Keep them out of range of magnetic fields such as those emitted from a TV and keep them away from direct sunlight. When not in use you should keep the exposed sections of the disk away from dirt and sticky fingers by returning the floppy disk back to its storage envelope.

THE VIC-1541 SINGLE DISK DRIVE

The Commodore VIC-1541 is a disk drive that can be interfaced to your Commodore 64. It should be connected to the serial port with the cable provided and if any other peripherals are required, the disk drive should be

first in the 'daisy chain'. Note that all power should be disconnected from the equipment before connecting.

When power is initially supplied to the disk drive it is very important that no disk is in the disk drive or the stored data will be corrupted.

There are two L.E.D.s on the front of the disk drive; the green L.E.D. signals that the power is on and the red L.E.D. signals that the disk drive is either not ready or that an error has occurred.

To insert a floppy into the drive, open the door of the drive and insert it with the label on the top and the write protect notch to the left, see Figure 2.1.1. Once inserted, close the door. The floppy disk will spring out once the door is opened.

STORING PROGRAMS ON FLOPPY DISK

We shall now see how to examine the disk directory and to store or retrieve program files on a floppy disk. For simplicity, we'll assume that we have just one drive. The first thing that we must do is open a logical file to set up a channel so that we can communicate with the disk drive.

Whenever a floppy disk has been inserted into the disk drive it must be initialised. This command aligns the read/write heads with the disk and transfers a copy of the disk directory to the DOS. If the disk is new it has to be formatted; this sets up the sectors and directory onto the disk. Having done this we can then transfer programs to and from the floppy disk.

OPENING A COMMAND CHANNEL

To communicate with the disk drive a 'command channel' has to be set up by opening a logical file assigned to the disk drive (device 8) specifying a secondary address of 15. Our reference number for the logical file may be changed to any number between 1 and 255.

example OPEN 15, 8, 15

INITIALISE DISK

The disk drive is initialised by sending the disk drive command 'INITIALISE dn' where dn is the drive number. In our case where we only have one drive the number is 0 but it may be omitted.

example PRINT# 15, "INITIALISE 0"

The command 'INITIALISE' may be abbreviated to 'I'

i.e. PRINT# 15, "I"

If the command channel had not yet been opened, the initialise command can be included in the OPEN statement.

example OPEN 15, 8, 15, "I"

Having done this, the read/write heads will be properly aligned and the DOS will know which files are on the disk. This command must be used every time a fresh disk is inserted into the drive.

LOADING THE DISK DIRECTORY

We can examine which files are on a disk by loading the disk directory and then displaying it. To do so we use the command

LOAD "\$dn", 8 or LOAD "\$",8

but you should remember that this will erase any program that is present in the computer's memory. The directory may then be inspected by LIST. The directory displayed will tell us the name of the disk, the disk id number, what version of DOS it was created with, how much space is still available and the file names present with their type and size.

The "disk id" is a number given to the disk when it is being formatted. It is advisable to give all your disks different id's since, if the id on the disk in the disk drive does not match that in the DOS, the computer rejects it. This avoids the possibility of corrupted disks by forgetting to initialise a fresh disk. In such a situation the disk drive would be using the previous disk directory on the new disk and so would be writing to the wrong locations.

THE DISK MAINTENANCE COMMANDS

NEW - Formatting a Disk

Whenever a floppy disk is brand new and unused, or an ld disk is to be re-used, the disk has to be formatted; this process sets up the sectors on the disk by imprinting magnetic signals on it. The required disk command for executing the formatting routine is 'NEW', not to be confused with the BASIC command NEW:

i.e. PRINT# 15, "NEW dn: disk name, disk id", 8

The disk name contains a maximum of 16 characters and the disk id is a two character alphanumeric value. The disk drive no., in our case 0, may be omitted since we are using a single drive and so the command 'NEW' can be abbreviated to 'N':

PRINT# 15, "N: disk name, disk id"

SAVE, LOAD and VERIFY

Before a diskette can be used, it must be formatted and initialised - read the manual or the notes in this chapter. This section presumes that you have initialised a disk. A program may now be saved using the command SAVE for device number 8:

SAVE "dn: program name", 8

The drive number, in our case 0 (i.e. just one drive is being used) may be omitted. To overwrite an existing file with the same name, the character @ is inserted at the start of the string

SAVE " @ dn: program name", 8

As with the cassette unit, we can verify that a program has been saved correctly using VERIFY:

VERIFY "dn: program name", 8

To verify a program that has just been saved, the following may be used:

VERIFY "***", 8

where "*" is a wild-card character meaning "the last program saved" - you can use the actual name if you prefer.

To load a selected program from a disk the command LOAD is used:

LOAD "dn: program name"

As before, the drive number 0 may be omitted.

If you use LOAD "*", the "*" wild-card means "the first program on the disk". If you have only one program on the disk, this can be handy.

Other disk commands include:

RENAME

A file name may be changed by the command 'RENAME' (or 'R'):

PRINT# 15, "Rdn: filename = old filename"

The drive number, in our case 0, may be omitted.

SCRATCH

Any unwanted file(s) may be deleted from the floppy disk and its directory by the command 'SCRATCH' (or 'S'):

PRINT# 15, "Sdn: filename, dn: filename,....., dn: filename"

The drive numbers, in our case all 0, may be omitted.

COPY

Duplicate copies of a file can be created by the command 'COPY' (or 'C')

PRINT# 15, "C dn: filename = dn: existing filename, ...

The drive numbers, in our case both 0, may be omitted.

ERRORS WITH DISK HANDLING

Any errors that occur with disk handling will be signalled by the red L.E.D. flashing on the front of the disk drive. To reset the device and obtain an

error message, the following routine should be called as a subroutine:

Program 46 - DISK HANDLING ERROR ROUTINE

```
10000 REM *****
10005 REM * DISK ERROR SUBROUTINE *
10010 REM *****
10015 REM
10020 INPUT#15,EN,EM$,ET,ES
10025 IF EN<20 THEN RETURN
10030 PRINT "***** DISK ERROR HAS OCCURED ****"
10035 PRINT "ERROR NUMBER      ";EN
10040 PRINT "ERROR MESSAGE      ";EM$
10045 PRINT "TRACK NUMBER       ";ET
10050 PRINT "SECTOR NUMBER      ";ES
10055 STOP
```

where

EN is the error number
EM\$ is the error message
ET is the track)
ES is the sector) where the error occurred.

The error number references a detailed description of the error in the VIC-1541 user's manual.

It is a good idea to check the error channel after every disk operation to check that everything has gone OK. However doing this after every PRINT# although wise can be very time consuming if vast quantities of data are involved and so some programmers often choose to omit this step. If your data is valuable or the disc is nearly full then this precaution should always be undertaken.

FILENAME MATCHING

The referencing of file names can often be made easier with the use of the wild-card characters '*' and '?'.

A '*' at the end of a file name means the remaining characters in the name

are insignificant

e.g. "DA*" could refer to any of:

"DA"
"DATA"
"DATA1"
"DATA 1 2 3 4"

A '?' in a file name means that the particular character is insignificant.

e.g. "C?T" could refer to any of:

"CAT"
"COT"
"CUT"

Care should be taken when using this technique to delete files from a disk as it is easy to accidentally erase the wrong file!

CHAPTER 22

THE VIC-1515 GRAPHIC PRINTER

The VIC-1515 graphic printer has been designed to enable you to obtain 'hard' copies of the Commodore 64's output. It is capable of printing all the upper and lower case alphabetic characters, the numeric digits and the graphic characters from the Commodore character set. In addition we can define some of our own characters and specify format instructions. Like some of the other Commodore peripherals, it has its own microprocessor and so the required processing is not undertaken by the Commodore 64's central microprocessor. It is interfaced to the Commodore 64's serial bus along with any other peripherals by linking them up in a 'daisy chain' fashion with the cables provided.

Just like any other peripheral, communication with the printer is via logical files; having opened a communication link, the Commodore 64 'talks' to the printer using PRINT#. The printer has a device number of 4 although this can be altered to 5 by a slide switch located at the rear (the same switch can be set to 'T' which gives a diagnostic printout for testing). The file may be opened with a secondary address of either 0 for the upper case character set or 7 for the lower case character set.

As a simple example, to send "COMMODORE 64" to the printer:

```
10 OPEN 4,4
20 PRINT# 4, "COMMODORE 64"
30 CLOSE 4
```

A secondary address of 0 is assumed by default and so the printing is done in upper case.

THE CMD COMMAND

The CMD command can be used to replace the primary output device, i.e. the screen, with any required device; the device involved is then said to be 'listening'.

Syntax

CMD <filename> (,"<string>")

Any device may be used providing it has been linked to the Commodore 64 via a logical file. Any string specified in the CMD command is also sent to the device. To redirect output back to the primary device a PRINT# command is used to send what would appear to be nothing, (i.e. a blank line) - the device is now said to be unlistening! Note that if the logical file that was created with the communication link is closed the device does not automatically become unlistening. If an error occurs with the communication to the peripheral all the logical files are closed but the device is not unlistened; thus, if an error occurs, the communication links should be recreated and a PRINT# message sent to unlisten the device.

LISTING A PROGRAM

To obtain a program listing at the printer the following instructions should be entered in command mode:

```
OPEN 4,4  
CMD 4 ←————— sends output to printer  
LIST  
PRINT# 4 ←————— unlistens printer  
CLOSE 4
```

MORE ADVANCED CONTROL OF THE VIC-1515 PRINTER

In addition to the standard characters of the Commodore 64 character set there are enhancements that we can make to the printed output by sending selected control codes with the PRINT# statement. The control codes are listed in appendix C.

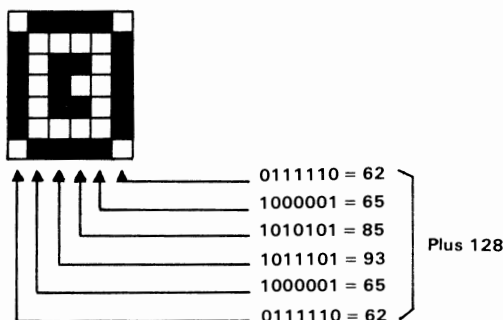
DOUBLE WIDTH MODE

By sending the character CHR\$(14) we can enter 'double width print mode' in which the characters are built on a grid of dots that is twice its usual width. The printer remains in this mode until the character CHR\$(15) is sent which returns the printer back to 'standard print mode'.

GRAPHIC MODE

By sending the character CHR\$(8) we can enter 'graphic mode' in which we can define any character that can be accommodated on a 7 x 6 matrix grid. To define such a character we need to specify 6 numbers which indicate which pixel dots are set, in a very similar fashion as to the way that we defined characters in standard display mode.

Consider for example the copyright character ©



Having designed our character we have to calculate the six numbers that represent the pixels in each column. Taking each column in turn, the pixel sequence is written as a binary number with a "1" representing each set pixel, a "0" representing a blank pixel, and the bottom pixel being the most significant bit. The six characters can then be evaluated as a decimal number to which the value 128 has to be added i.e. in the case of © the pixel data is

190, 193, 221, 213, 193, 190

The character is then sent to the printer by the following sequence of characters:

CHR\$(8), { six characters whose codes are
the pixel code numbers } ,CHR\$(15)

CHR\$(15) is used to return the printer back to standard print mode. By sending CHR\$(26) and CHR\$(N) before the pixel data, the character will be repeated N times.

CURSOR UP/DOWN

We may switch between the upper and lower case character sets by sending the characters:

CHR\$(145) for upper case & CHR\$(17) for lower case

REVERSE CHARACTER MODE

Similarly we may switch between reverse colour mode (i.e. white on black) and normal colour mode (i.e. black on white) by sending the characters:

CHR\$(18) for reverse colour & CHR\$(146) for normal colour

CONTROLLED PRINTING

By sending appropriate characters we can specify the character or pixel dot position at which an item is to be printed. Character positions range from 0 to 79 and pixel dot positions range from 0 to 479.

To specify a character position we must send the character CHR\$(16) followed by the position as a two digit number in quotes.

e.g. to print string variable A\$ in column 27:

PRINT# 4, CHR\$(16) + "27"; A\$

Similarly, to specify a pixel dot position we must send characters CHR\$(27) + CHR\$(16) followed by the two characters CHR\$(H) + CHR\$(L) where the dot position is equal to $H * 256 + L$.

e.g. to print A\$ in dot column 300

PRINT# 4, CHR\$(27) + CHR\$(16) + CHR\$(1) + CHR\$(44); A\$

These advanced printing commands are illustrated in Program 47 which plots a histogram for the values that are stored in the DATA statements.

Program 47 - HISTOGRAM

```

10 REM *****
15 REM *   HISTOGRAM   *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 OPEN 4,4
45 PRINT#4,CHR$(14);"SALES FIGURES";CHR$(15)
50 PRINT#4,SPC(8);CHR$(18);"1983";CHR$(15)
55 PRINT#4,CHR$(17);"COMPUTER SALES (£1000'S)";CHR$(
    15);CHR$(10)
60 Q$=CHR$(8)+CHR$(27)+CHR$(16)+CHR$(0)+CHR$(60)+CHR$(26)
65 READ N$,V :GET K$
70 IF N$="" OR K$=" " THEN PRINT#4 :CLOSE 4 :END
75 P$=N$+Q$+CHR$(V)+CHR$(255)+";"+CHR$(15)
80 PRINT#4,P$;V
85 GOTO 65
100 DATA JANUARY ,100
105 DATA FEBRUARY ,37
110 DATA MARCH ,243
115 DATA APRIL ,134
120 DATA MAY ,187
125 DATA JUNE ,145
130 DATA JULY ,67
135 DATA AUGUST ,145
140 DATA SEPTEMBER,110
145 DATA OCTOBER ,76
150 DATA NOVEMBER ,99
155 DATA DECEMBER ,69
160 DATA "" ,0

```


SALES FIGURES

1998

computer sales (£1000's)

JANUARY	100
FEBRUARY	37
MARCH	243
APRIL	134
MAY	187
JUNE	145
JULY	67
AUGUST	145
SEPTEMBER	110
OCTOBER	76
NOVEMBER	99
DECEMBER	69

THE COPY STATEMENT

Some versions of BASIC on other systems have the command COPY which duplicates a hard copy of whatever is displayed on the screen. Program 48 can be called as a subroutine to simulate this command. The main difficulty in writing such a routine arises because the PEEK/POKE codes used to store the screen in the screen memory do not correspond with the characters that are required to be sent to the printer - thus a conversion has to be done.

Program 48 - COPY SUBROUTINE

```
20000 REM *****
20005 REM * COPY SUBROUTINE *
20010 REM *****
20015 REM
20020 OPEN 4,4 :PRINT CHR$(142);CHR$(8)
20025 FOR I=1024 TO 2023 STEP 40
20030 L$=""
20035 FOR J=0 TO 39
20040 P=I+J :C=PEEK(P) :R=0 :C$=CHR$(160)
20045 IF C>127 THEN R=1 :C=C-128
20050 IF C>=0 AND C<=31 THEN C$=CHR$(C+64)
20055 IF C>=32 AND C<=63 THEN C$=CHR$(C)
20060 IF C>=64 AND C<=95 THEN C$=CHR$(C+32)
20065 IF C>=96 AND C<=127 THEN C$=CHR$(C+64)
20070 IF R=0 THEN L$=L$+C$
20075 IF R=1 THEN L$=L$+CHR$(18)+C$+CHR$(146)
20080 NEXT J
20085 PRINT#4,SPC(15);L$
20090 GET K$ :IF K$=" " THEN 20100
20095 NEXT I
20100 PRINT#4 :CLOSE 4 :PRINT CHR$(9)
20105 RETURN
```

CHAPTER 23

DATA STORAGE ON CASSETTE

Our first look at using data files will involve the C2N cassette unit which is a simple device for storing information; unfortunately it does have some severe limitations. We have already seen how to communicate with the unit via the serial bus and that the device number is 1; in most cases this number may be omitted as the Commodore 64 assumes 1 by default. It was also pointed out that since a section of tape containing data cannot be accessed quickly if the tape is wound a long way from that particular section, all data stored on the tape must be organised sequentially. In a sequential file, the records are stored one after another and usually grouped into blocks. The blocks, which are small enough to be held in main memory, are stored on the tape with small intervals between them which enables the cassette unit to have time to stop and start the tape, see Figure 22.1.

The Commodore 64 has a section of RAM located from address 829 to 1019 called the cassette buffer. When data is loaded from the tape, one block is loaded at a time and stored into the buffer and the data can then be accessed from it. Similarly, when data is being stored on tape it is stored in the buffer and sent to the cassette unit either when the buffer becomes full, in which case it can be refilled, or when the logical file assigned to the cassette unit is closed. This avoids the tape continually being started and stopped for each data item, and so enables the data to be more closely packed onto the tape.

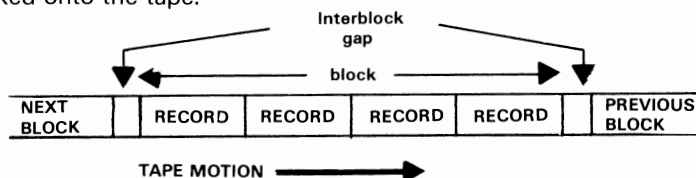


Figure 22.1 Sequential Storage on Tape

In large commercial installations, the slowness of sequential files on magnetic tape means that most amendments are made to large data files by using "transaction files". Consider, for example, a bank with all the current accounts of its customers stored on magnetic tape. During the day the value of some of the accounts will be altered as cheques are received or cashed. It would be very inefficient, if not impossible, to change the values of the accounts as each alteration occurs. Instead, a 'transaction file' is built up during the day which contains each amendment to an account as it occurs. At the end of the day, the transaction file is sorted using some unique key (probably the account number) until the records are in the same order as the records appear on the main "master" file. The two tapes can then be matched to update the accounts and produce a new "master file".

CASSETTE DATA FILES

Cassette data files are created by the following operations:

Firstly we must open a logical file to communicate with device 1, the cassette unit. In this case the general syntax of the statement is

OPEN file number, device number, secondary address, "filename"

When the logical file is opened, the secondary address should be specified and this determines the mode in which the cassette unit will function; in this case it should take a value of 0, 1 or 2. An assigned value of 0 corresponds to a 'READ' command, and a value of 1 or 2 corresponds to a 'WRITE' command - the first (1) will write with an End-of-File marker (EOF) and the second (2) with an End-of-Tape marker (EOT) when the file is closed. As usual, an unspecified secondary address will be, by default, assumed as 0. If no filename is specified then the operations will be undertaken on the first file encountered on the tape.

To output data to this file the PRINT# command is used with the corresponding file number specified. PRINT# works in the same fashion as PRINT except that the data is output as a block of bytes, that will eventually be sent to the cassette unit, rather than on the screen. For example, just as a comma between two print items causes the second item to be placed in the next print zone, a comma between two items in a PRINT# statement causes the second item to be placed at the first position in the next block of ten characters.

Likewise INPUT# and GET# are used to input data; GET# is used when the input data is limited to one byte at a time. More care has to be taken

when using INPUT# which enable several variables to be input. When several variables are input using INPUT they are separated by a comma and accepted by pressing RETURN. Similarly when data is set up on a tape data file that is going to have the items accessed using INPUT#, commas and the RETURN character (CHR\$(13)) are again included.

Finally, all files must be closed: if this is not done, some data could be left in the cassette buffer and not get written to the tape.

To demonstrate the file management commands that we have looked at, some examples follow.

WRITING A DATA FILE

The first example sends a list of 300 numbers to the tape; the file is opened without a filename and with a secondary address of 1 so that an EOF marker is written when the file is closed. In order that we can identify all the numbers separately, rather than one large number 12345...etc..., a carriage return character will be sent after each number. Also, at the end of the PRINT# statement is a semicolon which suppresses the carriage return and linefeed characters that are present. It is very important that when more than one variable is written using a single PRINT# statement, that separators between the variables are also written to the tape.

example 1

```
10 REM *****
15 REM * CASSETTE DATA FILE *
20 REM *****
25 REM
30 OPEN 1,1,1
35 FOR J=1 TO 300
40 PRINT#1,J;CHR$(13);
45 NEXT J
50 CLOSE 1
```

Data items may also be sent as strings, in the next example 10 strings of characters are stored on tape. Again a secondary address of 1 is given to write a EOF marker when the file is closed.

example 2

```
10 REM *****
15 REM * CASSETTE DATA FILE *
20 REM *****
25 REM
30 OPEN 1,1,1
35 READ N$
40 IF N$="" THEN CLOSE 1 :END
45 PRINT#1,N$;CHR$(13);
50 GOTO 35
55 DATA THIS,IS,THE,VOICE,OF,THE,MYSTERONS
60 DATA ""
```

The final write example sets up a file to include records containing names and telephone numbers which are entered by the normal INPUT statement. We shall later see a program that reads the file and searches for a specified name.

Program 49 - TELEPHONE PAD (PART 1)

```
10 REM *****
15 REM * TELEPHONE PAD *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 OPEN 1,1,1,"TEL PAD"
45 PRINT "TEL PAD "
50 PRINT "ENTER NAMES AND TELEPHONE NUMBERS "
55 PRINT "ENTER '*' STRINGS TO TERMINATE "
60 INPUT "ENTER NAME ";N$
65 INPUT "ENTER NUMBER ";T$
70 IF N$="" OR T$="" THEN CLOSE 1 :END
75 PRINT#1,N$;",";T$;CHR$(13);
80 GOTO 45
```

READING A DATA FILE

Reading a cassette data file is similar to the technique of creating one except that INPUT# is used with a logical file that is opened with a secondary address of 0. The next three programs read the data files we have just created. Remember from chapter 20 that a EOF is found when the function STATUS returns a value of 64.

example 1

```
10 REM *****
15 REM * CASSETTE DATA FILE *
20 REM *****
25 REM
30 OPEN 3,1,0
35 INPUT#3,N
40 PRINT N,
45 IF STATUS=64 THEN CLOSE 3 :END
50 GOTO 35
```

example 2

```
10 REM *****
15 REM * CASSETTE DATA FILE *
20 REM *****
25 REM
30 OPEN 3,1,0
35 INPUT#3,N$
40 PRINT N$;" ";
45 IF STATUS=64 THEN CLOSE 3 :END
50 GOTO 35
```

Program 49 - TELEPHONE PAD (PART II)

```
10 REM *****
15 REM * TELEPHONE PAD *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
```

```

35 REM
40 OPEN 3,1,0
45 PRINT "TELEPHONE PAD"
50 INPUT "ENTER NAME REQUIRED ";R$
55 INPUT#3,N$,T$
60 IF N$=R$ THEN PRINT "TELEPHONE NUMBER ";T$ :GOTO
   75
65 IF ST=64 THEN PRINT "RECORD NOT PRESENT " :GOTO
   75
70 GOTO 55
75 CLOSE 3

```


CHAPTER 24

DATA STORAGE ON DISK

When storing data on floppy disk, we have the choice of organising the data in either sequential or random access files. When sequential files are used, the file has to be read through from the beginning until the required record is found; in random access files any record can be accessed almost equally quickly. Unfortunately, random access files generally occupy more memory, thus the programmer has the choice of either minimising access speeds or of maximising memory efficiency. Usually, if the file requires less than 30K of data, sequential files are chosen since the access times are still within reasonable limits.

SEQUENTIAL DATA FILES

To utilise sequential files we need to know how to

- (1) Set up a file
- (2) List the file
- (3) Access records in the file
- (4) Delete records from the file
- (5) Amend records in the file
- (6) Add records to the file

Each of these will be explained and then all six will be illustrated in a program.

As usual, all communication is done via logical files which must first be opened to communicate with device 8 - the disk drive. In this case the general syntax of the statement is:

OPEN file number, device number, secondary address

“drive number: file, type, mode”

When the logical file is opened, the secondary address should specify a

communication link, called a 'channel', in the range 2 to 14. 15 is reserved for the command channel and 0 and 1 have specific uses with LOAD and SAVE. In our case, the disk drive number 0 may be omitted, the file type is 'SEQ' (or 'S') and the mode is either 'READ' (or 'R') or 'WRITE' (or 'W').

If a file is required to be opened for both READING and WRITING then two communication channels must be created.

SETTING UP A SEQUENTIAL DATA FILE

Sequential disk files may be created by the following operations:

Open a logical file to write to the disk.

example OPEN 2, 8, 2, "DATA, S, W"

As with sequential files on cassette tape, to output data to this file the PRINT# command is used with the corresponding file number specified.

example PRINT# 2, DATA\$

Finally the file must be closed which also puts an EOF marker at the end of the file. If the file is not closed the data cannot be read back from the file.

example CLOSE 2

LISTING A SEQUENTIAL FILE

Listing a file is simple; you just open a 'READ' communication link to the disk and then INPUT# and print each item.

example OPEN 3, 8, 3, "DATA, S, R"

and use INPUT# 3, DATA \$: PRINT DATA \$

SEARCHING FOR A RECORD

To search for a record in a file each record from the beginning of the file is read until the required record is found; this usually involves checking one or more fields in each record until the required record has been found. Careful thought may be needed as the value in a field may not be unique to one record. For example, do you want the first record or all of the records that have the specified values? If the EOF marker is reached then the

record is not present in the file and so some suitable error message should be given.

DELETING A RECORD

This involves reading each record in the file from the beginning to the EOF marker and, with the exception of the unwanted record, each record is written to a temporary file. Each record is tested, using some key (a particular field) as previously described. When the temporary file has been closed it can be renamed with the original file name.

AMENDING A RECORD

This is the same as deleting a record except that the record is updated and also written to the temporary file.

ADDING A RECORD

Again, a file is written to a temporary file and the new record is added onto the end before the EOF marker is written and the file renamed. In some cases they will be required to be sorted with the key in some order. In this case the file should be written to the temporary file until it reaches the position where the new record should be inserted. The new record can then be written to the temporary file and then the rest of the file after it.

When reading a sequential data file it is important that the program can recognise when the end of the file has been reached. This can be done in several ways:

- 1) Write the file with a fixed number of records so that a known number of records have to be read back.
- 2) Store the number of records in the file in the first record of the file. The file can be read by reading the first record and then the number of records specified.
- 3) Use the function STATUS which takes a value of 64 on reaching the EOF marker, which is written to the file when it is closed. See Figure 20.3.
- 4) Write your own record at the end of the file which your program can recognise as the terminating record.

Program 50 which functions as an address book demonstrates how a

sequential file can be created, and how records can be listed, amended, deleted and added. The fourth option above for recognising the end of file has been used as it is the clearest for the reader to comprehend.

Program 50 - ADDRESS BOOK

```

5 REM *****
10 REM * ADDRESS BOOK *
15 REM * SEQUENTIAL FILE *
20 REM *=====*
25 REM * MRH 1/7/1983 *
30 REM *****
35 REM
40 OPEN 15,8,15,"I" :GOSUB 10000
45 POKE 53280,14
50 PRINT "***** ADDRESS BOOK "
55 PRINT "***** PRESS: *****"
60 PRINT " F1 .. ACCESS RECORD"
65 PRINT SPC(20);" F2 .. CREATE FILE  "
70 PRINT " F3 .. AMEND RECORD"
75 PRINT SPC(20);" F4 .. LIST FILE  "
80 PRINT " F5 .. ADD RECORD  "
85 PRINT SPC(20);" F6 .. COMMAND MODE "
90 PRINT " F7 .. DELETE RECORD"
95 PRINT SPC(20);" F8 .. SCRATCH FILE "
100 GET K$ :IF K$="" THEN 100
105 K=ASC(K$) :IF K<133 OR K>140 THEN 100
110 ON K-132 GOSUB 1000,3000,5000,7000,2000,4000,
    6000,8000
115 GOTO 45

1000 REM ** ACCESS RECORD **
1005 POKE 53280,7
1010 OPEN 3,8,3,"DATA,SEQ,READ" :GOSUB 10000
1015 PRINT "***** ADDRESS BOOK "
1020 PRINT "***** ACCESS RECORD "
1025 INPUT "***** ENTER NAME ";IN$

```

```

1030 INPUT#3,N$,A1$,A2$,A3$,A4$,A5$,T$ :GOSUB 10000
1035 IF N$="E-O-F" THEN PRINT "END OF FILE " :
      GOTO 1090
1040 IF IN$CON$ THEN GOTO 1030
1045 PRINT " ";N$
1050 PRINT " ";A1$
1055 PRINT " ";A2$
1060 PRINT " ";A3$
1065 PRINT " ";A4$
1070 PRINT " ";A5$
1075 PRINT " ";T$
1080 GET K$ :IF K$="" THEN 1080
1085 GOTO 1030
1090 CLOSE 3 :GOSUB 10000
1095 PRINT "EXIT (Y/N) ?"
1100 GET K$ :IF K$="" THEN 1100
1105 IF K$="N" THEN 1010
1110 RETURN

```

```

2000 REM ** CREATE DATA FILE **
2005 POKE 53280,7
2010 PRINT "ADDRESS BOOK "
2015 PRINT "DATA FILE BEING CREATED"
2020 OPEN 4,8,4,"DATA,SEQ,WRITE" :GOSUB 10000
2025 N$="E-O-F" :GOSUB 9200
2030 CLOSE 4
2035 RETURN

```

```

3000 REM ** AMEND RECORD **
3005 POKE 53280,7
3010 OPEN 4,8,4,"DATATEMP,SEQ,WRITE" :GOSUB 10000
3015 OPEN 3,8,3,"DATA,SEQ,READ" :GOSUB 10000
3020 PRINT "ADDRESS BOOK "

```

```

3025 PRINT "ADD AMEND RECORD "
3030 INPUT "ADD ENTER NAME ";NN$
3035 INPUT#3,N$,A1$,A2$,A3$,A4$,A5$,T$ :GOSUB 10000
3040 IF N$="E-O-F" THEN 3130
3045 IF NN$<>N$ THEN GOSUB 9200 :GOTO 3035
3050 PRINT "ADD NAME      ";N$
3055 PRINT " ADDRESS      ";A1$
3060 PRINT "              ";A2$
3065 PRINT "              ";A3$
3070 PRINT "              ";A4$
3075 PRINT "              ";A5$
3080 PRINT " TELEPHONE ";T$
3085 PRINT
3090 INPUT "ADD NAME      ";IN$
3095 INPUT " ADDRESS      ";I1$
3100 INPUT "              ";I2$
3105 INPUT "              ";I3$
3110 INPUT "              ";I4$
3115 INPUT "              ";I5$
3120 INPUT " TELEPHONE ";IT$
3125 GOSUB 9100 : GOTO 3035
3130 PRINT"ADD END OF FILE "
3135 N$="E-O-F" :GOSUB 9200
3140 CLOSE 3 :GOSUB 10000 :CLOSE 4 :GOSUB 10000
3145 PRINT#15,"S:DATA" :GOSUB 10000
3150 PRINT#15,"R:DATA=DATATEMP" :GOSUB 10000
3155 PRINT#15,"S:DATATEMP" :GOSUB 10000
3160 RETURN

```

```

4000 REM  ** LIST DATA FILE **
4005 POKE 53280,7
4010 OPEN 3,8,3,"DATA,SEQ,READ" :GOSUB 10000
4015 PRINT "ADD ADDRESS BOOK "
4020 PRINT "ADD PRESS SPACE BAR FOR EACH RECORD "
4025 GET K$ :IF K$="" THEN 4025
4030 INPUT#3,N$,A1$,A2$,A3$,A4$,A5$,T$ :GOSUB 10000
4035 IF N$="E-O-F" THEN 4085.

```

```

4040 PRINT "***** ADDRESS BOOK "
4045 PRINT "*****";N$
4050 PRINT " ";A1$
4055 PRINT " ";A2$
4060 PRINT " ";A3$
4065 PRINT " ";A4$
4070 PRINT " ";A5$
4075 PRINT " ";T$
4080 GOTO 4025
4085 PRINT "***** END OF FILE "
4090 WAIT 197,191
4095 CLOSE 3 :GOSUB 10000
4100 RETURN

```

```

5000 REM ** ADD RECORD **
5005 POKE 53280,7 :F=0
5010 OPEN 4,8,4,"DATATEMP,SEQ,WRITE" :GOSUB 10000
5015 OPEN 3,8,3,"DATA,SEQ,READ" :GOSUB 10000
5020 INPUT#3,N$,A1$,A2$,A3$,A4$,A5$,T$ :GOSUB 10000
5025 IF N$<>"E-O-F" THEN GOSUB 9200 :GOTO 5020
5030 PRINT "***** ADDRESS BOOK "
5035 PRINT "***** ADD RECORD "
5040 PRINT "***** ENTER DETAILS "
5045 INPUT "***** NAME ";IN$
5050 INPUT " ADDRESS ";I1$
5055 INPUT " ";I2$
5060 INPUT " ";I3$
5065 INPUT " ";I4$
5070 INPUT " ";I5$
5075 INPUT " TELEPHONE ";IT$
5080 GOSUB 9100
5085 PRINT "***** EXIT (Y/N) ?"
5090 GET K$ :IF K$="" THEN 5090
5095 IF K$="N" THEN 5030
5100 N$="E-O-F" :GOSUB 9200
5105 CLOSE 3 :GOSUB 10000 :CLOSE 4 :GOSUB 10000
5110 PRINT#15,"S:DATA" :GOSUB 10000

```

```
5115 PRINT#15,"R:DATA=DATATEMP" :GOSUB 10000
5120 PRINT#15,"S:DATATEMP" :GOSUB 10000
5125 RETURN
```

```
6000 REM ** RETURN TO COMMAND MODE **
6005 POKE 53280,7
6010 PRINT "RETURN ADDRESS BOOK "
6015 PRINT "RETURNING TO COMMAND MODE !!"
6020 PRINT "PRESS SPACE BAR TO CONFIRM "
6025 PRINT "ANY OTHER KEY TO ABORT "
6030 GET K$ :IF K$="" THEN 6030
6035 IF K$<>" " THEN RETURN
6040 PRINT " " :POKE 53280,14
6045 CLOSE 15
6050 END
```



```

7000 REM  ** DELETE RECORD **
7005 POKE 53280,7
7010 OPEN 4,8,4,"DATATEMP,SEQ,WRITE" :GOSUB 10000
7015 OPEN 3,8,3,"DATA,SEQ,READ" :GOSUB 10000
7020 PRINT "***** ADDRESS BOOK "
7025 PRINT "***** DELETE RECORD "
7030 INPUT "***** ENTER NAME ";IN$
7035 INPUT#3,N$,A1$,A2$,A3$,A4$,A5$,T$ :GOSUB 10000
7040 IF N$="E-O-F" THEN 7060
7045 IF IN$<>N$ THEN GOSUB 9200
7050 IF IN$=N$ THEN PRINT "***** ";N$;" DELETED "
7055 GOTO 7035
7060 PRINT "***** END OF FILE "
7065 N$="E-O-F" :GOSUB 9200
7070 WAIT 197,191
7075 CLOSE 3 :GOSUB 10000 :CLOSE 4 :GOSUB 10000
7080 PRINT#15,"S:DATA" :GOSUB 10000
7085 PRINT#15,"R:DATA=DATATEMP" :GOSUB 10000
7090 PRINT#15,"S:DATATEMP" :GOSUB 10000
7095 RETURN

```

```

8000 REM  ** SCRATCH DATA FILE **
8005 POKE 53280,7
8010 PRINT "***** ADDRESS BOOK "
8015 PRINT "***** DATA FILE WILL BE SCRATCHED !!"
8020 PRINT "***** PRESS SPACE BAR TO CONFIRM "
8025 PRINT "***** ANY OTHER KEY TO ABORT "
8030 GET K$ :IF K$="" THEN 8030
8035 PRINT "***** CONFIRMED "
8040 IF K$=" " THEN PRINT#15,"S:DATA" :GOSUB 10000
8045 RETURN

```

```

9000 REM *****
9005 REM * WRITE DATA *
9010 REM *****
9100 PRINT#4,IN$;",";I1$;",";I2$;",";I3$;",";I4$;",";
    I5$;",";I6$;CHR$(13);
9105 GOSUB 10000
9110 RETURN
9200 PRINT#4,N$;",";A1$;",";A2$;",";A3$;",";A4$;",";
    A5$;",";T$;CHR$(13);
9205 GOSUB 10000
9210 RETURN

```

```

10000 REM *****
10005 REM * DISK ERROR SUBROUTINE *
10010 REM *****
10015 REM
10020 INPUT#15,EN,EM$,ET,ES
10025 IF EN<20 THEN RETURN
10030 PRINT "***** DISK ERROR HAS OCCURED *****"
10035 PRINT "ERROR NUMBER      ";EN
10040 PRINT "ERROR MESSAGE     ";EM$
10045 PRINT "TRACK NUMBER       ";ET
10050 PRINT "SECTOR NUMBER      ";ES
10055 CLOSE 3 :CLOSE 4: CLOSE 15
10060 STOP

```

RANDOM ACCESS DATA FILES

Although the concepts of random access files may seem initially complicated the only real problem that occurs for the beginner is to absorb the numerous commands that are required to be used. We said earlier that random access files used considerably more memory when compared to sequential files; this is because the length of each field in all the records is set to the same size and to that of the item with the maximum length expected. This may seem a waste of space but, because the length of a record is known, we have knowledge of the layout of the whole file and so can write to a record without having any fear of overwriting part of the next record.

ADVANCED DISK CONTROL

In order that we can use random access files we are going to have to delve a little deeper into the structure of the floppy disk and the operating system used by the disk drive.

The floppy disk used by the disk drive contains 35 concentric tracks, which are labelled from 1, for the outermost track to 35 for the innermost track, on which the programs and data are stored.

Each track is subdivided up into a number of sectors or blocks each of which can store 254 bytes of data plus a number of other items used by the DOS. The number of blocks per track varies depending on its position from the centre of the floppy disk; the outermost tracks have a longer circumference and so can store more data. The number of blocks for each track is listed in Figure 24.1

Track No.	Sector No.	Total No. of Sectors
1 - 17	0 - 20	21
18 - 24	0 - 19	20
25 - 30	0 - 18	19
31 - 35	0 - 17	18

Figure 24.1 Block (Sector) Distribution on a Floppy Disk

A program or data file which is stored on the disk may occupy several of the blocks. Since these blocks need not be stored sequentially on the disk each block contains a two byte pointer to indicate the position (track and sector) of the next block where the file continues.

Two blocks on the disk, positioned on track 18, sector 0 and 1, are reserved by the DOS for the 'Block Availability Map' (BAM) and the disk directory. The BAM consists of a bit map occupying 139 bytes where a bit is set on or off to signal whether a corresponding block is vacant or not. The disk directory gives information about the disk itself such as its name, id, and detailed information on the files stored on the disk such as the file type, position on disk, name and size.

By sending the disk command 'VALIDATE' (or 'V') to the disk drive the DOS will search through the blocks of data and construct a new BAM on the disk; any blocks which have been allocated but do not have an associated filename or files that were never closed properly will be deleted, freeing the space for future use.

```
i.e.  
OPEN 15, 8, 15  
PRINT# 15, "VALIDATE dn"  
or  
PRINT# 15, "V"
```

In our case the drive number 0 may be omitted. If a disk is ever changed and not re-initialised, the DOS will still be using the BAM and disk directory of the previous disk. This can cause corruption of data if the following precaution is not undertaken: when initialised, the DOS loads the disk id into local storage and then, whenever the disk is accessed, it identifies the id of the disk with the stored value and only obeys the disk command if they are identical. Thus, to avoid any mistakes you should always format all of your disks with different id numbers.

When using random access files, all communication between the Commodore 64 and the disk is undertaken via one of eight sections of storage available called disk buffers. The main advantage of this is speed since the computer can read/write data to the buffer faster than onto a magnetic medium such as a floppy disk. As a result the Commodore 64 can undertake other processing while the DOS transfers data between the buffer and blocks on the disk. By referring to Figure 24.2 it can be seen that there is basically five operations involved.

- (1) Transferring data from the computer into the buffer
- (2) Transferring data from the buffer onto the disk
- (3) Transferring data from the disk into the buffer
- (4) Transferring data from the buffer to the computer
- (5) Controlling the buffer pointer

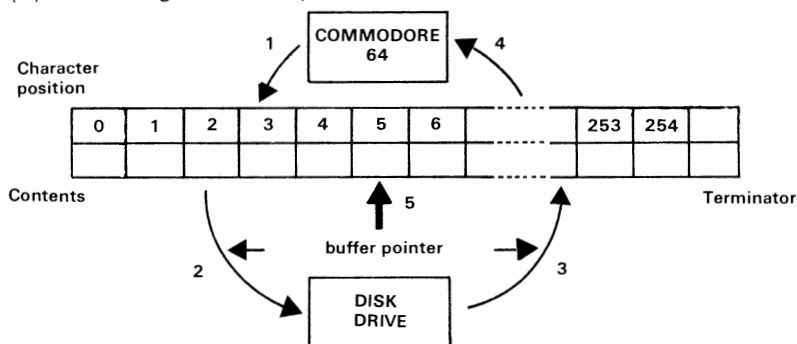


Figure 24.2 Communicating to Disk Drive Via Disk Buffer

The disk buffer, which contains 254 bytes available for data, is preceded by one byte which indicates the position in the buffer last written to and is followed by one byte that contains a terminator character.

We shall now see what commands are required to undertake the five operations shown in Figure 24.2. Since the syntax of these commands can take many forms it is advisable to refer to the Commodore VIC-1541 users manual once the basic techniques are understood.

COMMUNICATION CHANNELS REQUIRED FOR ADVANCE DISK CONTROL

So that the Commodore 64 and the disk drive are able to 'talk' to the disk buffer we have to set up three communication channels. This is done as before by opening logical files with secondary addresses that identify the link number. The first channel required is the command and error channel which is the same as in the previous disk operations that we have already seen - it is obtained by specifying a secondary address of 15.

i.e. OPEN 15, 8, 15

The two remaining channels link the Commodore 64 to the disk buffer and the disk buffer to the disk drive; the latter can also control the buffer pointer which indicates the location in the buffer to be written to. The first

of these channels may be created using any secondary address between 2 and 14 through we shall be consistent and always use the value 4. When creating this link we also have to specify which of the eight disk buffers we want to use. This is done by sending the instruction “#x” in the open statement where x is a buffer number 0 to 7; if x is omitted the DOS uses the first available buffer.

i.e. OPEN 4, 8, 4, “#”
links the DOS and the buffer.

The other channel utilises the command channel although it does require a separate logical file.

i.e. OPEN 5, 8, 15

Thus summing up so far, we have the three communication channels:

- 4 - Links computer to the disk buffer
- 5 - Links disk buffer to floppy disk
- 15 - Command and error channel

With three logical files opened, it is important that you are referring to the correct file and at the appropriate time. Channel 4 is used to PRINT# and INPUT# data to or from the buffer in the same fashion as to any other peripheral; channel 5 is used by the DOS to control certain other operations on the buffer. As before, channel 15 is the command and error channel for operations such as initialising the disk or for reporting “read errors”. When closing the files it is advisable that the command and error channel is closed last.

THE ADVANCED DISK CONTROL COMMANDS

When transferring data to and from the disk buffer we have to set the buffer pointer to the byte that we are referring to. The command to do so is ‘BUFFER-POINTER’ (or ‘B-P’) and should be followed by the channel number of the link between the computer to the buffer and the byte position in the buffer. Thus using the channel that we previously opened, to set the buffer pointer to the first location we would send

PRINT#5, “B-P:” 4; 1

Now that we have created all the communication channels required and set up the buffer pointer we can transfer data to the disk buffer by using PRINT#. Care has to be taken using PRINT# since, when sent to the

buffer, it will contain a sequence of data items followed by carriage return and linefeed characters. These two terminator characters can cause problems when reading back the data since we usually need only a single terminator to distinguish between the different items. We have seen earlier with cassette data files that a semicolon at the end of the PRINT# statement suppresses a carriage return and linefeed - thus, one solution to avoid any problems is to send the data required followed by a semicolon, then a linefeed character CHR\$(13) to distinguish the different data items and, finally, another semicolon to suppress any further terminator characters. If one PRINT# statement is used to transfer several items of data then they should be separated by the comma character, CHR\$(44).

For example, to send the variables A\$ and B\$ to the buffer position specified in the previous example we would use:

```
PRINT#4, A$, CHR$(44); B$; CHR$(13);
```

To transfer data to and from the buffer and a particular block on the disk we use the commands 'BLOCK-READ' (or 'B-R') and 'BLOCK-WRITE' (or 'B-W') respectively. After these commands we must specify the file number that linked the computer and buffer, the disk drive number and, finally, the particular track and sector on the disk.

Thus to write our data from the disk buffer to track 10 sector 5 we would use

```
PRINT#5, "B-W:" 4; 0; 10; 5
```

Likewise to read back the data from the same block we would use

```
PRINT#5, "B-R:" 4; 0; 10; 5
```

Before using INPUT# to transfer data back from the disk buffer to the Commodore we should ensure that the block pointer is in the correct position; in our example we know that the data starts at location 1 so we would use

```
PRINT#5, "B-P:" 4; 1
```

Finally, we can transfer the data from the buffer to the variables A\$ and B\$ stored in the computer by

```
INPUT#4, A$, B$
```

There are a series of disk commands called U1, U2, U3....U9 which

undertake user defined disk commands. These commands which are written in 6502 machine code and stored in the disk drives RAM are accessed by the above commands through a jump table of which detailed information is given in the VIC 1541 user's manual.

There are two special USER commands U1 and U2 which can be used to replace the BLOCK-READ and BLOCK-WRITE commands respectively which we have just looked at.

The format of these two commands are:

```
PRINT# file,"Ui";channel number;drive number;track;sector
```

These two commands have extensions over BLOCK-READ and BLOCK-WRITE. The main difference is the way they treat the character pointer byte (byte 0) in the disk buffer.

When INPUT#ing data using BLOCK-READ the position of the last valid character is stored in the character pointer byte and so INPUT# terminates on reaching this position. U1 however forces this byte to 255 and so the entire buffer is INPUT#.

Whereas BLOCK-WRITE sets the character position buffer to last position written to, U2 leaves this byte unchanged.

Of the many commands and syntax variations mentioned in the Commodore VIC-1541 users manual the two remaining important commands that should initially be learnt are 'BLOCK-ALLOCATE' (or 'B-A') and 'BLOCK-FREE' (or 'B-F'). Both of these commands should be followed by the drive number, and the track and sector of the disk involved.

'BLOCK-ALLOCATE' updates the copy of the BAM used by the DOS to show that a specified block has been allocated; as a result the DOS knows not to write to this block. A copy of this BAM is written to the disk when the relevant links are closed. Before writing to a block we should use 'BLOCK-ALLOCATE' since if the block has already been allocated a disk error will be returned. If we use INPUT# 15,EN,EM\$,ET,ES as shown in Program 46 then if no error has occurred, (EN=0), ET and ES will return the next available track and sector.

As an example, to make the block at track 10 sector 5 unavailable for use

```
PRINT#5, "B-A:" 0; 10; 5
```


Similarly the command 'BLOCK-FREE' releases a previously allocated block from the BAM.

For example to release the same block: PRINT#5, "B-F:" 0; 10; 5

BLOCK ALLOCATION & HASHING

Before we can write any programs using random access files we have two remaining problems. First we need to decide on how we are going to allocate the available disk space and, second, we need a method of being able to specify a key and to calculate the location on the disk where the record with that key is situated.

There are no fixed rules on how the records of a random access file should be distributed amongst the 254 byte blocks on a disk and so it is up to the programmer to decide on what he feels is the optimum layout for efficiency with both memory space and access speeds. In order to make the 'BUFFER-POINTER' commands simple it makes good sense to use sequential blocks on the disk and to set the start of each record at regular intervals.

Methods for calculating a records position from a key are called hashing algorithms for which many techniques, including some very complicated ones, exist. The complexity usually depends on whether the keys have a fixed pattern or are random; usually the more regular the pattern the simpler the hashing algorithm. In order to master random access files it is advisable to use keys with an obvious pattern to resolve the hashing algorithm to a linear formula of the type relative position = $a * \text{key} + b$.

If the keys are in a semi random order we can often use the techniques of sorting described in chapter 11 to order the file with reference to some key; we can then use a hashing algorithm to find approximately the location of a record and then search sequentially for the exact record from this point.

As an example of a hashing algorithm, consider the problem of storing each date in the year on disk along with all the names of friends whose birthdays lie on that date. Thus we need to allocate disk space for 12 months of up to 31 days and provide enough space to write the names of say up to 15 characters each. Remember that the method to be given is just one example of the many ways you could attack the problem.

We shall allocate two tracks on a disk to each month and store the data using sixteen blocks on the first of the tracks and as many blocks required

on the second track to make up the number of days in the month.

IMPORTANT - Care has to be taken to avoid track 18 sectors 0 and 1 since these contain the disk directory and BAM!!

Using tracks 1 & 2 for January, tracks 3 & 4 for February etc, the following algorithm returns a unique block for any date and avoids the disk directory and BAM.

Day d
Month M

IF $d \leq 16$
THEN track = $2 * M - 1$; sector = d
ELSE track = $2 * M$; sector = $d - 14$

e.g. 6th May = track 9 sector 6

Once the unique block for the date is known we can store the name on the disk. Each name is separated by the character CHR\$(13) and the last name on the block is followed by the character "/". Thus when a name is added the buffer pointer is set to the position where the character "/" is situated.

e.g. Program 51 - BIRTHDAY DATABASE

```
5 REM *****
10 REM *      BIRTHDAYS      *
15 REM * RANDOM ACCESS FILES *
20 REM *=====*
25 REM *    MRH  1/7/1983    *
30 REM *****
35 REM
40 PRINT "***** ENTER BIRTHDAY DATA DISK !!!"
45 WAIT 197,191
50 OPEN 15,8,15,"I"
55 OPEN  5,8,15
60 DIM NM(12) :FOR J=1 TO 12 :READ NM(J) :NEXT J
65 PRINT "***** BIRTHDAY DATABASE "
70 PRINT "***** PRESS: *****"
75 PRINT " F1 .. CLEAR  "
80 PRINT " F2 .. FORMAT  "
```

```

85 PRINT " F3 .. ENTRY  "
90 PRINT " F5 .. ENQUIRY "
95 PRINT " F7 .. QUIT  "
100 GET K$ :IF K$="" THEN 100
105 K=ASC(K$)-132 :IF K<1 OR K>5 THEN 100
110 ON K GOSUB 1000,2000,3000,4000,5000
115 POKE 53280,14
120 GOTO 65
200 DATA 31,29,31,30,31,31,31,31,30,31,30,31
1000 REM ** RESET DATE **
1005 POKE 53280,5
1010 PRINT "### BIRTHDAY DATABASE "
1015 PRINT "### RESET DATE "
1020 INPUT "### DATE ";D$ :D=D$
1025 INPUT " MONTH ";M$ :M=M$
1030 IF M<1 OR M>12 THEN 1010
1035 IF D<1 OR D>NM(M) THEN 1010
1040 PRINT "### BIRTHDAY DATABASE "
1045 PRINT "### DOB ";D;"/";M
1050 PRINT "### ARE YOU SURE (Y-N) ?? "
1055 GET K$ :IF K$="" THEN 1055
1060 IF K$<>"Y" THEN 1100
1065 GOSUB 6000 :REM HASH
1070 OPEN 4,8,4,"#" :GOSUB 10000
1075 PRINT#5,"B-R:"4;0;T;S :GOSUB 10000
1080 PRINT#5,"B-P:"4;1 :GOSUB 10000
1085 PRINT#4,"/";CHR$(13):GOSUB 10000
1090 PRINT#5,"B-W:"4;0;T;S :GOSUB 10000
1095 CLOSE 4
1100 RETURN

```

```

2000 REM ** ADD A RECORD **
2005 POKE 53280,5
2010 PRINT "### BIRTHDAY DATABASE "
2015 PRINT "### RECORD ENTRY "
2020 INPUT "### DATE ";D$ :D=D$

```

```

2025 INPUT " MONTH ";M% :M=M%
2030 IF M<1 OR M>12 THEN 2010
2035 IF D<1 OR D>NM(M) THEN 2010
2040 NN$="" :INPUT "ENTER NAME";NN$
2045 NN$=LEFT$(NN$,20)
2050 PRINT "YOUR BIRTHDAY DATABASE "
2055 PRINT "NAME ";NN$
2060 PRINT "DOB ";D;"/";M
2065 PRINT "ARE YOU SURE (Y-N) ?? "
2070 GET K$ :IF K$="" THEN 2070
2075 IF K$<>"Y" THEN 2145
2080 GOSUB 6000 :REM HASH
2085 OPEN 4,8,4,"#" :GOSUB 10000
2090 PRINT#5,"B-R:"4;0;T;S :GOSUB 10000
2095 P=1
2100 INPUT#4,N$ :GOSUB 10000
2105 IF N$="/" THEN 2120
2110 P=P+1+LEN(N$)
2115 GOTO 2100
2120 IF P+LEN(NN$)>250 THEN PRINT "BLOCK OVERFLOW !!
      " :WAIT 197,191 :GOTO 2140
2125 PRINT#5,"B-P:"4;P :GOSUB 10000
2130 PRINT#4,NN$;CHR$(13);"/";CHR$(13) :GOSUB 10000
2135 PRINT#5,"B-W:"4;0;T;S :GOSUB 10000
2140 CLOSE 4
2145 RETURN

```

```

3000 REM ** ACCESS A RECORD **
3005 POKE 53280,5
3010 PRINT "YOUR BIRTHDAY DATABASE "
3015 PRINT "RECORD ACCESS "
3020 INPUT "DATE ";D% :D=D%
3025 INPUT " MONTH ";M% :M=M%
3030 IF M<1 OR M>12 OR D<1 OR D>NM(M) THEN 2010
3033 GOSUB 6000 :REM HASH
3035 PRINT "YOUR BIRTHDAY DATABASE "
3040 PRINT"DOB ";D;"/";M;"

```

```

3045 OPEN 4,8,4,"#" :GOSUB 10000
3050 PRINT#5,"B-R:"4;0;T;S :GOSUB 10000
3052 PRINT#5,"B-P:"4;1 :GOSUB 10000
3055 FOR J=1 TO 20
3060 INPUT#4,N$
3065 IF N$="/" THEN 3080
3070 PRINT " ";N$
3075 NEXT J
3080 PRINT "X ----- "
3085 WAIT 197,191
3090 CLOSE 4
3095 RETURN

```

```

4000 REM ** RETURN TO COMMAND MODE **
4005 POKE 53280,5
4010 CLOSE 5 :CLOSE 15
4015 PRINT "X00 BIRTHDAY DATABASE "
4020 PRINT "X00 RETURN TO COMMAND MODE "
4025 PRINT "X00 PRESS SPACE BAR "
4030 GET K$ :IF K$="" THEN 4030
4035 IF K$<>" " THEN RETURN
4040 CLOSE 5 :CLOSE 15
4045 POKE 53280,14
4050 END

```

```

5000 REM ** FORMAT DISK **
5005 POKE 53280,5
5010 PRINT "X00 BIRTHDAY DATABASE "
5015 PRINT "X00 DISK FORMAT "
5020 INPUT "X00 ARE YOU SURE (YES/NO) ?? ";K$
5025 IF K$<>"YES" THEN RETURN
5030 FOR J=1 TO 5
5035 PRINT "X00 DISK WILL BE FORMATED.. OK ?? "
5040 NEXT J
5045 WAIT 197,191
5050 PRINT#15,"NEW:BIRTHDAYS,ZZ" :GOSUB 10000
5055 OPEN 4,8,4,"#" :GOSUB 10000

```

```

5060 FOR M=1 TO 12
5065 FOR D=1 TO NM(J)
5070 GOSUB 6000 :REM HASH
5075 PRINT#5,"B-P:"4;1      :GOSUB 10000
5080 PRINT#5,"B-R:"4;0;T;S :GOSUB 10000
5085 PRINT#4,"/"CHR$(13) :GOSUB 10000
5090 PRINT#5,"B-W:"4;0;T;S :GOSUB 10000
5095 NEXT D :NEXT M
5100 CLOSE 4
5105 PRINT "X00 BIRTHDAY DISK HAS BEEN FORMATTED "
5110 RETURN

```

```

6000 REM ** HASHING **
6005 IF D<=16 THEN T=2*M-1 :S=D :GOTO 6015
6010 T=2*M      :S=D-14
6015 RETURN

```

```

10000 REM *****
10005 REM * DISK ERROR SUBROUTINE *
10010 REM *****
10015 REM
10020 INPUT#15,EN,EM$,ET,ES
10025 IF EN<20 THEN RETURN
10030 PRINT "***** DISK ERROR HAS OCCURED X00"
10035 PRINT "ERROR NUMBER      ";EN
10040 PRINT "ERROR MESSAGE     ";EM$
10045 PRINT "TRACK NUMBER      ";ET
10050 PRINT "SECTOR NUMBER     ";ES
10055 STOP

```

There is one other method for block allocation and keeping track of whereabouts the information is situated on the disk. Let us suppose a record R\$ has been set up and is to be written to the disk so that it can be accessed directly by specifying a key K\$ which is a field contained in R\$.

Vacant space is found at Track T and Sector S by using the disk command BLOCK-ALLOCATE as described previously. Once the record is sent to the disk buffer using PRINT# and then written to the disk, we need to remember the position where it is stored. A suitable method is to keep a simple index such as 3 x n array in main memory which relates a key to a track and sector number. So that this index is not lost when the computer is switched off it should be written to the disk as a sequential file.

Reading back a record by specifying the key is now simple. One just searches through the index for the key and then transfers the data from the track and sector specified into the disk buffer. The record can now be read using INPUT#.

CHAPTER 25

DYNAMIC DATA STRUCTURES

Up till now, whenever we have required data to be contained in some fixed order we have stored the items in sequential elements of an array; such a data structure is called static. There are however occasions when we require a more flexible system of ordering and so we turn to several different methods which can be covered by the general title 'dynamic data structures'. While many of these techniques can be handled more suitably by other languages - for example ALGOL or Pascal - they can all be undertaken with Commodore 64 BASIC.

As an example of what dynamic data structures are all about, consider the following table which contains a list of customers' orders with a car dealer. All the records are kept in the order of the customer's surname as in Fig. 25.1, which enables the information referring to a particular customer to be found easily. Suppose further that a record for the customer HARRISON is required to be added to the list.

NAME	INITIAL	MODEL	COLOUR	DATA DUE
BANKS	M.	240 DL	Green	04.5.83
BEECH	G.	760 GLE	Silver	30.6.83
GRAHAM	R.	360 GLT	Red	25.5.83
HART	W.	240 GL	White	25.5.83
HOPTON	J.	360 GLS	Blue	17.6.83
TROWSDALE	D.	260 GLE	Black	19.6.83
TURNER	M.	340 DL	Red	17.6.83
HARRISON	M.	340 GL	Silver	30.6.83

Figure 25.1

In order that the table remains in order, the new record must be placed between HART and HOPTON. If these records were kept using normal office index cards any new record could be added by slotting the new card in at the appropriate position. Likewise, when the customer's order had been completed the corresponding record could be deleted from the system simply by removing the card.

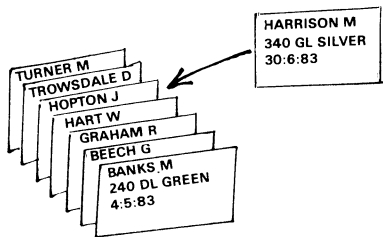


Figure 25.2

Our problem is to simulate such a system where we can add and delete records and, by reusing any vacated space, we keep the occupied memory down to a minimum. One data structure that can be used for this application is called a linked list. When using linked lists for small quantities of data the advantages of flexibility are offset by the disadvantages of complexity.

However, it is useful to learn the techniques since general and more complicated structures can then be handled in a similar fashion.

THE FORWARD LINKED LIST

Unlike static data structures, the ordering of a linked list is contained alongside the data. Each element in the structure will contain two pieces of information; the first item will be the data itself and the second item will be a pointer which relates the element with another element. The data could contain several fields, although the list in Figure 25.3 uses just one field.

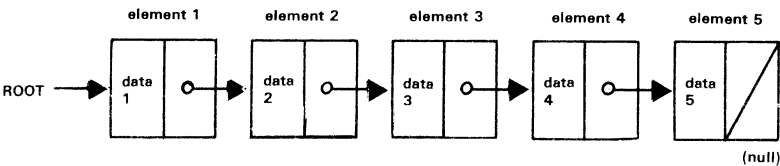


Figure 25.3

Since element 1 is linked to element 2 the pointer in element 1 will have the value 2; similarly, pointers 2,3 and 4 will have values 3,4 and 5 respectively. As element 5 is the last element in the list its pointer is null and so this could be indicated by a negative value, say -1.

Another method would involve setting the pointer to the address in the memory where the element is situated. This is a form of "indirect addressing" that we will not discuss further.

The structure shown in Figure 25.3 could be stored on the Commodore 64 by using a two dimensional array where $L(E,1)$ would contain the data of element E and $L(E,2)$ would contain the pointer to the next element. A variable ROOT can be used to point to the first element.

i.e.

ROOT = 1	
L(1,1) = data 1	L(1,2) = 2
L(2,1) = data 2	L(2,2) = 3
L(3,1) = data 3	L(3,2) = 4
L(4,1) = data 4	L(4,2) = 5
L(5,1) = data 5	L(5,2) = -1

To retrieve the data we access all the elements in the list; this is done by starting from ROOT and then moving along the elements in the list until the null pointer is reached.

An item can easily be deleted from a linear list by amending the pointer from the previous element so that it points instead to the next element. Thus, our routine above for retrieving data would search along the list and jump over the deleted item.

example - delete data item 4

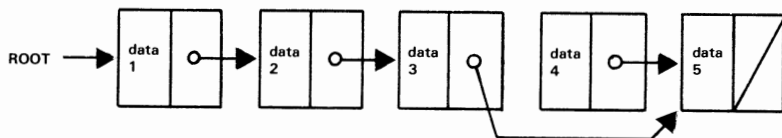


Figure 25.4

i.e. $L(3,2) = 5$

If instead we require to insert an item into our list there are two stages: firstly we must find where in the list the item should be placed and, secondly, amend the pointers to include the new item.

example - if data item 6 were to lie between items 3 and 4

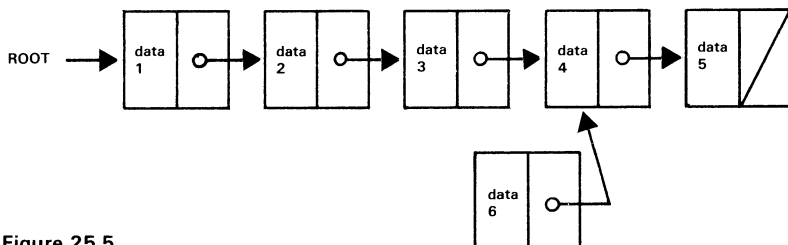


Figure 25.5

First the pointer from element 6 should be changed to point to element 4 and then the pointer of element 3 should be made to point at element 6:

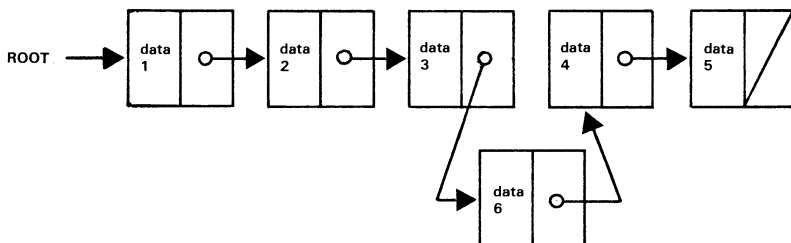


Figure 25.6

i.e. $L(6,1) = \text{data 6}$

$L(6,2) = 4$

$L(3,2) = 6$

In writing a general routine to undertake the addition of a data item, care must be taken when the data is to be placed at the beginning or end of the list.

Program 52 - LINKED LIST

```

10 REM *****
15 REM * LINKED LIST *
20 REM *=====*
25 REM * MRH 1/7/83 *
30 REM *****
35 REM
40 NULL=-1 :N=100 :DIM L(N,2) :GOSUB 1000
45 POKE 53280,14
50 PRINT"1000 LINKED LISTS"
55 PRINT"1000 PRESS: 0"
60 PRINT"F1 .. INITIALISE"
65 PRINT"F3 .. LIST"
70 PRINT"F3 .. DELETE"
75 PRINT"F7 .. INSERT"
80 GET K$ :IF K$="" THEN 80
85 K=ASC(K$)-132 :IF K<1 OR K>4 THEN 80
90 ON K GOSUB 1000,2000,3000,4000
95 GOTO 45

```

```

1000 REM **INITIALISE**
1005 POKE 53280,8
1010 PRINT"1 LINKED LISTS"
1015 PRINT"0 INITIALISE"
1020 ROOT=1 :NF=5
1025 L(1,1)=100 :L(2,1)=200 : L(3,1)=300 :L(4,1)=400
      :L(5,1)=500
1030 L(1,2)=2 :L(2,2)=3 :L(3,2)=4 :L(4,2)=5
      : L(5,2)=NULL
1035 PRINT"1000COMPLETED"
1040 GET K$ :IF K$="" THEN 1040
1045 RETURN

```

```

2000 REM **LIST**
2005 POKE 53280,8
2010 PRINT"1000 LINKED LISTS"
2015 PRINT"X LIST X"
2020 P=ROOT : IF ROOT=NULL THEN 2040
2025 PRINT P;TAB(4);L(P,1)
2030 P=L(P,2)
2035 IF P<>NULL THEN 2025
2040 GET K$: IF K$="" THEN 2040
2045 RETURN
3000 REM ** DELETE**
3005 POKE 53280,8
3010 PRINT"1000 LINKED LISTS"
3015 PRINT"X DELETE X"
3020 INPUT"XENTER DATA ITEM TO BE DELETED ";D
      :IF D<0 THEN 3020
3025 IF ROOT =NULL THEN 3055
3030 IF L(ROOT,1)=D THEN ROOT=L(ROOT,2)
      :PRINT"XITEM DELETED ": GOTO 3060
3035 P=ROOT
3040 PP=P: P=L(P,2) :IF P=NULL THEN 3055
3045 IF L(P,1)=D THEN L(PP,2)=L(P,2)
      : PRINT"X ITEM DELETED" : GOTO 3060
3050 GOTO 3040
3055 PRINT"X ITEM NOT IN LIST"
3060 GET K$: IF K$="" THEN 3060
3065 RETURN

```

```

4000 REM ** INSERT **
4005 POKE 53280,8
4010 PRINT"1000 LINKED LISTS "
4015 PRINT"X INSERT X"
4020 INPUT"X ENTER DATA ITEM TO BE INSERTED";D
      :IF D<0 THEN 4020
4025 NF=NF+1 :L(NF,1)=D
4030 IF ROOT=NULL THEN L(NF,2)=ROOT :ROOT=NF
      : GOTO 4065

```

```

4035 IF D<L(ROOT,1) THEN L(NF,2)=ROOT :ROOT=NF
      :GOTO 4065
4040 P=ROOT
4045 PP=P: P=L(P,2)
4050 IF P=NULL THEN L(PP,2)=NF :L(NF,2)=NULL
      :GOTO 4065
4055 IF L(NF,1)>L(P,1) THEN 4045
4060 L(PP,2)=NF :L(NF,2)=P
4065 PRINT"X ITEM INSERTED"
4070 GET K$: IF K$="" THEN 4070
4075 RETURN

```

One of the criteria that was stated when mentioning the requirements with data structures was that any vacated space could be reused. With Commodore 64 BASIC we need a method of releasing redundant elements from a data structure (some other languages do this automatically). One suitable method is to use another linear list that contains the elements that are available for usage.

Initially, this free storage list would include all the elements, but as the data list expanded the free storage list would shrink. This second linked list would be handled using the same algorithms as the first. If the data item is just a single item, then there is very little advantage using the linked list technique over static structure since the memory used would always be constant. However if the data items included several fields then the overheads of this second list would be offset by the amount of memory that could be saved.

MORE ADVANCED LISTS

Although most applications can be handled by forward linked lists there are a couple of amendments that can be made which enhance the power of the structures we have seen.

CIRCULAR LIST

In a circular list or ring the last element is made to point back to the first element. The main advantage of such a structure is that an item that precedes an element identified can still be accessed without having to restart at the ROOT.

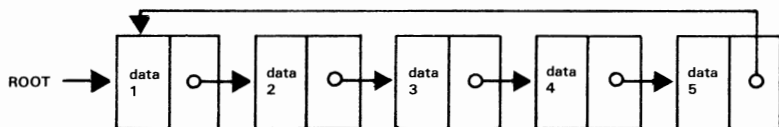


Figure 25.7

$$L(5,2) = 1$$

DOUBLE LINKED LISTS

Even greater power can be added to linked lists by including a backward pointer which links up an element with its predecessor in the structure.

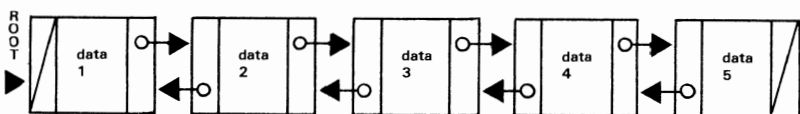


Figure 25.8

This enables the structure to be searched in either direction. By using more than one pointer it is possible to order the list in more than one type of order.

In small applications that we are likely to meet, the advantage of such a structure will be offset by an increase in the memory required for pointer storage.

STACKS AND QUEUES

There are two very useful linear data structures that are commonly used in computing called stacks and queues.

A stack is a method of storing and retrieving data in the computer with the basic principle that the most recent item of data stored will be the first retrieved. Storing information is known as pushing onto the stack and retrieving information is known as popping the stack. The top of the structure is indicated by a pointer called TOP.

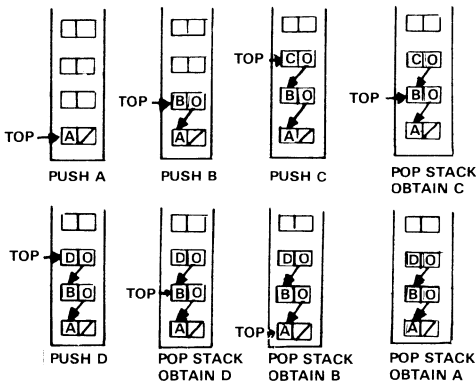


Figure 25.9

Although it is never obvious to the programmer, the system software uses a stack structure when dealing with subroutines. When control executes GOSUB, the line number of the statement plus 1 is pushed onto a stack and then control continues from the line specified in the statement. When control executes RETURN it continues from the line whose number is popped from the stack. This technique enables subroutines to be nested several levels deep although careful management is required by the programmer. If a RETURN is executed without a corresponding GOSUB statement an empty stack will be present and so when popped the error report 'RETURN WITHOUT GOSUB' is given.

A queue structure differs in that it has the rule that the most earliest item stored will be the first retrieved. Two pointers are required, HEAD and TAIL which point in the structure to the first and last items respectively.

A queue structure is used with the keyboard buffer where keys are stored as they are pressed. Since it is a queue, the earliest keys pressed are accessed before the more recent keys. Any released space is then made available to store future key presses.

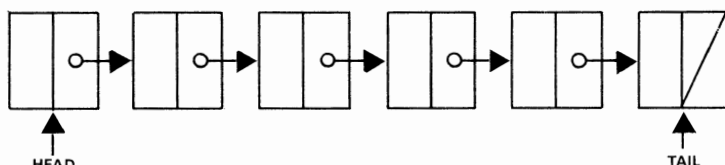


Figure 25.10

When an item is retrieved, HEAD is made to point to the next element in the structure, i.e. to the element that is pointed to by the pointer of the element that is being retrieved.

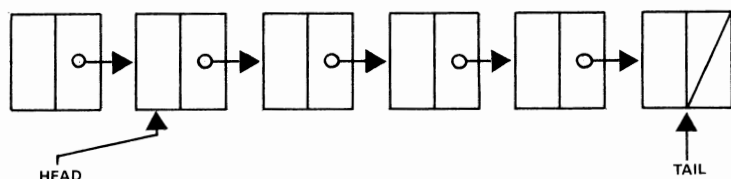


Figure 25.11

When an item is added both TAIL and the last element are made to point at the new item.

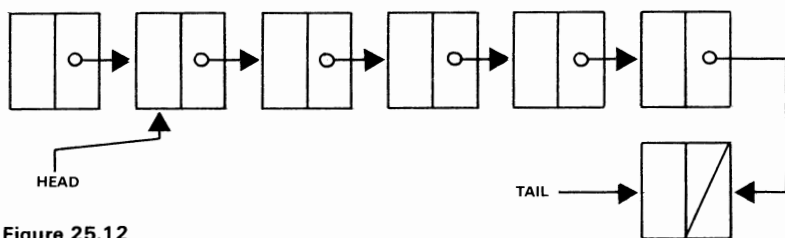


Figure 25.12

The main problem is that a queue gradually drifts through the memory as retrievals and additions occur. One solution is to use a circular list so that, if all the memory at the end of a structure runs out, then items can be added at the beginning. If TAIL ever reached HEAD then we have run out of space. A similar idea was used in Program 27 to control the SNAKE.

GRAPHS

Although the concepts of the linked lists are very useful as an insight into data structures, their use is severely limited as they can only function in one dimension, either forwards or backwards. In order to utilize these techniques with any practical ideas we often need to handle data structures in more than one dimension; such structures are called graphs.

Consider for example the following which represents the air routes that a certain airway company undertake.

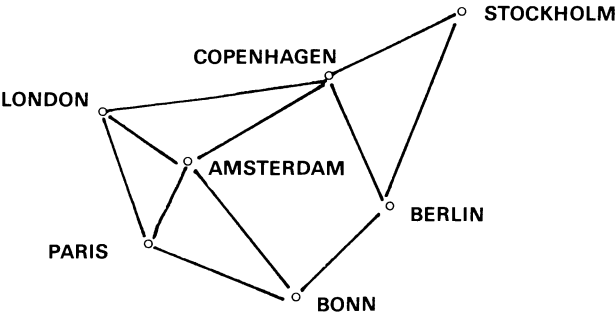
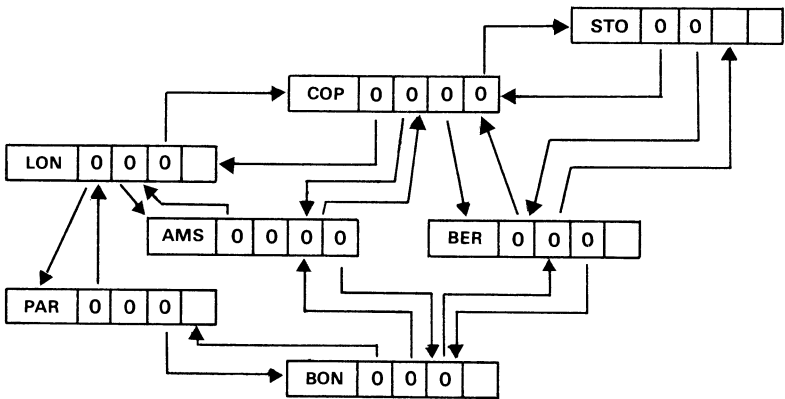


Figure 25.13

This could be represented by the following structure.

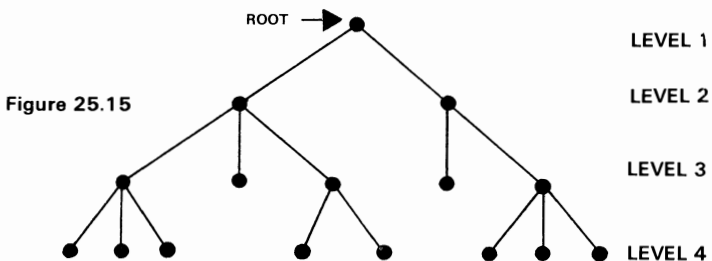


Additional data items could be included which, for example could give the cost, distance, etc. between two cities.

When referring to graphs; the intersecting points are called nodes and the links that join the nodes are called edges. Edges can be either directed or undirected and may or may not contain a value (e.g. in our above example cost, distance etc). Nodes can represent many things; such as towns or positions, production output, steps in a process etc.

TREES

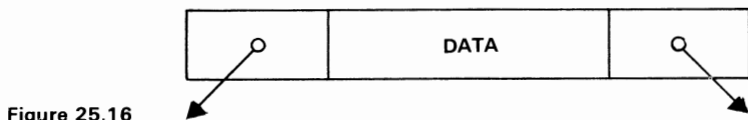
There is one special type of graph that is in common use in computing called a tree. A tree is a graph that contains no isolated nodes and no cycles, i.e. there is one and only one route for getting from one node to another. We are usually concerned with trees in which all nodes are directed away from one specific node called the ROOT; every node except for the root has exactly one edge, entering it.



The number of nodes away from the ROOT is often called the level where the ROOT is situated at level 1. The depth of a tree is its maximum level. Often a preceding node is called a parent node and a descending node is called a sibling.

In order to keep things simple we shall restrict ourselves to binary trees; these have a maximum of two edges leading from each node.

Binary trees may be stored on the Commodore 64 using arrays in a similar fashion to linked lists except that three fields are required; one is used to contain the data and the remaining two to contain the pointers.



As with linked lists we require the data to be set up in some logical order. For an example, let us see how we would store the following sequence of car names in alphabetical order in a binary tree.

MERCEDES, FERRARI, PORSCHE, LOTUS, VOLVO, BMW, SAAB

We commence with the first name, MERCEDES, as the root of our tree, and at this stage the node has no descendants.

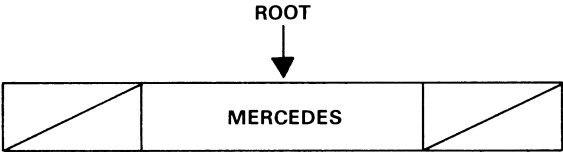


Figure 25.17

Next we introduce FERRARI and since it precedes MERCEDES alphabetically it will become a descendant on the left.

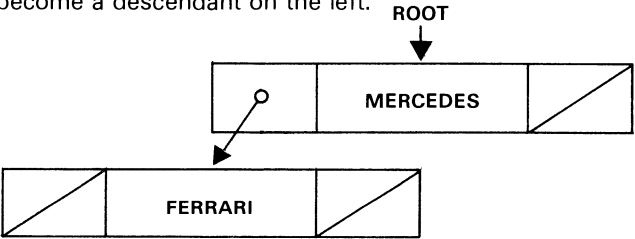


Figure 25.18

The next item, PORSCHE follows MERCEDES and so becomes a right descendent of it.

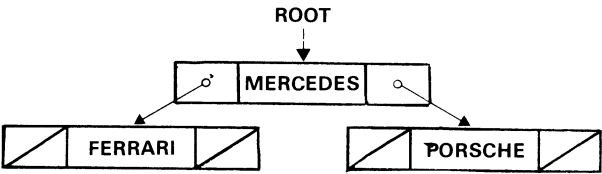


Figure 25.19

After examining all the information and arranging it alphabetically the tree will resemble:

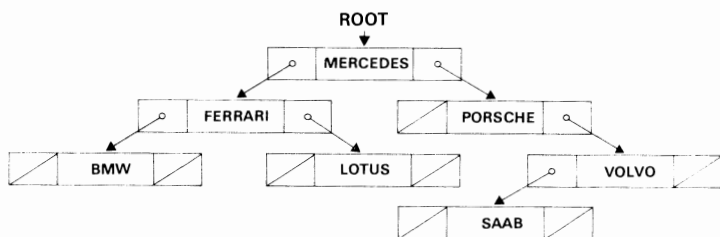


Figure 25.20

With Commodore BASIC we could use the arrays N\$ to store the data items and L & R to store the left and right pointers. Other data items such as details about the cars can be added to the structure by storing them in arrays and using the corresponding subscripts.

ROOT = 1

1 N\$(1) = "MERCEDES"	L(1) = 2	R(1) = 3
2 N\$(2) = "FERRARI"	L(2) = 6	R(2) = 4
3 N\$(3) = "PORSCHÉ"	L(3) = -1	R(3) = 5
4 N\$(4) = "LOTUS"	L(4) = -1	R(4) = -1
5 N\$(5) = "VOLVO"	L(5) = 7	R(5) = -1
6 N\$(6) = "BMW"	L(6) = -1	R(6) = -1
7 N\$(7) = "SAAB"	L(7) = -1	R(7) = -1

How a tree can be created with a sequence of data is illustrated later in part of Program 53.

The next obvious task is to be able to select a name and to request all the data that is associated to it. Having entered a name, the program searches down the tree starting at the ROOT and then travelling either left or right depending on the alphabetical order or the entered name and the one stored at the present node. If the null pointer is reached before the entered name is located then the name is not present in the tree. This method of searching is a form of what is called a binary chop and can be very fast, even with vast quantities of data. For example, by making just seven comparisons we can get down to level 8 of the tree which could mean that up to 255 elements have been searched through.

To output all the data in the tree in the order set up, we require a systematic method of travelling to each node once and in the order of the leftmost nodes before those to their right.

If we consider any node in the binary tree, providing its pointers are not null, it is linked to two binary subtrees. Similarly, all the nodes in each of these subtrees are joined to further subtrees. Thus a binary subtree may be abbreviated to

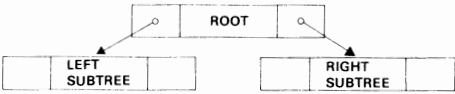


Figure 25.21

If we travel to each of these three components in a fixed order it is called a traversal.

There are three traversals that are commonly used:

1) Left Subtree	1) Left Subtree	1) Node
2) Node	or	2) Left Subtree
3) Right Subtree	3) Node	3) Right Subtree
(LNR)	(LRN)	(NLR)

Further traversals using the same fixed order should be made at each subtree. If we use the order LNR we can obtain our data in the order that the structure was set up with (i.e. alphabetic).

Consider the following tree under a LNR traversal

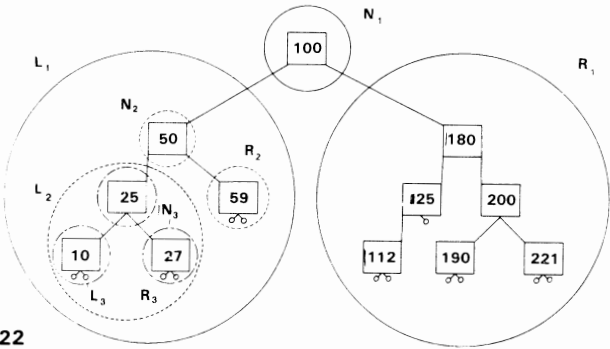


Figure 25.22

Thus our order would be L_1 , 100, R_1 . Subtree L_1 can be traversed LNR to give L_2 , 50, 59 and L_2 can be traversed LNR to give 10, 25, 27. Thus our order so far is 10, 25, 27, 50, 59, 100, R.

When R_1 has traversed LNR will have accessed the whole tree in order.

Program 53 demonstrates how to sort alphabetically a sequence of strings by creating a binary tree and then undertaking an LNR traversal.

The traversal is done by starting at the ROOT and continually travelling left popping the nodes onto a stack. Wherever a null pointer is reached we pop the stack and print out the data value of the node. Having gone as far as possible we then back-track to the parent node by again popping the stack and then printing its data value. Then if there is a right hand node made we undertake the same process on its subtree. This process is repeated for the whole tree.

Program 53 - TREE SORT

```

10 REM *****
15 REM *   TREE SORT   *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 NULL=-1 :M=150
45 DIM N$(M) :DIM R(M) :DIM L(M) :DIM S(M)
50 GOSUB 1035
55 POKE 53280,14
60 PRINT "TREE SORT "
65 PRINT "PRESS: M"
70 PRINT " F1 .. INITIALISE "
75 PRINT " F3 .. ADD TO TREE"
80 PRINT " F5 .. LIST TREE "
85 GET K$ :IF K$="" THEN 85
90 K=ASC(K$)-132 :IF K<1 OR K>3 THEN 85
95 ON K GOSUB 1000,2000,3000
100 GOTO 55
1000 REM ** INITIALISE **
1005 POKE 53280,10
1010 PRINT "TREE SORT "
1015 PRINT "INITIALISE "
1020 PRINT "ARE YOU SURE (Y/N) ?? "

```



```

1025 GET K$ :IF K$="" THEN 1025
1030 IF K$<>"Y" THEN 1065
1035 FOR J=1 TO M
1040 N$(J)="" :R(J)=0 :L(J)=0
1045 NEXT J
1050 ROOT=NULL :V=0
1055 PRINT "### COMPLETED "
1060 GET K$ :IF K$="" THEN 1060
1065 RETURN

```

```

2000 REM ** ADD TO TREE **
2005 POKE 53280,10
2010 PRINT "### TREE SORT "
2015 PRINT "### ADD TO TREE "
2020 D$="" :INPUT "### ENTER DATA ";D$
2025 V=V+1 :IF V>M THEN PRINT"### INCREASE SIZE OF
      ARRAY " :STOP
2030 IF D$="" THEN 2010
2035 IF ROOT=NULL THEN N$(V)=D$ :R(V)=NULL :L(V)=NULL
      :ROOT=V :GOTO 2075
2040 VV=ROOT
2045 IF N$(VV)<D$ THEN 2060
2050 IF L(VV)=NULL THEN L(VV)=V :GOTO 2070
2055 VV=L(VV) :GOTO 2045
2060 IF R(VV)=NULL THEN R(VV)=V :GOTO 2070
2065 VV=R(VV) :GOTO 2045
2070 N$(V)=D$ :R(V)=NULL :L(V)=NULL
2075 PRINT "### ITEM ADDED "
2080 PRINT "### EXIT (Y/N) ?? "
2085 GET K$ :IF K$="" THEN 2085
2095 IF K$<>"Y" THEN 2010
2100 RETURN

```

```

3000 REM ** LIST TREE **
3005 POKE 53280,10
3010 PRINT "### TREE SORT "

```

```

3015 PRINT "### LIST TREE  "
3020 PRINT "### PRESS ANY KEY TO LIST  "
3025 IF ROOT=NULL THEN 3075
3030 VV=ROOT :P=1
3035 IF VV=NULL THEN 3055
3040 S(P)=VV :P=P+1
3045 VV=L(VV)
3050 GOTO 3035
3055 P=P-1 :VV=S(P)
3060 PRINT N$(VV)
3065 VV=R(VV)
3070 IF P>0 THEN 3035
3075 PRINT "  -----  "
3080 GET K$ :IF K$="" THEN 3080
3085 RETURN

```

HEURISTIC PROGRAMMING

By now you should have some ideas of the flexibility that dynamic data structures can provide. Trees are the most common form found in computing and are used in a varied range of unusual subjects. Many games can be studied using trees, where different branches provide the various options available; by using some optimizing algorithm a computer can examine all the possible states that could occur for any feasible move it next makes.

A similar idea is used when a program learns from previous operations and remembers what results occur for certain moves by building onto the tree. The program commences with a tree consisting of a single node, the ROOT, and at that stage knows nothing; with time the tree will have expanded and soon can become "intelligent". Programs that learn with experience are called heuristic. Heuristic programming is demonstrated in program 54 which starts off with a smaller number of animals and facts in its memory. It asks you to think of an animal and then it tries to guess your animal by asking a series of questions to which you reply 'yes' or 'no'. If the computer is unable to guess the animal it will ask you for a question and answer that it can use in the future.

The program builds up a binary tree with animals stored at the bottom nodes in the tree and questions in the other nodes. By asking questions and analyzing the answers the program can decide on which path to travel down the tree. When an animal is reached the computer asks if it is correct; if not, it asks for a question that distinguishes between the animal in the tree and the animal chosen.

Program 54 - HEURISTIC ANIMALS

```
10 REM *****
15 REM *   ANIMALS   *
20 REM *=====*
25 REM * MRH  1/7/1983 *
30 REM *****
35 REM
40 GOSUB 300 :REM INITIALISE
45 PRINT "***** ANIMALS "
50 PRINT "***** THINK OF AN ANIMAL"
55 PRINT "***** THEN ANSWER (Y/N) TO THE FOLLOWING", " QUES
   TIONS *****"
```

```

60 P=1
65 IF L(P)=0 AND R(P)=0 THEN 95
70 PRINT " ";Q$(P); " ? ";
75 GET K$ :IF K$="" THEN 75
80 IF K$="Y" THEN PRINT "Y" :P=L(P) :GOTO 65
85 IF K$="N" THEN PRINT "N" :P=R(P) :GOTO 65
90 GOTO 75
95 PRINT " IS THE ANIMAL A ";Q$(P); " ?"
100 GET K$ :IF K$="" THEN 100
105 IF K$="Y" THEN PRINT "### I THOUGHT SO " :GOTO 170
110 IF K$<>"N" THEN 100
115 NF=NF+2 :IF NF>199 THEN PRINT "### INCREASE SIZE OF
      ARRAYS " :STOP
120 INPUT "### WHAT HAD YOU CHOSEN ";Q$(NF+1)
125 Q$(NF)=Q$(P)
130 PRINT "### ENTER A QUESTION THAT DISTINGUISHES"
135 PRINT " A ";Q$(NF); " FROM A ";Q$(NF+1)
140 INPUT "### ";Q$(P)
145 PRINT "### WOULD THE ANSWER TO THIS QUESTION BE"," Y
      OR N FOR ";Q$(NF)
150 GET K$ :IF K$="" THEN 150
155 IF K$="Y" THEN L(P)=NF :R(P)=NF+1 :GOTO 170
160 IF K$="N" THEN R(P)=NF :L(P)=NF+1 :GOTO 170
165 GOTO 150
170 PRINT "### WOULD YOU LIKE TO PLAY AGAIN ?"
175 GET K$ :IF K$="" THEN 175
180 IF K$<>"N" THEN GOTO 45
185 END
300 REM ** INITIALISE **
305 N=150 :NF=12
310 DIM Q$(N), L(N), R(N)
315 FOR J=1 TO 13
320 READ Q$(J),L(J),R(J)
325 NEXT J
330 RETURN
350 DATA "IS IT A BIRD",3,2
355 DATA "IS IT A MAMMAL",4,5
360 DATA "DOES IT FLY",6,7
365 DATA "DOES IT HAVE A TRUNK",8,9
370 DATA "FROG",0,0
375 DATA "ROBIN",0,0

```

```
380 DATA "OSTRICH",0,0
385 DATA "ELEPHANT",0,0
390 DATA "DOES IT HIBERNATE",10,11
395 DATA "SQUIRREL",0,0
400 DATA "DOES IT SWIM",12,13
405 DATA "SEAL",0,0
410 DATA "DOG",0,0
```

Since the topic of data structures is both large and complicated anyone interested in delving further should obtain one of the numerous books specializing on the subject. A recommended book is "Successful Software for Small Computers" by G. Beech, published by Sigma Technical Press. This contains numerous BASIC programs relevant to various data structures.

CHAPTER 26

MACHINE CODE PROGRAMMING

WHERE DO WE GO FROM HERE?

On reaching this part of the book you may have felt that there have been times when your Commodore 64 was operating just too slowly for your requirements. The fast execution of programs is vital for such uses as high speed action games or simulations and complex games like chess which may involve looking ahead at hundreds of possible moves, whilst getting a response within a reasonable time. If you have become sufficiently skilled in the techniques of Commodore BASIC programming you may like to go on and try machine code programming which will give you the extra speed. This involves writing the program directly in the form that is obtained when the BASIC interpreter converts a BASIC program into code that the computer can understand. Since the translating process is no longer required, a machine code program will execute many times faster than its equivalent high level program (a machine code program also has the advantage that it occupies less memory space). This section of the book is not intended to teach the intricacies of machine code programming but to, hopefully, wet the appetite of the Commodore 64 user enough to seek more information about the subject.

How does a machine code program differ from a BASIC program? Well, basically a machine code program looks like a long list of pairs of hexadecimal digits, with each pair representing a code for a certain instruction that the computer is required to undertake. Unfortunately the long list of digits are far less readable than the English-like structure of a BASIC program. The actual number of instructions that are available is immense and so, even the most experienced machine code programmers have to refer to reference tables for the correct codes. The instructions available are listed in terms of assembler language in Figure 26.1 but each of these have certain variations that require checking in a book specialising

on the 6510A microprocessor. Each individual machine code instruction usually has a very simple task - the most common is storing a value at a particular location. Other operations carried out by the 6510A include addition, subtraction, logical AND, OR, exclusive-OR, compare, the shifting and rotating of the bits, incrementing or decrementing a value by 1, the manipulation of individual bits, conditional jumps, stack operations and I/O operations. In addition to the addresses in the RAM, there are several other locations, called registers, that can store values; each of these, which have specific names, are called:

1. The Accumulator - holds the result of arithmetic and logical operations.
2. The x Index Register - used by specific instructions.
3. The y Index Register - used by specific instructions.
4. The Program Counter - contains the address of the machine code instruction that is currently being obeyed. After this instruction has been carried out the program counter is incremented by 1. It is possible to make a jump in a machine code program and this will cause the value of the program counter to change to the address of the instruction to which the jump refers.
5. The Stack Pointer - this contains the address of the top of a stack that is used by the 6510A. By using machine code instructions, data may be popped from and pushed onto the stack.
6. The Process Status - this register contains a group of 8 bits, each bit indicates a different specific condition relating to the result of arithmetic and logical operations, i.e. overflow flag, zero flag, carry flag etc.

Once written, a machine code program can be stored in the memory by POKEing each instruction into sequential addresses of the RAM. It can then be executed by using either the system command SYS or the functionUSR.

Syntax:

SYS <address>
USR (<numeric>)

SYS will commence the execution of a machine code program from the address specified until the instruction RTS, return from subroutine, is executed (all rather similar to calling a BASIC subroutine). Similarly USR

will execute a machine code program from the location specified at address 785/6 (0311/2 (hex)). The argument to the function is stored in a floating point form at address 97(0061(hex)) for the exponent and addresses 98-101 (0062-0065(hex)) for the mantissa - this location is called the 'floating point accumulator' and is used in machine code programming to carry out mathematical operations on values other than the integers 0 to 255. Again RTS will return the program back into BASIC mode. If there is no RTS instruction the computer will remain in the machine code system indefinitely. If you run a BASIC program containing errors, the computer will stop execution and display the error reports. However, a machine code program containing errors will probably crash

and there will be no way of stopping the program. Pressing **RUN STOP** will have

no effect and so the only way of regaining normality will be to disconnect the power supply, which unfortunately will mean the contents of the memory will be lost!!

6510 MICROPROCESSOR ASSEMBLER LANGUAGE INSTRUCTION SET

Figure 26.1

ADC	Add to Accumulator with carry.
AND	Logical AND with Accumulator
ASL	Shift one bit to left
BCC	Branch on Carry clear
BCS	Branch on Carry set
ADC	Add to Accumulator with carry.
AND	Logical AND with Accumulator
ASL	Shift one bit to left
BCC	Branch on Carry clear
BCS	Branch on Carry set
BEQ	Branch on result zero
BIT	Test specific bit
BMI	Branch on negative result
BNE	Branch on non zero result
BPL	Branch on positive result
BRK	Break
BVC	Branch on Overflow clear
BVS	Branch on Overflow set
CLC	Clear Carry Flag

CLD	Clear decimal mode
CLI	Clear Interrupt Disable bit
CLV	Clear Overflow Flag
CMP	Compare with Accumulator
CPX	Compare with Index X
CPY	Compare with Index Y
DEC	Decrement by 1
DEX	Decrement Index X by 1
DEY	Decrement Index Y by 1
EOR	Exclusive Or with Accumulator
INC	Increment by 1
INX	Increment Index X by 1
INY	Increment Index Y by 1
JMP	Jump
JSR	Subroutine jump
LDA	Load Accumulator
LDX	Load Index X
LDY	Load Index Y
LSR	Shift one bit to right
NOP	No operation
ORA	Logical OR with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pop Accumulator from Stack
PLP	Pop Processor Status from Stack
ROL	Rotate bits one bit to left
ROR	Rotate bits one bit to right
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract from Accumulator with borrow
SEC	Set Carry Flag
SED	Set decimal mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator
STX	Store Index X
STY	Store Index Y
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

If at this stage you are feeling that machine code programming is tedious to use - then you are right - it is! Fortunately there are some software tools available from Commodore which can make machine code programming a lot more enjoyable.

The Commodore Assembler Development Package (Commodore Product No. ASM 6440) is a series of software tools that enables the Commodore 64 user to create, assemble, load and execute 6510 assembly language code. By using an assembler it is possible to write a machine code program using mnemonics, as in Figure 26.1, for each instruction rather than the pair of hexadecimal digits that represents the actual instruction.

The fact that you are writing mnemonics that have an obvious meaning (such as JMP for jump, ADC for add) makes life a lot simpler.

Commands included in the Commodore package allow you to:-

- enter a line of assembly code
- compare two areas of memory
- disassemble code (this is the process of converting machine code back into assembler)
- fill a range of memory locations with a specific byte
- search through memory
- display memory in ASCII format
- display memory in hexadecimal format
- relocate in absolute memory with an offset
- display the contents of registers
- transfer data from one area of memory to another
- and much more

The assembler development package is available on diskette and includes a comprehensive manual containing full instructions.

In addition, the Commodore Assembler Tutor package (Commodore Product No. AST 6440/6420) is available for programmers who have no experience with machine code or those requiring more knowledge on the subject. The package is written about the 6502 microprocessor from which the 6510 was derived. The package is available on either diskette or cassette and comprises the three modules

- representation of information
- hardware and instructions
- software and programming

A user handbook is provided with the package but the vast majority of teaching information is displayed on the screen.

Finally there is one other method of obtaining a machine code program - right back in Chapter 1 we learned that a machine code program could be obtained from a high level language source program by a process known as compilation. When compiled, a program is converted into a separate machine code form; whenever the program is run the machine code version is executed. A BASIC compiler from Commodore called PETSPEED (Commodore Product No. PSP 6440) is available which enables a machine code version of a BASIC program to be executed up to 40 times faster than the original. When executed, the fact that a program is running under PETSPEED is completely user transparent. The PETSPEED BASIC compiler is available on diskette.

APPENDIX A

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
	0		17	"	34	3	51
	1		18	#	35	4	52
	2		19	\$	36	5	53
	3		20	%	37	6	54
	4		21	&	38	7	55
	5		22	.	39	8	56
	6		23	(	40	9	57
	7		24	)	41	:	58
DISABLES  	8		25	*	42	;	59
ENABLES  	9		26	+	43	<	60
	10		27	,	44	=	61
	11		28	-	45	>	62
	12		29	.	46	?	63
	13		30	/	47	@	64
	14		31	0	48	A	65
	15		32	1	49	B	66
	16	!	33	2	50	C	67
D	68		97		126		155
E	69		98		127		156
F	70		99		128		157
G	71		100		129		158
H	72		101		130		159

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	
I	73		102		131		160	
J	74		103		132		161	
K	75		104	f1	133		162	
L	76		105	f3	134		163	
M	77		106	f5	135		164	
N	78		107	f7	136		165	
O	79		108	f2	137		166	
P	80		109	f4	138		167	
Q	81		110	f6	139		168	
R	82		111	f8	140		169	
S	83		112			141		170
T	84		113			142		171
U	85		114			143		172
V	86		115			144		173
W	87		116			145		174
X	88		117			146		175
Y	89		118			147		176
Z	90		119			148		177
[	91		120			149		178
£	92		121			150		179
]	93		122			151		180
↑	94		123			152		181
←	95		124			153		182
	96		125			154		183
	184		186		188		190	
	185		187		189		191	

CODES
CODES
CODE

192-223
224-254
255

SAME AS
SAME AS
SAME AS

96-127
160-190
126

APPENDIX B

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
(a		0	C	c	3	F	f	6
A	a	1	D	d	4	G	g	7
B	b	2	E	e	5	H	h	8
I	i	9	%		37		A	65
J	j	10	&		38		B	66
K	k	11	.		39		C	67
L	l	12	(		40		D	68
M	m	13	)		41		E	69
N	n	14	*		42		F	70
O	o	15	+		43		G	71
P	p	16	,		44		H	72
Q	q	17	-		45		I	73
R	r	18	.		46		J	74
S	s	19	/		47		K	75
T	t	20	0		48		L	76
U	u	21	1		49		M	77
V	v	22	2		50		N	78
W	w	23	3		51		O	79
X	x	24	4		52		P	80
Y	y	25	5		53		Q	81
Z	z	26	6		54		R	82
[		27	7		55		S	83
£		28	8		56		T	84

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
]		29	9		57		U	85
↑		30	:		58		V	86
←		31	;		59		W	87
SPACE		32	<		60		X	88
!		33	=		61		Y	89
"		34	>		62		Z	90
#		35	?		63			91
\$		36			64			92
		93			105			117
		94			106			118
		95			107			119
SPACE		96			108			120
		97			109			121
		98			110			122
		99			111			123
		100			112			124
		101			113			125
		102			114			126
		103			115			127
		104			116			

Codes from 128-255 are reversed images of codes 0-127.

APPENDIX C

SCREEN CONTROL CHARACTERS

CHR\$(147) ☐ CLEAR SCREEN.
CHR\$(19) ⌂ CURSOR HOME.
CHR\$(145) ☐ CURSOR UP.
CHR\$(29) ⏏ CURSOR RIGHT.
CHR\$(157) ⏏ CURSOR LEFT.
CHR\$(17) ⌵ CURSOR DOWN.

CHR\$(146) ⌂ REVERSE CONTRAST OFF.
CHR\$(18) ■ REVERSE CONTRAST ON.

CHR\$(144) ■ COLOUR BLACK.
CHR\$(5) ■ COLOUR WHITE.
CHR\$(28) ■ COLOUR RED.
CHR\$(159) ▲ COLOUR CYAN.
CHR\$(156) ■ COLOUR PURPLE.
CHR\$(30) ■ COLOUR GREEN.
CHR\$(31) ■ COLOUR BLUE.
CHR\$(158) ■ COLOUR YELLOW.
CHR\$(129) ■ COLOUR ORANGE.
CHR\$(149) ■ COLOUR BROWN.
CHR\$(150) ■ COLOUR LIGHT RED.
CHR\$(151) ■ COLOUR GREY 1.
CHR\$(152) ■ COLOUR GREY 2.
CHR\$(153) ■ COLOUR LIGHT GREEN.
CHR\$(154) ■ COLOUR LIGHT BLUE.
CHR\$(155) ■ COLOUR GREY 3.

CHR\$(14) ⏏ UPPER/LOWER MODE.
CHR\$(142) ■ UPPER/GRAPHIC MODE.
CHR\$(8) ⏏ DISABLE CHARACTER MODE SWITCH.
CHR\$(9) ⏏ ENABLE CHARACTER MODE SWITCH.

CHR\$(141) ■ SHIFTED RETURN.

* *
*Not *
Directly
*From *
Keyboard
* *

PRINTER CONTROL CHARACTERS

CHR\$(10)	LINE FEED.
CHR\$(13)	CARRIAGE RETURN.
CHR\$(8)	GRAPHIC MODE.
CHR\$(14)	DOUBLE WIDTH PRINT MODE.
CHR\$(15)	STANDARD WIDTH PRINT MODE.
CHR\$(16)	SET PRINT POSITION.
CHR\$(27)	SET DOT POSITION.
CHR\$(26)	REPEAT GRAPHIC CHARACTER.
CHR\$(145)	UPPER CASE MODE.
CHR\$(17)	LOWER CASE MODE.
CHR\$(18)	REVERSE ON MODE.
CHR\$(146)	REVERSE OFF MODE.

APPENDIX D

MOS 6566 VIDEO INTERFACE CONTROLLER (VIC)

Address (hex)	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	Address (decimal)
D000	Sprite 0 x position								53248
D001	Sprite 0 y position								53249
D002	Sprite 1 x position								53250
D003	Sprite 1 y position								53251
D004	Sprite 2 x position								53252
D005	Sprite 2 y position								53253
D006	Sprite 3 x position								53254
D007	Sprite 3 y position								53255
D008	Sprite 4 x position								53256
D009	Sprite 4 y position								53257
D00A	Sprite 5 x position								53258
D00B	Sprite 5 y position								53259
D00C	Sprite 6 x position								53260
D00D	Sprite 6 y position								53261
D00E	Sprite 7 x position								53262
D00F	Sprite 7 y position								53263
D010	Most Significant Bit of Sprites (0-7) x position								53264
D011	Raster Compare	Extended colour Text	Bit map	Blank screen	24/25 Row Text	y value used for vertical scrolling			53265
D012	Compare		IRQ		Read Raster/Write Raster				53266
D013	x position of light pen								53267
D014	y position of light pen								53268
D015	Sprites (0-7) On/Off Flags								53269
D016	Unused	0	Multi-colour	3/40 Column Text	x value used for horizontal scrolling				53270
D017	Sprites (0-7) Vertical Expansion/Contraction Flags								53271

		BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0		
Address (hex)	D018	Video Matrix Base Address Relative to VIC				Character Dot Data Base Address Relative to VIC				53272	Address (decimal)
	D019	Req Condition Flag	Light pen IRQ				Sprite/Sprite Collision IRQ	Sprite/B/ground Collision IRQ	Raster Compare IRQ	53273	
	D01A	IRQ Mask Register								53274	
	D01B	Set for Sprites (0-7) Priority Over Background								53275	
	D01C	Sprites (0-7) Multicolour Flags								53276	
	D01D	Sprites (0-7) Horizontal Expansion/Contraction Flags								53277	
	D01E	Sprite/Sprite Collision Flags								53278	
	D01F	Sprite/Background Collision Flags								53279	
	D020	Border Colour								53280	
	D021	Background Colour 0								53281	
	D022	Background Colour 1								53282	
	D023	Background Colour 2								53283	
	D024	Background Colour 3								53284	
	D025	Sprite Multicolour Register 0								53285	
	D026	Sprite Multicolour Register 1								53286	
	D027	Sprite 0 Colour								53287	
	D028	Sprite 1 Colour								53288	
	D029	Sprite 2 Colour								53289	
	D02A	Sprite 3 Colour								53290	
	D02B	Sprite 4 Colour								53291	
	D02C	Sprite 5 Colour								53292	
	D02D	Sprite 6 Colour								53293	
	D02E	Sprite 7 Colour								53294	

APPENDIX E

MOS 6581 SOUND INTERFACE DEVICE (SID)

Address (hex)	BIT 7 BIT 6 BIT 5 BIT 4 BIT 3 BIT 2 BIT 1 BIT 0								Address (decimal)
D400	Voice 1 Frequency Control - (Low Byte)								54272
D401	Voice 1 Frequency Control - (High Byte)								54273
D402	Voice 1 Pulse Waveform Width - (Low Byte)								54274
D403	Voice 1 Pulse Waveform Width (most significant bits)								54275
D404	Random Noise Waveform	Pulse Waveform	Sawtooth Waveform	Triangle Waveform	Disable OSC 1	Ring MOD OSC 1 with OSC 3	Sync. OSC 1 with OSC 3	Gate Bit	54276
D405	Envelope 1 Attack Cycle Duration				Envelope 1 Decay Cycle Duration				54277
D406	Envelope 1 Sustain Cycle Duration				Envelope 1 Release Cycle Duration				54278
D407	Voice 2 Frequency Control - (Low Byte)								54279
D408	Voice 2 Frequency Control - (High Byte)								54280
D409	Voice 2 Pulse Waveform Width - (Low Byte)								54281
D40A	Voice 2 Pulse Waveform Width (most significant bits)								54282
D40B	Random Noise Waveform	Pulse Waveform	Sawtooth Waveform	Triangle Waveform	Disable OSC 2	Ring MOD OSC 2 with OSC 1	Sync. OSC 2 with OSC 1	Gate Bit	54283
D40C	Envelope 2 Attack Cycle Duration				Envelope 2 Decay Cycle Duration				54284
D40D	Envelope 2 Sustain Cycle Duration				Envelope 2 Release Cycle Duration				54285

Address (hex)	BIT 7 BIT 6 BIT 5 BIT 4 BIT 3 BIT 2 BIT 1 BIT 0								Address (decimal)
D40E	Voice 3 Frequency Control - (Low Byte)								54286
D40F	Voice 3 Frequency Control - (High Byte)								54287
D410	Voice 3 Pulse Waveform Width - (Low Byte)								54288
D411	Voice 3 Pulse Waveform Width (most significant bits)								54289
D412	Random Noise Waveform	Pulse Waveform	Sawtooth Waveform	Triangle Waveform	Disable OSC 3	Ring MOD OSC 3 with OSC 2	Sync. OSC 3 with OSC 2	Gate Bit	54290
D413	Envelope 3 Attack Cycle Duration				Envelope 3 Decay Cycle Duration				54291
D414	Envelope 3 Sustain Cycle Duration				Envelope 3 Release Cycle Duration				54292
D415	Filter Cutt Off Frequency (least significant bits)								54293
D416	Filter Cut Off Frequency - (High Byte)								54294
D417	Filter Resonance				Filter External Input	Voice 3 Output	Voice 2 Output	Voice 1 Output	54295
D418	Cut off voice 3 Output	Filter High Pass	Filter Band Pass	Filter Low Pass	Output Volume				54296
D419	Analogue/ Digital Converter 1								54297
D41A	Analogue/ Digital Converter 2								54298
D41B	Oscillator 3 Random Number Generator								54299
D41C	Envelope Generator 3 Output								54300

APPENDIX F

ABBREVIATIONS FOR BASIC KEYWORDS

When typing in a BASIC command or program line the majority of the BASIC keywords can be entered in an abbreviated form using the initial of the keyword followed by pressing the second letter of the keyword simultaneously with **SHIFT**. e.g. to obtain CHR\$ we would press C, **SHIFT** H. A program entered using abbreviations will be LISTed with the keywords in their expanded form.

The few exceptions to the rule are:

CLOSE	- C,L, SHIFT O	RETURN	- R,E, SHIFT T
GOSUB	- G,O, SHIFT S	STATUS	- S,T
LEFT\$	- L,E, SHIFT F	STR\$	-S,T, SHIFT R
PRINT	-?	TIME	-T,I
RESTORE	- R,E, SHIFT S	TIME\$	-T,I,\$

The following keywords cannot be abbreviated:

COS
FN
GET#
IF
INPUT
INT
LEN
LOG
NEW
ON
OR
POS
REM
TAN

APPENDIX G

SOFTWARE AVAILABLE FOR THE COMMODORE 64

Even at such an early stage in the life of the Commodore 64 there are numerous software packages available for such uses as accounting, club management, education, financial planning, games, graphics, mathematics, music composition, programming utilities, stock control, word processing, etc - the list is endless. To obtain an up to date view of what software is available one should refer to one of the many personal computer magazines.

The following brief notes on a section of Commodore software may be of some interest.

SIMON'S BASIC

SIMON'S BASIC (Commodore product No. SIB6410) has been designed to enable programmers of all levels to easily utilise the potential of their Commodore 64. There are three sections to the package:

- Toolkit: removes some of the more tedious aspects of computer programming
- Graphic and Sound Commands: provides some additional BASIC commands to use the graphic and sound feautre more easily - no more POKEing around in the VIC and SID's control registers.
- Structured Programming Commands: provides some additional BASIC commands that enable programs to be written in a structured form. This technique avoides the GOTO statement and as a result we do not get large programs with lots of jumps that are difficult to keep track of.

Other commands that are available enable:

- screen formatting
- input validation
- character string manipulation
- number system conversions
- screen scrolling
- and much more

The package is available on a cartridge which can be slotted on at the rear of the Commodore 64.

Other Computer Languages

In addition to BASIC, several other computer languages are available from Commodore on diskette.

COMAL - Common Algorithmic Language (Commodore Product No. CML 6440/6410) COMAL combines the ease of BASIC with the format of a structured language like ALGOL, Pascal etc. COMAL is an ideal language for programmers of all levels and because the programs are structured, long programs can be easily read. Also available in cartridge form.

PILOT - (Commodore Product No. PLT 6440) PILOT is a computer aided learning language which allows tutors to compose lessons on the computer for their pupils to answer through the keyboard.

LOGO - (Commodore Product No. LG 6440) LOGO is a computer graphic language which is often used in schools in a learning/teaching environment (PILOT and LOGO may be used in conjunction with each other)

PROGRAMMER'S UTILITIES

The programmer's utilities package (Commodore Product No. UTL 6440) contains a wide variety of useful programs for the BASIC and assembler programmer to get the most from the Commodore 64. The package, which is available on diskette, contains the following utilities:

- a number of disk handling commands
- a machine code monitor
- a user defined character editor
- a sprite editor
- a program to experiment with the SID

- a PET emulator which enables earlier versions of Commodore BASIC to be executed
- a screen editor with input validation
- and much more

MACHINE CODE TOOLS

A comprehensive list of Commodore software dealing with machine code on the Commodore 64 was given in Chapter 26.

DISK OPERATING SYSTEM SUPPORT

Provided with the VIC 1541 disk drive and with certain other Commodore packages is a program call the "C64-Wedge". Once loaded the Commodore 64 will appear to be as normal, with the exception of a few hundred less bytes, but the disk operating commands can be obtained more easily. The LOAD command is replaced by a "/" and the device numbers need not be specified. The "" and ">" characters are used in place of PRINT# to send disk commands to the disk drive. The disk directory is obtained by either '\$' or >\$' and if an error occurs this is signalled by the red LED, then '' or '>' will return the error message.

For a detailed up-to-date list of Commodore's educational, business and programming utility software contact:

Commodore Information Centre
675 Ajax Avenue
Slough
Berkshire
SL1 4BG

INDEX

A

ABS	57
Accumulator	276
Accuracy	19
ADSR	185
Addition	22
Address	111
Algebraic expression	21
Algorithm	25
Alphabetical order	45
Alphabetical sorting	93
AND	104
Animation	81
Argument	53
Arithmetic	19
Arithmetic and logic unit	4
Array	86
Arcsine	52
Arcosine	52
ASC	68
ASCII	68
ASCII codes	69
Assembler	279
Assignment statements	29
Attack	185
ATN - arctangent	60

B

BAM	241
Bank selection	123
BASIC	15
Binary notation	95
Binary tree	265
Bit	96

Bit mapped graphics	140
Block	224
Block allocation	241, 246
Block pointer	242
Boolean algebra	106
Brackets	22
Bubble sort	88
Buffer - cassette	224
Buffer - disk	242
Buffer - keyboard	13
Byte	96

C

Cassette buffer	224
Cassette interface	224
Character arrays	86
Character code	68
Character generator	118
Character set	118
Circular lists	261
CHR\$	68
Clear screen	49
CLOSE	29
CLR	49
CMD	29, 218
Collision detection	170
Colour display	128
Colour codes	128
Colour control characters	127
Colour memory	113
Complement - two's	99
Computer system	3
CONT	30
Control	4
Control statements	37
Control variables	41
Copy - screen to printer	223
COPY - disk command	214
COS - cosine	60
CTRL	77
Cursor	77
Cursor control characters	77

D

DATA	32
Data file	225
Datasette	206
DEF	65
DEL - delete key	77
DIM - dimension	47
Disk	207
Disk buffer	242
Disk drive	208
Disk operating system	210
Display code	131
Display screen	132
Division	22
DOS	210
Doubly linked lists	261
Dynamic data structure	254

E

Editing	17
END	49
Envelope	185
EOF - end of file	231
Error control, disks	214
Error reports	239
EXOR	106
EXP	55
Expansion ports	10
Exponential	20

F

Files - cassette	224
Files - disk	230
Files - sequential	224
Files - random access	240
Floppy disk	209
Flowchart	26
Formatting disks	210
FN	52

FOR ... NEXT	41
FRE	66
Functions	52

G

GET	36
GET#	225
GOSUB RETURN	45
GOTO	37
Graphs	264
Graphics - keys	13
Graphics - characters	13
Graphics - user defined	121

H

Hardware	3
Hashing	246
Heuristic programming	272
Hexadecimal notation	100
High resolution	139

I

IF ... THEN	44
Input	4
INPUT	36
INPUT#	225
Insertion sort	89
INT	53
Integer	19
Interpreter	112
Interrupts	177
I/O commands	36

J

Jiffy clock	83

K

Kernal	112
Key	13
Keyboard	13
Keyboard buffer	13

L

LEFT\$	69
LEN	69
LET	31
Linked list	255
LIST	31,218
LOAD	29
Logic	103
Logic Expressions	104
Logic Operators	104
Logic tables	104
Logical files	197
LOG	57

M

Machine code	275
Masking	109
Master file	225
Memory	111
Memory address	111
Memory map	111
MID\$	70
Mnemonics	277
Multiplication	22

N

Negation	22
Nesting of FOR loops	41
NEW - BASIC	31
NEW - disk command	213
NEXT	41
NOT	105
Number systems	95
Numerical arrays	86
Numerical variables	21

O

ON	47
OPEN	29
Operand	52
Operator	23
OR	104
Output	4

P

PEEK	66
PEEK/POKE codes	52
Peripherals	4
PI	61
Pitch	187
Pixels	126
Plotting techniques	139
POKE	131
PRINT	37
PRINT#	225
Printer	217
Pulse waves	181
Pulse width	181

Q

Queue structure	262
Quick sort	91

R

Radian	61
RAM	4, 111
Random access files	240
Random number generator	54
READ	32
Records	224
Register	178
Release	185
REM	50
RENAME - disk command	214
RESTORE - basic	32
RETURN - keyboard	226
Reverse video	14
RIGHT\$	270
RND	54
ROM	4, 111
RUN	29

S

SAVE	29, 207
Sawtooth wave	181
SCRATCH - disk command	214
Screen display	132
Screen editor	112
Screen memory	113, 117
Scrolling	155
Sector	210
Seed	54
Sequential files	224
SGN	57
Shell sort	90
SID	113
SIN - Sine	60
Software	6
Sorting - bubble	88
Sorting - insertion	89
Sorting - quick	91
Sorting - shell	90
Sorting - tree	265
Sound	179

Sound waves	179
SPC	52
Sprites	157
Sprite colours	159
Sprite position	160
Square root	57
Stack - structure	262
Static data structures	254
STATUS	66
STEP	41
STOP	50
STOP key	12
STR\$	52
String	21
String addition	68
String array	86
String function	68
String variables	68
Subscript	86
Subtraction	22
Sustain	185
Syntax	30
System commands	30
SYS	51

T

TAB	52
TAN - tangent	57
Three dimensional plots	150
TIME	83
TIME\$	84
Timing facilities	83
TO	41
Track	210
Tree structures	265
Triangle waves	181
Truth tables	104
Two's complement	99

U

Updating files	225
User defined graphics	121
USR	52

V

VAL	52
Variables - numeric	21
Variables - string	21
Variables - subscripted	86
VERIFY	29, 207
Vibrato	194
VIC-II	113, 115
VIC-1515 printer	217
VIC-1541 disk drive	208
Voice	190
Volume control	182

W

WAIT	51
White noise	182

A message from the publisher

Sigma Technical Press is a rapidly expanding British publisher. We work closely in conjunction with John Wiley & Sons Ltd. who provide excellent marketing and distribution facilities.

Would you like to join the winning team that published these highly successful books? Specifically, **could you successfully write a book that would be of interest to the new, mass computer market?**

Our most successful books are linked to particular computers, and we intend to pursue this policy. We see an immense market for books relating to such machines as:

DRAGON
THE BBC COMPUTER
APPLE
TANDY
SINCLAIR
OSBORNE
ATARI
IBM PC
SIRIUS
NEWBRAIN
COMMODORE
and many others

If you think you can write a book around one of these or any other popular computer — or on more general themes — we would like to hear from you.

Please write to: Graham Beech
Sigma Technical Press,
5 Alton Road,
Wilmslow,
Cheshire, SK9 5DY,
United Kingdom.

Or, telephone 0625-531035

SUCCESSFUL "64" PROGRAMMING — and much more!

Mark Harrison takes the complete newcomer through a comprehensive programming course right up to advanced projects. It's a meaty book, with contents that make clear all those things you never could understand in the manual.

You'll see how to produce dazzling graphics, "sprite" displays and super sounds! Once you've mastered your 64, you'll want to understand how to use disk drives, cassette recorders, printers and the other peripherals that turn your 64 into a system. The small business man will see — easily — how to design forms on the screen and to keep track of all his valuable business records.

Mark is one of our most successful authors. This is bound to be another sure-fire success for him.

Some Other Books you'll like:

Computer Based Learning: practical microcomputer methods: by Graham Beech

Maths + Computers = Fun: by Geoffrey Childs

Commodore 64 Mindstretchers, by Ian Creasey

Microcomputer Game Design: by Gordon Hamlett and Tony Hetherington

Learn New Skills with a 64 : by David and Philip McKirgan

And Lots more. In fact, we've got a catalogue of over 50 Super Titles — so, write today to:

Sigma Technical Press
5, Alton Road,
Wilmslow,
Cheshire SK9 5DY

ISBN 0 905104 59 5