DENNIS JARRETT

THE COMPLETE COMMODORE 64



A COMMODORE USER BOOK

THE COMPLETE COMMODORE 64

THE COMPLETE COMMODORE 64

By the same author

The Good Computing Book
The Electronic Office – A Management Guide
Getting The Most From Your Vic-20
The Sirius Book
The Apricot Book
The IBM Personal Computer Book

DENNIS JARRETT

THE COMPLETE COMMODORE 64

HUTCHINSON
London Melbourne Sydney Auckland Johannesburg

An HCP Product

A COMMODORE USER BOOK

Hutchinson & Co. (Publishers) Ltd

An imprint of the Hutchinson Publishing Group
17-21 Conway Street, London W1P 6JD

Hutchinson Publishing Group (Australia) Pty Ltd
PO Box 496, 16-21 Church Street, Hawthorne,
Melbourne, Victoria 3122

Hutchinson Group (NZ) Ltd
32-34 View Road, PO Box 40-086, Glenfield, Auckland 10

Hutchinson Group (SA) (PTY) Ltd
PO Box 337, Bergvlei 2012, South Africa

First published 1984

© Dennis Jarrett, 1984

Printed and bound in Great Britain by
Anchor Brendon Ltd, Tiptree, Essex

British Library Cataloguing in Publication Data

Jarrett, Dennis
The Complete Commodore 64
1. Commodore 64 (Computer)
I. Title
001.64'04 QA76.8.C64

ISBN 0-09-154981-7

A dedication

This one is for Mum and Dad, and not before time.

Contents

Section 0 Preliminaries

... Dedication
... Acknowledgements
... About this book

Section 1 Getting to know the 64

Chapter 1 Why Buy a 64? 15
Chapter 2 A Brief History of Commodore 18
Chapter 3 Setting Up 27
Chapter 4 A Tour of the Keyboard 36

Section 2 Programming

Chapter 5 First Steps in Programming 49
Chapter 6 Program Flow 54
Chapter 7 Variables 58
Chapter 8 Getting Something In 67
Chapter 9 Getting Something Out 72
Chapter 10 READ, DATA, RESTORE and Arrays 78
Chapter 11 Handling Characters 86
Chapter 12 Handling Numbers 92
Chapter 13 Handling Memory 98
Chapter 14 Graphics 105
Chapter 15 Sound 112
Chapter 16 Extensions to Basic 117
Chapter 17 Writing Faster Programs 124

Section 3 Extra hardware

Chapter 18 What's a File? 133
Chapter 19 Cassettes 139
Chapter 20 Disk 146
Chapter 21 Printers and Plotters 157

Section 4 The 64 in business

Chapter 22 Buying Business Software 179
Chapter 23 Filing Packages 183
Chapter 24 Spreadsheet Calculators 187
Chapter 25 Word Processing 193
Chapter 26 Business Accounting 198
Chapter 27 The SX 201

Section 5 Resource Directory

Chapter 28 Error Messages 209
Chapter 29 A Package of POKEs 212
Chapter 30 A Practical Glossary 219
Chapter 31 Now Read On 225

Section 6 Summaries

Appendix 1 Basic 231
Appendix 2 The Memory Map 239

Section 7 Index 255

About this book

Like most home computers today, the Commodore 64 is both simple and complicated. It's simple because you get up and running quickly with a 64; you can have it doing something for you within a few minutes of plugging the thing in. That's the scope of the little handbook provided with the 64.

But it's also complex because it is after all a computer – which means there probably isn't anyone anywhere who understands *everything* about it. You could spend several years discovering new things about the 64 ...

Commodore has thoughtfully made the 64 inherently more interesting from that point of view by building all kinds of idiosyncracies and awkwardnesses into the way the 64 operates. So not only is there a whole lot to learn about what it *can* do; there's much to find out about *how* it does it.

Another complication derives from the thing's sheer versatility. In part that is a measure of its success: so many people have bought the 64 that it's worthwhile for program-writers and electronics wizards to design, build and sell all kinds of software and hardware add-ons for this computer. The 64 probably has more of those extras available than any other single computer in the world, and the combination means that the 64 can be used for just about anything you might want a computer to do. The range of possibilities seems limitless – and that's another reason why it isn't necessarily a *simple* computer.

This book is an attempt to pull together all the different threads – or as many of them as I can locate ends for.

So Section 1 is a swift intro: doubtless it will repeat much of what the average 64 owner already knows about setting up and using the computer, but it adds some practical comments and tips based on my own hard-earned experience. You'll also get some background to Commodore and to the 64, and hopefully you'll start seeing how the 64 fits into the home computer business generally.

Section 2 is mostly about Basic (though it includes a Chapter on other programming languages). It's not exactly a tutorial, since I don't regard myself as a particularly good teacher (which in other words is a cop-out: I would have written a tutorial if I felt I was capable of doing so). Instead you get a discursive reference section talking about how to program, how to work with the individual Basic statements (with uses and limitations highlighted), and how to write better and/or faster programs. Again, you'll be getting the fruits of experience – my own and that garnered from many other people.

Section 3 starts getting into the heavier bits with a look at files and filing. Since that inevitably means getting closer to the way hardware works, I've followed it with Chapters on the 64's main peripherals – tape, disk and printers.

And then there's a lengthy chunk on using the 64 in business, particularly with reference to what kind of business packages you can buy for it. There's a chapter here on the rather odd SX-64 portable too.

Non-business extras come as part of the Section called 'Expanding your 64'. That's a kind of pot-pourri which looks at some of the more interesting and less predictable uses for the computer.

From there on the book is a collection of facts and comments without much rabbiting. Section 5 is a Resource Directory, a reference for many of the one-off considerations that you may or may not have. The Appendices are even more abrupt, being a collection of summaries relating to the 64.

And if you think that I've tried to cram too much into the one book, you're right. There's a case for writing a collection of books to cover the same ground, some look-up reference works and some more chatty read-in-the-bath coverage. But while using the 64 I got bored having a dozen separate manuals and books around the thing, so I tried to pull it all together. I've also tried to include as much practical experience as I have acquired.

Inevitably I'd be fooling me and you if I claimed that this is the last book you'll ever need to use a 64. But I'd hope that it goes some way in that direction.

Thanks page

Any book about any home computer that doesn't carry a substantial list of acknowledgements is to my mind suspect. Could be that the author is indeed a genius who has spent he previous year doing nothing else but finding out about the computer: I guess that's just about possible; but it's not likely. Home computers in general, and the Commodore 64 in particular, have a great deal of power ... which, when it comes down to it, means *complexity*. There's always something new to find out.

But one of the nice things about home computing is the implicit sense of belonging to a wide and diversified information-exchanging club. Which is a long-winded way of saying this book owes much to numerous sources of information, advice, enlightenment and sign-posting from conversation, magazine articles, other books.

It's almost inevitably the case that I've forgotten where some of those pointers and eye-openers came from. The problem is compounded somewhat because I see so many contributions from different authors to my own magazine **Commodore User**. I hope that in this book I haven't inadvertently ripped off anyone else's original thoughts or words: but I do acknowledge a specific debt to several people – notably **Jim Butterfield**, who manages to combine a broad and deep knowledge with a writing style that makes his articles the most reliable and the most readable exposition of the 64 that you'll find.

Running him a close second on the technical side are four more of the mag's regular contributors: **Mike Todd**, **Chris Preston**, **Chris Durham** and **Fred Reid**. And **Bohdan Buciak**'s wind-up–and-go enthusiasm for discovering New Facts is also reflected substantially in these pages. **Sean Harrington** read the typescript, a task I wouldn't wish on anyone.

There are others, of course, but those must be my principal acknowledgements. Perhaps a blanket 'thank you' to the Commodore user community will suffice to cover everyone else? It's a great pleasure to be part of that world.

I must also thank what you might call the Support Staff. For this book the A-Team includes several of the people at Paradox, notably **Nicky Chapman** who's my PA and did much of the legwork when things were going well and cuddling when things weren't; and of course my editor at the publishing house, Hutchinson's – **Doug Fox** never really got too heavy when the project's delivery date slipped wildly under the pressure of my other work. Thanks, Doug.

And thank you for buying the book. I hope you enjoy it; and I hope you get something useful from it, enough at least to justify the cost, time and effort you'll put into it. Any comments and criticisms you can send to me c/o Paradox Group Limited, The Metropolitan, Enfield Road, London N1 5AZ.

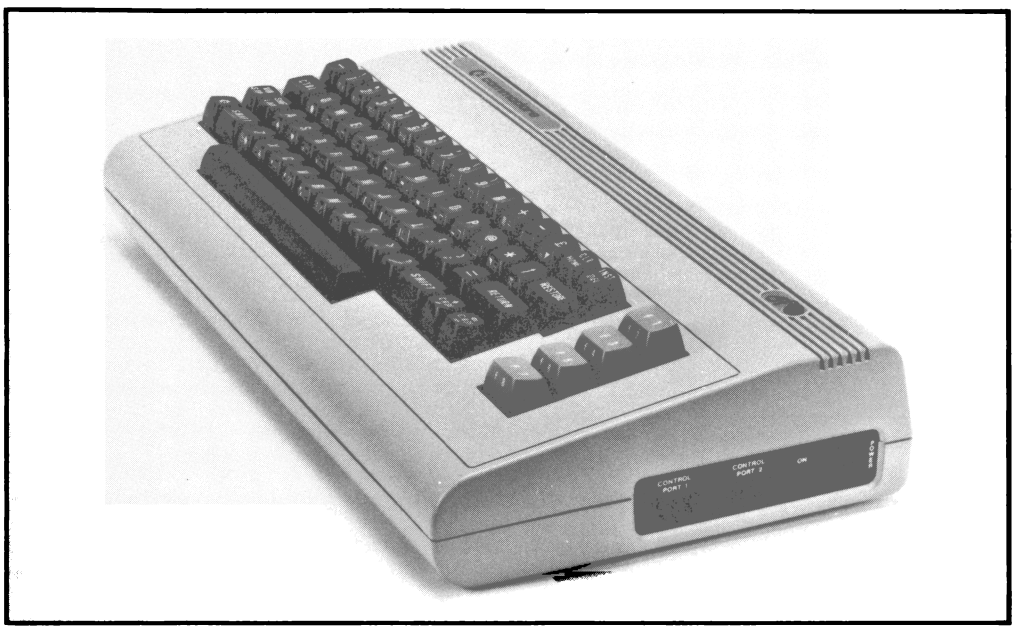
Dennis Jarrett
Islington, June 1984

Section 1

Getting to know the 64

CHAPTER 1
Why Buy a 64?

If Commodore's sales claims are to be believed, the 64 is the world's best-selling computer. Why is that so – particularly when there are several rivals (most of them British) that offer rather more for the same kind of money? And does the competition for the 64 mean that you should regret your decision to buy one?



At first sight the 64 looks rather like a brownish version of its predecessor, the Vic-20 (two million sold). But the similarities end with the layout of the keyboard and the design of the case.

Inside

The most notable features of the circuitry inside the 64 are three:

- its **memory** – a full 64K bytes' worth of memory chips, which is the maximum amount of memory that an eight-bit microprocessor can comfortably handle without complicated programming.

Several of the 64's competitors don't give you as much as 64KB; but on the other hand a lot of that memory isn't normally available to you as the 64 itself takes up a good deal of it. You're left with about 38KB, which is more than some rivals and less than others

- the **microprocessor** Commodore used is the MOS Technology 6510 (MOS Technology is a chip-maker owned by Commodore).

No other computers use this micro; it is however derived from the 6502 as used in the Vic, the Atari 400/800 and other members of the Atari family, and the Apple II. In fact the 6510 has the same machine-language instruction as the 6502, which means programmers *didn't have to relearn anything about machine-code programming when moving on to the 64 from the Vic.*

The difference is simply the addition of some extra integrated circuits on the chip that allow it rather cleverly to flip between different types of memory for certain parts of its addressable range – in some cases it expects certain addresses to be in RAM and part of the user's program, in other circumstances it uses pre-specified routines in ROM for those addresses. That way the 64 can in fact have a total of 84K bytes of memory chips inside it; only 64K's worth is in use at any one time.

- the VIC (Video Interface Chip) isn't so remarkable – the kind of graphics and sprite effects it can produce are shared by several other computers. But the **SID (Sound Interface Device) chip** is something rather special among music synthesisers and sound effects generators on home computers.

SID allows the 64 to produce three independently-controlled 'voices', which means for instance that the computer can be programmed to play three-part music. Each voice can have a nine-octave range. You also get that on a fourth 'voice', a 'noise' generator that's useful for percussive effects in music and sounds like surf or explosions.

That alone isn't exactly remarkable these days; what is unusual is the full sound synthesis features built into the SID chip. You get a full ADSR envelope on each sound (which means you can specify very precisely how the sound is actually played). And you can set up different waveforms for each voice to have them sound like different instruments; on one waveform you can also set ring modulation effects.

The result is a quite staggering capability in terms of variety and precision in sound generation, still ahead of virtually every other home computer and comparable in terms of musical control to some professional sound synthesisers.

Outside

The initial impression of the 64 will probably be its **keyboard**. When the 64 was launched this keyboard was streets ahead of comparably-priced home computers in terms both of its layout (more or less like that on a typewriter) and its 'feel' – touch-typists generally find it a bit too light and non-positive, but to most of the rest of us it was a revelation.

Today there are several rivals with keyboards that 'feel' better, and the 64's layout does have some oddities about it (like using only two keys for cursor control rather than four). But considering the constraints of space and money, the 64's designers have done a good job overall.

The chubby look of the **casing** is not to everyone's taste, particularly if your preference is for the lower and deeper design of something like the BBC computer. Personally I like it: being relatively high means the keys are at a pretty comfortable height, and the keyboard is physically quite small and takes up little space on a desk or table (so it can more easily be positioned to suit the way you want to work).

The rounded edges and unused space around the key layout gives you somewhere to rest your palms comfortably; and the fashionable (but highly relevant) concern for good ergonomics is seen in the way the whole thing is raked slightly to present the keys at the best angle, with the tops of the keys themselves 'dished' so that the fingertips get a positive feel on the keys.

The other notable feature of the external view of a 64 is the number of **sockets** you get. Two for joysticks, for instance: that means you can have two-player joystick games running. These can alternatively be used to attach other things.

Around the back there's also sockets for plug-in ROM cartridges (which automatically occupy some of that 'switchable' memory; a TV connector; a DIN-plug socket for audio input and output and video output; the serial port, used to connect the 64 to a printer, a disk drive, and some of the other extras you might buy (it can also handle a type of RS232 communications, RS232 being one of the standard methods of connecting devices to computers); a socket for a cassette unit (only Commodore's own tape decks attach easily, though); and the 'user port', which with some esoteric programming and advanced electronics can control other devices.

That's a lot of connectability for one computer.

Using the 64

Strong features of the 64 that emerge when you start to use it include the very clever **screen editing** capabilities, whereby you can quite easily (and quickly) amend programs and enter commands. This is hardly unique among home computers these days, but it's worth noting that Commodore was one of the pioneers in screen editing.

The other key attribute is the provision of **Basic**, and not a new version of Basic either. The fact is that Commodore Basic version 2.0 as used on the Vic has been around for a while on Commodore's computers; and there are pros and cons to that. On the plus side we can count the virtues of having a tried and tested programming language that is more or less free of the kind of bugs that inevitably appear in any new and unproven software.

On the other hand ...

The minus side has a major entry for that Basic, too. The 64 may be loaded with advanced facilities like sprite graphics and its sound synthesis: but Commodore hasn't provided us with a version of Basic that allows easy access to and use of those capabilities.

Not as standard, anyhow: there are extra-cost plug-in cartridges and software on disk or tape that do add such commands to the 64's Basic. But sound and graphics commands *are* provided as standard on most of the other home computers that compete with the 64.

Instead 64 owners have to pay more – or struggle with a lot of additional (and in some cases quite

complex) programming in order to use those undeniably excellent capabilities.

The 64 also omits some of the more advanced Basic commands, particularly those that help out with complex screen displays and printing: things like PRINT USING, where you can pre-specify a format in which you want something printed. Notable absentees also include ON ERROR commands, which provide a relatively simple but really useful way of telling the computer to do something when it spots an error in the user's input.

Another group of statements not to be found on the 64 are those that have especial relevance to what's called 'structured programming' – conditional branches, for instance. Nor will you find on the 64 the kind of built-in 'programmer's aid' commands like automatic line numbering and renumbering that many of its rivals feature. Other program development facilities you can expect from the competition include deletion of groups of lines, search-and-replace facilities for quick and easy changes, helpful listings of the variables your program has set up so that you can check them, a HELP command to show you exactly where and why a program stopped dead ...

The collection of sockets also have their limitations. Just about every printer available apart from Commodore's uses one of two standard plug-and-socket connections, 'Centronics parallel' or 'RS232 serial'. RS232 can also be used to connect a lot of other devices. The 64 has neither of these (well, it does have a bastardised version of RS232: but it doesn't really work like standard RS232).

Instead it uses a different method of associating wires with electrical signals, the IEEE standard ... except that the 64's IEEE isn't like other people's idea of IEEE.

The result? Only devices sold by Commodore – and those from manufacturers who are keen enough to follow Commodore's standards – will attach easily to the 64. If you want to use anything else, you'll have to pay for a converter box of some kind.

The killer punch

So there are arguments for and against the 64. The quality of its keyboard and the excellence of its sound and graphics facilities probably tip the balance in its favour by comparison with many of its competitors ... but not all.

The most telling factor in the 64's favour, however, doesn't have much to do with the thing's technical specification. There are so many 64s around now that the independent manufacturers and software producers regard it as worthwhile to produce programs and add-on hardware for the 64: and the result is that the 64 probably has a wider range of options and extras available than any other computer.

That fact alone outweighs most of the minus points. You can overcome virtually all of the 64's technical deficiencies by buying converters and amendment packages: you can find programs to do just about anything you want to do, from adventure games to astrological calculation via business accounting, word processing, spreadsheet calculators, database packages ...

And those products are generally very good, too. That's partly because they are able to take advantage of the built-in attributes of the 64 – like its good ways with colour and graphics, the availability of function keys for single-key command inputs, the sound chip.

But more important is the question of competition: a lot of 64s means a lot of suppliers, and to be successful the independent 64-oriented suppliers really have to look to their laurels. Besides, the more users there are the more feedback there is which the suppliers can use to improve their products.

To say you should feel confident about buying a 64 because a lot of other people have done so may sound a bit like damning with faint praise. In truth the 64 is a pretty good little computer. But it's the availability of all that independent material which elevates it to star status.

CHAPTER 2
A Brief History of Commodore

Commodore is currently the world's largest supplier of computers, with something approaching four million users around the world. How did it get to be that big?

Plug-in-and-go computing

Jack Tramiel was old by the standards of the micro business when Commodore launched a pioneering computer called the Pet (it stood for 'Personal Electronic Transactor', but they thought of that after they'd decided to call it the Pet anyway. "It was a time when "Pet Rocks' were all the craze' said the Pet's designer Chuck Peddle later. "We just wanted to associate with that.")

Tramiel had been born in Poland of Jewish parents and survived the nastiest the Nazis could be towards Polish jews – including a spell in Auschwitz. Liberated by the Americans, he made his way to New York and drove a cab for a time before joining in the US Army. While in uniform he supervised the maintenance of office equipment the Army had in New York; and when he left the service he joined a typewriter repair firm.

That didn't last long: "they didn't reward me well enough for the work I'd done, so I decided that setting up my own company was the only solution". Self-employed typewriter maintenance went well, and he also got a hefty contract to supply some new typewriters as well as some mechanical adding machines.

He founded Commodore in Toronto in 1958 as the Commodore Portable Typewriter Company, and the next year it began selling its own 'Everest' brand typewriters. In 1961 it also started importing mechanical adding machines from Japan; and clearly it saw the way the electronic wind was blowing.

So by the late 1960s Commodore was still heavily into all kinds of office thingies, including furniture (in 1962 it bought an Ontario manufacturer of office desks). It nearly folded in 1965, but a sharp-eyed financier called Irving Gould stepped in to become chairman and principal shareholder. He provided some funds and apparently kicked the company into another gear – and into more electronics. Commodore started operating all over North America as importer of and middleman for a line of Japanese calculators. In 1967 it signed the most important of those distributorship deals, with Ricoh for adding machines and with Casio for the AL-1000 electronic calculator – a bulky beast, retail price just under $1,500.

That in turn lead to Commodore making its own calculators: first appearance 1969, major appeal their low price. I've still got a Commodore pocket calculator – cost me £3.99 at Tesco five years ago. The keys are a bit small and the display is readable only at certain angles, but it *is* a five-function calculator with a memory; and I use it all the time.

Commodore's calculators got cleverer and cheaper, especially its 'scientific' models, and it got really into mass-marketing with the C108 – a bargain at $200 (you got five cents change too).

But by 1974 the competition was really hot and the price war had brought several of its calculator competitors to their knees. And in the following year, things got worse. Chip prices dropped from $12 to $1; Commodore found itself sitting on a lot of chips purchased at the old price while some competitors were able to buy at the new price. And then Texas Instruments – which had been Commodore's principal supplier of chips – decided it would go into the calculator business itself, which meant it would be keeping most of its production for its own products. One in the eye for a smallish manufacturer who relied on regular supplies of components, particularly if that company had large stocks of expensively-purchased chips and suddenly uncompetitive made-up products.

But rather than pull in its horns, Commodore (or more likely Tramiel himself) characteristically decided to tackle the problem in a big way. Said Big Way consisted of 'vertical integration', which meant making *everything* it needed itself – and that in turn meant setting up factories and/or buying all of the component suppliers for its calculators. This way it could control the source of supply, rely on no-one, minimise costs, and spread expenditures over whatever period it felt like. So Tramiel cancelled all his outstanding contracts with suppliers, won 22 lawsuits they brought against him, and went looking for companies to buy.

The integration included a small electronics manufacturer called MOS Technology, brought into the fold during 1976. It was followed within 18 months by acquisition of a small but complementary Los Angeles chip maker called Frontier Manufacturing and a Texas-based firm called Micro Display Systems which made liquid crystal displays.

Those two illustrate Commodore's thinking at the time: it had an immediate requirement for chips, so buying Frontier made sense – Frontier's expertise was in low-power CMOS and that fitted well with MOS Technology's skill in NMOS chips. But Commodore had left the calculator market and was getting out of a short flirtation with digital watches, so MDS must represent an investment in the future. Commodore still doesn't have a mainline mass-market product that uses liquid crystal displays (which typically appear in watches and calculators as an alternative to LEDs). And as one of the official PR blurb-writers put it, "This expansion programme meant that Commodore had now acquired in-house expertise and production in more key technologies than many much larger companies".

There was one product in which Frontier's chips and the MDS displays did actually appear, but it was all too brief. The stillborn HHC-4 was a Tandy/Panasonic type hand-held 'computer' originally promised for an Autumn 1981 launch and shown finally at US shows in Spring 1983. The insides were CMOS chips, the outsides included a single-line 24-character liquid crystal display.

Meanwhile MOS Technology has been booming. Currently it's making 60,000 silicon wafers a month (they don't all pass the quality control tests, but then that's the norm in chip-making) and the 1983 Annual Reports reckons this means it's North America's fifth largest in-house semiconductor manufacturer.

Meanwhile back in 1976 Commodore moved out of Canada to the Bahamas, something of a offshore tax haven (and anyhow the weather's better than Toronto). Today the financial HQ is still in the islands; operations and manufacturing remain on the mainland, in Pennsylvania.

At the time of its acquisition by Commodore, MOS Technology happened to be working on an eight-bit micro called the MOS 6502; more by good fortune than any inherent excellence, the future of the 6502 processor was further assured by the way Apple and Atari both espoused the new chip from the new chip-maker.

MOS Technology also happened to employ a very bright electronics engineer named Chuck Peddle. Peddle it was who saw the appeal of a microcomputer designed for the average home buyer rather than for the electronics hobbyist or the programming nut. He'd already designed the 6502 itself.

The Apple story captures the flavour of the times – Steve Wozniak, 23 years old at the time and one of a dedicated band of enthusiasts In California called the Homebrew Computer Club, was keen to design his own computer. He bumped into Peddle at a 1975 electronics trade fair; Peddle was selling the brand-new MOS 6502 for $20 apiece. As Wozniak put it, "It wasn't like trying to think "Which microprocessor will make a better computer?' – it was sort of like, 'Which one can you get for twenty bucks?" All the alternatives cost over $150, so he bought a 6502 and built the Apple I. Someone else persuaded him to put that bare-board computer into a neat box, and he called it the Apple II ...

The Pet computer?

MOS Technology did cover the wild-eyed late-night enthusiast fringe with the KIM, a single-board 6502-based kit. But the Commodore Pet was the prime result of Peddle's inspiration, launched just after the KIM in 1977 and also built around the 6502.

The Pet was actually designed for Tandy to sell, but the leather goods specialist – which was just getting stuck into electronics shops – opted to go for a rather different style of computer, the machine that became the TRS-80.

In fact, Jack Tramiel's interest in the Pet was pretty nebulous. He subsequently put it like this: "Our aim is to sell components. To be able to sell those components, sometimes we have to have end products. We decided at that time to produce a product ... we still didn't know specifically what the computer was going to be used for". One appearance at the 1977 Consumer Electronics Show in Chicago pulled $3 million worth of orders for the Pet in two months, though, so Commodore knew it had a product that would sell anyhow.

And to some extent it was the Pet that sparked off the boom in 'personal' computers. It was ugly, had an awful calculator-style keyboard, came with a mere 8KB of memory, and displayed only 40 characters per line on its built-in screen. But you could take it home, unpack it, plug it into a wall socket, and start using it right away.

Commodore continued to sell components, and during 1978 reckoned to have supplied something or other inside more than half the watches made in Hong Kong that year. But computers, and home computers, were increasingly where the company made its real money.

In 1979 Commodore added its first peripherals, disk drives and printers which it bought in from the manufacturers (mostly Japanese) and rebadged with the Commodore logo.

In 1980 the Commodore 4000 and 8000 appeared – Pets with no-built-in cassette unit, a 'real' screen with 24 lines of 40 or 80 characters respectively, a 'real' keyboard (not *very* typewriter-like, but better than the Pet's keys), 32K memory, and an improved version of the Pet's Basic.

But the major milestone of that year was the launch of the Vic. It was even more of a plug-in-and-go computer, and it was a lot cheaper: significantly it was shown to the world first at the giant US National Computer Convention but it was first offered to the punters at the Seibu department store in Tokyo ("the Japanese are coming", said Tramiel; "therefore, we must become the Japanese").

Again, the Vic wasn't an especially innovatory product: it wasn't the first, the cheapest, or the most powerful home computer of its time. But it got near the mark on all three counts – and it did benefit from having neat packaging with a (comparatively) excellent keyboard. And a very good marketing campaign, too, which has taken this machine to over

two million sales worldwide in four years ... despite it modest memory endowment, despite the inadequacy of its Basic, despite the largely non-standard attachment points provided for externally-connected goodies.

The Commodore 64 was launched in August 1982. It retained two of those minus points; but it was the first home computer to feature 64K bytes of memory, the theoretical maximum that you can use easily with an eight-bit microprocessor. Well, you can't actually use it – the computer itself takes up all but 38K of the total. But at least it's there.

By the end of 1982 the 64 had already overtaken monthly sales of the Apple II, the yardstick everyone used to judge the success or otherwise of a small computer. Mind you, the 64 did have the benefit of Commodore's largest ever advertising campaign ...

In 1983 Commodore hit sales of over $680 million, a sixth successive year of record growth that took it to the Number One spot in terms of numbers of computers sold around the world. The *64 Programmer's Reference Guide* has for some time been the top-selling computer book in the States.

Commodore in Europe

Europe had always been the base of Commodore's success, with sales there of the Pet and then the Vic outstripping North American shipments for a long while.

That's partly due to the energetic activities of the marketing people in Britain and Germany in particular, partly as a result of the much less intense competition on this side of the Atlantic (Commodore was one of the first micro makers to treat the world as its market rather than something that happens more or less incidentally the other side of Long Island).

It opened a factory in Braunschweig, West Germany, to help meet the demand from Europe. Initially building Pets, this quickly took on Vics and then 64s. Why Germany? Because it had had a typewriter-manufacturing plant in Berlin back in the 1960s, having bought out an office machine company there in 1962; and because the financial incentives were better than anywhere else in Europe.

Towards the end of 1984 Commodore announced it was starting work on a factory and new headquarters building in the ex-steelworks town of Corby, Northants. It received a chunk of public money for this – £2 million in grants from the British Government's Office and Service Industry Scheme and another £4 million or so in loans from the ECC's European Coal and Steel Agency. Both sources of cash are intended to alleviate the problems of steel closures by tempting employers into industrially-depressed areas.

The total investment at Corby should top £20 million by the time it's finished. Commodore reckons to double its UK staffing to 600 by the time the facility goes fully live in the Autumn of 1984. Actual manufacturing reportedly got under way in the Spring ("over 5,000 computers a day") and some support people have also been based there for a while; the final target is 1,000 jobs within two years and 200,000 micros a month (with the capacity to churn out 350,000 per month).

During 1983 Commodore also opened extra manufacturing facilities in Canada, the USA, and Hong Kong. With Corby, that lot took the company's worldwide floorspace for manufacturing and warehousing from 600,000 square feet to 1,500,000.

In February 1984 Commodore received the Royal Warrant – which means for the next ten years it can describe itself as "By Appointment to Her Majesty The Queen, manufacturers of Computer Business Systems". All a bit ironic, really, since Commodore's major success in terms both of sales and in quality of products hasn't been in 'Computer Business Systems' at all; a Royal Warrant for home computers I'd understand, but Commodore has never been big in *business* computers – not least for the very good reason that other people have had better business systems.

So why didn't someone like Sinclair, or ICL, or ACT get the Royal Appointment? Seems they just didn't apply for it at the right time ...

Up to date

The vertical integration seems to have stopped now. It's probably no longer necessary; most microcomputer companies realise that as long as they can field the designers and guarantee the size of their sales, they're in a a very strong position to buy in technology from specialists without actually having to find something for those people to do in the intervening slack periods.

Licencing is now the name of Commodore's game; it's doing deals for hardware (like a joint venture with the Japanese firm Mitsumi Electric to design and build floppy disk drives), for components (a licencing deal with Zilog to manufacture the Z8000 16-bit processor and another with Micron Technology on 64K RAM memory chips), for operating systems (the Unix-like COHERENT from Mark Williams Company as well as the *de facto* standards on 16-bit micros MS-DOS from Microsoft and the CP/M family from Digital Research), and for packages (numerous examples, including the British word processor Easyscript from Precision Software).

Then Tramiel resigned suddenly. Jack Tramiel had built Commodore from a tiny typewriter-sales operation into a substantial company. He was (to say the least) a somewhat volatile character, disinclined to suffer fools gladly (prime examples of fools being managers who didn't make their sales targets and computer-press journalists).

The most extreme illustration of his less than deft way with underlings was probably the extraordinarily acrimonious departure of Chuck Peddle at Christmas 1981 under circumstances about which neither man would say anything printable; it seems though to have been a row about future product development, with Tramiel stomping on Peddle's ideas about building an up-market business microcomputer not unlike the Sirius.

Peddle left, saying he'd been sacked. Tramiel wished him good riddance and said he'd quit. Commodore successfully went to law for the return of 162,000 shares that (as Commodore puts it) were "improperly received and retained" on the basis of a share-purchase option. Peddle has been countersuing, alleging breach of contract and claiming no less than $152,500,000 in damages (well, why think small?). Meanwhile Peddle set up his own company and *did* build the Sirius ...

Tramiel had often threatened to quit as President and Chief Executive Officer, but Chairman Gould had reportedly done wonders with the emollients. In January 1984 though Gould didn't work too hard to counter Tramiel's latest resignation threat: reputedly he already had a replacement in mind, an accountant unknown to Wall Street or the computer industry called Marshall F Smith. Smith was appointed two days after Tramiel's official resignation.

Tramiel left Commodore holding about seven per cent of Commodore's shares, valued at around $100 million at the time. He's said he will be selling some of them, but in any case gets a deal to stay on as independent 'consultant' – Gould's official statement complimented him on putting into place "strong and talented" management while commending the way he had "served the company brilliantly".

No-one is saying exactly what caused the bust-up. Tramiel himself attributed it to "personal reasons" and said he'd do "everything possible" to help what he termed the "transition".

In the wake of Tramiel's resignation, four other senior executives also quit. Don Richard, acting President of Commodore's US operation, had been Tramiel's right-hand man for some years. Also departed are marketing vice-president Myrrdin Jones, systems engineering director Bill Miller, and Roy Thomas – director of materials in charge of US chip-making and computer assembly.

Commodore said the resignations are unconnected with Tramiel's departure. Jones, who is now at Sperry, has contradicted that. Meanwhile Tramiel went quiet for a couple of months – then popped up with a successful bid for Atari, once one of the market leaders in home computing but more recently a bit of dog for Warner Bros. Reportedly one of the companies he beat to the acquisition was Commodore itself. Now, Atari's current products sell in direct competition with Commodore's home computers: the next couple of years should be fun for Tramiel-watchers ...

The products that aren't

All companies fly a few kites from time to time. They start developing products which they decide not to proceed with; they launch and start trying to sell stuff which subsequently is withdrawn.

But Commodore has a quite extraordinary reputation for announcing products in a blaze of publicity – and then either changing the name or dropping them. Characteristically, it does both.

One result of this is the scepticism with which an unbiased observer might regard all claims from the company about its ability to predict or populate its future in computing. It does seem an expensive way to conduct market research and/or indulge the whims of its bosses ...

So I offer this catalogue of samples as an object lesson. Most of these are simply non-existent; some do have a ghostly impermanence about them that suggests you should actually touch the thing before you write the cheque.

- **The Colour Pet**: Back in early 1981 a full-colour version of the determinedly monochrome Pet was being developed in Canada by BMB Compuscience, a Canadian company that had developed an early disk-sharing inter-Pet local network system of modest ambition but effective enough operation called Mupet. A prototype was shown at the Hanover Fair that April ... and promptly disappeared for good.

- **The HHC-4**: This Tandy/Panasonic-type hand-held 'computer' was first promised for an Autumn 1981 launch but showed up eventually at US computer shows in Spring 1983.

The US price quoted was $199; for that you would have got something that looked like a king-size calculator, with 'chiclet' calculator-style keys (QWERTY plus a separate numeric keypad) and a single-line 24-character liquid crystal display. Inside there was 4K RAM, 1K of which was taken by the system; it could be expanded to 16K. There was also 20KB of ROM. An extra-cost plug-in connector box included a built-in mini-printer and sockets for cassette and TV.

It might have sold at that price, but not many people bought hand-held computers and the fad fizzled out. Commodore's entrant never reached the shops.

- **Commodore 9000**: Another 1981 flag-waver, also known as the 'Micromainframe' and 'MMF9000' and the 'SuperPet'. It was another two-microprocessor design – it included a Motorola M6809 as well as a 6502, had 192KB RAM and 36KB ROM, featured a built-in RS232 interface for the first time on a Commodore computer, offered compatibility with existing CBM 8000 peripherals and software, and looked like an 8000 since it lived in the same kind of box.

In addition though it also ran mainframe-type languages like Fortran, APL, Pascal and a Business Basic – all deriving from Waterloo University in Canada, home of much advanced Fortran work for micros and reportedly the effective designers of the 9000.

The main target audience was expected to be mainframe programmers who wanted a lowish-cost 'personal' computer on which to develop programs that could then be transferred to run live on big machines.

It is still featured in Commodore's price list (at £795 – no disks included) but it's said to be still stuttering around various back rooms of Commodore's software development places around the world. Probably waiting for someone to decide just why on earth anyone would want to buy it.

- **Vic 10**: At the Spring 1982 Commodore Show in London Commodore was blithely talking about five new computers using the then-new 6509 processor. They were promoted enthusiastically in the Show's Catalogue; and the 64 did indeed appear. The Vic 10 was the most notable of the absentees from Commodore's stands.

The Vic 10 – also known in its short but troubled life as 'Max' and 'Ultimax'– was described as "a games computer and music synthesiser ... due in Britain in September".

It was to be a Sinclair competitor of sorts, a lightweight slab-like little number flat computer using an awful flat membrane keyboard laid out more or less like a 64's. It had only 2K of RAM and to do anything on it you had to buy cartridges to plug in; that even applied to the Basic. Still, you were going to get a 64-style approach to sound, graphics and colour; the SID chip with three voices and ADSR capability was standard, the thing had a resolution of 320x200 for graphics, and text was shown as 40 lines of up to 25 characters – all of which were features of the 64.

Prices quoted were around £100 in the UK, $180 in the States.

In the late Summer of 1983 Commodore finally discarded the product and filed a $30 million suit against a Massachusetts company called AMP Keyboard Technologies. AMP was making the Max/Ultimax/Vic 10 keyboards, and Commodore alleged that the high rate of defective units meant it had had to scrap the whole project after building 35,000 of the things.

- **Vic 30**: Another of the Spring 1982 promises, characterised as "the natural progression from the Vic" – or more specifically from "the computing power of the Vic 20 and the colour and capabilities of the Vic 10".

The Vic 30 was basically a 16KB 64 to sell at around £250 (when the 64 itself was slated to cost something like £400). It was originally called the Super Vic when previewed in the States back in January 1982, but the name had changed by the Spring: the Commodore 16 that has been shown a year later is obviously the latest incarnation of it ...

- **Vic 40**: This was the original name for the Commodore 64, and the one it was launched under. By the time the thing was reaching the retailers the name had changed.

- **700 Series:** Also known (in North America) as the 'B' Series, and mellifluously described as "Commodore's handsome new computer for business use". The most impressive thing about the 700 is its styling, all low-drag-factor curves and swirls. Reputedly it was styled in Germany by Porsche, partly to improve on the ageing US-derived casing first used for the Pet and partly to meet European ergonomic standards (which among other things call for word processing machines to have a detachable keyboard – which the Pet-based boxes don't have).

Two models of the 700 were announced in Spring 1982 with deliveries promised for September of that year. The 720 was to be the principal 700 family member, with twin disks (only 170KB apiece – a bit less than competitive when compared with other eight-bit business microcomputers) and 256KB to 896KB of memory (which is more than most, and in theory allows the 700s to run much more complex programs). Its retail price was to be £1,595 in Britain.

The 720 was basically what the States was offered as the B256, though there was another model called the BX256 – the spec for that US-only product included the 8088 as standard in its $2,995 price tag.

US punters were also shown the B128 at $1,695, the same as the 710 in Europe – a similar machine but with 128KB memory and no second processor. You also don't get disks included in the UK price of £995. At least the 710 is in the current Commodore catalogue, along with the 700B (same system, no monitor: £695) and the 715B (same system but 256KB: £1,195). The 720 is ostentatiously omitted.

There's a professional businessy feel to the 700 Series keyboard – 94 keys including a numeric pad and 10 Function keys – and to the screen too (an 80x25 display for text). You also got a real IEEE interface and a real RS232 port too, so plugging in printers and the like from vendors other than Commodore wasn't too difficult.

The interesting option however was for a second processor. All 700s can have a plug-in Zilog Z80 running the CP/M operating system; in theory that means your Commodore computer can run any of the zillion CP/M programs around. Naturally it ain't that simple, since practically none of those CP/M programs is available in a format that Commodore disk drives can read.

And some 700 models were also going to get a second processor as standard, *in addition to* the

Z80-plus-CP/M option. Said second processor would be an Intel 8088, as used in the IBM PC, the Sirius, and a dozen other business micros claiming 16-bit operation; for the 700 it could be supplied (at extra cost) with one of the 16-bit operating systems – MS-DOS, CP/M-86, or the clever multi-tasking Concurrent CP/M-86. So you could run standard 16-bit programs or standard Commodore programs as you felt the need. Assuming you could get the software, and you couldn't.

In fact you couldn't even get the 16-bit operating systems to *run* the software.

Currently some 720s *are* reportedly getting through to dealers, but not with the add-on processor options. Seems there's been no interest in standard CP/M-80 for the eight-bit Z80 add-on and problems in getting MS-DOS and CP/M-86 to run on the 8088 cartridge.

Anyhow, Commodore doesn't really push business systems too hard; so the 700 is not making much of an impact. Besides, Commodore seems keener to shift old 8000s in new Porsche/700-style clothes – notably a bargain-basement version of the 96KB member of the Commodore 8000, the 8096D, which has the built-in floppies featured on the 700s (standard 8000s need a separate cable-connected box for a pair of floppy disk drives).

Why punt the 8000 around now, particularly when it's obviously outmoded in technical terms even by Commodore's own products? There's a lot of software ready to go for the 8000s; not much is available for the 700s (in theory the 700 line will run 8000 software but in practice some conversion is necessary). Besides, it's cheap.

The 700s came with the standard Commodore Basic 4.0 from the 8000 series, Pet-style block graphics but no high-resolution capability, a lot of memory (128 or 256KB RAM for starters, only 3K of which was taken for the system: and you could have up to 61K for user programs in Basic with the balance for data and variables, or the whole 61 to 253KB range was available if you eschewed Basic and ran only machine-code programs).

- **Commodore 500**: "The scientific/professional computer" is the consensus from various Commodore PR blurbs, though one catch-all copywriter did fall back on the meaninglessly comprehensive "ideal for small businesses, scientific applications, education, and entertainment". That should cover it, John.

The 500 was also known before its launch by the commendably modest appellation 'Commodore 2', but it arrived in the USA as the P Series. Launched at the same time as the 700, the 500 was effectively a 700 without disks or the built-on screen (it used an external user-supplied monitor or a TV set to display 40x25 characters) but including the 700's 128-to-896KB memory range. And from the 64 it borrowed the 16 colours, high-resolution graphics, SID sound chip, and joystick-type control ports.

Starter price for a 510 system with 128KB was to be £595 in the UK, $995 US (later repriced down to $795).

The 500 was one of a batch of new products promoted vigorously around the Spring of 1982, but a year later it seemed to have been dropped. Or rather it wasn't being promoted any more: without actually saying the 500 had been killed, Commodore people were talking about a "specialised" demand for scientific and process control applications rather than the mass market originally promised.

Part of the reason for that, according to some dealers, was the number of bugs in its operating system and the absence of anyone interested in fixing them. Nor was there any applications software forthcoming; and to cap it all the thing had no worthwhile manuals. You really had to be keen to want to use one ...

- **Commodore 104**: Not too much is known about this one. Half-hearted rumours circulated briefly in 1982, leading to much ill-informed press comment. The 104 sounded great. It was said to use a new MOS Technology processor called the 0104 which offered a degree of microprogrammability (as is found on fancier minis, whereby you can add your own machine-language instructions to the thing) and the (equally advanced) ability to handle some 16- and 32-bit operations even though it was still basically an eight-bit micro.

That latter feature meant that the chip could cope with massive memory ranges – from 1MB to 16MB. It was also reported to steam along at 8MHz, which is **very** fast for any micro ... particularly by comparison with the feeble 1MHz of the 64.

Just about everything in the 104 would be software-controllable. So programmers could for instance alter the character set being used and the relationship of keys to characters they produced. The screen layout was also to be software-driven, with the highest possible resolution being 1056x480 (a very funny size) and normal text displayed as anything between 20x10 characters per screen to 132x60.

A second processor – a conventional 6502 – would look after the screen display and manage the screen memory, so that wouldn't impinge on the space available for user programs. That presumably made the screen-handling both versatile and very quick.

The other exciting claim for the 104 was the availability of plug-in ROM 'emulation packs' that could make it behave like a whole variety of other computers – thus enabling it to run software from them. The emulators available were said to include all the computers Commodore itself had built, the

Superbrain, Atari 800, and Apple II and III; "coming soon" were British-developed emulators for the Sinclair ZX81 (!) and the IBM 360/370 mainframe. Something for (or from) everyone there, I should think.

The price for the wonderbeast? A starter 64K system with one emulator was put at $1,600; with all eight you'd pay nearer $5,800.

I still don't know if the fabled 104 was a spoof perpetrated on gullible Commodore-watchers by some bored Commodore-watcher. Must have been, come to think of it. But then with Commodore anything's possible ...

- **Digi-Drum**: Circa 1982 it was actually demonstrated with much assurances about it being a prototype even though production was just about to get under way. It was/is a three-pad synthesised drum kit (it simulates snare, bass, high-hat cymbals) and was promised for Spring 1983 at about $60.

Would have made for really great cheap DIY synth music on the 64 when used with ...

- **The Synthesiser Keyboard**: Also shown around the same time, this full-size four-octave organ-style keyboard attached to the 64 via a black box that contains three more SID chips – giving you 12 programmable voices in all.

Another amazing price: $100. Another Commodore product distinguished by its absence ...

- **SX-100**: One of the original names for the SX-64
- **Executive 64**: ... and another
- **DX-64**: Promised implicitly in the Commodore documentation for the SX-64 and explicitly by Commodore salespeople – the 'S' standing 'single-disk', 'D' denoting 'double-disk'. Somehow Commodore just hasn't been able to get the twin-disk portable 64 together though; word has it that the disk drives simply won't work when they're that close together.

Coming soon ...

And here's the latest batch of announcements. By the time this book appears, these may all be in the Commodore catalogue – but I doubt it.

- **264**: The major new home computer announcement from Commodore during 1984 – though whether it *is* a 'home' computer is distinctly debatable.

Its major characteristics are a much sleeker styling than the Vic/64 design, an interesting set of arrow-shaping cursor direction keys, four function keys along the top of the main keyboard rather than along the side, and a vastly improved version of Basic.

Commodore calls that Basic version 3.5, thereby implying with good grace that it's not as advanced as the Basic 4.0 found on the 8000s, the 9000 and the 700s. It does however do a lot more than the Basic 2.0 from the 64; it sits in a sizable 32KB of ROM and has some 75 commands – including most of the notable absentees from previous Commodore Basics, things like programmer's aids and explicit graphics commands. Essentially you get all the commands added to the 64 when you buy the Super Expander.

You also get a built-in machine-code monitor with a dozen commands, which also implies that this is a computer aimed at programmers. And of the 64KB memory, fully 60KB is available for user programs – on the 64 you lose nearly half the available memory to the system even before you write a line of code.

On the other hand it does have the somewhat curious provision of a total of 121 colours (15 separate colours, in fact, with eight luminance levels on each, plus black). The 264 also shares the 64's 320x200 resolution for graphics, 40 characters per line and 25 lines per screen for text.

There again, in some respects it's dressed up as a business computer. The 67-key keyboard includes Help, Escape, and Reset; it obviously owes a lot to the SX-64's keyboard. Most important, there's the 'integrated software' options for plug-in ROMs or cartridges – typically Magic Desk (minimal word processor, limited filing system, modest spreadsheet calculator).

The PR blurbwriters settled for this one on the claim that it "makes professional applications possible for private as well as commercial users". Certainly there's no games-playing orientation despite the provision of two 'control' ports (normally used for joysticks) and the colour range. The PR promises that the role of the 264 is "to get things done, more things learned and more things enjoyed in less time than ever before"

According to the now-departed US marketing supremo Myrridon Jones, "the 264 is really not aimed at quite the same target market as the Commodore 64, because the 64 is oriented toward music and sprites and gaming ... The 264 has strength in built-in programming capability and in built-in software to suit a specific need ...".

What the 264 *doesn't* have is more interesting. It lacks the 64's sprites and three-voice nine-octave sound capabilities; in those respects it's more like a throwback to the Vic. Not that you'll be able to use either Vic or 64 software on the 264 without a good deal of modification; there's a new microprocessor, the 7501, which replaces the 6502/6510 chips found in older Commodores. At least the 7501 appears to be largely similar to the 65xx micros; that most certainly doesn't apply to the internal organisation of the 264, especially in terms of its use of memory – there's a brand new operating system for a start, and nearly all memory locations are different.

Third-party suppliers were heavily involved in the pre-production planning. The 264 was announced with several dozen packages, including translations of EasyCalc and SuperScript from independents. The 'silicon software' packages-on-ROM-chips were largely done by Data 20 Corp, a Californian company that's responsible for some of the cleverer add-on hardware for the Vic and 64; Data 20 provided the word processing, spreadsheet and business graphics facilities.

- **Plus 4**: Basically, this is a 264 with one of the promised ROM packages.

This built-in firmware combines word processing, spreadsheets, database, and business graphics with a 'windows' capability that lets you see 'snapshots' of the activity in different jobs and different files on the screen at the same time. You can also transfer information from one 'window' to another.

- **364**: Another family member. Despite the claim that the 364 is "a truly innovative personal computer at a highly competitive price", it's a 264 with a couple of extras – most obviously, the addition on an elongated keyboard of a 19-key calculator-style numeric keypad. That's why the function keys were moved from the right-hand position they occupy on the 64 and Vic to an upper left location – so the 264 and 364 could look like part of the same homogeneous family while still allowing the 364 to have its keypad.

The other difference is the built-in speech synthesiser. This has a 250-word vocabulary, which can be extended if you buy or create extra lexicons on disk or plug-in cartridges.

The 364 is the least likely of this bunch to survive in its specified form. No-one's yet heard the speech synthesiser, for instance, and that does sound rather like high-tech for its own sake and/or pie in the sky.

- **Commodore 116**: A 264 with only 16KB memory, 12K of which is available to the user. This one was "specially developed for the European market" because it "meets the rapidly rising desire of young people to learn and play with the computer." Or to put it another way the Germans (and perhaps the British) decided they'd like a modestly-priced high-powered programmer's machine with the built-in machine-code monitor and the 75-command Basic version 3.5 to sell to schools. This one also looks a prime candidate for not a lot of exposure.

- **Commodore 16**: "A very advanced home computer with features normally found only on computers costing much more ... designed with first-time users in mind ... an ideal machine on which to learn or develop programming skills ..." In other words, it looks like the replacement for the Vic 20.

It comes in the familiar Commodore 64 case, though the keyboard is arranged differently (four cursor keys rather than two, and a 'Help' key as well. You also get Basic 3.5 rather than the 64's less than wonderful Basic 2.0; and the 264's machine-code monitor is there too.

So while it *looks* like a 64 with only 16KB memory, it isn't. You get the 264's two tone generators (or one tone plus noise) producing Vic-type sound rather than the rich range of the 64's SID chip. And you get the 264's graphics and colour capabilities rather than sprites.

The 16 should lead to simpler programming – but the computer can't do as much. It will be interesting to see whether it maintains Commodore's position among the first-time buyers of computers.

- **Commodore PC**: Now this one is a real departure for Commodore – not so much as a nod towards compatibility with its own existing products: instead Commodore is rather belatedly trying to ride the wave of popularity generated by the IBM Personal Computer.

The PC is an IBM-compatible portable computer. Basically it's the Bytec Comterm Hyperion from Canada modified in some respects to Commodore's requirements. The deal with Bytec enables both companies to share technology, and to purchase circuits and other sub-assembly parts from each other with a royalty being paid on any products that use them. Inside the Commodore PC is an Intel 8088 microprocessor (as used by IBM and others) with the MS-DOS operating system (functionally similar to IBM's PC-DOS) and a pair of IBM-style double-sided 320KB floppies. A colour card add-in emulates the PC (and the PCjr, if that's ever going to be a consideration); and the 640x400 screen resolution is also the same as the IBM PC's.

Differences include the healthy starter memory of 256KB (maximum is only 512KB, though, which is less than IBM and most other PC-compatible computers). It's luggable, too, of course. And it has an integral 7in screen with a 12in cable-connected monitor "soon" to be available as an option.

The Hyperion is a neatly-styled piece of work. And while Commodore is probably too late to become the dominant opponent for IBM's PC, it should do pretty well with what looks to be a solid enough computer.

- **The Z8000:** This one is being called the 'Zee' already. It's the least developed of the 1984 clutch of announced, a multi-user Unix-based system built around a chip from Zilog and an operating system from a smallish independent.

Late in 1982 Commodore signed a two-year agreement with Zilog to receive details of the Zilog Z8001 16-bit processor chip; it immediately became apparent that MOS Technology had abandoned work on its own 6500-family 16-bit micro. (In fact that was reportedly a novel design codenamed the 650000 – or 65000, depending on which industry-watcher you believed – which combined multiple eight-bit pro-

cessing units to provide data manipulation over just about any multiple of 8. Sounds heavy? Must have been ...)

Zilog has been running a distant third to Intel (with the 8088/8086 line) and Motorola (the M68000) in the market for 16-bit processor chips, its only other significant business-computer customer in this area being Olivetti with its M20 line.

Basically the deal gives Commodore the rights to build Zilog Z8000 chips itself, provided it doesn't then try to sell them to anyone. In other words it can make the chips for its own use only. Before then however Commodore is committed to take a substantial quantity of the components from Zilog's own manufacturing facility before the volumes get high enough to warrant in-house production.

Some aspects of the Z8000 are really impressive – notably the screen. A separate block of 128KB is reserved for it, and the Zee is capable of displaying a 1024x1024 screen resolution (no colour though). A bundle of high-level languages includes Basic as standard with Pilot, Assembler, and the C language as extra-cost options.

The COHOS operating system used in the Z8000 is the Mark Williams Company's COHERENT, highly regarded by the cognoscenti and highly compatible with Unix Version 7 (the latest implementation of that high-grade operating system). Early Z8000s will be able to run two users at the same time, eight later.

CHAPTER 3
Setting Up

Unpacking the box and assembling the 64 is something the eager owner will do before taking off any overcoats, galoshes and the like immediately after arriving home with the thing. This chapter duplicates some of the information in the owner's handbook, so it's short; but it also adds some comments from experience about how to organise yourself with a 64 ...



When you unpack your 64 you should end up with:

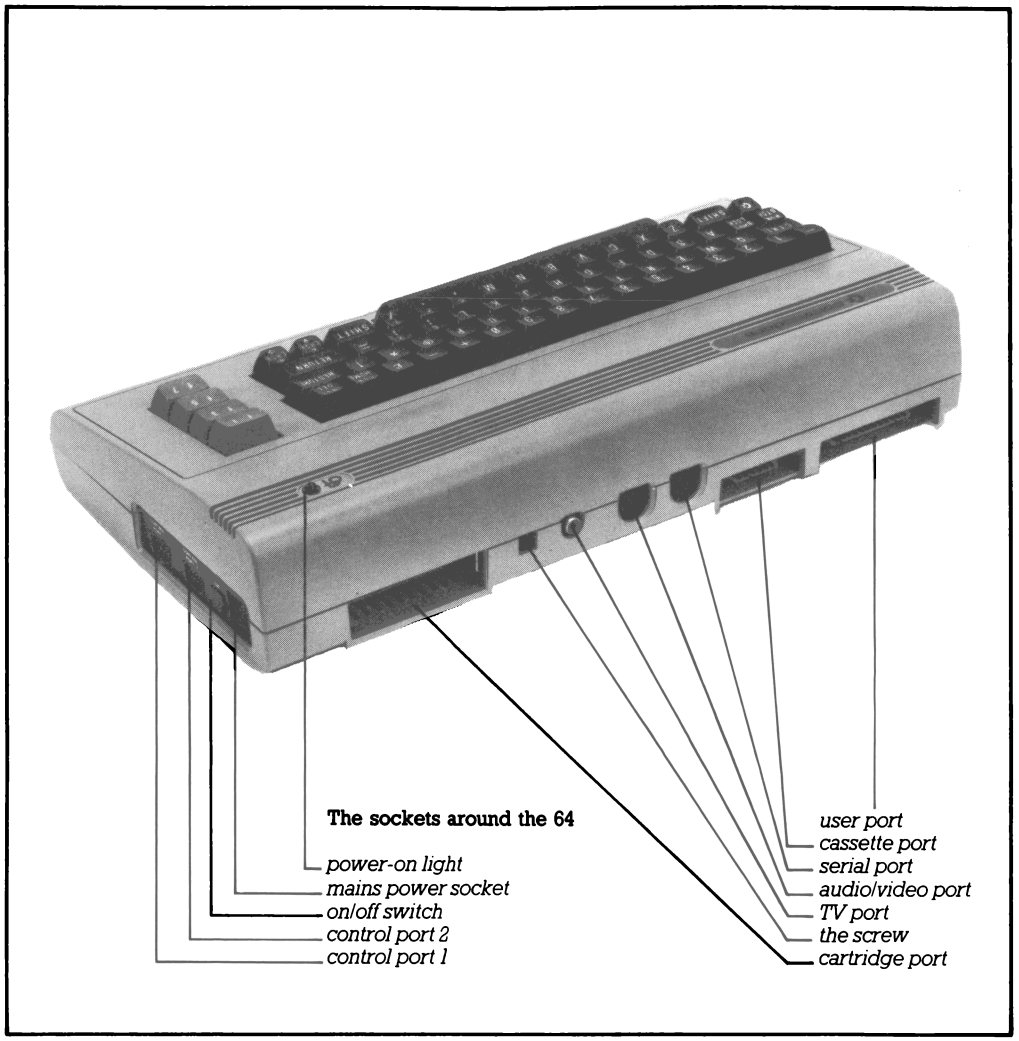
- **The Commodore 64 computer**: which looks more or less like a typewriter keyboard with some extra keys. It's surprisingly light

- **The mains adapter**: the surprisingly heavy thing, a hand-sized lump with two leads coming from it. This converts the 240 volts of the mains electricity power to the 9 volts the 64 needs. One lead goes to a mains socket, the other has a 'DIN' plug that goes into the 64 (DIN plugs are cylindrical and have a number of prongs visible at the open end)

- **The TV lead**: a length of cable about 2ft 6ins long with what's called a 'phono' plug on one end (that is another standard type of plug, also cylindrical but much smaller and with only one one prong which sticks out). This goes into the back of the 64. At the other end there will be a standard aerial socket connector for a television set – it's exactly the same as the plug you get get on a normal TV aerial and it goes into exactly the same place

- **The instruction booklet**: a spiral-bound lie-flat book of around 160 pages called the *Commodore 64 MicroComputer User Manual*

You may actually have bought more than this – but this lot is what you should get inside that long polystyrene box with the pic of the 64 on it. And if you don't have *at least* all of the items listed above, take the 64 back to your dealer right away ...



Around the 64

Figure 3.2 summarises all the connections and sockets around the 64. And there's quite a few ...

- **on/off switch**: for the mains power. At least it's obvious enough, though it's a bit small for my taste (I like large, chunky, *unmissable* on/off switches).

Some computers have a 'Reset' button which clears the memory of anything that's in there and returns the screen to its start-up state; but with the basic 64 the only way of doing this is to type ...

```
SYS 64738
```

... or turn it off and on again, which is simpler. You'll find yourself doing this quite a lot, but it is potentially dangerous to the 64's innards since the sudden surges of power can in theory overload some of the circuits. It will certainly cause wear and tear on the switch.

Oh well, maybe I'm just being pedantically over-cautious: I've never actually heard of a 64 failing because of repeated use of this rocker switch.

Incidentally, the on/off switch only controls the power to the 64 itself. When it is 'off' there is still power getting to the mains converter and it's beavering away at turning 240V into 9V whether or not the 64's running. Overuse will wear it out; so switch off at the mains whenever you can.

- **the mains socket**: for mains power, arriving as 9 volts having gone through the converter lump. This is not the normal kind of socket that takes a pronged plug; instead it's for a DIN plug, a tube-like thing containing a number of pin connections – you might be familiar with these from a stereo set-up

- **two control ports**: which in fact can connect a variety of things like light pens and paddles, but which are normally used to plug in one or two joysticks. If you have a single joystick (which is usually the case) it's plugged into one of these sockets leaving the other free: normally it goes into the one labelled 'CONTROL PORT 2', but it depends on the program that uses the joystick. You may see the Control Ports referred to as 'games' ports. (For no particularly good reason the home computer industry regards it as essential to use the word 'port' to mean 'socket')

- **the cartridge slot**: it's the slot at the back nearest the power switch, and inside you can see a thin strip of 'edge connector' – which is actually part of the main circuit board, complete with a collection of metal platings that represent connections to the rest of the circuitry. This port is used to plug in games or extra program-writing facilities, both of which can come in the form of flattish plastic 'cartridges' that themselves contain additional bits of circuitry including a couple of ROM (Read-Only Memory) chips. The mouth of the cartridge slides over the edge-connector in the slot and grips it tightly enough to make a connection with those metal-plated lines, so that the stuff inside the cartridge effectively becomes part of the computer's circuitry

- **the TV connector**: this one is not labelled, but it's very characteristic; it's the small silvery single-pin socket that juts out slightly. It takes a phono plug carrying the lead for a TV set – it passes sound and pictures from the 64 to the television

- **the audio/video port**: another DIN plug socket, round and black with a circle of holes in it. It looks very like the one next to it, but this is the one located nearer to the power switch. It can be used to connect the 64 to a high-quality sound system (for better sound than you'd get from your TV) and/or to a monitor to get a better picture – more about this later

- **the serial port**: the other round DIN-plug socket at the back, next to the video port. This is the one with fewer holes: the plug for an audio/video connector won't fit into it. It is used to connect the 64 to a printer, a disk drive, and some of the other extras you might buy

- **the cassette port**: the smallest of the three slots on view at the back of the 64. It too allows direction connection to the internal circuitry; the lead from a cassette unit plugs in here (the cassette doesn't have a 'real' plug, it uses a flat multi-way connector that fits straight on to the exposed edge of the main circuit board

- **the user port**: this is the third slot at the back, again allowing direct connection to the circuit lines. It's called the 'user' port because with advanced programming and some clever electronics construction you can control other devices via this connection; some standard off-the-shelf products also plug in here, things like modems to allow your 64 to send and receive information over phone lines

- **the screw**: also round the back there's an odd little screw. The *User Manual* refers to a 'channel selector' switch which you're blithely informed "is used to select which TV channel the computer's picture will be displayed on". That doesn't apply in Europe and this switch was omitted. We get a screw instead. Ignore the comment in the *User Manual*.

Other items

To set up your 64 you'll also need to have two other things. One is a **plug** for a mains socket, since Commodore doesn't feel able to give you one for free on the power cable.

An ordinary 13 amp plug will do, but you're advised to get a fuse with a lower rating – 5 amp should be enough. That's a simple precaution, since there's a (remote) chance that the 13 amp fuse you normally get in plugs won't give enough protection: the lower-rated fuse, which you can buy for pennies at any electrical and most hardware stores, will blow before too much of an electrical overload screws things up for the 64.

Actually, I've never heard of a 13 amp fuse being too generous and causing problems. But you can't be too careful ...

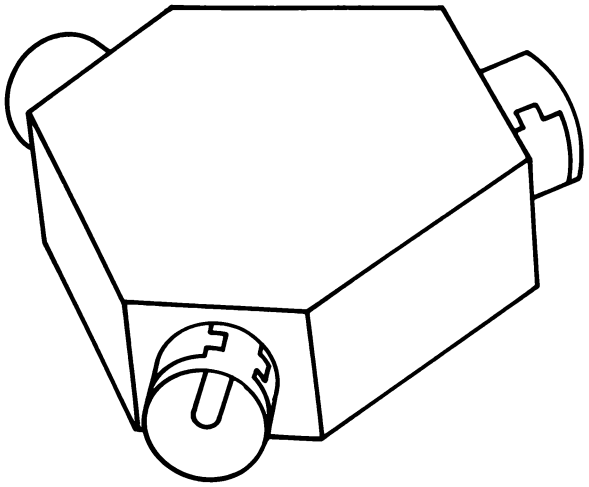
You'll also need a **television set**. It doesn't have to be portable and it doesn't have to be colour, but both are preferable. Positioning a full-size TV such that it's comfortable to use with the 64 *and* out of the way of the rest of the family can be a hassle; TVs tend to live in a corner of the living room, while you'll want it in a quiet place and on a table or desk with the 64.

As for colour, well, the 64 will work ok in black and white: but one of its real strengths is its ways with colour. And it's a bit pointless buying a colour computer and not seeing the colour ...

Ideally, try for a second TV to avoid conflicts with the rest of the family's unaccountable preference for *Dynasty* over your programming efforts. (*Dynasty* may be awful, but even it will make for better viewing than watching your back as you clack away at the keyboard putting a lot of letters on to the screen).

So buy a second TV if you can, preferably new portable – prices these days start below £150. And if you can't afford a new colour set, consider buying an ex-rental TV (prices from about £50).

One other extra that I always recommend is a thing called a **video** (or **aerial**) **splitter**. This plugs into the aerial socket of a television set and allows you to leave both the TV aerial and the 64 connected all the time. Otherwise you'll be constantly plugging and unplugging both the 64 and the aerial, which



An aerial splitter – lets you have both the TV aerial and the 64 connected to a TV at the same time

won't do either plug or the aerial socket any good at all. You can get one from most TV or hi-fi shops (or for that matter by mail-order from **Commodore User** magazine); an aerial splitter shouldn't cost you more than three quid at most.

Layout

It pays to put some thought into where and how you're going to set up a 64. If you don't do it now, you probably won't ever bother doing it.

For a start, choose your working space. The different bits of the 64 can sprawl a bit and one of the more irritating things about the 64 is the spaghetti-like tangle of leads you can end up with.

Begin with a working surface that's bigger than you think you will need. Why? Because while you're using the 64 you will want somewhere to put books (probably more than one at a time), notepads (it's amazing how much scribbling on paper there's involved in using computers), and cups of coffee (vital if you're getting seriously into the 64).

So don't get cramped – give yourself enough room. Fairly instant worktop solution, provided you've got the room: go along to a wood shop for a 2ft 6in wide flush interior door (£10 or so) and put it on a couple of trestles (about £20) or some other instant legs. Those doors don't bend like purpose-built laminated chipboard worktops, they're bigger, and they are cheaper ...

Site the TV at least a foot or so away from the keyboard to avoid eyestrain. It also makes sense to have the TV set a bit higher than the keyboard, if possible at eye height. It's more comfortable to use that way. An easy option if you're using a portable TV is to build a simple stand with a long flat surface (for the TV) and flat side pieces (to raise it above the 64). This gives the 64 somewhere to slide out of the way when it's not in use. It can involve three pieces of laminated chipboard cut to size by the wood shop, four plastic angle connectors, and about five minutes' work.

Make the chipboard thick enough to take the weight of the TV and leave enough room on top for a disk drive and printer (you'll want both of them sooner or later even if you don't start with either).

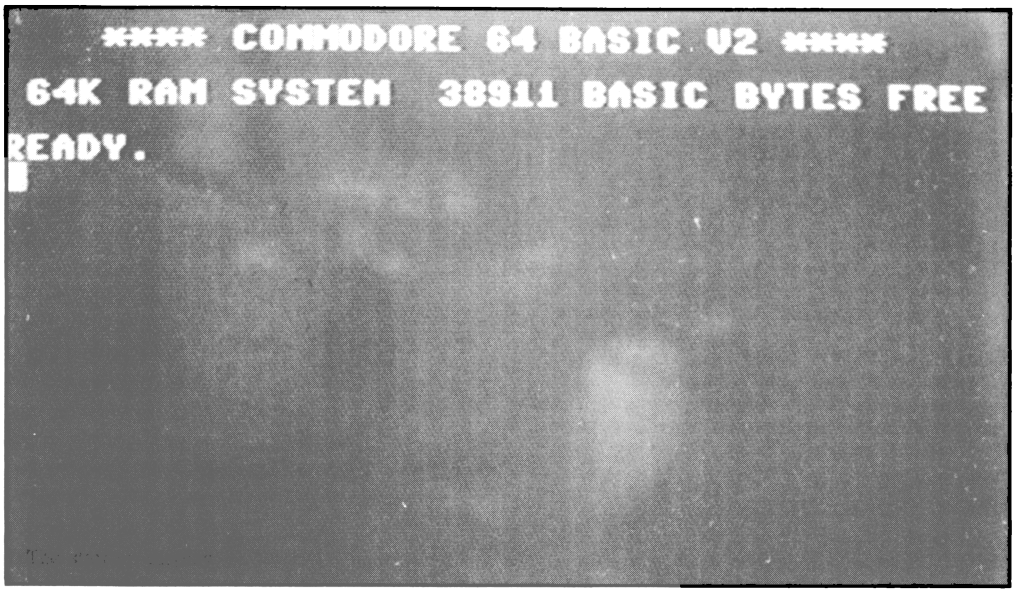
A cassette player is best sited it as far away from the TV set as possible, since the unit has been known to pick up unwanted radio emissions from the television set. It should also be well away from the mains lead; the interference from 'leaking' mains or the TV can both cause faults on storing programs on tape or loading something from cassette. If you have to have the tape unit close to the television and you do get faults, try putting it on the other side of the TV.

You could screw a three- or four-way extension socket on to the stand, too – the 64 takes one socket, the TV needs another, and mains connections will also required for a disk drive and the printer. Extension sockets are a good thing anyhow, since they prevent too many power leads wandering around the floor.

The hefty power transformer certainly ought to be well out of the way. If you knock it off the desk it could do some damage to your foot and/or your floor, not to mention the 64.

Connections

Connecting things together is pretty obvious. The **TV lead** obviously fits between the TV connector socket on the back of the 64 and your TV's aerial



socket – you can't plug them in the wrong way around, and the TV connector won't fit into any other socket.

The **cassette unit** plugs into the smallest of the three slots at the rear of the 64. Don't try forcing the plug – it only fits one way round. The plug is divided into two, one gap being narrower than the other; and the edge connector in the 64 is also divided like that, with one contact larger than the other.

Once it's in there, it is a good idea to mark the top of the plug with a blob of paint or something. That way you'll always know which way round it should go. Why didn't Commodore put some kind of mark on there for you? Don't ask me.

Now make sure the on/off switch is 'off' and switch off the power at the wall socket too. The **power lead** from the mains adapter goes into an unmistakable round multi-pin DIN-plug socket by the switch on the side of the 64, and again there's only one way it will go into the socket – the groove in the barrel of the the plug should be on the upper side – so don't force it!

Action

And now you can switch on at the mains and turn on the 64. The red 'power' light at the top right of the keyboard should come on: if it doesn't, check the connections and start again.

Still no red light? Could be the fuse in the mains plug: there's a fuse in the 64 to which might blow. Could be that the little bulb is dead. Or could be that the whole thing's dead ...

Try getting a picture anyhow. Turn on the TV and select an unused channel. If you're very lucky you will immediately see the 64's opening display – but first time you try it that's most unlikely. Twiddle the tuning knob on the TV for the channel you're on; you may pick up TV stations before you reach the 64's signal, but sooner or later you should see something like Figure 3.4 – light blue words on a dark blue background with a light blue border and a light blue square flashing at you.

- **the border**: you can change its colour if you want, but there's not much else you can do with it. You can't put text or set up pictures in the border area

- **the background**: this area is yours to fill with whatever you want; you can change its colour, too

- **the message "COMMODORE 64 BASIC V2"**: 'BASIC' is the name of the programming language you have available to you: 'COMMODORE BASIC' is Commodore's very own version of it (practically every micro you can buy comes with Basic, but that doesn't mean all versions of the Basic are the same – in fact they're practically all different in terms of the commands you get, what you can do with them, and how they're used): 'V2' means 'version no. 2' (Commodore itself goes in for different versions of Basic on its different computers)

- **the message "64K RAM SYSTEM"**: 'RAM' is 'random-access memory', a somewhat daft name indicating the kind of electronic storage that you use for your programs (it's daft because there is nothing particularly random about the way you access the contents of memory). A 'byte' is the internal way the 64 has for storing a single character, letter or symbol: 'K' is a symbol used by computer people and advertising agencies to indicate 'times 1,000' (or, more accurately, 'times 1,024', since K is really an arithmetic constant for that number). So '64K' means '64K bytes', which

in turn means enough memory in total to hold 65,536 characters of one kind or another. In fact this message is a bare-faced lie, since not all of the 64's 64K memory is RAM; and the 64 itself uses some of it for its own functions, anyhow

- **the message "38911 BASIC BYTES FREE"**: which tells you how much memory you have been left to play with after the 64 has taken what it wants for itself. It says 'BASIC' bytes because the memory will mostly be used for programs written in Basic: the total length of your program plus the extra working areas it needs can be as much as 38,911 bytes. If you don't use Basic, you can have more memory for your programs – with clever (and complicated) messing around you can actually use as much as 60K bytes, provided you're prepared to write programs in what's called 'machine code'

- **the message "READY"**: means the 64's waiting for you to do something

- **the blinking square**: this is the 'cursor' and it indicates the next position on the screen at which something you type will appear.

If you can't find this display after turning the tuning knob through its whole range, switch off and wait about ten seconds then try again. If it's still not there check all the connections and retry. And if you don't get it then, give up and take the thing back to the shop.

Subsequently when you turn on the 64 there will be a short delay before you get the 'READY' display. That's because the 64 is checking itself out internally.

Tweaking the picture

The picture on most colour TVs will be adequate – though you should if possible get a demo in the shop of a 64 running with the same kind of TV set as you'll be using (which is one good argument, albeit about the only one, for buying your 64 from a high-street electrical goods shop that sells TVs as well as computers).

All TV sets seem to have slightly different colour characteristics, only some of which can be attributed to internal adjustments that you probably won't be able to make yourself anyway. Fiddling with the controls might be enough to get a decent picture from a TV station, but you may well find that the much less complex mix of colours, shapes and movements produced by the 64 will show up any deficiencies in the TV.

Adjustment of the TV set's controls may help, of course, and you (or your dealer) can also try getting inside the 64 for some delicate tweaking of the picture-generating circuitry actually inside the 64.

Be warned, though – unscrewing the 64 can invalidate your Guarantee, because Commodore can decline to fix a bust computer if there's any evidence that you've been fooling around inside it yourself. So if you're going to try this, be extremely circumspect ... which means 'don't leave any evidence'.

More important, don't even attempt this unless you know what you're doing – or unless you can be extremely delicate in your movements (electronics are sensitive!) and you're willing to avoid the temptation to fiddle with anything other than the adjusters I'm about to indicate. You will also have to avoid the electricity that's coursing around the computer; not only could you get a nasty shock by inadvertently shorting something, you could really damage the 64.

Any doubts? Skip the rest of this section. Otherwise follow these instructions *rigidly*.

For a start, I assume you've followed the instructions in Chapter One of the *Commodore 64 MicroComputer User Manual* and now have at least something on view to prove that the TV and the 64 are in contact with each other. Now switch off the 64; up-end it carefully – *very* carefully! – and use a cross-head screwdriver to remove the screws that hold the top to the bottom; put them somewhere safe. Hold the two halves together and put the 64 back down the right way up. Lift off the top.

With the internals exposed, look for the bit of circuitry nearest the socket for the TV lead. It should look something like Figure 3.6 or 3.7 – most British 64s come with the layout in Figure 3.6 but a few don't. The bit labelled 'modulator' is provided there because no two TV sets respond in quite the same way to to video signals from the computer, and those round screw-head thingies you can see are the adjusting screws on top of potentiometers.

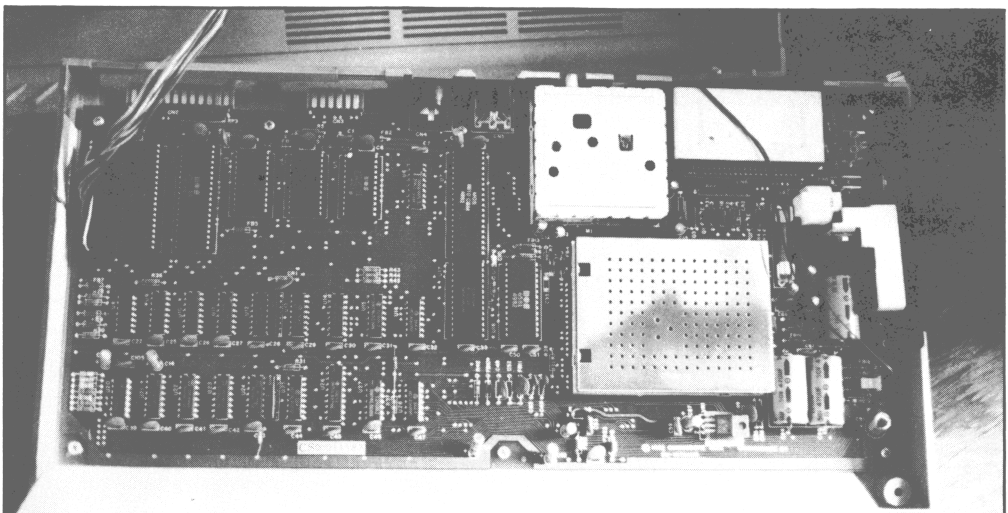
Whichever you've got, start by plugging the mains and TV leads back in and switch everything on. Stay well clear of the power-in bit, and use an insulated electrical screwdriver and a steady hand. Turn the brightness and colour controls on the TV as high as they'll go and make sure you have a picture from the 64.

Insert the screwdriver into the screwhead labelled **V**: move it delicately in both directions – you'll see that this controls the steadiness of the picture. Now adjust the TV set's colour, contrast and brightness controls till you get a reasonable picture.

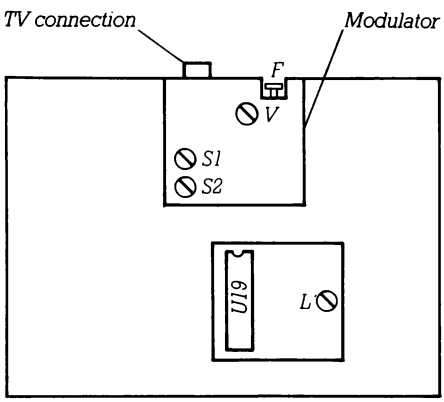
If you've got the layout in Figure 3.7, you can also adjust the screwhead labelled **C** (this handles colour contrast) and **L** (which should be turned until you can see no horizontal 'scanning' lines on the display).

Finally adjust **F** slightly to get the sharpest possible picture. For the Figure 3.6 layout this is a little screw at right angles to the rest; for Figure 3.7 it's a screwhead like the others.

When you've got the best mix you can find, switch everything off and screw the 64 back together again before you start using it – unless you want to take the opportunity to adjust the sound quality too while you've got the thing in bits.

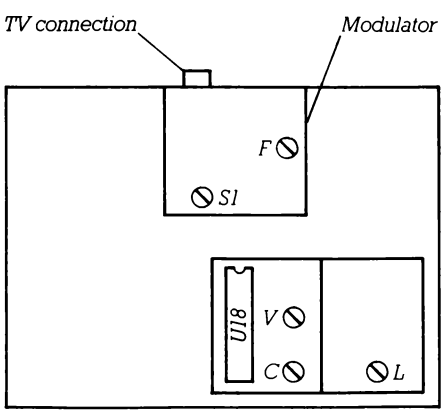


Inside the 64 – instant invalidation of your warranty?



F (fine video adjustment)
L (line scan adjustment)
V (video adjustment)
S1 (sound adjustment)
S2 (background sound adjustment)

Modulator layout for 7-pin audio/video socket



F (fine video adjustment)
L (line scan adjustment)
V (video adjustment)
C (colour contrast adjustment)
S1 (sound adjustment)

Modulator layout for 5-pin audio/video socket

Tweaking the sound

The 64 doesn't have a built-in loudspeaker of its own. Instead it passes sound signals to the TV, and your average TV set doesn't have a great sound-generating system – a small loudspeaker, a squeaky little amplifier. You can live with it, though, and if there's any real problems on the sound you can again do some tweaking inside the 64.

Before that however you'll need to have a test program. The one on pages 80 of the user's handbook will do, if you add a new line to repeat the sound continuously:

```
80 GOTO 10
```

You can type this in and SAVE it on tape or disk before you unscrew the 64; then when you've separated the two halves of the computer, you can reLOAD and RUN it. Alternatively take the thing apart first, plug in, and type the program then.

While it's RUNning apply your trusty electrical screwdriver to **S1** and turn it gently until you get the best possible sound. If your 64 has the Figure 3.7 layout, there'll also be the screwhead I've labelled **S2**; this can be adjusted to minimise any background noise.

The best solution for colour ...

No matter how much fine-tuning you can do, an ordinary TV set will almost invariably give you a degree of 'bleeding' – where the edges of characters and shapes are slightly fuzzy and the boundaries between two colours aren't particularly sharp.

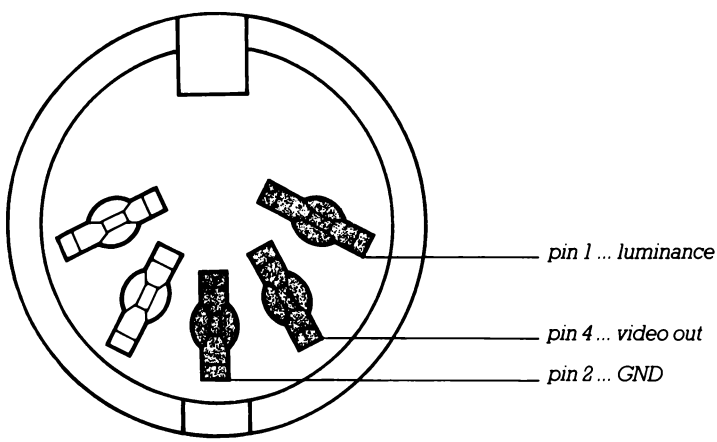
You'll get a better picture from a purpose-built **monitor**, which in practice is a TV set without the gubbins required to receive and decode TV signals and to switch channels. Omitting that circuitry means the picture from the computer will be much crisper and the colours much sharper. (You can also use them with a video recorder, and the pictures there will also be much better than a standard TV.)

Monitors for the Commodore 64 are about the same size as a portable TV and prices are currently in the range £170-£250. Monitors are connected to the audio/video port rather than the TV-output socket, which means they need a different kind of lead with a different plug (a DIN plug rather than a phono plug). The monitor will have a built-in loudspeaker too, just like ma TV set, so you'll probably use all the connections on the audio/video socket.

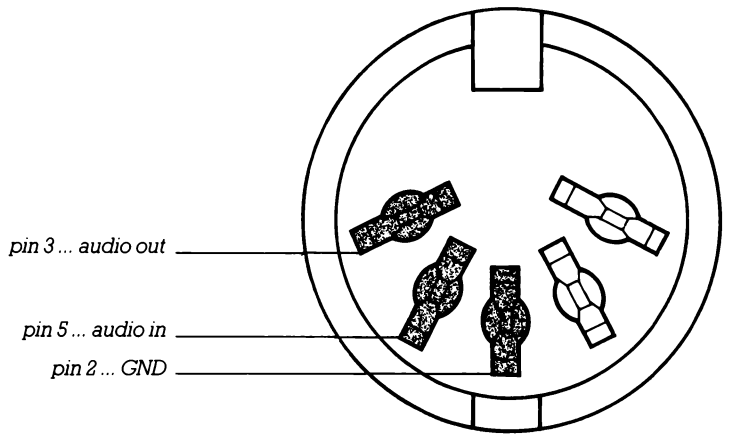
Commodore's monitor comes with the right lead; but if you use a different monitor that doesn't have the appropriate lead, don't worry too much – any electrical shop or computer dealer should be able to make one up for you. Show the shop Figure 3.8, which is what's got to happen at the 64's end, and take along the monitor's own handbook so that the other end can be wired appropriately.

Or it's not too difficult to do yourself. Buy a five-pin DIN plug and some five-core cable; wire it according to Figure 3.8. The other end will probably have a clutch of phono plugs for sockets in the monitor, but you might need to refer to the monitor maker's manual to check that.

Video connections on the audio/video port



Audio connections on the audio/video port



By the way, don't be seduced into buying a more expensive display called an 'RGB monitor' on the promise of top-quality picture reproduction. RGB monitors are so called because they assemble a video display from three different colour signals (Red, Green and Blue – hence the name). And they do indeed produce the very best pictures ... provided the computer delivers its video signal in the RGB format.

The Commodore 64 doesn't. Instead it uses the much simpler (and cheaper) 'composite video' format, which is the one employed by standard TVs. The picture will still be considerably better than the TV's, though, and it probably won't suffer very much by comparison with an RGB monitor's. So you save your money and buy the least expensive colour monitor you like.

... and for sound

You might not need better sound reproduction – TV loudspeakers aren't great, but these days they are probably adequate for most of things you'll hear from your 64. You can get better sound if you want it, though; and you probably *will* want it if you're using a voice-output device or a music package. Anyhow, on a TV you will probably notice a background hum that changes according to what exactly is on the screen; this is the video signal interfering slightly with the sound signal, and it would be nice to get rid of it ...

The solution again lies in that thoughtfully-provided audio/video socket on the back of the 64. With a lead to this you can hook the 64 up to one of the inputs on an amplifier – an ordinary hi-fi home-stereo amp, for instance, in which case go via the AUX sockets: or even a 'professional' instrument amplifier, like a practice combo amp with a built-in speaker.

Going into a stereo amp won't give you stereo sound, of course, since the 64 doesn't produce two sound channels – just the one. So you'll need to set the hi-fi system to 'mono' to hear sound through both speakers.

Again, making up a cable isn't difficult: wire to pins 2 and 3, and you'll have the audio signal. If you are using a monitor with a ready-made cable, you can probably take the two audio leads into an amp simply by buying extension leads for those two wires. If you're using a TV for the picture, you can make up a sound cable with just those two pins connected – but then you'll get the sound duplicated on the TV set too, so turn down the volume on the television.

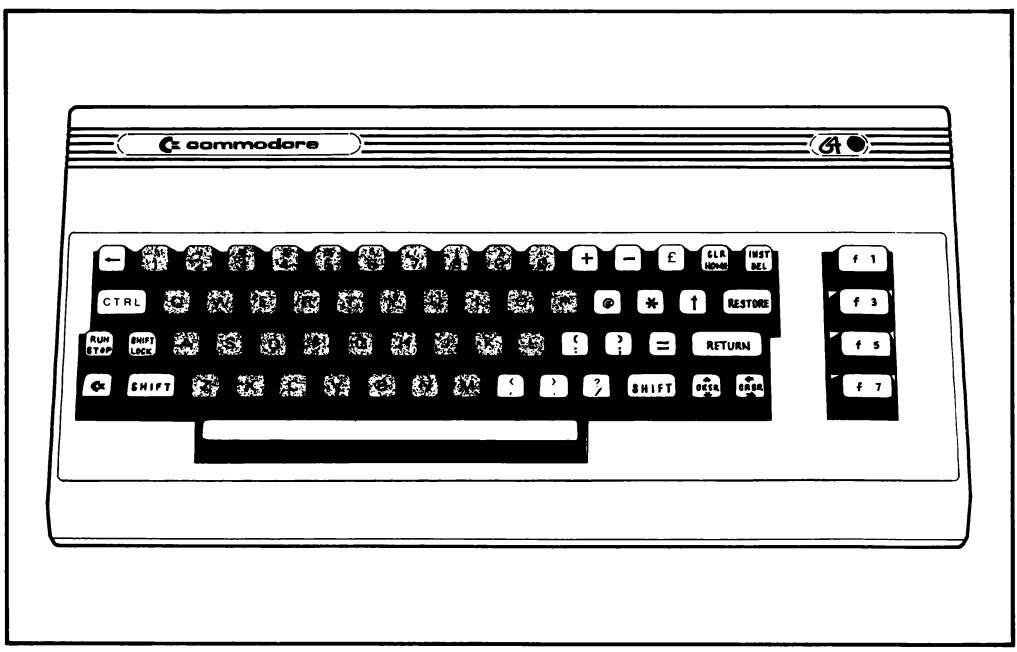
Pin 5, 'audio in', could be fun since it also you to *input* external sound to the 64 – should be interesting if used with music programs. But I don't know anyone who *has* used it satisfactorily ...

CHAPTER 4
A Tour of the Keyboard

The 64's keyboard might look like a typewriter at first sight. But most of the typewriter-style keys have at least two symbols on them and many have three; and most of them can produce three different results, while some don't produce any ...

In essence, though, there are five kinds of operation provided by the 64's keys:

- displaying characters (letters and numerals)
- displaying graphics
- moving the cursor and editing
- controlling other keys and general operation of the computer
- user-defined functions

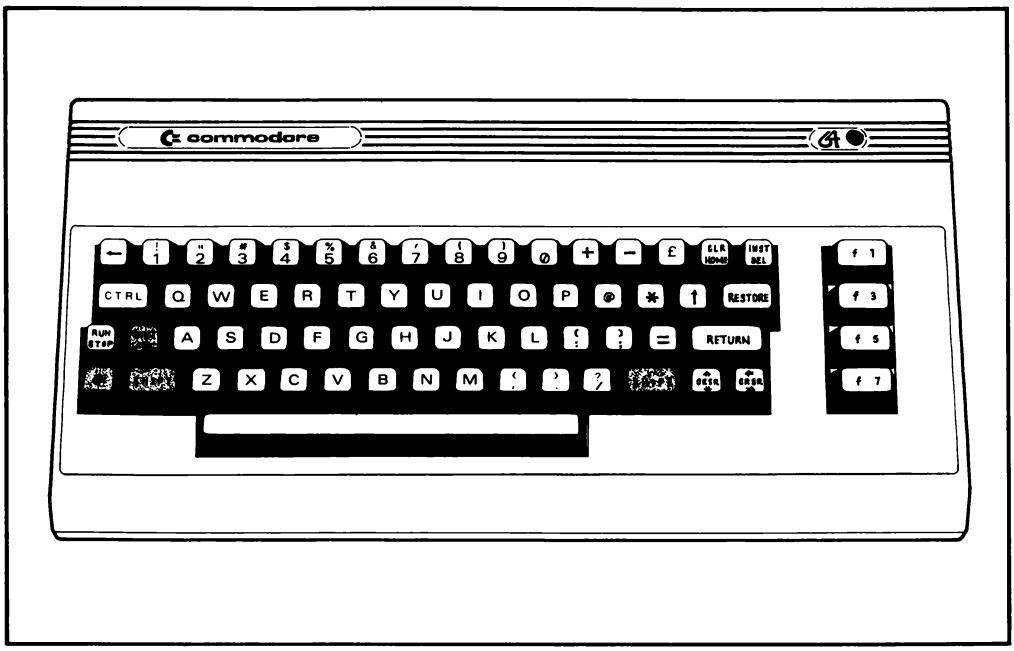


Displaying characters

The keys that print something you'd recognise if you were used to typewriters are more or less in the same position as for typing – the alphabetic keys are arranged in the conventional QWERTY format and the numbers run along the top row with punctuation marks and other symbols.

And those symbols and the numbers work just as on a typewriter. You hit the key to get the number, you hold down Shift and hit the same key to get the symbol or punctuation mark engraved above the number.

(Either of the two big Shift keys on the bottom row of the keyboard can be used. You can also 'lock' the



Shift key by pressing Shift Lock: that way everything you type will be shifted until you next hit Shift Lock and release it.)

The letters don't work like that, however. On a typewriter you'd get lower-case letters if you pressed a key without using the Shift and capitals if you did. On the 64 you get *capitals* when you hit the key without Shift. And if you do hold down the Shift and type a letter, you'll find that you get one of the two graphic symbols marked on the front of the key – the one on the right, in fact. (You'll also get graphic symbols when you type the letter keys with Shift Lock engaged.)

This is because the 64 has two entirely different sets of characters available to it – one group consisting of capital letters and a whole bunch of graphics symbols along with the numerals, punctuation marks and the like, and this is what you get at start-up.

The other character set gives you lower-case letters as well as capitals, and some extra graphics symbols too. But you don't get as many graphics symbols in this set. That's because there is a fixed limit to the number of different characters the 64 can recognise at one time; and to give you a load of graphics shapes on the keyboard the designers had to leave out something. Which in the case of the characters available at start-up was the group of lower-case letters.

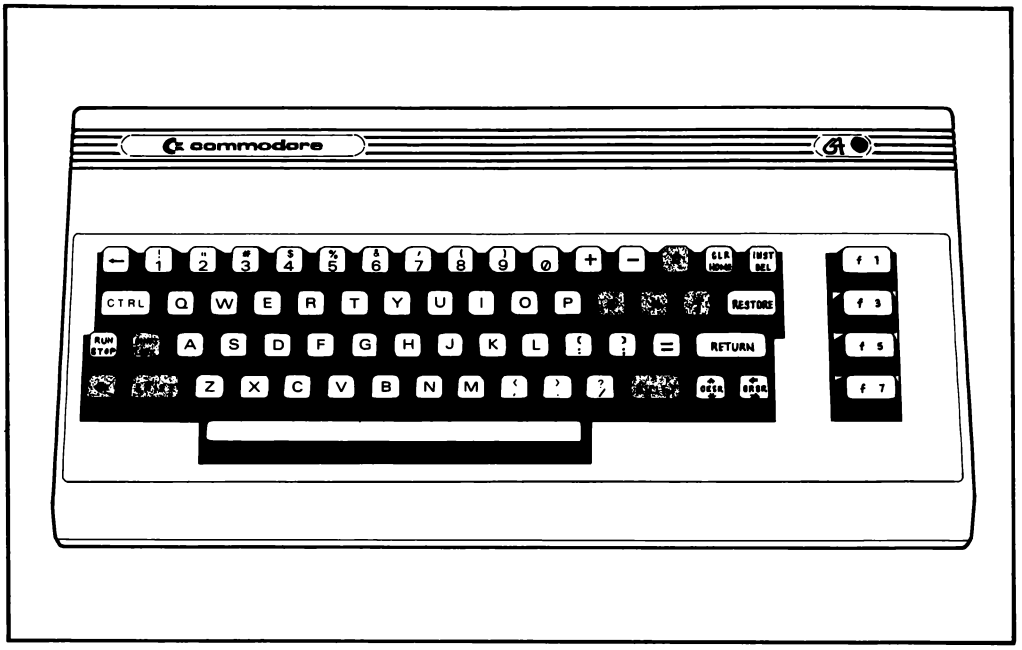
To get to the 'typewriter' style with upper- and lower-case letters – and to use the *other* graphic shape marked on the front of a key, the one one the left – you hold down one of the Shifts and press the key at the bottom left of the keyboard. That curious symbol on it is Commodore's own corporate logo, and this key works as a kind of extra Shift: you'll see it referred to as the 'Commodore' key or the 'CBM' key.

You can't display characters and shapes from both character sets at the same time. The 64 detects that you've pressed a particular key and then checks to see which character set you're using before it decides what it's actually going to put on the screen. Thereafter it will change what's on the screen whenever you press Shift and the CBM key together.

When you write programs you can include commands to flip between one mode and the other, however. In other words you'll probably use the 'graphics' mode (capital letters plus lots of graphics) to write your program; but you could include a command to switch the display into 'typewriter' mode when the program is RUN, so that the display which appears uses lower-case letters and different graphics.

Displaying graphics

There are a total of 62 different graphic symbols on the 64's keyboard. They can be used individually to give emphasis to something you display on the screen: or they can be used in combination to build up quite sophisticated 'drawings'.



The graphics don't appear in a kind of ruled box such as you get on the front of the keys: that box is just there to indicate the (invisible) outline of the rectangular space on the TV screen which the 64 uses for a single letter or character.

Each key in the alphabetic layout produces three symbols in each mode. Take 'Z', for example. In the start-up 'graphics' mode you'll get a capital Z when you press it. Hold down Shift and type Z again; and you'll get the diamond, graphic symbol that's on the front of the key to the right. Hold down the CBM key and you'll get the symbol on the left.

Now switch to 'typewriter' mode by pressing the CBM and Shift keys together, and just about everything on the screen will change. You can see how and why by repeating the 'Z' combinations – by itself the key now produces a lower-case 'z', with a Shift it gives a capital Z, and using the CBM key provides the same left-hand cut-off-corner graphic.

So switching between modes can give you a total of four different symbols on this one key. That's true for most of the alphabetic characters and many of the standard symbols (though not the numerals – using the CBM key with them changes the colour of the cursor and anything you type subsequently. More of this anon.)

So using the Shift and Commodore keys, all of the 'printing' keys on the 64 keyboard produce the characters marked on the key in question. What the handbook doesn't tell you is that there are four other graphics you can get which *aren't* marked on the keys.

You get to them by pressing the Commodore and Shift keys simultaneously. Then do this:

press Shift and @ keys to get 

press Commodore and * keys to get 

press Shift and £ keys to get 

press Shift and ↑ keys to get 

Punctuation marks

The punctuation marks can be used conventionally for punctuation, but several have other rather special features associated with them as well.

- **Double quotation marks**: in Basic, double quotes are used to indicate that the text between them is to be treated as a string of characters – which thereby distinguishes them from variables

- **full stop**: doubles as a decimal point; and in fact outside quotation marks that's its *only* use – in other words, to persuade a Basic program or direct-mode statement to put a full stop on the screen without a ?SYNTAX ERROR message it's got to be somewhere inside a pair of quotes or numbers. More about this in Chapter 7, but the implication is that it's impossible for a

Commodore Basic program or a Basic statement used in direct mode to print or display double quotes

- **comma**: again, it won't be accepted as a comma if you use it outside quote marks – and it won't be recognised if you try to split up long numbers with commas (so type 5532861 rather than 5,532,861).

As well as a punctuation mark for text inside quotes, the comma has two other uses. In Basic commands it can be used to separate various items that the computer has to check out before it gets on with things – as in LOAD 'PROGRAM',8 (that will actually try to load a program named 'PROGRAM', from a disk drive which in this case is identified as '8'.

And in the lines of a Basic program the comma between separate PRINT commands means that separate items to be printed will appear one every ten columns – it's a kind of automatic 'tab' in fact.

- **semi-colon**: this is another one that functions as a punctuation symbol only when used inside a pair of quote marks, since outside the quotes it has a specific meaning to Basic. When it separates different PRINT statements on the same line of a program, it means the different bits of PRINTing to be done will appear on the same line.

- **colon**: likewise, this acts as a colon inside quote marks – it has a totally different meaning elsewhere. One is to separate different Basic statements appearing on a single line of a program:

```
10 FOR T=1TO5: PRINT "HELLO": PRINT
"NICE DAY OUT?": PRINT: NEXT
```

Here there are five separate statement (FOR, three PRINTs, one NEXT) all on the one line.

The other use of a colon is to make Basic programs more readable. Ordinarily the 64 will take out any unnecessary spaces you've left in front of a statement while typing in a program. So if you were to type in this ...

```
100          REM ASK "NICE DAY OUT?"
```

... and then you LIST it, it would appear with the reduandant space closed up:

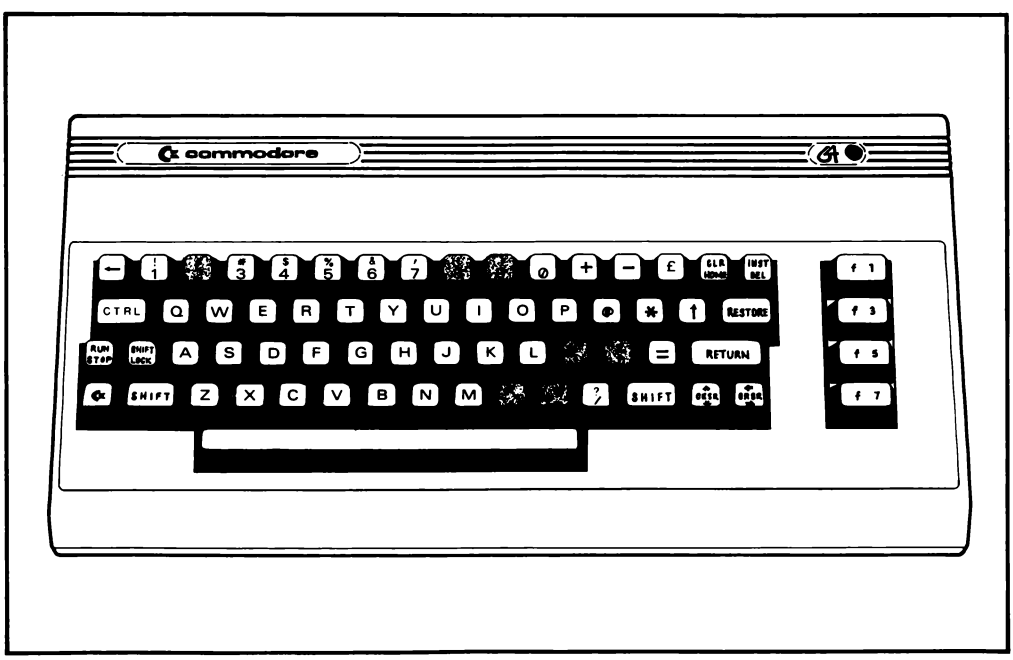
```
LIST
100 REM ASK "NICE DAY OUT?"
```

This means that long program LISTings tend to look much the same. But sometimes you might want a way of changing the physical shape of the listing, for instance to indicate where each new section of program starts. Putting a colon in there after the line number will give you the spaces:

```
100:          PRINT "NICE DAY OUT?"
```

So the next time you LIST it, this line will stand out.

- parentheses: also tend to have a special meaning on the 64, apart from their use inside quotes. (For a start, though, *square* brackets are only square brackets: you can use them only inside quotes.) The parentheses, the curved brackets are additionally they used as in maths to indicate an order of calculation: 44/(1.66+0.275) pro-



duces a different answer to (44/1.66)+0.275 because you're telling the 64 to do the arithmetic in the brackets first.

Parentheses are also used with some Basic statements to indicate a predefined value. For instance, CHR$(36) is a way of referring to one of the characters on the keyboard – it's the dollar sign. And PRINT TAB(8) 'HELLO' puts the word 'HELLO' on the screen eight places from the left.

Arithmetic

The keyboard has all the normal arithmetic signs plus some extras.

- **addition**: use the plus sign as usual – but *not* with the Shift key, which on most typewriters still gives you a plus sign. On the 64 holding down Shift and pressing the plus gives a character which *looks* like a plus, but in fact it's a cross – a graphic symbol

- **subtraction**: use the dash as a minus key as normal (if you try holding the Shift down you'll get a symbol that looks nothing like a minus

- **division**: use the slash key – it has the same effect as the normal division symbol

- **multiplication**: use the asterisk – it has the same effect as the 'multiply' symbol.

- **pi**: pressing the up-arrow key while holding down Shift to get the constant pi to eight decimal places. You could see that if you were to type 'PRINT π', and of course you can also use the symbol in arithmetic (PRINT 2*π*376 finds the circumference of a circle with a radius of 376 on the basis of the familiar 2πr formula – note that you have to explicit about the multiplication)

- **exponentiation**: the up-arrow key without Shift provides exponentiation, which means 'raise to the power'. The symbol is used between two numbers like this ...

```
2↑3
```

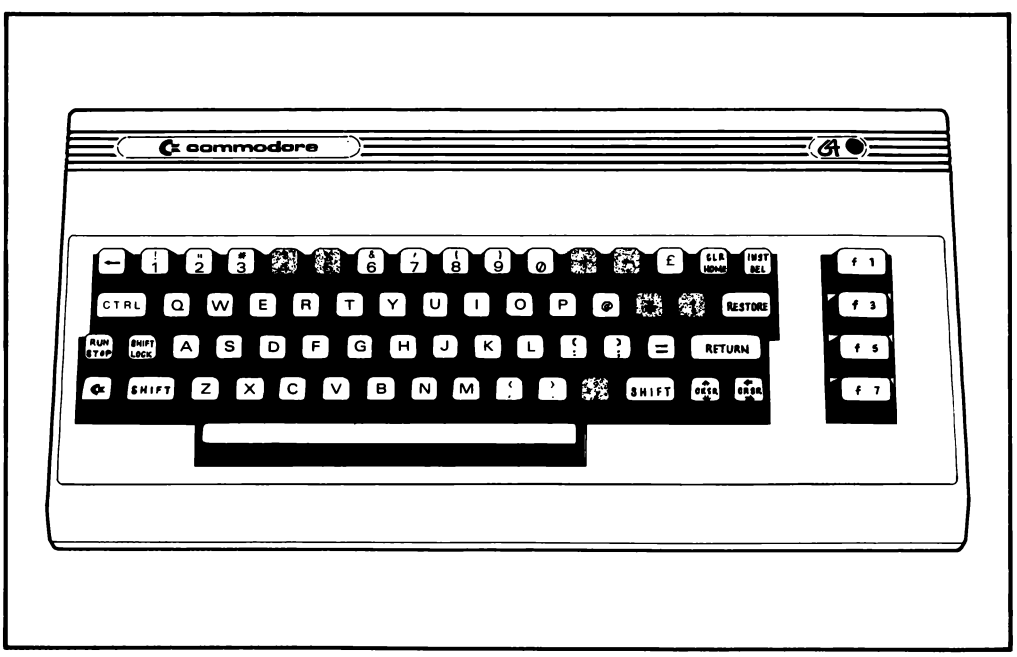
- **percent sign**: as well as the normal use in text – within quotation marks, that is – the percentage symbol is used to indicate what type of variable you're assigning. Basically there are three types – character strings (a string is just a bunch of characters or digits), real numbers (with a decimal point if necessary) and integers (where only the bit to the left of any decimal point is used).

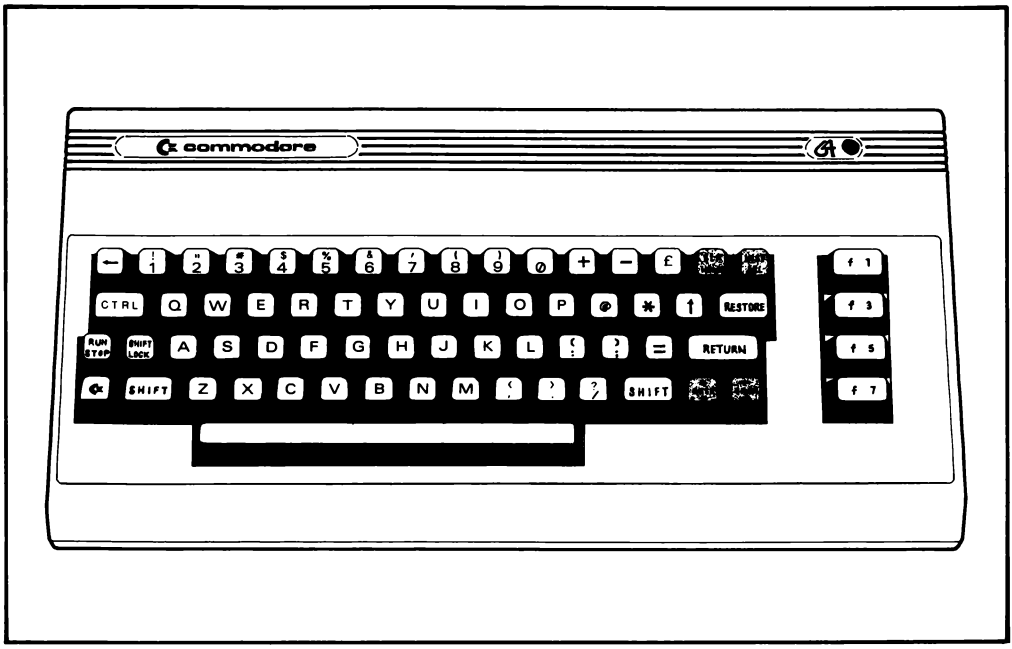
Variables are given names, and the type of variable in question is indicated by the symbol alongside the name. Real numbers don't get any at all: integers are denoted by the percent mark

- **dollar sign:** character strings are denoted by the dollar. This little program illustrates all the combinations:

```
10 A$="123.4"
20 A=123.4
30 A%=123.4
40 PRINT A$, A, A%
```

The character string has to be enclosed in quotes; neither of the other two can have quotes anywhere.





If you RUN the program, you should see the three things associated with the three variables appear on the screen – the first one being the character string, the second the 'real' number, and the third the integer (which has lost the decimal point and everything following it).

Cursor control and editing

The two 'CRSR' keys at the bottom right and the 'HOME' key on the top row all move the cursor around. Once you've got where you want to go, you can play with the 64's clever 'screen editing' facilities.

- **The CRSR keys:** use the CRSR keys without Shift to move down and right, use the Shift with them to move up and left. Holding the 'cursor down' key will move the cursor down the display until it hits the bottom line, at which point it will carry looking at the 'next' line and so force anything already on the screen off the top edge.

You can't get that text back by doing the reverse procedure, holding down the 'cursor up' key. That will take the cursor up the screen until it reaches the top edge – and there it will stop.

'Cursor left' will keep the cursor moving until it reaches the right-hand edge of the line, whereupon it will automatically move on to the start of the next line – and so on until it hits the bottom, where it too will go on to the 'next' line and force stuff off the top of the display.

'Cursor right' works much the same way in the other direction, but when it gets to the top left corner it won't move any more.

- **HOME:** takes you straight to the top left corner of the display without deleting anything

- **CLR:** Shifted HOME gives you CLR, which does the same thing but also clears the display for you

- **DEL:** deletes the character immediately to the left of the cursor. If you hold it down it will carry on deleting, so be careful ... If you keep it pressed, the cursor will step back deleting everything in its path until it gets to the top left corner

- **INST:** Shifted CLR gives 'insert', which opens up a space in the text your cursor is currently sitting in. You can then type something in there. Hold the keys down to insert several spaces – it'll carry on inserting until the spaces plus the text total 80 characters, two lines of the screen; then it'll come to a stop. If you try to insert a lot of spaces on the bottom line of the screen, the text being shoved to the right by the inserted spaces will be forced on to a new line and again you'll lose whatever was on the top line of the display

Controls

We've come across a few of what might loosely be termed the 'control' keys on the 64's keyboard, since they control something – usually in combination with other keys:

- Shift
- Shift Lock
- Commodore key

There are a number of others, though.

- **Return**: this is probably the single most important key on the keyboard. Pressing it tells the 64 to take whatever's on the line where the cursor happens to be and do something with it – either enter it into memory, if it starts with a number (in which case it's assumed to be a line of a program to be executed later when you RUN the program): or to execute it immediately (it's assumed to be a direct-mode command if there's no opening line number).

The really clever bit comes in the phrase 'the line where the cursor happens to be'. The cursor does not have to be at the *end* of the line before you hit Return – it can be hanging around *anywhere* in the line for *all* of that line to be accepted by the computer. And the 'line' isn't necessarily one horizontal strip across the display; the 64 can work with lines of up to 80 characters in length, so the cursor can be anywhere in the first or second 'lines' of a long command or statement.

The cleverest part of all though is the way you can take the cursor back to a line that's anywhere on the screen, make some changes if you want to, and then press Return to execute or accept it. With many computers you can't do this; even though you may be able to move the cursor back like that, you would have to retype the whole line anyway if you want to repeat or slightly modify it.

Incidentally, on some computers the Return key is called 'Enter'. That's actually much more sensible, since it more accurately describes its function. 'Return' is a hangover from the early days of computing (and typewriting), when display screens weren't the norm and instead you got a printed copy of input and output. 'Return' is short for 'Carriage return', and pressing that key 'returned' the printing element and/or the typewriter carriage to the lefthand position on the next line. I suppose it's still just about justifiable, since Return does 'return' the cursor to the left-hand position on the next line ... 'Enter' would still have been better, though.

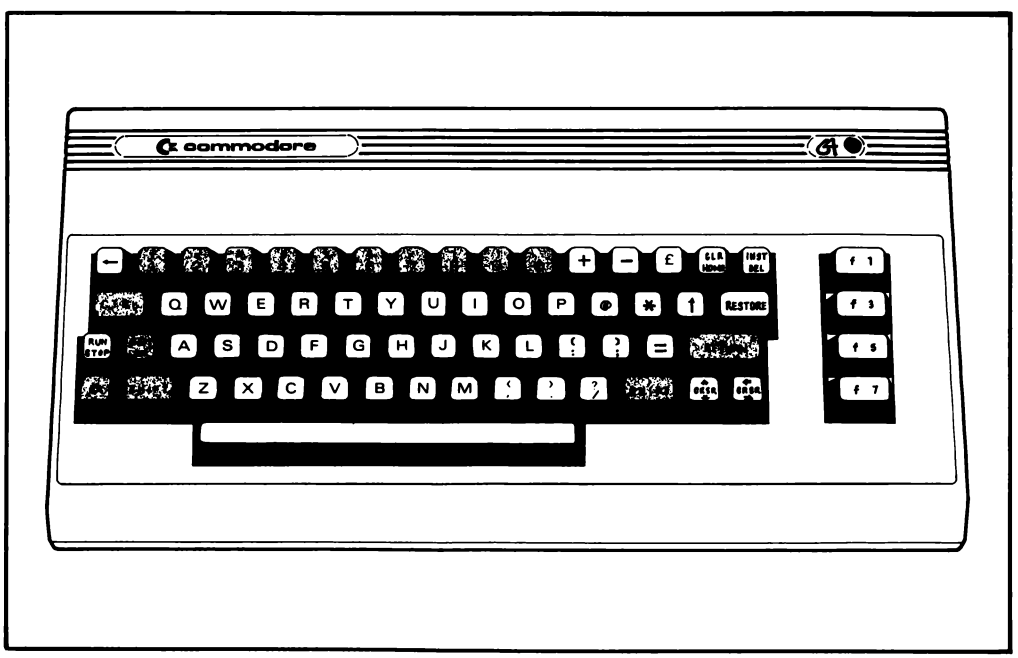
- **CTRL**: The CTRL key is the biggish one at the top. Press it when you are LISTing or RUNning a program, and things will slow down.

- **CTRL and colour changes**: under the number keys 1 to 8 are marked eight of the colours in which the 64 can display text on the screen; you pick your preferred colour, and the cursor will immediately change to that with all text and graphics subsequently appearing in it too. (You can also specify colour combinations for the border and background, but that's done in a different way). The abbreviations for colour are all fairly obvious – from left to right they are black, white, red, cyan (a yukky light blue), purple, green, blue (darker than cyan – more of a royal blue) and yellow.

The CTRL key is used as yet another 'Shift' key to get at the colour-change functions marked along the front of the number keys in the top row.

By pressing CTRL and one of the colour keys you change the cursor's colour right away; and anything you type subsequently will be in the new colour. You can change these colours as often as you want by using CTRL with one of those numbers.

The table summarises the colours available:



Colour selections with the CTRL key

Press CTRL key and ... ... to get:

1 black
2 white
3 red
4 cyan
5 purple
6 green
7 blue
8 yellow

Colour selections with the CBM key

Press CBM key and ... ... to get:

1 orange
2 brown
3 light red (pink)
4 dark grey
5 medium gray
6 light green
7 light blue
8 light grey

- **CTRL and reversed characters**: you also use CTRL to switch in and out of 'reverse' display for anything you type – CTRL and RVS ON (the key has '9' on top) turns it on, CTRL and RVS ON (the key with '0' on top) switches it off. You won't notice any difference to the cursor; but when you type something you'll see it come up with the light and coloured areas of the character exchanged

Actually, CTRL can be used with other keys on the keyboard to duplicate various functions more easily and/or more obviously produced by other means. For instance:

CTRL Q moves the cursor down
CTRL R switches on reversed printing
CTRL T takes the cursor back to the *end* of the previous line (could be useful in editing
CTRL ↑ changes to green
CTRL S duplicates the HOME key
CTRL ; moves the cursor forward one character
CTRL = changes to blue
CTRL N changes to lower-case
CTRL M duplicates Return
CTRL Shifted Q moves the cursor down one line
CTRL Shifted T moves the cursor back one character

- **CBM and colour changes**: there are another eight colours that aren't marked on the keys; those you get at by using the CBM key in much the same way as CTRL – press a number while holding down CBM. The table summarises the colours available:
- **RUN/STOP**: this has two functions. Used without Shift, it will STOP a program currently RUNning and tell you which line number of the program was being executed when you halted things. If there's no program running, pressing RUN/STOP won't have any effect at all.

When you hold down Shift and press the same key, you have a quick and easy way of starting to read something from the cassette unit – you'll be invited to 'PRESS PLAY ON TAPE' just as if you'd typed 'LOAD'.

- **RESTORE**: the biggish key on the right of the second row down labelled 'RESTORE' works only if you press it *while holding down the RUN/STOP key.* It clears the screen and leaves the cursor blinking at you underneath 'READY'. This resets things as if you'd just turned the 64 on the first time (it 'RESTOREs' the original status).

But there is an important plus: if you turn the 64 off then on again, you'll lose any program that was in the memory. Using RUN/STOP and RESTORE *doesn't* erase the memory, and if you type LIST you will be able to see a program that you previously had in memory.

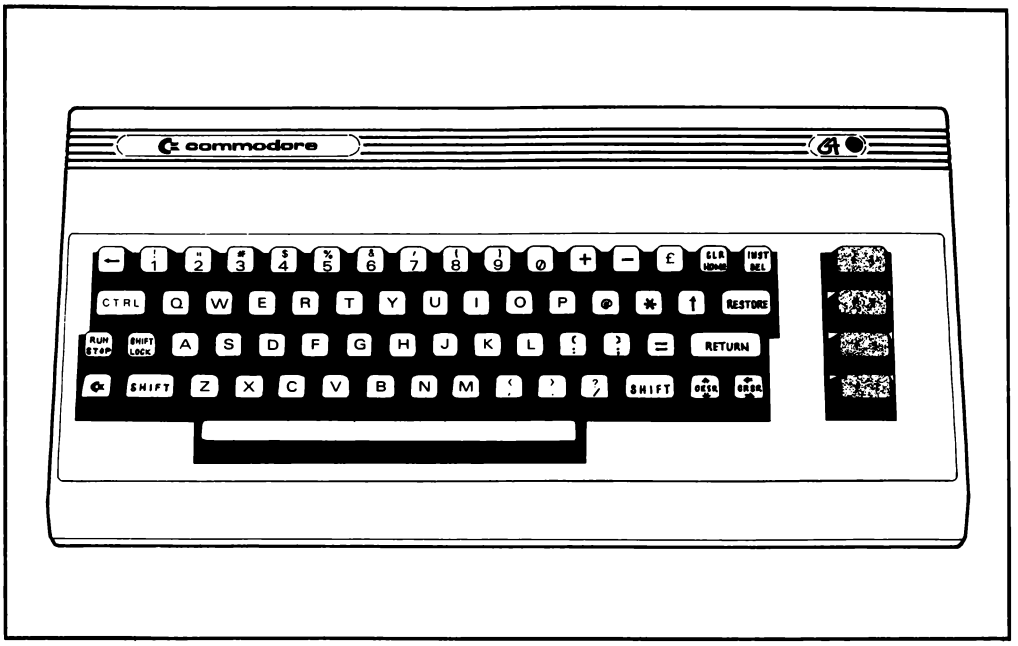
The function keys

The four brown keys on the right are labelled f1 to f4 on top, f5 to f8 on the front: the 'f' stands for 'function'. These keys don't do anything unless a program tells them to. In other words, they have a *function* given to them by a program.

They are used quite a lot in games, for instance. The person who wrote the game might tell the 64 that when 'f1' is pressed, it should display the instructions: and when f2 is pressed, it should start the game rolling. Load in a second program, though, and it may well give totally different functions to those keys.

Those four keys actually give you 12 possible functions. For four of them, numbers f1 to f4, you just type the key. For f5 to f8 you hold down Shift and hit the key. And what the 64 handbook doesn't tell is that you can have four *more* functions (f9 to f12, as it were) by holding down the CBM key and pressing the brown keys.

Of course, you don't *have* to use any of the function keys in your programs. But it's often helpful to do so; we'll show you how later in the book.


commodore
CTRL
Q W E R T Y U I O P
RESTORE
RUN STOP
SHIFT LOCK
A S D F G H J K L
RETURN
SHIFT
Z X C V B N M
SHIFT

Conventions

Through this book I've used a number of different ways of explaining how the 64's commands work; and I've also tried to simplify the reproduction of the cursor and colour control characters.

- When you see a statement with something in *italics*, that's either a description of what goes in there or something you type in yourself

- For clarity I put more spaces into program lines than is absolutely necessary. When trying these you probably won't need to use so many spaces; and in some cases you'll have to shorten the lines anyhow for they won't fit on to the 64's screen otherwise

- 'CTRL' means the ConTRol key; 'CBM' means the key with the Commodore logo; 'Shift' means either of the two Shift keys

- anything enclosed by the 'greater than' and 'less than' brackets means 'press a key' – as follows:

<**cup**> ... *cursor up*
<**cud**> ... *cursor down*
<**cul**> ... *cursor left*
<**cur**> ... *cursor right*

<**home**> ... *CLR/HOME*
<**clr**> ... *Shifted CLR/HOME*
<**del**> ... *DELete*

<**blk**> ... *black – CTRL and 1*
<**wht**> ... *white – CTRL and 2*
<**red**> ... *red – CTRL and 3*
<**cyn**> ... *cyan – CTRL and 4*

<**pur**> ... *purple – CTRL and 5*
<**grn**> ... *green – CTRL and 6*
<**blu**> ... *blue – CTRL and 7*
<**yel**> ... *yellow – CTRL and 8*

<**orng**> ... *orange – CBM and 1*
<**brn**> ... *brown – CBM and 2*
<**lred**> ... *light red – CBM and 3*
<**gr1**> ... *dark grey – CBM and 4*
<**gr2**> ... *medium grey – CBM and 5*
<**lgrn**> ... *light green – CBM and 6*
<**lblu**> ... *light blue – CBM and 7*
<**gr3**> ... *light grey – CBM and 8*

<**rvs**> ... *reverse on – CTRL and 0*
<**rvo**> ... *reverse off – CTRL and 0*

<**space**> ... *space bar*

<**f1**> ... *function key 1*
<**f2**> ... *function key 2*
<**f3**> ... *function key 3*
<**f4**> ... *function key 4*
<**f5**> ... *function key 5*
<**f6**> ... *function key 6*
<**f7**> ... *function key 7*
<**f8**> ... *function key 8*

- when I'm suggesting you press a particular key more than once, it will either appear like this:

```
PRINT "<cud><cud>"
```

Or, when it's more than one or two presses, I use a 'times' symbol – meaning 'press that key the number of times indicated'. This asks for you to press the space bar four times:

```
PRINT "<space x 4>"
```

Section 2

Programming

CHAPTER 5
First Steps in Programming

A computer is a fast rule-following idiot. It's an idiot because all it can do is follow rules, doing what it's told: and it can't make judgements about the efficacy or efficiency of those rules – it can't detect and amend an inappropriate rule, it won't take any decisions for itself.

The rules are contained in a *program*, and that's all that a program is: a list of rules, instructions, commands for the 64 to follow blindly. The rules can actually be quite complicated, telling the computer to do different kinds of things depending on circumstances at the time. But they're still no more than a bunch of orders or instructions.

Some background: when you switch the 64 on it automatically gets going on a big built-in program that's permanently there, embedded back at the factory in Read-Only Memory (that is actually a type of memory chip rather different to the chips used for memory whose contents you can alter, and unlike the latter – which is called 'RAM' – the contents of a ROM chip are not lost when you switch off the power).

This built-in program does a lot of things that you don't notice particularly; the main effect is to provide a means of getting information into and out of the computer. The 64 actually operates with streams of electrons; the resident program gives you a way of redirecting and manipulating those electrons by allowing you to use real words.

Its main task is a massive and continuous conversion job – like figuring out whether any keys have been pressed, deciding *which* keys were pressed, determining what you're trying to tell the 64, translating that into terms the electronic innards can understand ...

Part of that translation process includes setting up some words for you to use, words that you can type in on the keyboard and see on the screen. Its 'dictionary' isn't particularly large, though; there are only a few dozen words that it can recognise and translate. Which is bad news in one way, since you can't just type in whatever you want: that internal converter won't recognise it. But it's good news too, since the relatively small number of recognisable, permitted words means you don't have too many to remember.

As well as being limited in its total vocabulary, the 64 is also limited in terms of its grammar. You have to use the words according to a specific set of grammatical rules, just as you do for English or French.

So when you switch on and the screen illuminates with the start-up messages and READY, the 64 is sitting there waiting for you to type in something that it recognises – and in a form that it recognises.

What you can type in is 'Basic'.

The basics of Basic

Some revolutions are quick, big and noisy: some are gently, small and quiet. Nearly 20 years ago a couple of blokes in New England started one of the latter. What Tom Kurtz and John Kemeny did at Dartmouth College was to devise a way of letting 'ordinary' people get a computers: they developed a programming 'language' called Beginners' All-purpose Symbolic Instruction code.

Like any other language, programming languages have a vocabulary – a set of acceptable words with specific meanings. And they have a grammar, the rules by which those words are strung together into 'sentences' and longer collections of meanings.

Basic wasn't the first programming language by any means, but it was the first that was deliberately designed to be relatively simple. All the other programming languages around at the time were really quite complicated, demanding a lot of learning and expertise before they could be used sensibly; with Basic you could start writing programs pretty well straight away.

That's one reason why Basic is the programming language offered as standard on just about all small computers. And in the case of Commodore's computers (as with many others) the programming language is actually built into the machine at the factory. The words and grammatical rules that the 64 recognises are taken from the Basic language.

There are other reasons why Basic is so widespread: like it's relatively easy and quite cheap to give a small computer this language. And thanks to the efforts of a company called Microsoft, a very reasonable version of the original Dartmouth College Basic was available to microcomputer makers just at the time when they needed something – anything – to put into their electronic marvels.

Basic is no longer alone as an easy-to-use quick-to-get-going programming language, and for a variety of reasons it's probably not the best programming language for beginners anyway – nor is it the bees' knees for more advanced programmers. But it's a reasonable compromise, and in any case it's what you get with the 64 when you switch on ...

Commodore Basic

No micro these days uses an exact version of the original Dartmouth Basic. New facilities have been added, with new words and new grammatical rules; and today you'll find very few Basic-computers computers with identical versions of Basic. Even Commodore's own computers exhibit this trait: Basic on the Vic is the same as Basic on the 64, but it's different to Basic on the Pet/8000/700 computers and both are different to Basic on the 264/364 line.

On the Vic and 64 you get Basic 2.0, meaning version 2 of Commodore's implementation of Basic. Version 2 has more facilities than version 1, which appeared on the original Pets; more important, Version 2 includes a lot of error-correction that fixed some of the bugs in Version 1. Version 4 on the bigger Commodore computers have an upgraded Basic featuring all of the facilities in Version 2 with several extensions and new commands. And by comparison with Versions 2 and 4, Version 3.5 on the newer 264/364 machines has *many* extra commands available for you to use.

'Direct' vs 'Program' mode

Three other important points about programs: they can be altered and amended quite easily, they won't actually start working until you set them in motion somehow, and they can be saved for re-use sometime in the future.

All that distinguishes programs from commands entered in what's called 'Direct' or 'Immediate' mode.

The Basic you get with the 64 allows you to type stuff into the computer in one of two ways: you can write programs, where the end of one line is signaled by your pressing the Return key – but nothing will happen until you type 'RUN' to start the program.

Or you can type one or more instructions that will be acted upon as soon as you hit Return. And that is 'Direct' or 'Immediate' mode, since the 64 does what you tell it *immediately*.

For instance, you could try typing ...

```
PRINT 5 + 55
```

PRINT is one of the words Basic recognises, an instruction that tells the 64 to display something on the screen. The 64 doesn't care whether you typed it in capitals or in lower-case; and in fact you could have omitted the spaces – 'PRINT5+55' will work just as well.

This is actually an instruction to display the resulting of adding 3 and 55. Why 'PRINT'? Because it's a hangover from the days when computers didn't have screens: they all used to use typewriter-like printers to show you what you'd done.

You tell the computer you've finished by pressing the big key labelled RETURN. (Why 'Return'? Because it takes the cursor on to the start of the next

Order of calculation

When a number of operators are used together in one expression, the order in which they are evaluated and operated upon is ...

1. ↑ exponentiation
2. – negation
3. * multiplication
   / division
4. + addition
   – subtraction
5. = equality
   <> not equal to
   < less than
   > greater than
   <= less than or equal to
   >= greater than or equal to
6. NOT logical NOT
7. AND logical AND
8. OR logical OR

Expressions within brackets are evaluated first, however, so parentheses can be used to force a different order of calculation.

Operators at the same level of priority are evaluated in the order in which they appear.

line – the cursor 'returns' to its start position on a line.) Pressing Return should immediately give you the answer to 5+55 on the screen, with the 64 telling you it's READY for your next command.

But why can't you just type '3+5=' as on a pocket calculator? Well, the 64 needs an instruction of some kind to tell it which of the many functions available you actually want. That's one of the key elements of Basic: you (almost) always need word(s) that instruct the 64 to do something.

You can string together several arithmetical operations. Try another sum:

```
PRINT 19*6.6-21.672/2
```

Use the asterisk for multiply (it's '*' to avoid any confusion with the letter 'x'), the slash ('/') for divide. You can also include brackets to force the 64 to do its arithmetic in the right order. So ...

```
PRINT 3.4/67+0.25
```

... will produce a different answer from this:

```
PRINT 3.4/(67+0.25)
```

And you can include the two ready-to-go arithmetic functions on the 64's keyboard, the upwards-point arrow (the exponentiation symbol, meaning 'to the power of') and the mathematical constant *pi* (use Shift on the same key – *pi* is taken as 3.14159265):

```
PRINT π * 23.5 ↑ 2
```

This is that old familiar πR<squared>, the formula for calculating the area of a circle from its radius given here as 23.5. (23.5 ↑ is 23.5 to the power of 2, which is the same as 23.5 squared.)

Characters

You can use PRINT like this to display anything you want: it doesn't have to be the result of a calculation. But you do have to put quotation marks around what you want displayed, otherwise the 64 assumes it's something to be worked out. For instance:

```
PRINT "3.4/(67+0.25)"
```

Try typing that, and the 64 will display everything in the quotes *without* doing the calculation. So obviously you can put letters inside the quotes:

```
PRINT "HELLO MUM"
```

Syntax errors

'PRINT' is one of the words that the 64 understands. If you type in a misspelled command or anything else the 64 *doesn't* recognise, it'll let you know. For instance, try typing HELLO MUM without the PRINT command – you'll get this:

```
HELLO MUM

?SYNTAX ERROR
READY
```

'SYNTAX ERROR' means the 64 does not recognise 'HELLO' as a word in its vocabulary. You would get the same result if you'd mistyped the command ...

```
PRONT "HELLO MUM"

?SYNTAX ERROR
READY
```

And it'll happen again if you omit the quotes:

```
PRINT HELLO MUM

?SYNTAX ERROR
READY
```

Numerics vs characters

In the latter case you're getting the SYNTAX ERROR because one of the rules in Basic makes a distinction between *numeric* information and *strings*. A 'string' is a collection of characters – a 'string' of characters, in fact, though the string can be a single character.

In other words, a string is just about anything that can be typed at the keyboard – letters, numbers, punctuation marks, other signs and symbols. From Basic's point of view a string is anything that's enclosed in quotation marks; and if the quote marks aren't there, it will assume that what follows is a number.

Clearly when we tried PRINT HELLO MUM the 64 discovered that while it was expecting a number (no quotes) it didn't get one. Hence the SYNTAX ERROR.

Note that you can have numerals in a string as well as without the quotes. The difference? In the latter case the numerals can be used in arithmetic; as part of a string the numbers are just characters, like any other characters, and they can't be used as values in arithmetic.

Programs

Nearly all of the words Basic recognises can be used in Direct mode. A few can't, though, because they have to refer to *line numbers*.

Line numbers are what distinguishes Program mode from Direct mode. If the 64 sees that something you've typed starts with a number, it assumes that it's part of a program.

Each line in a program has a number, and the numbers are used by the 64 to work through the program in the correct sequence: when you press Return after typing in a numbered line, the computer will tuck it away in memory.

If you later type a new line that has the same line number as something already in the memory, the original will be junked and the updated line will replace it at that line number.

Your lines don't have to be numbered '1', '2,' '3' and so on: you can use any numbers you like, for the 64 will just start at the lowest number you've used and then go on to the next lowest. (You can have line numbers going as high as 63339.)

This can be really useful in indicating different sections of your program: one can start at line 10, say, with another one at line 100 and a third at line 1000. It makes life a bit easier when you subsequently come to amend the program – and be warned: no program is perfect the first time around, they all need some kind of alteration before the finished version emerges.

What's more, you can type a line that starts with the number '20' *before* you key in one numbered '10'; and the 64 will automatically sort them into the correct sequence for you.

This means you can type in an entire program and later add lines that will be put into the middle of the program. Because you *can* do this, it makes sense to use widely-spaced line numbers the first time around: that way you'll be able to add new lines as you think of them, without having to overwrite any existing line numbers.

For this reason most programs you see (including those in this book) number lines in increments of 10; that gives you nine free line numbers between two existing ones for any additions.

Once you have a series of lines sitting in the 64's memory, you can start the program by typing 'RUN' – without a line number: in fact you're using this command in Direct mode.

Editing what's on the screen

If you have made any mistakes in your typing, don't worry: one of the really good aspects of all Commodore computers, a feature that has now been adopted by several rivals, is the way you can correct errors on the screen very easily.

This applies whether you're writing a program or doing something in Direct mode.

The two keys at the bottom right labelled 'CRSR' move the cursor around the screen. The right-pointing arrow means the cursor will move right when you touch the key: the left arrow moves it left when you hit this key while holding down either of the two keys marked 'SHIFT'. The other cursor control key works similarly to take the cursor down (unShifted) and up (with Shift).

To correct a mistake, then, you can use the CRSR keys to move up and down and/or from side to side – until the cursor is at the mistake.

- If you typed a wrong letter and want to *replace* it, put the cursor on top of the mistake and just type the correct key.
- If you want to *delete* something, sit the cursor on the letter immediately to the right of the offending character and hit the key marked INST/DEL. That should delete the error and close everything up.
- If you want to *insert* something, put the cursor at the point where you want the insertion and press INST/DEL while holding down a Shift key. This will start inserting spaces and moving text on the right of the cursor further along the line. You can insert as many spaces as you need, until the the text being shifted to the right hits the limit of 80 characters (two lines of the screen) which Basic imposes on your input. Once you've got enough spaces you can start typing in them; if you haven't inserted enough spaces you can put in some more just by hitting INST/DEL with SHift again. And if you've inserted too many spaces for the addition, it doesn't really matter – but they won't be closed up automatically by the 64, so if you want to get rid them you'll have to use INST/DEL to delete the unnecessary gap.
- And if you want to *add* something to the end of the line, you can just take the cursor to the position following the existing text and type away with your addition. Again, you can add anything you like until the total length of the 'line' (which is actually *two* lines on the screen) reaches 80 characters.

If you don't take your finger off the cursor movement keys or INST/DEL, they'll carry on moving anyhow.

Instant amendment

This editing of the screen isn't just for mistakes: you can overtype, delete and/or add *anything*. As soon as you've pressed Return after making a change anywhere on the screen, the revised contents of that line are taken as Gospel.

Which in Direct mode means that an instruction will be executed immediately with the amendment; and in program mode the revision will replace the existing contents of that line number.

Or you can actually alter the line number of a program, in which case the original line isn't affected at all – but a new line with the new line number is created for the program. If you've altered nothing else on that line, the new line will be exactly the same as the old one apart from its number. This is really handy, since it means much quicker programming. You can very quickly write a whole series of lines that are repeated with similar but not quite identical contents simply by typing the first one, using the CRSR keys to go back to it, altering the line number and changing whatever you want in its contents, and pressing Return ... then you can do it all again to create a third line, and a fourth, and as many as you need.

If you need to put in more lines than you've left line numbers for, you have a couple of options. Basic includes two commands that can Direct the 64 to go to some other line number, so you could just tack your addition on to the end of the program and include one of those 'go to' commands at the place where ideally you'd like to have put the amendment.

Alternatively you can just renumber lines by using the CRSR keys and overtyping a new line number on each of the lines that you want to have a higher number. If you're doing this, though, be careful that your new line numbers aren't already being used by a line that you want to retain in your program: new line numbers mean the new line displaces anything that currently has the same number.

And don't forget that this so-called 'renumbering' will just *copy* the original line against a new line number. It doesn't *move* the original to the new line number; that original line will still be there unless you then overwrite it or delete it.

Deleting lines is easy enough: you just type the line number followed by Return. That tells the 64 the line isn't required any more/

LISTing programs

To recall a program that has been RUN you'll have to get it back on to the screen. You do this by typing LIST (in Direct mode – no line number needed) and hitting the Return key.

The LIST command puts on to the screen whatever program is currently in the 64's memory; you can then edit and amend it at will, and try RUNning the revised version.

CHAPTER 6
Program Flow

Unless it's told otherwise, Basic will run through a program in line number sequence – executing the instructions on the lowest line number first, then those on the next highest, and so on. But in the last Chapter I mentioned in a throw-away remark that a program needn't necessarily RUN in the order of its line numbers. This Chapter considers some of the methods you can use to switch the order of execution.

GOing TO a different line

The GOTO command (it can be used as 'GO TO') is the most obvious way of switching the order in which the 64 executes program lines. It's used simply to tell Basic which line you want executed next; the computer will go off to that line and carry on executing statements in line number sequence from there.

```
10 PRINT "THIS IS THE OPENING LINE"
20 GOTO 100
30 PRINT "THIS IS THE THIRD LINE OF THE
PROGRAM"
40 PRINT "BUT IT'S THE THIRD ITEM TO BE
PRINTED": GOTO 200
100 PRINT "THIS IS THE FIFTH LINE OF THE
PROGRAM"
110 PRINT "BUT IT'S THE SECOND ITEM TO BE
PRINTED"
120 GOTO 30
200 PRINT "THAT'S ALL FOLKS"
```

Here line 10 is executed first; then the 64 gets to line 20, which switches it to do line 100. There it starts looking for the next number in sequence, finds and executes line 110, and reaches 120 – which takes it back to line 30. It executes the PRINT at line 30 and again starts dutifully on the next line in sequence ... which is a multiple-statement line containing another GOTO. And that one switches the program to line 200, after which it terminates since there aren't any more lines to be done.

You can switch the program around like this as much as you want. But you'd be daft to: the very presence of the GOTO command is regarded by some as a real blot on Basic's escutcheon, since the ability to hop around like that encourages really sloppy programming. With a bit of thought you can often avoid GOTOs: sloppy programs take up more memory and run more slowly, but more important you can easily lose track of what it is you're trying to do if the program isn't organised into neat and logical chunks.

Once helpful application for GOTO though is effectively to kill the READY message and the blinking cursor at the end of a program – in practice by making sure that the program never ends:

```
9999 GOTO 9999
```

This would be a bit pointless for many programs, since the only way to break out of it would be the somewhat drastic RUN/STOP and RESTORE. But it could be useful if you have a program that puts pretty graphics on to the screen which you don't want interrupted by READY.

Switch IF ...

Probably the most useful feature in Basic gives the computer the ability to make decisions – and this is also the most helpful use of GOTO, as it happens.

The IF/THEN construction does this: you have to give it something to work with (to test IF something is the case) – and you have to give it something to do THEN. The general format is ...

IF *expression to test* **THEN** *next action*

For example:

```
30 IF ALF < JON THEN PRINT "JON IS
OLDER": GOTO 100
40 PRINT "ALF IS OLDER"
100 PRINT "... BUT THEY'RE BOTH PRETTY
OLD"
```

Here the program takes decisions based on a comparison of the contents of two variables, ALF and JON: the action it takes depends on the results of that comparison. In line 30, if the value in ALF is less than that in JON the message gets PRINTed. The next statement Basic gets to is that GOTO 100, which means line 40 gets skipped.

But line 30 also illustrates an important effect of IF: if ALF is not less than JON – in other words, if the test didn't come out with a 'yes' result of some kind – *everything else on the line containing the IF is*

ignored. So in this case if ALF is the same age as JON or older, that first PRINT won't get PRINTed (obviously) *and* that GOTO won't be executed. The next line in sequence, which confirms the real position, is then executed; since there are no conditions attached to it, the conclusion will be PRINTed.

Either way the comment in 100 will be printed. With ALF younger than JON it'll be done because of the GOTO: if ALF is older it will be done anyway because it's the next statement in sequence.

Here's another way of doing the same thing with a few more extra, illustrating also how handy GOTO can be:

```
30 IF ALF < JON THEN GOTO 100
40 IF ALF > JON THEN GOTO 200
50 PRINT "ALF AND JON ARE THE SAME AGE"
60 GOTO 500
100 PRINT "JON IS OLDER"
110 GOTO 500
200 PRINT "ALF IS OLDER"
210 GOTO 500
500 PRINT "... BUT THEY'RE BOTH PRETTY
OLD"
```

Here's another important use of IF/THEN combined with GOTO: it's the GET statement described in Chapter 8. GET continuously checks the keyboard and tells the 64 what's happening there; if you press a key, GET will detect it immediately. So if you're asking the user to press 'Y' for 'Yes' and 'N' for 'No', an IF/THEN can test for that.

But once GET has got something, the program moves on to the next line. And if *nothing* is happening at the keyboard GET will pass on that information too. So you have to make sure that GET doesn't get passed by:

```
10 INPUT "PRESS ANY KEY"; K$
20 GET K$: IF K$="" THEN GOTO 20
30 PRINT "YOU PRESSED " K$
```

The INPUT statement is used with a variable, in this case one that I'm calling K$. The GET K$ in line 20 looks at the keyboard, and the IF statement there tells it to go back and look again if no key is being pressed. Only when something other than 'no key' is picked up by GET does the program proceed on to line 30, which tells you what key you did press.

In fact the use of IF ... THEN GOTO is so frequent in Basic that there are a couple of ways of abbreviating it – you can omit the word 'GOTO' in IF/THEN statements like this, or alternatively you can leave out the 'THEN'. These are both acceptable:

```
20 GET K$: IF K$="" THEN 20

20 GET K$: IF K$="" GOTO 20
```

Complicating IF/THEN statements

It's possible to use a kind of multiple IF/THEN test to make quite complex decisions which may be based on a lot of information. You do that by stringing together several expressions to be tested before you get to the THEN bit that decides what happens next; and you can use AND, OR and NOT for that.

```
30 IF ALF < JON AND ALF < JOE THEN 100
40 IF ALF > JON AND JON > JOE THEN 200
50 IF ALF > JON AND JON < JOE THEN PRINT
"ALF IS IN THE MIDDLE": GOTO 500
100 PRINT "ALF IS YOUNGEST"
110 GOTO 500
200 PRINT "ALF IS OLDEST"
210 GOTO 500
500 PRINT "... BUT THEY'RE ALL PRETTY
OLD"
```

Loops

GOTO can make a program jump around within itself; IF/THEN and GOTO can make it switch around according to circumstances. But GOTO makes only the one jump at a time: and if it jumps *back*, to a lower line number, the program will loop around indefinitely – every time it works through the line numbers and reaches that GOTO it will blithely leap back and do it all over again.

The FOR/NEXT combo gives you the chance to control the number of times a loop happens. There are several circumstance in which you'd want a collection of instructions to be executed a specified number of times; FOR/NEXT means you don't have to repeat them over and over again.

You put a FOR statement before the instructions, type them in once, and terminate the lot with a NEXT. The FOR includes an indicator of how many times they are to be executed:

FOR *indicator=start value* **TO** *end value*

The 'indicator' bit is a numeric variable – more about those in Chapter 7, but basically it's a letter or two that acts as a label. The start and end values can be anything you want, though usually you'll pick something like ...

```
10 FOR T=1 TO 10
20 PRINT "WE ARE THE GREATEST!"
30 NEXT T
40 PRINT "NEXT STATEMENT ..."
```

When it finds a FOR the 64 sets up a kind of invisible internal counter. This program starts the counter at 1, PRINTs the message once, gets to the NEXT statement, goes back to increment the counter, PRINTs it again – and carries on like that until the counter has reached 10. Only then will it go on to the next line in sequence.

But the count doesn't have to start at 1; it doesn't have to use whole numbers as the start and end values (you can put compound expressions in there); and it doesn't have to move by one at a time – you can tack on a STEP command which tells the

FOR statement how many increments it makes per cycle. For example:

```
10 PRINT "LEAP YEARS:"
20 FOR Y=1980 TO (1980+(4↑4)) STEP 4
30 PRINT Y
40 NEXT Y
```

You can omit the identifier with the NEXT statement. It's there though so that the right FOR loop gets executed at the right time when one FOR is running inside another – and yes, that can be done: it's called 'nesting' the loops.

Subroutines

If you have a bunch of Basic statements that are used frequently, you could have a GOTO to go to them whenever that was necessary ... except that it could get complicated and messy trying to keep track of all those GOTOs and making sure that the lines you've gone to don't themselves lead you into another GOTO.

Basic provides the neat and clean answer in the form of subroutines. These are clumps of program lines that can be located anywhere in the program. If the 64 gets to them in the normal course of things as it executes one line after another, it will try to execute the lines in the subroutine too: they're no different to any other program lines. But then the 64 will hit a RETURN statement and crash in puzzled confusion with a ?RETURN WITHOUT GOSUB error.

That RETURN marks the end of the subroutine. You have in the main body of the program a line like ...

```
200 GOSUB 1000
```

This works exactly like GOTO in that it switches program flow to the line number given. Unlike GOTO, though, the 64 can get back to the main program after it's executed the lines of the subroutine – and that is what the simple 'RETURN' statement tells it to do: it switches the program flow back to the line number immediately following the GOSUB.

One difference between GOSUB and GOTO: the GOSUB must be followed by a line number. Unlike GOTO, it can't decide where to go by checking on the current value of a variable or working out some compound expression.

Messages to the reader

Basic includes a helpful 'comment' facility that my programs employ a good deal; the REM statement lets you put messages of explanation into a program listing that helps to indicate what's going on where.

When Basic comes across a REM it ignores everything following it on the line, so you can be as generous as you like in your use of English. This means that while you can use REM on multiple-statement lines, the REM *must* be the last thing on the line – otherwise any statements following will be ignored.

This aspect of REM is also quite handy for debugging programs. You might want to delete a line temporarily and then re-insert it later; the way to do that is simply to INSert the word 'REM' after the line number (which means the line will be passed over by Basic) and then delete that REM when you want to restore the line. So much easier than deleting the whole line and retyping it ...

The anti-LISTing REMark

One interesting trick with REM: if the REM is followed by a Shifted L, the program magically becomes unLISTable – any attempt to LIST it results in a ?SYNTAX ERROR.

This doesn't exactly provide blanket protection for anyone trying to get at your programs, since someone who knows why it's crashing (like someone else who's read this book!) will simply be able to delete the offending REM line. Then the program will LIST normally. But you can make it difficult, by putting the REM with Shifted L on the end of a line containing vital variable-setting instructions or something like that: deleting the REM then deletes all the important stuff too.

And you can be really awkward: put the REM Shifted L on several lines of the program – all of them, maybe. No-one will be keen to do several dozen deletes and retypings.

Or you could start your program with 10 or 12 REMs containing the Shifted L: that'll fox the would-be LISTer.

Mind you, you'd better make sure *you* don't want to LIST the program yourself; keep a 'clean' copy somewhere, one without the Shifted-L REMs in it.

The pro-LISTing REMark

When you do LIST a program, you get a look at every line number in sequence. If you'd like better-looking LISTings, though, you can have a title preceding the main program without a line number. Put this as the first line of your program:

```
0 REM ""<del><rvs>TTTTTT<rvo>PUT
YOUR HEADING HERE"
```

(I came across this interesting effect with REM in an American magazine – sadly I've lost the original, which means I can't credit the author. Sorry, whoever you are ...)

Another trick to smarten up listings and make them more readable: a line number followed by a colon

and nothing else *will* be LISTed, so a couple of them followed by a REM line makes a good and legible distinction between two parts of a program –

```
998 :
999 :
1000 REM SUBROUTINES
```

The colon also allows you to include spaces in a program line, and that too can make listings more readable:

```
998 :
999 :
1000 REM:              SUBROUTINES

998 :
999 :
1000 :                 REM SUBROUTINES
```

END and STOP

Traditionally a Basic program ends with the statement END. This causes the program to stop running and returns you to the READY prompt.

It's of considerable value when your program has subroutines tacked on to the end of the main code, something like this:

```
10     REM initialisation
100    REM main program execution
starts here
110    REM includes jumps to
subroutines at line 1000 on
1000   REM subroutines start here
1010   REM includes RETURNs to main
program
```

If you didn't have some method of telling the 64 that your main program has completed, it would just run on into the subroutines and start executing the statements in there. Which means eventually it would come across a RETURN; and since it hadn't got there via a GOSUB, the program would die on you with a ?RETURN WITHOUT GOSUB error.

Inserting an END just before the subroutines, as with

```
999 END
```

... will guarantee that the subroutines will *only* be executed as the result of a GOSUB.

END has a couple of other uses, though. Once a program has ENDed it is possible to restart it with the CONT command in Direct mode – assuming there are more program lines to be executed. So END can be used as a debugging tool during program development, halting the program midway through so that you can examine variables or whatever just to make sure that everything's ok; if it is, you can get going again with CONT – whereupon the next line in sequence will be executed.

You can also use END to set up two programs in memory at the same time. To the 64 they would look like one single program, but there could be an END statement midway through.

Say you have one program numbered with lines 10 to 450; make line 460 an END and code your second program as lines 1000 onward. When you type 'RUN' the first program will be executed, and it will end when it hits the END statement. Then you could use 'RUN 1000' to start the second program.

The STOP command is identical in operation to END, except that it's more obviously designed to help program development: the program halts when it finds the STOP and tells you in what line it found the command. With END you don't get the BREAK IN *line number* message.

CHAPTER 7
Variables

The 64 can hold information for you until you want to use it. But to do that you have to give the information a kind of label or name for both of you to remember it by; subsequently you can refer to that information by using the label.

Once you've set up a label, you can use it irrespective of the actual contents. So you can tell the 64 to use the name 'SUM' when it's printing the total of a list of numbers; and provided there is an actual number that you've somehow associated with that name, the value labelled 'SUM' will be printed. This means the statement PRINT SUM can be used to print your total *whatever the value happens to be at the time.*

In other words, the value identified by a particular label name can vary. That's why the names are called 'variables'. And as you might surmise, variables can be a really powerful tool in Basic.

The invisible command: LET

You don't have to tell the 64 explicitly that you're setting up a variable. It just spots the variable the first time you assign a value to something it regards as a valid type of name, and it records it internally for you.

The variable assignment command is LET, and it works like this:

LET *variable name = value*

But in fact there's a shorthand form of the command which omits 'LET' completely. The 64 assumes that you're setting up or changing the value of the named variable when it comes across ...

variable name = value

Nobody uses the word 'LET' in variable assignments. You can if you want, but I don't bother. Using variables means you don't have to know exactly **where** in its memory the 64 has put a particular; and you don't have to know exactly **what** the value is – you can always use the variable name to get at it.

Variables work like this. Say you want to work out how much you are spending. You could have a different variable for each type of expense, and you might put different values into them:

```
10 PHONE = 34.55
20 FOOD = 32.05
30 MISC = 121.77
```

It's as simple as that. And you can use expressions to assign a value to a variable:

```
10 PHONE = 11.27 + 34.55
20 FOOD = 4.68 + 8.12 + 7.35 + 5.5
```

What's more, you can assign new values to variables based on whatever happens to be in *other* variables ...

```
100 SUM=PHONE + FOOD + MISC
```

... and you can use variables inside expressions:

```
100 SUM=(PHONE + FOOD + MISC) * 1.1
```

(That makes sure there is a contingency allowance of 10 per cent in your expenditure total, just in case there are some bills you forgot). You could also have done that like this:

```
100 SUM=PHONE + FOOD + MISC
110 SUM=SUM*1.1
```

There you're multiplying the *original* value of the variable SUM by 1.1, and assigning the *new* value that results to the *same* name. So the next time you use the variable SUM it's value will have changed; and you'll have lost the original value in SUM. If you didn't want that to happen, of course, you could have created *another* variable for the grand total:

```
100 SUM=PHONE + FOOD + MISC
110 GT=SUM*1.1
```

Once you've got your variables you can utilise them in other Basic commands. The most obvious is PRINT:

```
120 PRINT "SUM =" SUM
130 PRINT "GRAND TOTAL =" GT
```

And since PRINT can use compounded expressions, line 130 might have been ...

```
130 PRINT "GRAND TOTAL WITH CONTINGENCY
=" SUM*1.1
```

Names for variables

The names for variables can be practically any length you like, but in fact the 64 looks at only the first two of them. In the example I've used names longer than two characters simply for the sake of clarity, and several times in this book you'll find that I have done this: but in fact I'd get the same results had I stuck to two characters:

```
10 PH = 34.55
20 FO = 32.05
30 MI = 121.77
100 SU=PH + FO + MI
110 GT=SU*1.1
120 PRINT "SUM =" SU
130 PRINT "GRAND TOTAL =" GT
```

Not that it has to be two characters: a variable name can be a *single* character.

```
10 P = 34.55
20 F = 32.05
30 M = 121.77
100 S=P+F+M
110 G=S*1.1
120 PRINT "SUM =" S
130 PRINT "GRAND TOTAL =" G
```

From that you can see that a large number of variable names are available to you. All must begin with a letter: so you can have 26 single-character names for starters. The second character can be a letter or a number, which means you can additionally have 676 two-letter names ('AA', 'AB' and so on: then 'BA', 'BB' ... to 'ZZ': making 26x26 possibilities) plus 260 names made up of one letter and one numeral ('A0' to 'A9' etc, to 'Z9' – which means 26x10 alternatives).

And that gives you a total of 962 variables in use at any time. (Actually, you can't use all 962 of them because half a dozen two-letter combinations are reserved by the 64 for specific purposes.)

Two things to watch – first, some home computers (notably Atari) will recognise variable names longer than two characters. So if you're transferring from one of those on to a 64, bear the restriction in mind; otherwise you might fall foul of the second potential problem, which is the fact that you can be seduced by the way you can use variable names of any length for the purposes of clarity. It's all too easy to forget that *only* the first two characters are recognised: you might want to distinguish between CAR costs and CAB expenses, but the 64 will see *both* of them as the same variable CA.

Types of variable

So far we've just been talking about assigning numbers to variables. You can also put text or characters into a variable; but you can't mix numbers and text in the one variable, not if you want to be able to use the number in arithmetic.

That gives two types of variable – *numeric* and *string*. There's a third type too, *integer* variables: they can only hold whole numbers without a decimal point.

The three types of variables are distinguished by an identifying mark following the name:

- a **numeric** variable has *nothing* following the name. Examples: A, AA, A1
- a **string** variable has *a dollar sign* following the name. Examples: A$, AA$, A1$
- an **integer** variable has *a percent sign* following the name. Examples: A%, AA%, A1%

Note that you can use the same characters in the names for different types of variables: so AA, AA$ and AA% are accepted by the 64 as three different variables. Which more or less trebles the total number of variables that you can have at any one time.

Once you have defined the type of variable by using a particular style of name, you are immediately restricted to assigning only that type of information to it:

- **numeric** variables can take only numbers. The numbers can be whole numbers or fractions, but you can't assign character strings to a numeric variable – or at least you can, but you'll find that the 64 rather snottily assigns a number (zero) to that name. For example:

```
10 A=1
20 B=1.25
30 C=SOME CHARACTERS
40 PRINT A, B, C
RUN
1     1.25    0

READY
```

There isn't a lot of restriction on the maximum size of number you can assign to a numeric variable. It can be up to nine digits in length, excluding decimal point and preceding sign if any: which means you can have a number like -1.02967854 (but nine 9s will be rounded up).

You can also have variables holding numbers much larger, though, because the 64 uses an exponent form to hold them – so it doesn't have to cram all the digits into the nine spaces available. That means you can have numbers up to several billion.

- **string** variables *must* be assigned one or more characters – which usually means a string of characters enclosed in quotes, though it could be one or more CHR$ codes (more about these in Chapter 14, but the CHR$ codes are a way of referencing particular characters directly in terms of the internal code number the 64 uses for them).

The string can include letters, numbers (though you can't use the number in calculations), colour and cursor controls ... anything in fact that you'd expect to be able to put inside quotes:

```
10 A$="HI THERE"
20 B$="1.25"
30 C$="<clr><rvs><red>A NEW SCREEN"
40 PRINT A$
50 PRINT B$
60 PRINT C$
```

If you omit the quotes, however, the 64 won't like it much:

```
10 A$="HI THERE"
20 B$=1.25
30 C$="<clr><rvs><red>A NEW SCREEN"
40 PRINT A$
50 PRINT B$
60 PRINT C$
RUN
?TYPE MISMATCH ERROR

READY
```

That error message refers to line 20 and means that you're trying to assign the wrong information for that type of variable: no quotes means it's a number, and the dollar sign after the variable name B$ means the 64 demands a string for it.

You'll get the same error message if you try to assign a string in quotes to a numeric or an integer variable.

The maximum size of a string assigned to a variable is 255 characters – excluding the quote marks. That's quite a lot; but note that it's the total length you can assign, which means that if you're (as it were) stringing together several strings with plus signs to assign to a single variable the whole lot added together cannot top 255 characters.

- an **integer** variable takes only the integer part of a number. That means you'd normally assign only whole numbers to an integer variable; and if you assigned a fractional number, only the part before the decimal point would be accepted. Some examples:

```
10 A%=125
20 B%=1.25
30 C%=.25
40 D%=INT(23.5/1.2)
50 E%=TEXT STRING
110 PRINT A%
120 PRINT B%
130 PRINT C%
140 PRINT D%
150 PRINT E%
RUN
 125
 1
 0
 19
 0

READY
```

Here A% is a whole number, so it was PRINTed in full. Only the part in front of the decimal point was used for B% and C%; and in the latter case that was assumed to be 0. For E% I tried to assign some text, but the 64 ignored it and made E%=0; had the text been enclosed in quotes I'd have got a ?TYPE MISMATCH ERROR.

D% illustrates a Basic expression that's very handy in assigning integer variables; putting INT before an expression that's enclosed in brackets means that only the integer part of the result will be used. In fact 23.5 divided by 1.2 is 19.5833333, but INT ensured that only the 19 was assigned to variable D%.

In fact INT is superfluous here. You'd have got the same result if INT had been omitted:

```
D%=23.5/1.2: PRINT D%
 19
```

But I used the INT example to show that it's there and it's possible. You're most likely to use INT with integer variable assignment in quite complex expressions, particularly when you want the number rounded up – neither INT nor the integer variable assignment will do any rounding for you: they just take the integer part of the number, irrespective of whether it's the nearest whole number. If you want rounding-up, add 0.5 to the number:

```
D%=INT(23.5/1.2+0.5): PRINT D%
 20
```

... or ...

```
D%=23.5/1.2+0.5: PRINT D$
 20
```

If you attempt to assign a number greater than 32767 or less than -32768 to an integer variable you'll get an ?ILLEGAL QUANTITY ERROR. Integer variables can't take numbers outside that range.

A quick reference: some example variable names

Variable name	Result
A A% A$	All are ok – any single alphabetic character can be used as a variable name, and names can be duplicated for different types of variable
AB AB% AB$	All are ok – most two-letter combinations are acceptable, again with names being repeated for different variable types
A1 A1% A1$	All are ok – all two-character letter-plus-number combinations are acceptable provided they start with the letter. Again, names can be being repeated for different variable types
1B 1B$ 1B%	No – variable names must start with a letter
11 11$ 11%	No – variable names must start with a letter
TI	No – these variable names are all reserved for the 64's internal clock
TI$	
OR OR$ OR%	No – OR is a Basic word
COUNTER CHARS$ INTEGERS%	Yes – variable names can be longer than two characters, though only the first two will be recognised and used
START ORANGE$ TOTAL%	No – the first two characters of all these are reserved words (ST, OR, TO)
ST	No – ST is a reserved word. But ...
ST$ ST%	Yes – both are valid variable names (only ST, the numeric variable name, is reserved)
COUNT$ CONTROL$	No – since only the first two characters will be recognised and used, these apparently different names in fact are both treated as the one variable CO

'Reserved' words

The words Commodore 64 Basic uses for its own statements may not be used within variables:

ABS
AND
ASC
ATN
CHR$
CLOSE
CLR
CMD
CONT
COS
DATA
DEF
DIM
END
EXP
FN
FOR
FRE
GET
GET#
GOSUB
GOTO
IF
INPUT
INPUT#
INT
LEFT$
LEN
LET
LIST
LOAD
LOG
MID$
NEW
NEXT
NOT
ON
OPEN
OR
PEEK
POKE
POS
PRINT
PRINT#
READ
REM
RESTORE
RETURN
RIGHT$
RND
RUN
SAVE
SGN
SIN
SPC(
ST
STEP
STOP
STR$
SYS
TAB(
TAN
THEN
TI
TI$
TO
USR
VAL
VERIFY
WAIT

'Reserved' variables

A number of variable names are reserved for use by the 64, and you will probably get a ?SYNTAX ERROR if you try to assign values to them as variables.

A couple are still variables of a kind, but they have a predefined role in the 64:

- **ST**, which is a 'STatus' variable into which the 64 puts information about what's happening to movement of information to and from disk and tape
- **TI**, used by the 64 to hold a 'TIme' counter

There is also one reserved character variable:

- **TI$**, the text form version of TI that holds the time in the form of hours, minutes and seconds rather than 'jiffies'

In addition there are some two-letter Basic statements that means you can't use the same letters for variable names:

- **TO**, which the 64 will assume is the 'TO' in the FOR statement
- **OR**, one of the 'conditional operators'
- **IF**, again a Basic word – part of the IF/THEN statement
- **GO**, which will be treated as part of a GOTO or GOSUB statement

This reduces the total number of acceptable names numeric variables by six to 954. And TI$ means you can use only (*only*?) 961 string variables. There are no reserved integer variables with the percent sign following, so you have the full panoply of 962 to choose from – making a grand total of 2,881.

On top of that you can define a whole wadge more as *array* variables: more about them later.

The point is you should never run short of possibilities for variable names, so you should never have to be in the position where you're accidentally re-using a variable name. You'll never ever need to get anywhere near the 64's limit on the maximum permitted number of variables; and in any case the 64 will run out of memory to handle variables long before you get near the limit, since it keeps an internal reference of all variable names and what's currently assigned to them – and that can soak up a lot of memory.

Handling time and using timers

Two of those reserved variables deserve special attention here. We've seen how a delay loop using FOR/NEXT can usefully be incorporated into programs. But the 64 can be cleverer than that at counting time – and even telling you what the time is.

The computer has an internal clock which starts ticking away as soon as you switch on. It can be read and restarted by you, too. Actually the 64 has *three* built-in clocks – we'll come to others later. But the easiest to use is what's generally called the TI clock.

That's because there is a 'reserved' variable called TIME, usually abbreviated to TI since the 64 only looks as the first two characters of the variable name anyhow. (Which means you can't use 'TI' as one of your own variable names, though you *can* have that pair of letters somewhere other than the first two positions in a long variable name – 'NIGHTTIME' is acceptable as a variable name, 'TIDINESS' isn't.)

Whenever you reference TI the computer goes away to check how long it's been since you switched on or since you reset the clock. It does that by looking at a bunch of memory locations (which happen to be 160-162, though you needn't necessarily use them directly). In those the 64 keeps a counter that is incremented 60 times a second; and in Commodore parlance, 1/60th of a second is a 'jiffy'.

If you simply look at TI, you might not get too helpful a result – for instance:

```
PRINT TI
 9723
```

That means 9,723 jiffies have elapsed since you switched on (obviously the number you actually get when you try it will depend on exactly how long it's been since you *did* switch on). But you can convert it into something more meaningful simply by dividing by 60. Had you done that, you'd have seen ...

```
PRINT TI/60
 162.05
```

And that would be the 9,723 jiffies converted into seconds.

Using TI in timing loops

The conventional way to build a pause or some other kind of time delay into a program is to use a FOR/NEXT loop ...

```
100 FOR T=1 TO 999: NEXT T
```

You can however use TI as an alternative, and that way you'll know just how long the delay will be:

```
100 T=TI
110 IF TI<T+120 THEN 110
```

That gives a delay of precisely two seconds (120 jiffies = 120/60 = 2 seconds). Alter the number to give a longer or shorter delay.

Using TI to time actions

If you want to time something like a reaction test, the simple way is to take a snapshot of TI as the start time in one variable and do that again as a second variable when the action is complete. Subtract the first variable from the second and you have the elapsed time in jiffies. Divide that by 60 and you've got seconds.

Here's a simple speed reaction test that illustrates it. As soon as the timed action starts, a snapshot of TI is taken as variable TA by line 130; when a key is pressed, TB holds the stop time.

```
100 T=TI
110 PRINT "PRESS ANY KEY WHEN YOU SEE THE
SCREEN CLEAR"
120 IF TI<T+120 THEN 110
130 PRINT "<clr>": TA=TI
140 GET A$: IF A$="" THEN 140
150 TB=TI
160 PRINT "THAT TOOK " (TB-TA)/60
"SECONDS"
```

TI and tenths of a second

The result of that little program is likely to show reaction time to nine decimal places. To make it a bit neater, you'll probably want to show tenths of seconds to two decimal places only. The following procedure works for any use of TI:

- divide by 60 to get seconds from the jiffy count
- multiply by 100 to get the two most significant digits
- add 0.5 to make sure it's rounded
- use INT to lop off the rest of the fraction
- divide the result of that lot by 100 to restore the decimal point.

And to summarise that lot as a single expression ...

tenths = INT((time/60)*100+0.5)/100

So you could replace line 160 above with this:

```
160 PRINT "THAT TOOK " INT(((TB-TA)/
60)*100+0.5)/100 "SECONDS"
```

And if you're going to want to use tenths of a second frequently in your program, it might be worth using DEF FN to define a function for it:

```
5 DEF FNTENTHS(T) = INT((T/
60)*100+0.5)/100
10 program continues ...
160 PRINT "THAT TOOK " FNTENTHS(TB-TA)
"SECONDS"
```

Resetting the jiffy clock

You can't reset the TI clock directly – in other words, to restart the jiffy counter you can't just say ...

```
TI=0
```

You have two options to do that. One is to get directly at the TI memory locations and POKE zero into all of them:

```
100 FOR T=160 TO 162: POKE T,0: NEXT
```

The other way is to use a related variable, another 'reserved' one, called TIME$ – more usually in its abbreviated form TI$.

As the dollar sign indicates, this is a string variable; but the string it contains states the time in the form of six digits – three pairs corresponding to hours, minutes and seconds in that order. It doesn't come out with clear delineations between the three, but it has the format

hhmmss

If you were to type ...

```
PRINT TI$
```

... the 64 would rush off to check the TI counter, convert the jiffies into hours/minutes/seconds, and display the result.

It works in reverse, too: you can set TI$ to a specific time and the 64 will then use it to reset the TI clock. You can see that at work by typing this line in direct mode:

```
PRINT TI, TI$: TI$="000000": PRINT TI,
TI$
 10864         000301
 0             000000
```

The first time you PRINTed TI and TI$, you got a response of 10,864 jiffies. Dividing that by 60 will tell you 181 and a bit seconds had elapsed, which is three minutes and one second: exactly what the first TI$ produced – no hours, three minutes, one seconds.

The you've filled TI$ with zeros – note that you need six digits of one kind or another when you're setting TI$, and those quote marks must be there. And the

second lot of PRINTing shows that both TI and TI$ are now at zero; no time at all has elapsed since the 64 executed the TI$ reset, since the PRINT was the next thing it did ... and it did it almost immediately.

A more practical demonstration. Type this:

```
10 INPUT "COUNT"; C
20 TI$="000000"
30 FOR D=1 TO C: NEXT
40 PRINT "TIME =" TI, "TIME$ =" TI$
50 GOTO 10
```

When you RUN this, try different values for C. Here's one run:

```
RUN
COUNT? 10
TIME = 1        TIME$=000000
COUNT? 100
TIME = 8        TIME$=000000
COUNT? 1000
TIME = 63       TIME$=000001
COUNT? 10000
TIME = 679      TIME$=000011
COUNT? 100000
TIME = 7023     TIME$=000157
```

Note how TI$ doesn't pick up fractions of a second, so it won't get incremented at all until the number of jiffies in TI tops 60.

You can of course set TI$ to any time you want. The only restrictions are that the first two digits must be no greater than 23 (TI$ uses a 24-hour clock, with midnight being 0 o'clock and 11pm being 23 o'clock): the next pair (minutes) shouldn't be greater than 59: and nor should the last pair (seconds). All very obvious really.

You can use the string-handling functions to separate the hours, minutes and seconds. This example just puts colons between them:

```
100 PRINT LEFT$(TI$,2) ":"
MID$(TI$,3,2) ":" RIGHT$(TI$,2)
```

This one works out the AM/PM bit and presents the time more conventionally, just like the Speaking Clock:

```
100 INPUT "TIME NOW"; TI$
110 AMPM$="AM": HOURS =
VAL(LEFT$(TI$,2)
120 IF HOURS >12 THEN AMPM$="PM"
130 IF HOURS >13 THEN HOURS=HOURS-12
130 PRINT "TIME IS " HOURS ":"
MID$(TI$,3,2) AMPM$ " AND "
RIGHT$(TI$,2) " SECONDS"
```

(You'll need to use keyword abbreviations to get all that on to line 130.) Practical applications include timing how long a program takes to run – set TI$ to '000000' at the start of things, PRINT TI$ at the end – and creating alarm clocks for yourself (set TI$ to the right time, then have the 64 beep at you or flash the screen somehow when TI$ reaches your alarm time).

The bug in TI and TI$

Sometimes, though, TI and TI$ appear to lose time. This happens after a SAVE or LOAD from disk to tape, and after you use a printer. That's because the internal housekeeping routines which increment the TI clock 60 times a second also check the cassette and serial ports – and if something is happening there the TI increment doesn't get done as regularly as it should.

There's nothing you can do about this except be aware that it happens ... Or use one of the alternative clocks.

The *other* clocks

In addition to TI and TI$, the 64 has not one but two extra clocks; and unlike TI/TI$ they work independently of the processor, so that anything else going on has no effect on them.

These two clocks are duplicated at locations 56328-56331 and 56584-65687, as follows:

clock 1 location	*clock 2 location*	*function*
56328	56584	counts tenths of seconds
56329	56585	counts seconds
56330	56586	counts minutes
56331	56587	counts hours and indicates AM or PM

What happens is that the tenths-of-seconds counter is incremented until it reaches 10, then the seconds counter is incremented ... and so on until the hours counter is incremented. When that one hits 12, it flips the AM/PM indicator (in bit 7) and resets itself to 0.

Because these clocks tick away all the time once they're started, it could be tricky to read them – after all, if you happen to do that when it's 11:59 and 59.9 seconds, the time could change while you're looking at the clock. And since you have to read the four locations one by one, the time you end up with could be 11:00:00:00 if you read the hours first just as the others are changing. Or if you read the *seconds* while *hours* were changing you might get 12:59:59:59 ...

Fortunately the clocks have a helpful built-in feature: they freeze as soon as you read the hours. And it gets better: the clock actually keeps on running, and as soon as you read tenths of seconds the whole lot gets reset with the correct time again to pick up where it left off.

So all you have to do is ensure that you read the clock locations in reverse order: hours, then minutes, then seconds, then tenths. Not exactly a hardship, since that's the way you would normally display the clock anyhow. But if you're using the

clock as a timer, and you're interested only in seconds and tenths, remember that you have to PEEK at 56331 (or 56587, depending which clock you're using) before you can get an accurate reading from the rest of the clock. And if you're using a simple hours-and-minutes clock, don't forget that you have to PEEK at 56328 (or 56584) before you can restart.

The other essential is the location that sets the clock frequency – location 56334 for the clock starting at 56328, location 56590 for the other one. Bit 7 of these determine the frequency, and it's set to 1 for 50Hz or 0 for 60Hz. This is provided because 64s are sold around the world, and there are two common mains frequencies in the world – 50Hz like what we use here in Britain, 60Hz as in the States. Which among other things is why the TI/TI$ clock operates in 'jiffies' – 1/60ths of a second.

Now, the Hz setting determines how many electrical pulses the 64 gets per second; and all 64s sold in Britain are set up for the 50Hz rate. But lots of the 64's internal operations reflect the 60Hz from the land of its designers; and since the 64 uses those pulses as an internal timer to decide how quickly it does things, just about everything happens inside the 64 at the 60-times-per-second rate appropriate to the US.

The two clocks we're considering here are about the only aspects of the 64 which could be adversely affected by the 50/60Hz rate. Unless you alter the frequency setting in 56334 or 56590, the 64 will use the 60Hz rate – which means the clock will expect to be incremented *60* times a second. But because the electrical supply is coming in at *50*Hz, the clocks will run 10 seconds slow every minute.

Unless you set bit 7 or 56334 or 56590 to 1. Those locations have many other functions, so you can't just POKE them with decimal 128 to get bit 7 set to 1; you have to use an OR ...

```
POKE 56334,PEEK(56334) OR 128
```

... or ...

```
POKE 56590,PEEK(56590) OR 128
```

And that makes sure the clock runs at the right speed.

There's one other problem in actually reading the clock. The designers deliberately made it easier to use from machine code rather than Basic, and that means they employed a number system called 'Binary Coded Decimal'. Instead of storing the clock values as the binary equivalent of a decimal number – like '47', say, which you might expect to find as '00101111' – they're held as two decimal values, '4' and '7'. And the actual value stored in binary form is then obtained by multiplying the left-hand digit by 16 and adding the right: in location 56329, 47 seconds would be represented as '01000111' (decimal 71) because it's (4 x 16) + 7 = 71.

Sounds complicated? Well, it's done like that because the processor finds it easier to get at the decimal equivalent of the number – it takes only four machine-code instructions to divide by 16 to find the left-hand part of the number. Otherwise it would have to divide by 10, which is much more complicated for it.

Single-digit numbers don't need any conversion – obviously a number like '9' is treated as '09' and translates into BCD as (0 x 16) + 9 = 9.

```
10     REM enter the time
20 INPUT "CURRENT TIME (HH-MM-SS)";T$
30 IF LEN(T$)>8 THEN PRINT "WRONG-TRY
AGAIN"
40 GOTO 20

50     REM input for AM/PM and hours
60 HOUR=VAL(LEFT$(T$,2))
70 IF HOUR<0 OR HOUR>24 THEN PRINT
"WRONG-TRY AGAIN": GOTO 20

80     REM set minutes
90 MIN=VAL(MID$(T$,3,2))
100 IF MIN<0 OR MIN>60 THEN PRINT
"WRONG-TRY AGAIN": GOTO 20

110    REM set seconds
120 SEC=VAL(RIGHT$(T$,2))
130 IF SEC<0 OR SEC>60 THEN PRINT
"WRONG-TRY AGAIN": GOTO 20

200    REM convert to BCD if necessary
210 IF HOUR>9 THEN HOUR=INT(((HOUR/
10)*16)+((HOUR/10)-10))
220 IF MIN>9 THEN MIN=INT(((MIN/
10)*16)+((MIN/10)-10))
230 IF SEC>9 THEN SEC=INT(((SEC/
10)*16)+((SEC/10)-10))

300    REM set frequency
310 POKE 56334,PEEK(56334) OR 128:
CLOCK=56328

320    REM set the time
330 IF HOUR>12 THEN HOUR=HOUR+128
340 POKE CLOCK+3,HOUR
350 POKE CLOCK+2,MIN
360 POKE CLOCK+1,SEC

400    REM now start the clock...
410 POKE CLOCK,0
```

Here the first four lines request the current time in a string variable T$ and checks that you've put in enough characters (use zeros if necessary to make sure there *are* enough characters). Numeric variables for hours, minutes and seconds are set up by taking the VALue of the relevant parts of the string in lines 60, 90 and 120 respectively.

If any of them are greater than 9 – in other words, if any take more than one digit – the complicated-looking calculations in lines 200-230 do a decimal-to-BCD conversion. To run through this:

(HOUR/10)*16) ... takes the left-hand digit of the two by dividing the number by 10. It's then multiplied by 16 to get the BCD equivalent

(HOUR/10)-10) ... dividing the number by 10 and then subtracting ten gives the right-hand digit

(HOUR/10)*16)+((HOUR/10)-10)) ... adding the two BCD components together now gives a BCD number that can be encoded as binary in the variable 'HOUR'

INT(((HOUR/10)*16)+((HOUR/10)-10)) ... the division by 10 is likely to produce a fractional number, and using the INT function will round it up into an integer for use as a whole number of hours (you're getting the fractions of hours in the MINS and SECS variables)

Then line 310 sets the frequency and simplifies things a bit by setting up a variable called CLOCK which contains the clock's start address of 56328. Line 330 sets the AM/PM indicator in bit 7 of the hours counter (adding 128 ensures that the binary pattern has a 1 in bit 7). 340 to 360 then POKE in the hours, minutes and seconds that you've entered; POKE CLOCK, 0 sets the clock going.

Thereafter you can use the clock in a program or display it. The clock will freeze as soon as you PEEK at CLOCK+3 500. But don't forget that to calculate the time you'll need to convert back from BCD – use something like ...

```
HOUR = (PEEK(CLOCK+3) AND 112) +
(PEEK(CLOCK+3) AND 15)
```

... to add the two parts of the number. AND 112 screens the lower three bits in the top half, those that correspond to 'tens' in the decimal number (use AND 128 separately to check the AM/PM flag); AND 15 compares all the bits in the lower half of the byte, which will give the rest of the digit.

CHAPTER 8
Getting Something In

You can't really have a program that doesn't require something to be typed in for it to get to work on – well you can, but it's hardly usual. There are a couple of Basic commands that you can use; and while they both have quirks, neither is too difficult to get on top of.

The obvious option: INPUT

Your first acquaintance with inputs should probably be the reasonably-named INPUT command. It looks like this:

```
10 INPUT X$
```

If you now type RUN you'll see a question mark appear. That means the computer has reached the INPUT and is waiting for you to put in something: so type whatever you feel like (not more than 255 characters, though that ought to be enough!) and press RETURN.

A bald question mark on the screen isn't too helpful, though. It makes more sense for the program to ask something specific, like 'WHAT'S THE DATE' or 'DID YOU SLEEP WELL'. You *can* do this by having the screen PRINT out a question and following it by an INPUT:

```
10 PRINT "HELLO. SLEEP WELL"
20 INPUT A$
```

There's a neater way, though. You can put the question actually in the INPUT command:

```
10 INPUT "HELLO. SLEEP WELL"; A$
```

Note that you don't need the question mark – INPUT will put that there for you. And you can't miss out that semi-colon after the quotes which close the message.

INPUTting numbers

It doesn't have to be text that you can INPUT – you can request a number rather than a bunch of characters. If you had INPUT A (with no dollar sign on the variable) the computer would expect a number to be typed in; and if you tried typing some *letters* you'd get a curt ?REDO FROM START message.

On the other hand if you're asking for a string of characters, as with INPUT A$, you can type in anything you like – including numbers. But you won't then be able to *use* those numbers in arithmetic, just as you can't use a spelled-out number like 'SEVEN' in a calculation: the dollar sign means a character variable, and the numerals 0 to 9 are treated as characters when they're given in response to a character input.

The only thing to watch for is a restriction on the length of the message you can have with an INPUT statement. This applies whether you're putting the message after the INPUT itself or before it in a PRINT statement (messages used like this are often called 'prompts', by the way, since they're prompting the user to react somehow). Your prompt can't be more than 44 characters long. And what happens if it is? Try this:

```
10 INPUT "A TEST FOR THE MAXIMUM INPUT
LINE LENGTH. WHY NOT TRY TYPING ANY
RESPONSE AFTER THE QUESTION MARK TO SEE
WHAT HAPPENS WHEN THE PROMPT LINE IS
LONGER THAN 44 CHARACTERS IN LENGTH";
A$
20 PRINT A$
```

When you RUN this, the prompt will lose everything after the first 44 characters. Nothing disastrous will happen; you just won't see the rest of your question.

Multiple INPUTs

Another idiosyncracy with INPUT is the way it doesn't like commas in your response. If you have a program that asks ...

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

... and you respond with 'JEREMY BIGGS, ESQ' you'll get an ?EXTRA IGNORED message; and PRINT A$ will show that the INPUT picked up only the words before the comma. Not exactly a vital consideration, but *any* restriction is annoying.

It's because the comma has a special usage with INPUT – you can ask for more than one input per INPUT. For instance:

```
10 INPUT "TYPE THREE NUMBERS";A,B,C
```

The 64 will expect you to put in three numbers – the first for variable A, the second and third for B and C (you'll get a double question-mark prompt for those). And it won't let you proceed until you've put in all three.

You can mix different types of variable with INPUT:

```
10 INPUT "WHAT'S YOUR NAME AND AGE";
A$,A
20 PRINT  "HELLO,"A$
30 IF A<18 THEN 50
40 PRINT "SO YOU STARTED VOTING" A-18
"YEARS AGO": GOTO 10
50 PRINT "SO YOU START VOTING IN" 18-A
"YEARS": GOTO 10
```

Beating the restrictions on INPUT

So you can't use commas in response to an INPUT. That restriction also applies to the colon, though I don't know why (INPUT doesn't have any special uses for colons). If you use one in typing an answer to the INPUT prompt, you'll again lose everything after the colon (including the colon itself).

And you can't get past an INPUT by inputting nothing – by hitting Return in an attempt to skip the command. That you might want to do for example if your INPUT is part of a series of form-filling commands:

```
10 INPUT "FIRST NAME"; N1$
20 INPUT "SECOND NAME"; N2$
10 INPUT "THIRD NAME"; N3$
```

If you don't *have* a second name, you'd want to skip that second INPUT. But the 64 won't let you. Try Return; if you don't have any other commands in your program you'll just jump right out of it and back to the READY prompt. If there is more to the program, INPUT will just come back with the question-mark prompt and sit there aggressively waiting for you to type *something* it's prepared to accept.

There's one easy answer to all these restrictions, but it's not exactly elegant: just about *anything* will be accepted if it's enclosed in double quotation marks, including colons, commas, and Return. The way INPUT works, it will strip off the quotes before using the input. Try this:

```
10 PRINT "EVERYTHING TYPED IN RESPONSE
TO INPUT SHOULD START WITH DOUBLE
QUOTES"
20 INPUT "TEST"; A$
30 PRINT "A$"
```

And RUN it, typing in some of those forbidden punctuation marks. You'll see when line 30 prints A$ that the quotes have disappeared.

Note that you don't really need closing quote marks – the system will ignore them in any case. And of course if you type quotes followed by Return you won't get the chance to put them in anyhow.

The only thing that *won't* be accepted is a quotation mark ... Since all the quotes *are* removed, INPUT won't like your attempt to put in any quote marks and you'll get a ?REDO FROM START message.

So the user could to be told to key in his or her response starting with double quotes. Alternatively you could do that automatically for him/her.

How? Well, INPUT works by reading the keyboard buffer (which is a small patch of memory devoted to reflecting anything typed in at the keyboard – it occupies locations 631 to 640 inclusive, in fact) on the basis of the input buffer counter (another memory location that tells Basic how many characters there are in the keyboard buffer). So if you can persuade INPUT that there's something already there in the buffer, and if the counter agrees with you, that will be taken as coming before anything you type in.

You can stuff something into the keyboard buffer by POKEing location 631; and you can force up the counter by POKEing 198. So let's put a double-quotes mark into the buffer (its ASCII character code is 34, so we can POKE 631,34) and tell the counter that there is already one character sitting there (POKE 198,1). Add this line to the INPUT operation:

```
5 POKE 631,34: POKE 198,1
```

Now when you RUN it, the INPUT's question-mark prompt will immediately be followed by the character that's already in the buffer – the quotation mark. And when you've finished typing in your *real* input, the 64 will accept the whole lot as the response.

Killing the query with INPUT

You'll *always* get a question mark when the 64 reaches any INPUT statement. This can be a bit irritating if you want to say something like 'PLEASE TYPE YOUR NAME', since that isn't a question.

The simple way around this is to frame all INPUT prompts as a question ('WHAT IS THE DATE'). That's a bit clumsy, of course. But the only way to lose the question mark is to get into some advanced programming: so for now why don't you just limit yourself to asking questions.

Keying information without Return: GET

The other big problem about INPUT is that you always need a Return to be pressed before things can proceed. This can be irritating at best, since it represents yet another demand the 64 makes on the user. And of course it's totally useless for anything required fast response, like control of something in a game.

The alternative is a potentially helpful command called GET. This works in much the same way as INPUT – you use it with a variable, as in ...

```
10 GET A$
```

... and it takes a character from the keyboard buffer to assign to the variable A$. But you won't get a prompt like the INPUT question mark, and you can't incorporate any prompting text: GET takes only one character at a time, too, and it does the assignation automatically without you needing to press Return.

So when you RUN a program containing a GET and you type something, that goes into the keyboard buffer and GET immediately reads it. It's working continuously, so you can just use it like this:

```
10 GET A$
20 PRINT A$
```

You wouldn't be able to type fast enough for GET when you RUN this; it would just pick up the first character in the buffer, which since you're just starting up would be a blank. So you have to have some way of telling GET to hang around waiting for a real keyboard input:

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

This does just that. Line 15 tells the program to cycle back to 10 if the character in the keyboard buffer is a blank; and GET will wait until it finds something that isn't a blank before the program's allowed to continue on to line 20.

GET won't pick up number variables, and it won't accept more than one character at a time. But you can use GET to get any character at all from the keyboard – including commas, colons and Returns (and numbers that are treated as characters). Add a line 30 to switch back and do it all again, then try any key on the keyboard. Some of them won't put anything on the screen – the space bar, Return, function keys – because they don't have any characters associated with them.

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

GET is very handy for pausing things, for instance to allow your user to read a screenfull of instructions and to proceed when he or she wants to:

```
10 PRINT "PRESS ANY KEY TO CONTINUE"
20 GET A$: IF A$="" THEN 20
30 ...
```

This will just hang around waiting for a key to be pressed. When that happens it'll just continue on to the next line. (Actually, 'any' key isn't quite right: GET won't pick up the keys that are normally used in combinations with another – Shift, Shift Lock, Restore, the Commodore logo key, CTRL.)

But the main use of GET is as a single-character-only version of INPUT. It's particularly useful when you want the user to choose an item from a 'menu' of options on the screen – for instance:

```
10 PRINT "PICK ONE (PRESS THE
APPROPRIATE NUMBER)"
20 PRINT
30 PRINT "1. ANDY WARHOL"
40 PRINT "2. PRINCE PHILIP"
50 PRINT "3. SUSAN SONTAG"
60 PRINT "4. COMPUTERWIT"
70 GET A$: IF A$="" THEN 70
80 IF A$=1 GOTO 100
85 IF A$=2 GOTO 200
90 IF A$=3 GOTO 300
95 IF A$=4 GOTO 400
100 PRINT "ONE'S COMPANY, TWO'S A
CROWD, THREE'S A PARTY"
110 GOTO 10
200 PRINT "I AM INTERESTED IN LEISURE"
210 PRINT"IN THE SAME WAY AS A POOR MAN
IS INTERESTED IN MONEY"
220 GOTO 10
300 PRINT "COGITO, ERGO BOOM"
310 GOTO 10
400 PRINT "TO ERR IS HUMAN, BUT TO REALLY
FOUL THINGS UP"
410 PRINT "YOU NEED A COMPUTER"
420 GOTO 10
```

The little GET statement buried in that is the key to the whole thing. RUN this, and you'll be able to select an aphorism appropriate to your mood by keying one of the numbers 1 to 4.

Using function keys with GET

This seems a good point to introduce programming for the function keys, since they're often useful in selecting menu options. Function keys don't produce any characters on the screen unless you've associated a particular text string with them. But the keyboard scan can still detect when a function key is pressed (otherwise they wouldn't have any effect at all) and that means they can be used for example in one of those 'PRESS ANY KEY' versions of GET.

You can also check to see which function key has been pressed, though. Replace lines 80 to 95 in the menu program with this lot:

```
80 IF A$="<f1>" GOTO 100
85 IF A$="<f3>" GOTO 200
90 IF A$="<f5>" GOTO 300
95 IF A$="<f7>" GOTO 400
```

If you press the relevant function key after the opening quotes when typing this in, the hieroglyphic associated with those keys should appear. Here's an alternative:

```
80 IF A$=CHR$(133) GOTO 100
85 IF A$=CHR$(134) GOTO 200
90 IF A$=CHR$(135) GOTO 300
95 IF A$=CHR$(136) GOTO 400
```

What's happening here is that we're checking directly for the ASCII codes produced by the function keys – these are the key-code values that would be put into location 631, and all eight function keys produce different values. The CHR$ function is a handy little item that tells you what character (CHaRacter, d'you see?) the accompanying number code refers to. So the line ...

```
80 IF A$=CHR$(133) GOTO 100
```

... compares the character in A$ to the character that's encoded as 133; and if they're the same, the program branches off to line 100.

More about that in a moment. But whichever route you use to pick up on the function keys, you'll be able to amend the start of the program slightly to tell the user which keys to press:

```
10 PRINT "PICK ONE (PRESS THE
APPROPRIATE FUNCTION KEY)"
20 PRINT
30 PRINT "F1. ANDY WARHOL"
40 PRINT "F3. PRINCE PHILIP"
50 PRINT "F5. SUSAN SONTAG"
60 PRINT "F7. COMPUTERWIT"
```

Pressing the appropriate function key now brings up the relevant saying.

You can check how any of the keys (including the function keys) generate different codes by trying this little program.

```
10 GET A$: IF A$="" THEN 10
20 PRINT ASC(A$)
30 GOTO 10
```

The ASC function is the converse of CHR$. It tells you what ASCII code a particular character has (CHR$ tells you what character relates to a particular ASCII code). RUN this program; and when you press any of the character-producing keys – including Space and Return among them, for the 64 treats these as 'characters' – you should see a number appearing on the screen. This is the numeric code that the 64 uses to identify individual keys: not the idiosyncratic codes you get in location 197, but those that appear in the keyboard buffer and the ones used by INPUT and GET.

Press the function keys one by one and you'll see the code sequence runs like this:

```
133          ...    function key 1
137          ...    function key 2
134          ...    function key 3
138          ...    function key 4
135          ...    function key 5
139          ...    function key 6
136          ...    function key 7
140          ...    function key 8
```

The sequence may look a bit odd, but it does correspond to the way the function keys are laid out:

```
UNSHIFTED:
133          ...    function key 1
134          ...    function key 3
135          ...    function key 5
136          ...    function key 7

WITH SHIFT:
137          ...    function key 2
138          ...    function key 4
139          ...    function key 6
140          ...    function key 8
```

Using GET when holding down a key

But what you can't do is hold down a particular key and expect GET to see it more than once.

Say you're programming a game that requires movement of a space fighter, or traversing and continuous fire of your laser cannon. For that you may need to be able to hold down a single key; and the way the keyboard works means that the 64 will indeed pick up a 'repeated' key if you keep it pressed – the 64 'scans' the keyboard several times a second to see if anything's happening there, so as far as it is concerned every time it comes across the key you're holding down it thinks of it as it another press.

GET won't see things that way. It picks up whatever's in location 631, and the keyboard scanner routine won't bother to transfer your repeated key depression to the buffer if that character is already there.

There's a solution in the shape of location 197. This is a halfway house; it's where the 64 puts the character code of whatever key is being pressed *before* passing it on to the buffer.

Sounds unnecessarily complicated? Sounds as though the keyboard scan could just dump the character code straight into 631? Well, there's an idiosyncracy in Commodore's handling of characters. The codes the 64 associates with keys on the keyboard aren't the same ASCII codes it uses to put characters on to the screen. In other words, the scanning routine decides what key has been pressed and puts the code for it into 197; and then the 64 converts that to ASCII before putting a *different* code for the key it into 631.

And the 64 won't bother transferring the code if there's one of them already there in 631.

On the other hand, it does this quite happily if you release the key and press it again – because in the fraction of the second that your finger is off the button, the scan of the keyboard detects that no key at all is being pressed. That 'no key' code is put into 197 and because it's not the same as the last thing in 631 the 64 transfers the 'no key' code to the buffer. The next time you hit the key a different code is detected for 197, it's converted, and it is put into 631.

So if you want to be able to hold down the key rather than beat a tattoo on it, the solution is to force the 64 to think the key you're holding down is *another* depression. This you can do by stuffing a 'no key' code into 197 yourself:

```
5 POKE 197,64
10 GET A$
15 IF A$="" THEN 10
20 PRINT A$
30 GOTO 5
```

RUN this and you'll be able to hold down any key. That '64' is the code for 'no key pressed'; line 30 is amended to go to the POKE 197,64 after each key detection.

Even cleverer menus with ON

There's a neater way of handling menus where the option-selecting input can be a series of numbers. It's a modest little command called ON, it's very simple to use, it's neat and economical, and for some reason it's largely ignored by programmers.

ON has applications whenever you have a number of different IF/THENs to switch the program in different directions depending on the value of a single input. ON is used with GOTO or GOSUB, and you'd have it like this in our menu program:

```
10 PRINT "PICK ONE (PRESS THE
APPROPRIATE NUMBER)"
20 PRINT
30 PRINT "1. ANDY WARHOL"
40 PRINT "2. PRINCE PHILIP"
50 PRINT "3. SUSAN SONTAG"
60 PRINT "4. COMPUTERWIT"
70 INPUT A
80 ON A GOTO 100, 200, 300, 400
```

The ON/GOTO replaces the original lines 80 to 95 and it causes the program to GOTO line 100 if A is '1', to 200 if it's '2', and so on. ON works only with numbers, though, so I've had to change the GET A$ (a character variable) to INPUT A (a numeric variable) since GET doesn't work with numbers.

The default INPUT

If you want a 'default' value for the input variable – something that's going to be selected and used unless the user decides otherwise – here's one way of doing it:

```
30 INPUT "<cur x 3>ANDY WARHOL<cul x
2><cul x 11>
```

In other words you use an INPUT prompt that opens with quotes, has three cursor rights, some text, two cursor lefts, and then as many cursor lefts as there are characters in the text. That text becomes your default value; on the screen it will appear *after* the query mark (because of the three cursor rights) with the cursor sitting on the first character (because those cursor lefts moved it back there).

You can now press Return to accept the default, or simply overtype it with an alternative – you will have to remove *all* the default characters, though, or else they'll be taken as part of the input.

CHAPTER 9
Getting Something Out

The PRINT command doesn't PRINT, at least not in the sense of persuading a printer to produce something on paper – there's a separate (though related) command for that.

Instead PRINT 'prints' something on the screen; a term like 'DISPLAY' might therefore have been more explicit.

But that possible confusion is just about the only thing wrong with PRINT. It's my favourite command; it's so *versatile* ...

Without the ability to display something, a computer would be more or less useless. That makes the PRINT command so important. Fortunate, then, that it's so easy to use – and so flexible.

It gets that flexibility simply from the variety and number of options you can use with it. The basics of PRINT, however, are as follows:

- **the command can be followed by one or more things to be printed** – *numbers* as constants, formulae or variables: or *strings* within quotes, string expressions (which may include the string functions like MID$ and friends), string variables, or CHR$ codes – all of which can include colour and cursor controls as well as text.

- **the command can be followed by the TAB or SPC functions** which can be of use in formatting the way things look when they're printed – both effectively insert spaces into PRINTed text.

- **the command can be followed by nothing at all**, in which event you just get a new line

- **the stuff to be printed can be separated** by *nothing* (in which case all the items to be printed will appear on the same line one after another): by *commas* (in which case the next item to be printed will appear at the start of the next 'print zone' – the 80 characters on a displayed line are divided into eight 'zones' of 10 characters, effectively pre-set tab points): or by *semi-colons* (which kill the normal new-line effect at the end of a PRINT, leaving the cursor on the same line at the position following the last item printed so that anything displayed subsequently goes on that line too)

- **the command is terminated by a Return** – and unless there's a semi-colon at the end of the PRINTed line this means the cursor will automatically move on to the start of a new line

- **the command can be abbreviated to a question mark.** PRINT is used so frequently that this single-key abbreviation is really handy; get into the habit of using it!

The best way to get on top of PRINT is to experiment; the examples following and those scattered through the book also illustrate the use of this command. But let's some run through some of the key features.

Printing numbers, text and blank lines

Numbers are printed with a preceding space if they're zero or positive, with a leading minus sign if they're negative. Getting rid of that space is possible, though a bit boring, by turning it into a STRing:

```
10 INPUT "NUMBER"; N 20 PRINT N
30 N$=STR$(N)
40 PRINT RIGHT$(N$, LEN(N$)-1)
```

Line 20 prints the numeric form, line 40 prints the string version with the space for the sign lopped off by LEN(N$)-1.

Numbers are also given a following space. Strings don't get either; if you want a space there you'll have to put it in yourself ...

```
100 A$ = "HELLO"
200 B$ = "HOW ARE YOU TODAY?"
120 INPUT "NAME"; C$
130 PRINT A$ C$ B$
```

```
RUN
NAME? DENNIS
HELLODENNISHOW ARE YOU TODAY?
```

Those short PRINTs also illustrate how character-handling functions can be used. It's worth observing that many types of expression can be mixed in a PRINT statement:

```
100 PRINT "NUMBER" N$ "IS" INT(N*1.2)
CHR$(13) "<blk><rvs>... AND THAT IS
MORE THAN" A% "<cud>"
```

The CHR$(13) there is the character code for Return, so the colour change and the text following would appear on a new line. PRINT used on its own has the same effect, and you can have as many forced new lines as you like by using it:

```
10 PRINT "HELLO"
20 PRINT: PRINT: PRINT: PRINT: PRINT
CHR$(30)
30 PRINT "CARE TO COME BOPPING TONITE?"
```

Line 10 prints 'HELLO' and terminates with a Return that forces one new line. Line 20 pushes the cursor down five more lines, the last PRINT being a colour change (to green) that also forces a new line.

Printing control characters

The cursor controls (cursor up, down, left, right, home) and the screen-clear facility can be incorporated into PRINT statements; so can the text colour controls.

There are two ways of doing this. The obvious one is to put them inside quotes; as soon as you open a pair of quotation marks after a PRINT you can use CTRL or CBM plus a number to switch colour or reverse the text displayed, and everything that the PRINT command prints after them will be in the newly selected look. Similarly you can drop in <clr> (just press Shift and CLR/HOME) or the cursor controls.

CHR$ codes can be used as a handy and interchangeable alternative to control characters inside quotes. You end up having to type a lot more characters, but the result is easier to distinguish than the hieroglyphics produced in quotes mode.

Besides them, there are eight special effects that can't be entered in quotes mode at all – for those you'll need the CHR$ code in any case. The table lists the correspondences; and in the examples following you'll also see that occasionally I've preferred to use CHR$ codes for other characters too, just for the sake of clarity.

One interesting use of this, incidentally, is to PRINT all control character codes – in other words, to include them in a PRINT statement so that they are actually displayed rather than having their usual effect. This you can do by using CHR$ to print

CHR$ codes and quotes-mode characters

Effect	Quotes mode keys	CHR$ code
Black	CTRL + 1	CHR$(144)
White	CTRL + 2	CHR$(5)
Red	CTRL + 3	CHR$(20)
Cyan	CTRL + 4	CHR$(159)
Purple	CTRL + 5	CHR$(156)
Green	CTRL + 6	CHR$(30)
Blue	CTRL + 7	CHR$(31)
Yellow	CTRL + 8	CHR$(158)
Orange	CBM + 1	CHR$(129)
Brown	CBM + 2	CHR$(149)
Light red	CBM + 3	CHR$(150)
Dark grey	CBM + 4	CHR$(151)
Medium grey	CBM + 5	CHR$(152)
Light green	CBM + 6	CHR$(153)
Light blue	CBM + 7	CHR$(154)
Light grey	CBM + 8	CHR$(155)
Reverse on	CTRL + 9	CHR$(18)
Reverse off	CTRL + 0	CHR$(146)
Cursor right	CRSR Right/Left	CHR$(21)
Cursor down	CRSR Up/Down	CHR$(17)
Cursor left	Shifted CRSR Right /Left	CHR$(157)
Cursor up	Shifted CRSR Up /Down	CHR$(145)
Home	CLR/HOME	CHR$(19)
Clear	Shifted CLR/HOME	CHR$(147)
Space	Space bar	CHR$(32)
Return (new line)	not possible	CHR$(13)
Shifted Return	not possible	CHR$(141)
DELete	not possible	CHR$(20)
INSert	not possible	CHR$(148)
Switch to lower case	not possible	CHR$(14)
Switch to upper case	not possible	CHR$(142)
Disable Shift /CBM	not possible	CHR$(8)
Enable Shift /CBM	not possible	CHR$(9)

quotes (CHR$(34) is the code) and then *opening* some quotes using the normal quotation-mark key – followed by the control characters. For instance:

```
10 PRINT CHR$(34)
"<clr><cup><cdn><rvs><blu><red>"
```

What happens is that the quote mode you initiated by keying the quotation mark remains in effect until the PRINTing starts; then the 64 comes across the first set of quotes – those from CHR$(34) – and treats the rest as a mere character string. You can even try deleting the quotes that are printed by immediately following CHR$(34) with the delete code CHR$(20).

TAB and SPC

These two do however have some value. TAB is used with PRINT, and it's followed by a number which indicates the position at which the next PRINTing will be done – starting from the first position on the next line, TAB(0).

You're not limited to the 79 positions of the line, in fact; you can have TABs up to 255, which prints in position 15 of the sixth line on:

```
PRINT TAB(255) "HELLO"
```

Or try this in Direct mode:

```
FOR C=1 TO 255: PRINT TAB(C) "HELLO";
NEXT
```

That can be useful in displaying something with gaps between the text that otherwise you'd have to force by a series of new-line commands.

But you're more likely to use TAB to space out text along the current line. This and the following examples PRINT a preliminary row of position numbers to indicate where the TABs fall:

```
100 PRINT
"<clr>01234567890123456789012345678
90123456789"
110 PRINT TAB(4) "COLUMN ONE" TAB(22)
"COLUMN TWO"
```

Note that you can string together several TABs followed by strings in the same PRINT statement. As the FOR/NEXT demo above suggested, you can also calculate a TAB position rather than give it explicitly:

```
10 INPUT A$, B$
20 PRINT
"<clr>01234567890123456789012345678
90123456789"
30 PRINT "<clr>" TAB(39/3-(LEN(A$)/
2)) A$ TAB(2*(39/3)-(LEN(B$)/2)) B$
```

In this example the two text INPUTs are centered in columns. The first TAB position is calculated by dividing the line length by three to provide two equally-spaced columns across the screen (yes, I know 39 divided by 3 is easy enough to work out in your head – it's just an *example*). Then the LENgth of the text is halved and the result subtracted from the column position; so when A$ is printed, it will be centered around that column position. For column two the centering is down the same way except that the starting point has to be position 26 – hence the 2*(39/3).

(When an expression like this is evaluated, TAB always takes the integer portion of the result ... obviously, since you can't move the cursor to a point *between* two print positions; but there is no rounding up, so if the expression evaluates to 9.99 you'll start printing at position 9 rather than 10.)

Similarly, TAB can be used to right-justify your printing:

```
100 PRINT
"<clr>01234567890123456789012345678
90123456789"
110 PRINT TAB(35-LEN(R$)+1)
```

That will print R$ with its final character in position 35. You need to add 1 because the PRINT will otherwise stop one character too soon.

The comma in PRINT statements overrides the effect of TAB. In this case the PRINTing of 'TWO' in line 30 doesn't start at position number 5 but at the beginning of the next print zone after the TAB position:

```
10 PRINT
"<clr>01234567890123456789012345678
90123456789"
20 PRINT "ONE" TAB(5)"TWO"
TAB(10)"THREE"
30 PRINT "ONE", TAB(5)"TWO"
TAB(10)"THREE"
```

This also illustrates another effect – TAB can't cause the cursor to backspace. So if you're requiring that two things be PRINTed in the same positions, the second will just be butted on to the first.

SPC is similar to TAB in that it's used to separate displayed items – except that it inserts a specified number of *spaces after the last item PRINTed*, while TAB starts printing at the *position* specified. You can see the difference by running this:

```
10 PRINT
"<clr>01234567890123456789012345678
90123456789"
20 PRINT "ONE" TAB(5)"TWO"
TAB(10)"THREE"
30 PRINT "ONE" SPC(5)"TWO"
SPC(10)"THREE"
```

Both SPC and TAB print the word 'ONE' at position 0. But TAB starts 'TWO' at position 5; while SPC begins at position 8, because it's putting five spaces after the end of the previous word.

SPC doesn't in fact print *spaces*, it just moves the cursor without overwriting anything underneath it.

SPC can be used like TAB to centre and justify text; like TAB it can be used with numbers, variables or expressions to a maximum value of 255; and like TAB it is affected by the use of a comma forcing a 'print zone' tab –

```
10 PRINT
"<clr>01234567890123456789012345678
90123456789"
20 PRINT "ONE" TAB(5)"TWO"
TAB(10)"THREE"
30 PRINT "ONE" SPC(5)"TWO"
SPC(10)"THREE"
40     REM ... and now with commas
50 PRINT "ONE", TAB(5)"TWO"
TAB(10)"THREE"
60 PRINT "ONE", SPC(5)"TWO"
SPC(10)"THREE"
```

Neat ways with PRINT formatting

Elsewhere in the book I've suggested some 'pretty print' tricks for presenting numbers – in descriptions of STR$ and the chapter on numbers, particularly. Here's another one, a kind of equivalent to the PRINT USING statement found in most versions of Basic other than Commodore's.

This is a subroutine that enables numbers to be formatted, and it's particularly helpful when displaying columns of figures in financial format (meaning with two decimal places for pence, cents, centimes and the like). It allows you to specify the currency symbol you'd like, prevents the 64 trying to display exponent-format numbers (by preventing you from putting in numbers that are so large they'd have to be displayed that way!), and handles the automatic appendage of zeros and commas.

```
10     REM example program to use the
routine
20 INPUT "AMOUNT"; NUM
30 INPUT "CURRENCY SIGN"; SYM$
40 GOSUB 1000

1000    REM 'PRINT USING' routine
1010 NUM = INT(NUM * 100 +0.5)/100
1020 IF INT(NUM) > 999999999 THEN PRINT
"NUMBER TOO BIG!": RETURN
1030 NUM$ = STR$(NUM): NL = LEN(NUM$)
1040 IF NL-1 < 2 THEN 1070
1050 IF MID$(NUM$, NL-1, 1) = "." THEN
NL=NL-2: NUM$=NUM$+"0": GOTO 1080
1060 IF MID$(NUM$, NL-2, 1) = "." THEN
NL=NL-3: GOTO 1080
1070 NUM$=NUM$+".00"
1080 COMMA=1: NF$= RIGHT$(NUM$, 3)
1090 IF NL < 2 THEN NUM$=SYM$+NUM$; GOTO
2050
2000 IF COMMA > 3 THEN NUM$="," + NUM$:
COMMA=1
2010 NF$=MID$(NUM$, NL, 1) + NUM$:
COMMA=COMMA+1: NL=NL-1
2020 GOTO 1090
2030 IF LEN(NUM$) > 12 THEN 2050
2040 NUM$=""+NUM$: GOTO 2020
2050 IF NUM < 0 THEN NUM$="-" + NUM$
2060 RETURN
```

Specifying print position

How about neat ways of displaying text, though?

Your screen display is likely to get pretty cluttered if a program has a number of possible messages to PRINT. The obvious solution is to use **PRINT '<clr>'** whenever you want to put something else on the screen, but that's a bit drastic – you might not want the *whole* screen erased.

An alternative is to wipe only the message from the screen. This you can do by pre-defining a string of spaces:

```
10 SP$=" ": FOR S=1TO39: SP$=SP$ + " ":
NEXT
```

This should be part of the initialisation procedure. It creates a variable SP$ containing 40 spaces (the FOR/NEXT loop adds 39 spaces to the one already in SP$). An alternative set-up would use CHR$(32), the code for the space character:

```
10 SP$=CHR$(32): FOR S=1TO39: SP$=SP$
+ CHR$(32): NEXT
```

Thereafter you can use SP$ after printing a message. You need to include a couple of 'cursor up' commands to get on to the line that holds the text you want to wipe:

```
10 SP$=CHR$(32): FOR S=1TO39: SP$=SP$
+ CHR$(32): NEXT

100 PRINT "PRESS ANY KEY TO PROCEED"
110 GET K$: IF K$="" THEN 110
120 PRINT "<cup>" SP$
```

The problem then is that you might not always know exactly where the message will be printed – so you might not know how many cursor ups you need. That you can solve by predefining a routine which specifies a 'message line' for any printing:

```
20 COL$=CHR$(29): FOR C=1TO39:
COL$=COL$ + CHR$(29): NEXT
30 ROW$=CHR$(17): FOR C=1TO24:
ROW$=ROW$ + CHR$(17): NEXT
```

Now we have two more variables, COL$ holding a full line's worth of cursor right characters (CHR$(29) is the code) and ROW$ with 25 of the CHR$(29) cursor-down codes.

Whenever you want to PRINT a message you can GOSUB to a small routine that calculates a print position:

```
8000 END
9000    REM print-positioning
subroutine
9010 PRINT PRINT CHR$(147) LEFT$(COL$,
C+1) LEFT$(ROW$, R+1) M$
9020 RETURN
```

That END is in there to prevent the program running on uninvited into the subroutine lines. The CHR$(147) in this subroutine returns the cursor to the 'home' position momentarily – that's the top left corner, and this prevents the screen scrolling while ensuring a common start point on the 'page' displayed for all messages. The rest of the line works out the start positioning for the message text in M$.

Your GOSUB must be preceded by setting up two variables for row and column position, and one more holding the text of your message – for instance:

```
100 INPUT "ROW"; R
110 INPUT "COLUMN"; C
120 INPUT "MESSAGE TEXT"; M$
130 GOSUB 9000
140 GET K$: IF K$="" THEN 140
```

And you can use the same subroutine to overwrite the message:

```
200 PRINT CHR$(147): INPUT "NEXT
MESSAGE"; M$
210 GOSUB 9000
```

You need that 'home' code in there – CHR$(147), though you could as easily have pressed the CLR/HOME key while in quotes mode – because the INPUT line when it appears on the screen adds an extra line to the cursor position. Most uses of the row/column positioner wouldn't ask for input anyway, since you'd be setting R, C and M$ directly; as long as nothing extra is put on to the screen, the cursor's position won't be affected and so the 'home' in the subroutine will be enough to take it accurately to the 'print message' point.

The routine can also be used to erase a message at that point:

```
300 M$=SP$
310 GOSUB 9000
```

This prints spaces across the whole line. You could be a bit fancier and erase *only* the message:

```
300 M$=LEFT$(SP$,LEN(M$))
310 GOSUB 9000
```

Just to tidy things up a bit, it's worth forcing a garbage collection to clear away unused variables (you do that simply by referencing FRE). This REM-filled summary also has a screen-clear command added:

```
5     REM initialisation
10 SP$=CHR$(32):FOR S=1TO39:SP$=SP$
+CHR$(32):NEXT
20 COL$=CHR$(29):FOR C=1TO39:
COL$=COL$+CHR$(29):NEXT
30 ROW$=CHR$(17):FOR C=1TO24:
ROW$=ROW$+CHR$(17):NEXT
40 X=FRE(0):PRINT"<clr>"

50     REM main program
100 INPUT"ROW";R
110 INPUT"COLUMN";C
120 INPUT"MESSAGE TEXT";M$
130 GOSUB 9000
140 GET K$:IF K$=""THEN 140

150    REM second message
200 PRINT CHR$(147):INPUT"NEXT
MESSAGE";M$
210 GOSUB 9000
220 GET K$:IF K$=""THEN 220

250    REM erase
300 M$=LEFT$(SP$,LEN(M$))
310 GOSUB 9000

8000 END

9000   REM print-positioning
subroutine
9010 PRINT PRINT CHR$(147) LEFT$(COL$,
C+1) LEFT$(ROW$,R+1) M$
9020 RETURN
```

All of this incidentally gives the effect of the single command PRINT AT provided in many non-Commodore versions of Basic.

POSition of the cursor

POS is one of the more pointless provisions in Basic. It seems particularly appropriate that it is used with a 'dummy' variable: POS(X) tells you the current position of the cursor in the current line – and remember that a 'line' can be up to 80 characters line, even though to get that many you'd see *two* lines on the screen.

I have however found one use for POS. If you are PRINTing a long line of text, a word that starts towards the end of the first *displayed* line will 'wrap around' on to the next. With some of its characters are on one line and some on the other, the text is ugly and difficult to read. POS can tell you if that is about to happen.

It's a bit awkward, though. Use POS(X) to note the current cursor position and LEN to check the length of the next string: if POS(X) plus LEN exceeds 39 (the opening POSition is number 0, so the last on the first displayed line is 39) you can PRINT CHR$(13) to force a carriage return. The next word to be printed then starts on the next line.

Here's one pretty crude way of doing that:

```
10 DIM TEXT$(5)
20 FOR T=1 TO 5
30 INPUT W$
40 TEXT$(T)=W$
50 NEXT

100 FOR C=1 TO 5
110 SP$="<space>"
120 IF LEN(TEXT$(C))+POS(X)>39 THEN
SP$=CHR$(13)
130 PRINT SP$+TEXT$(C);
150 NEXT
```

This sets up an array containing five text items and prints them one by one on the same line (hence the semi-colon in line 130). Each printed item is preceded by whatever's in SP$ – which ordinarily is a space, unless LEN plus POS tops 39; then it's a Return.

An alternative way of using this is to check the amount of space remaining *before* the PRINT:

```
10 DIM TEXT$(5)
20 FOR T=1 TO 5
30 INPUT W$
40 TEXT$(T)=W$
50 NEXT

100 FOR C=1 TO 5
110 IF LEN(TEXT$(C))<39-POS(X) THEN
P$(C)=P$(C)+CHR$(13)
120 PRINT TEXT$(C)+"<space>";
130 NEXT
```

This time the amount of line positions remaining is checked in line 110 by comparing the length of the text item against displayed line length less the current position of the cursor. If the word won't fit a

carriage return character is tacked on to it before 120 does the PRINTing. The space between words in this example is part of the PRINT line.

And apart from that I don't know of any sane use for POS. Any ideas?

CHAPTER 10
READ, DATA, RESTORE and Arrays

If you have a whole collection of variables to assign, you could do it one by one – with a series of LET or INPUT statements, for instance. But there's an easier way

READing DATA

The easier way is to use the READ statement, typically with a FOR/NEXT loop. It works almost exactly like INPUT – except that instead of using information you key in at the keyboard the 64 looks for a line of DATA somewhere else in the program and takes its input from that.

In other words, you'd have something like this:

```
10 READ A$
100 DATA "A TEST SAMPLE"
200 PRINT A$
```

Or, since it would be easy enough to use a LET or INPUT for a single variable assignment like that, you'd be more likely to READ a series of DATA items:

```
10 READ A$, B$, C$, D$
20 PRINT D$, A$, B$, C$,
100 DATA "ONE", "TWO, "THREE", "TESTS:"
```

Here the program READs something into each of the four text variables and then prints them. Note that the READ works more or less like a subroutine – it rushes off to do its READing until all the variables specified with it have something in them, and only then will it continue with the rest of the program.

READ is quite clever. It'll go and look for DATA anywhere in the program and start READing from the first bunch it finds. This means that DATA can actually be positioned *before* the READ, or a long way away from it at the end, or buried somewhere in the middle of the program. And the first READ will take the first DATA:

```
10 DATA "TESTS:", 1.5, 2.5
100 READ P$, T1, DABBLE%
120 DATA "MORE TESTS:", 0.75, 7.5
200 PRINT P$, T1, DABBLE%
```

Here you'll find that the READ takes the DATA from line 10 and ignore line 120 – even though you might have expected it to be the next line to be executed by the 64.

You can have any number of items in a single DATA statement, up to the maximum line length of 80 characters total; the READ will just take as many as it needs to fill up its variables. Individual items on a DATA line must be separated by commas but you can omit any spaces that aren't part of a character string.

On the other hand, you'll get an ?OUT OF DATA ERROR message if there are more variables to fill than there are DATA items. Somewhat unhelpfully, the line reference given with the message will be the READ statement's line number and not the DATA.

Be careful when typing in DATA items – a missing comma, a misspelling, or an inadvertent repetition can foul things up. Because of that it makes sense not to cram too much on to a single DATA line. The READ doesn't care how many DATA lines it has to get through, and it'll just carry on reading as many items as it needs irrespective of how many lines they're on.

You can READ any kind of variable:

```
10 READ P$, T1, DABBLE%
100 DATA "TESTS:", 1.5, 2.5
200 PRINT P$, T1, DABBLE%
```

But obviously the DATA items must match the variable type you're READing, otherwise you'll get a ?SYNTAX ERROR. In fact most of the usual rules about variables apply – though character string DATA items needn't be enclosed in quotes unless they include the kind of thing that wouldn't normally be accepted as part of a string: if you have colons, commas, spaces, graphic characters or cursor controls in the string you should put quotes around the DATA item.

That's why I had quotes around 'TESTS:' (the program would otherwise have assumed that colon meant I was starting a new statement on the same line, and clearly I wasn't). In fact, to be on the safe side I generally put any text at all in quotes; apart from anything else this helps you see that text matches a string variable when your READ is reading a mixed collection of variables.

One wrinkle with string DATA: you can use quotes within them as part of the string provided that the quotes mark isn't the first thing in the DATA item.

```
10 READ P$, Q$
100 DATA TEST FOR "QUOTES", "I SAY!" HE
SAID
200 PRINT P$, Q$
```

Here you'll crash with a ?SYNTAX ERROR before anything gets PRINTed because the first DATA will be accepted as A$ but the second won't.

Another restriction: the items in your list of DATA can't be expressions, and they can't themselves be or use variables. Meaning you can't have things like these as DATA:

```
3.5*(A-1)
INT(ABS(-2.44))
LEFT$(TI$,2)+"HOURS"
A$
```

The first three would crash with a ?SYNTAX ERROR; the last one would be assumed to be a string reading 'A$' even if your program had previously defined A$ as being a string variable containing some text.

On the other hand, you can have an 'empty' DATA item just by putting two commas together. The second comma will be assumed to indicate a zero if you're assigning numeric variables (which makes it a handy and space-saving shorthand) or an empty space if it's a character variable. This is useful if you want to re-use a list of DATA items with one or two entries deleted but without changing the READ statement.

A Basic program can have any number of READ statements. When the first one has been satiated with something for each of its variables, the next READ will start READing at the *next* DATA item, whether or not it's on a new line:

```
10 DATA "TESTS:", 1.5, 2.5
100 READ P$, T1
120 READ DABBLE%
200 PRINT P$, T1, DABBLE%
```

The second READ here will pick up the next available DATA item, which happens to be the third one in the list.

Re-reading with RESTORE

DATA item can normally be read only once. In other words, READ runs through a list of DATA items only once; and two or more READs in a program will start READing from the first unread DATA item ...

```
10 DATA ONE, TWO, THREE, FOUR, FIVE
20 READ A$, B$
30 READ C$, D$, E$
100 PRINT A$, B$, C$, D$, E$

RUN

ONE   TWO   THREE  FOUR   FIVE
```

And while you can re-use variables, you can't normally re-use DATA:

```
10 DATA ONE, TWO, THREE, FOUR, FIVE
20 READ A$, B$
30 PRINT A$, B$
100 READ A$, B$, C$
110 PRINT A$, B$, C$

RUN

ONE   TWO
THREE  FOUR   FIVE
```

Unless, that is, you use the RESTORE statement. As soon as the 64 comes across a RESTORE in a program, it treats the next READ as if it were the *first* READ encountered and begins taking DATA from the start of the list again.

```
10 DATA ONE, TWO, THREE, FOUR, FIVE
20 READ A$, B$
30 PRINT A$, B$
40 RESTORE
100 READ A$, B$, C$
110 PRINT A$, B$, C$

RUN

ONE   TWO
ONE   TWO   THREE
```

One use for RESTORE is to look up individual items in a DATA list with a series of READs. Here's a simple directory:

```
10 INPUT "COLOUR"; C$
20 IF C$="Z" THEN END
30 READ A$, B$
40 IF A$<>C$ THEN GOTO 30
50 IF A$="ZZ" THEN PRINT "NOT IN
DIRECTORY": GOTO 10
60 PRINT "KEY " B$
70 IF B$ < "9" THEN PRINT "PLUS CTRL"
80 IF B$ > "9" THEN PRINT "PLUS CBM"
90 RESTORE
100 GOTO 10
110 DATA BLACK, 1, WHITE, 2, RED, 3,
CYAN, 4
120 DATA PURPLE, 5, GREEN, 6, BLUE, 7,
YELLOW, 8
130 DATA ORANGE, 1, BROWN, 2, LIGHT RED,
3, DARK GREY, 4
140 DATA MEDIUM GREY, 5, LIGHT GREEN, 6,
LIGHT BLUE, 7, LIGHT GREY, 8
150 DATA ZZ, 00

RUN

COLOUR? CYAN
KEY 4
PLUS CTRL
```

This is a ready-reference to the colour key combinations on the 64. You INPUT a colour (or 'Z', which quits the program); the opening two DATA items are read, and if they don't correspond to your colour line 40 loops back to read the next pair.

If the colour you entered isn't in the list at all the READs will hit line 150, an end-of-DATA warning, and the message in line 50 will be printed.

And if it is there, the second of the two variables read (the number) will be printed (line 60). The program also includes a test to see whether it's less or greater than 9 and tells you which of the two Shift keys to use as well.

It then hits the RESTORE, sets the READ/DATA pointer back to the start of the DATA list, and cycles around to ask for another colour.

Arrays

The READ/DATA combo is at its most useful when you have a lot of information to put into variables for future use. And the best way to do that is to stuff the DATA into an array.

What's an array? Well, basically it's a list of different variables with a special type of name.

Before you can use an array you have to tell the program how big it's going to be – and to specify those dimensions you use the DIM statement:

DIM *array name* (*number*)

The *number* is the number of spaces for variables that you want in that array; but '0' is counted as the first of them, so in practice you get one more than the number. The *array name* can be any of the standard variable combinations – a single letter, a pair of letters, or a letter with a number from 0 to 9. (As with other variables you can use a longer collection of letters like a whole word, but in practice the 64 will only recognise the first two letters.) And you can specify the array as being numeric (no suffix), string (with the dollar suffix) or integer (put the percent mark on the end). So these program lines ...

```
10 DIM A$(5)
20 DIM AB(10)
30 DIM A%(4)
```

... set up a string array with space for five variables, a numeric array with ten, and an integer array with four.

Thereafter you can put values into the array's variables in the usual way – with LET, or INPUT, or READ, or whatever. And you can reference individual variables within the array simply by indicating the array name and the variable's number: so A$(4) would be the fourth item in the string array called A$, and A%(0) is the first in the integer array A%.

So you can think of an array as a list of separate items (they're called 'elements') which are variables of the same type. And once you've defined them, you can use the elements like any other variable:

```
10 DIM A$(6)
20 A$(0)="BEAUTIFUL"
30 A$(1)="TODAY IS "
40 A$(2)="WEATHER"
50 A$(3)="AND IT'S"
60 A$(4)="MONDAY"
100 PRINT A$(1) A$(4) A$(3) A$(0) A$(2)
```

A bit trivial, that. This is a tadge more useful:

```
10 DIM A$(4)
20 INPUT"DAY OF WEEK"; D$
30 A$(0)"TODAY IS"
40 A$(1)=D$
50 A$(2)"AND IT'S"
60 INPUT"WHAT'S THE WEATHER LIKE"; W$
70 A$(3)=W$
80 A$(4)"WEATHER."
90 PRINT A$(0) A$(1) A$(2) A$(3) A$(4)
```

And rather than all those LET statements, you can quite neatly use a READ and some DATA to fill up the array:

```
10 DIM A$(4)
20 FOR C=0 TO 4: READ A$(C): NEXT
30 INPUT "DAY OF WEEK"; D$
40 INPUT "WHAT'S THE WEATHER LIKE"; W$
50 A$(1)=D$: A$(3)=W$
60 PRINT A$(0) A$(1) A$(2) A$(4) A$(5)
100 DATA "TODAY IS ",,"AND IT'S
",,"WEATHER."
```

The way to stuff something into an array with a READ, then, is use the format of line 20 – set up a FOR/NEXT loop the same size as the array you've DIMensioned, let the FOR counter decide which array element you READ some DATA into, and then do the NEXT one. (Note that here I'm skipping some of the array elements because I'm using a couple of INPUTs to fill them.)

Getting at DATA in a different order

READ normally works by taking the first DATA item, then the second, and so on until all the variables have been satisfied. But in practice you might want to start READing DATA from some other point than the first item – you may have a bunch of subroutines containing READs that take only part of the DATA list.

With a little care you can use RESTORE and a pair of READs rather neatly to start taking DATA items from some other point than the start of DATA. Take a look at this:

```
10 DIM A$(4)
20 INPUT "PEOPLE, STREETS, OR TOWNS";
I$
30 IF I$="S" GOSUB 200
40 IF I$="T" GOSUB 300
50 FOR A=1 TO 4: PRINT A$(A)
60 RESTORE: GOTO 20
100    REM read PEOPLE
110 READ B$: IF B$<>"PEOPLE" THEN 110
120 FOR N=1 TO 4: READ A$(N): NEXT
130 RETURN
200    REM read STREETS
210 READ B$: IF B$<>"STREETS" THEN 210
220 FOR N=1 TO 4: READ A$(N): NEXT
230 RETURN
300    REM read TOWNS
310 READ B$: IF B$<>"TOWNS" THEN 310
320 FOR N=1 TO 4: READ A$(N): NEXT
330 RETURN
```

```
1000    REM data
1010 "PEOPLE","NICKY","ANNABEL",
"ROLAND","HYWEL"
1010 "STREETS","WORPLESDEN ROAD",
"ACACIA AVENUE","WEMBLEY WAY","NEW
ROAD"
1030 "TOWNS","LONDON","LONDON",
"BIRMINGHAM","EPSOM"
```

Say the INPUT selection is 'T' for 'TOWNS'; line 310 here READs each DATA statement in turn but won't proceed unless it finds 'TOWNS'. Line 320 then starts READing DATA into the array *starting at the items following 'TOWNS'*.

When it's done, the subroutine RETURNs to print out what you've got in the array. Then the program RESTOREs everything and cycles around for another selection.

Fancier arrays

That's what they call a 'one-dimensional' array, and you can think of it as a list of items in a single column – RUNning the last of those programs would give you a list like this:

array element	*contains*
A$(0)	TODAY IS
A$(1)	answer to "DAY OF WEEK"
A$(2)	AND IT'S
A$(3)	answer to "WHAT'S THE WEATHER LIKE"
A$(4)	WEATHER.

You can however have rather fancier *two*-dimensional arrays, which look more like a rectangular box of cells. You still have one array name, but this time the DIM statement has to specify what you might call the array's 'width' as well as its 'height':

DIM *array name* (*number of rows, number of columns*)

Again, those numbers start at 0; so this DIM statement will set aside space for one more rows and one more columns than you've indicated. Thereafter you can refer to a particular item in the array in much the same way as before, except that you have to give its row and column position. Say you have this DIM statement to give you an array of three columns by four rows:

```
10 DIM P$(3,4)
```

You can think of it looking rather like this:

	col 0	col 1	col 2	col 3	col 4
row 0					
row 1					
row 2					
row 3					

... with the variables in it identified by our DIM as follows:

	col 0	col 1	col 2	col 3	col 4
row 0	P$(0,0)	P$(0,1)	P$(0,2)	P$(0,3)	P$(0,4)
row 1	P$(1,0)	P$(1,1)	P$(1,2)	P$(1,3)	P$(1,4)
row 2	P$(2,0)	P$(2,1)	P$(2,2)	P$(2,3)	P$(2,4)
row 3	P$(3,0)	P$(3,1)	P$(3,2)	P$(3,3)	P$(3,4)

And what would you use a two-dimensional array for? Well, anything that can usefully be held like this. For instance, here's a pretty basic one-month appointments diary:

```
10      REM set up basic calendar
20 DIM DIARY$(5,8)
30 FOR DAY=1 TO 7: READ DIARY$(0,DAY):
NEXT
40 FOR WEEK=1 TO 4: READ DIARY$(WEEK,
0): NEXT
50 DATA MONDAY,TUESDAY,WEDNESDAY,
THURSDAY,FRIDAY,SATURDAY,SUNDAY
60 DATA WEEK ONE,WEEK TWO,WEEK THREE,
WEEK FOUR
70 :
100     REM do what?
110 INPUT "ENTER OR CHECK APPOINTMENTS,
OR QUIT"; A$
120 IF A$="Q" END
130 IF A$="E" THEN GOSUB 200
140 IF A$="C" THEN GOSUB 300
150 GOTO 110
160 :
200     REM enter appointments
210 INPUT "DAY NO."; D
220 INPUT "WEEK NO."; W
230 INPUT "APPOINTMENT WITH"; P$
240 DIARY$(D,W)=P$
250 RETURN
260 :
300     REM check appointments
310 INPUT "DAY NO."; D
320 INPUT "WEEK NO."; W
330 PRINT DIARY$(W,0), DIARY$(0,D),
DIARY$(D,W)
340 RETURN
```

Line 20 sets up the array as a grid of five rows by eight columns. Why 5x8? Because I find it hard to cope with '0' being the first number in a sequence; I need four rows for the weeks in a month and seven columns for the days, and I want to be able to refer to the first week as 'week one' – and so on.

That means I'm not going to bother using column number 0. So when line 30 fills in the first row of the array I'm calling DIARY$, I will READ seven DATA items into the array starting at the *second* of the cells in row 0 of DIARY$. This is referenced as DIARY$(0, DAY); so the first time round that translates as DIARY$(0,1), which means the second column (column 1) of the first row (row 0). You see why I prefer to ignore '0' as the first number in a sequence?

Estimating array sizes

type of array	*example*	*bytes required*
one-dimensional numeric	DIM A(R)	7 + 5 × (R + 1)
one dimensional integer	DIM A%(R)	7 + 2 × (R + 1)
one-dimensional string	DIM A$(R)	+ 3 × (R + 1) + number of characters being stored
two-dimensional numeric	DIM A(R,C)	9 + 5 × (R + 1) × (C + 1)
two-dimensional integer	DIM A%(R,C)	9 + 2 × (R + 1) × (C + 1)
two-dimensional string	DIM A$(R,C)	9 + 3 × (R + 1) + number of characters being stored
three-dimensional numeric	DIM A(R,C,L)	11 + 5 × (R + 1) × (C + 1) × (L + 1)
three-dimensional integer	DIM A%(R,C,L)	11 + 2 × (R + 1) × (C + 1) × (L + 1)
three-dimensional string	DIM A$(R,C,L)	11 + 3 × (R + 1) × (L + 1) × number of characters being stored

On the other hand, I'm only skipping column 0 in row 0. It suits me to use columns 1 to 7 of row 0 because that's where I'm putting my titles. Line 30 gives me a row that looks something like this:

	col 0	col 1	col 2	col 3	col 4	col 5	col 6	col 7
row 0	(empty)	MON	TUE	WED	THU	FRI	SAT	SUN

Similarly, line 40 also omits the very first position of the grid and puts my week number title in the rest of column 0:

	col 0
row 0	(empty)
row 1	WEEK 1
row 2	WEEK 2
row 3	WEEK 3
row 4	WEEK 4

And so the READ/DATA combination is giving me a grid that can be thought of as looking like this:

	col 0	col 1	col 2	col 3	col 4	col 5	col 6	col 7
row 0	(empty)	MON	TUE	WED	THU	FRI	SAT	SUN
row 1	WEEK 1							
row 2	WEEK 2							
row 3	WEEK 3							
row 4	WEEK 4							

The rest of the positions in the array will now be filled in when the appointments are actually entered. For that I have the INPUT options in line 110, with the program skipping off to different subroutines depending on whether the user want to enter new appointments (or overwrite existing ones) or check what appointments have been made. If you want to leave the program you can type 'Q' for 'Quit'.

The subroutines are similar. You tell the program which day number of which week you're referring to; and for new appointments you're asked who you'll be seeing. That name is assigned to the string P$, and line 240 dumps P$ into the variable specified as DIARY$(D,W) – which means day number D in week number W.

RETURN takes you back to line 150, which in turns cycles round to ask for the 'enter/check/quit' input again.

For the 'check appointments' subroutine I'm simply printing the contents of the array element specified by DIARY$(D,W). As a confirmation this is preceded by the week number and the day, which you can get at in the same way – week numbers are in column 0, so DIARY$(W,0) finds the right text for week number W; days are in row 0, and DIARY$(0,D) therefore does the same for the days.

This program could do with some tidying up, and it's a bit long-winded. It also suffers from the irritating habit of dying every time you press 'Q', since the contents of the array are not preserved when you stop and re-RUN the program – not very handy for an appointments diary, since you'd have to leave it permanently on if you wanted to refer to it.

Here's another two-dimensional array that may be of more use, since it acts as a kind of calculator:

```
10    REM set up worksheet
20 DIM CALC(6,9)
30 FOR COL=1 TO 7: READ CALC(0, COL):
NEXT
40 FOR ROW=1 TO 4: READ CALC(ROW, 0):
NEXT
50 DATA 1, 2, 3, 4, 5, 6, 7
60 DATA 1, 2, 3, 4
70 :
100   REM do what?
110 INPUT "ENTER, CHECK OR QUIT"; A$
120 IF A$="Q" END
130 IF A$="E" THEN GOSUB 200
140 IF A$="C" THEN GOSUB 300
150 GOTO 110
160 :
200   REM enter amount
210 INPUT "ROW NO."; R
220 INPUT "COL NO."; C
230 INPUT "AMOUNT"; A
240 CALC(R,C)=A
250 RETURN
260 :
300   REM check amount
310 INPUT "TOTALS (Y/N)"; T$
320 IF T$="N" THEN 600
330 FOR ROW=1 TO 4: FOR COL=1 TO 7
340 CALC(ROW,8)=0: CALC(5,COL)=0:
NEXT: NEXT
350 FOR ROW=1 TO 4: FOR COL=1 TO 7
360
CALC(ROW,8)=CALC(ROW,8)+CALC(ROW,COL
370
```

```
CALC(5,COL)=CALC(5,COL)+CALC(ROW,COL
380 NEXT COL
390 NEXT ROW
400 :
500     REM print totals
510 INPUT "DAILY OR WEEKLY"; R$
520 IF R$="W" THEN 540
530 FOR COL=1TO7: PRINT CALC(0,COL)
"=" CALC(5,COL): NEXT: GOTO 999
540 FOR ROW=1TO4: PRINT CALC(ROW,0)
"=" CALC(ROW,8): NEXT: GOTO 999
550 :
600     REM check individual entry
610 INPUT "ROW NO."; R
620 INPUT "COL NO."; C
630 PRINT "COL. NO." CALC(0,C),"ROW
NO." CALC(R,0)
640 PRINT "CONTAINS " CALC(R,C)
650 :
999     REM return
1000 RETURN
```

This one works in much the same way. It's a simple expenses calculator for one month, telling you how much you spent each week during the month and (somewhat pointlessly) the total spent during the month for each day. Well, it could be more useful if the seven headings along the top of the columns were types of expense ...

Line 20 DIMs an array of six rows by nine columns – six rows because row 0 will hold a heading of sorts for the days, rows 1 to 4 are the four weeks, and row 5 will hold totals: and similarly, nine rows because there's one for each of the seven days plus row 0 for the week headings and row 8 for totals. Line 30 reads a simple column heading (numbers 1 to 7) into row 0, line 40 reads row titles 1 to 4 into column 0.

At line 110 you get the chance to enter new information, check what's there, or exit. The 'enter amount' subroutine at 200 onwards works like the 'enter text' routine in the previous example; the 'check' routine also lets you look at a particular entry – lines 600 to 640. I've added a totalling facility that lets you see totals by row or by column; if you don't skip the INPUT at line 310 the program automatically computes both lots of totals for you. It does columns first, then rows (the FOR loops starting at 350); for rows it builds up a total in column 8 by adding to it the value in each column of each row (line 360) – then with line 370 it does the same for column totals, keeping them in row 5.

Again, this example is somewhat exaggerated and simplified for the purposes of showing how a two-dimensional array can be used – for example, in the Real World you'd have a few 'range checks' to prevent ?BAD SUBSCRIPT ERRORS appearing when you inadvertently try to specify too many columns or rows.

It also illustrates one of the real problems with arrays: you can't mix different types of variable in the one array. In this case, for instance, it would have been nice to have real textual headings for the rows and columns – but text headings would be character strings, and the rest of the mini-calculator obviously needs numeric data.

Last words on DIM

You don't *have* to DIMension arrays before you use them – if you start referring to array variables without having first encountered a DIM, the program assumes a default value for the array. For one dimensional arrays, the default is 10; for two-dimensional arrays it's assumed to be (10,10) if there's no DIM.

On the other hand you're strongly advised to use DIM as a matter of course. Why? Three reasons: for one, you may need more array elements than the default – and you may not realise that until you're well into your program.

That's not especially important, but this is: you may not need as many array elements as the defaults give you. And even though you don't use all the elements, space is still reserved for them. For a graphic demonstration, try RUNning this little test:

```
10 DIM A(7,7)
20 A(1,1) = 1
30 PRINT 65536 + FRE(0)
```

... then type '10' to delete line 10 and RUN it again. In both cases you're setting up one array element of a two-dimensional array and printing out how many bytes of memory you've taken up. The first one has an explicit DIM, the second will assume a default DIM of A(10,10). And although the second version is one line shorter it still takes 70 per cent more memory.

To conserve memory then it makes sense to set up arrays no larger than you need. And the third reason for using DIM as a matter of course is the most important of all: it's simply good practice, because it acts as a reminder of what the array is called and how big it is. That will (hopefully) cut down the number of mistakes you make, and it'll certainly simplify the job of anyone – like you – who subsequently has to go through the program to amend or debug it.

You can however save a little memory by defining more than one array per DIM. It doesn't make things particularly readable, but a single DIM can specify the size of more than one array, as in ...

```
100 DIM A$(5,7), B%(12), C(5,5,4)
```

Separate the array specifications with commas. This line DIMs a string array of 5 rows by 7 seven columns, a 12-element one-dimensional integer array, and a three-dimensional numeric array.

A three-dimensional array? Yes, you can have 3D arrays too. And four or more dimensions, if you think your mind can handle them. Mine can't: you'll have to look elsewhere for information.

Actually a 4D array is likely to require the 64 to reserve more memory than it has. If memory space is likely to become a problem, you can guesstimate the size of arrays from the table.

Arrays don't have to be set up in the conventional manner by giving their size explicitly – they can be DIMensioned by using expressions or variables. For instance, all of these are acceptable:

```
10 INPUT "ROWS"; R
20 INPUT "COLUMNS"; C
30 DIM ARRAY(R,C)

10 INPUT "TEXT"; T$
20 DIM ARRAY$(LEN(T$))

10 DIM ARRAY%(18*B-1,COS(3.2*A%+7))
```

The 52-card trick

A common use of arrays – and a good illustration of how they work – is to shuffle the deck for a card game. Here's how it might be done:

```
10      REM set up array elements
20 DIM PACK%(52): PACK%(0)=-1
30 FOR P=1 TO 52
40 PACK%(P)=INT(RND(1)*52)+1
50      REM make sure number isn't
already being used
60 C=0
70 FOR A=1 TO 52: IF PACK%(P)=PACK%(P-
A) THEN C=1
80 NEXT A
90 IF C=1 THEN 40
100     REM ok? then get next number
110 NEXT P
```

What's happening here is that you're creating an integer array of 52 variables, setting the first – PACK%(0) – to '-1' to make sure that it doesn't foul up the subsequent assignments. Then you start to fill it up with random integers (line 40), with line 70 checking to make sure that the number in question hasn't been used once already (if it has the program tries again until it finds a unique number and shoves that into PACK%(P) before proceeding).

This kind of shuffling has lots of applications in games and the like, but for card games you'd probably use it as a subroutine to shuffle cards in a suit with a separate randomiser to select suits.

Incorrect DATA?

Typing in a chunk of DATA statements can be a drag, and it's easy to make mistakes – a missing comma, a number omitted, a number typed twice. Here are some tips that may help:

- If you are READing and POKEing into memory, make your line numbers for DATA correspond to the memory locations. For instance, say you are creating eight custom graphic characters in locations 14336 onwards and you have a READ statement like this:

```
150 FOR I = 14336 TO 14336+(8*8): READ M
160 POKE I,M: NEXT I
```

You could have your DATA at the end of the program, eight items to the line, with line numbers like these:

```
14000 REM DATA follows
14336 0,40,16,146,214,254,254,0
14334 0,12,60,60,255,60,0,0
14352 ...
```

- If you persistently get ILLEGAL QUANTITY ERRORS because there's a mistake, it may help to insert a temporary PRINT between the READ and assignment statements:

```
150 FOR I = 14336 TO 14336+(8*8): READ M
151 PRINT I,M
160 POKE I,M: NEXT I
```

This will show the DATA as it's being POKEd, which DATA item is going to which iteration of the FOR/NEXT loop. When the program stops with the ILLEGAL QUANTITY you know the last item printed is causing the error.

- Another trick when you get an ILLEGAL QUANTITY ERROR: locations 63 and 64 contain the line number of the last DATA statement used. Now, you'll get your ILLEGAL QUANTITY ERROR in the line containing the READ even though the problem is almost sure to be in one of the DATA statement lines. But try typing this in direct mode (without a line number) when you get the error:

```
PRINT PEEK(63) + PEEK(64)*256
```

That should give you the offending line number – which you can then LIST and correct.

- If your DATA is on lines within the main body of a program, you might want to signal the end of the DATA statements. You might want to do that anyway. The easiest way is to make your last DATA a minus quantity if you can, and then include a test for a negative number after the READ:

```
150 FOR I = 14336 TO 14336+(8*8): READ
M: IF M<0 THEN 200
160 POKE I,M: NEXT I
200 REM program continues
```

For alphabetic data, an asterisk as the last DATA statement works as well:

```
140 DIM S$(10)
150 FOR S = 1 TO 10: READ S$(I): IF S$="*"
THEN 200
160 NEXT I
200 REM program continues
```

- For numeric items, one simple (if somewhat laborious) method of ensuring that you've typed in all the DATA you should is to use the 64 or a trusty pocket calculator to sum them all. Then you can have a test counter running:

```
10 COUNT=0
20 FOR R=1 TO 8: READ A
30 COUNT=COUNT + A
40 NEXT R
50 IF COUNT <> 255 THEN PRINT "ERROR IN
DATA LINES": GOTO 999
60      REM rest of program follows ...
999 END
1000 DATA 1, 2, 4, 8, 16, 32, 64, 128
```

CHAPTER 11
Handling Characters

Any program will have to work with words, if only to PRINT instructions or INPUT prompts on the screen. But the 64's Basic is really quite clever at handling characters.

Basics first: characters are termed 'strings' in the 64 lingo. In practice that means anything inside quote marks – letters, numerals, punctuation symbols, graphics characters, cursor and screen control hieroglyphics. Strings can be one character or many. And they are differentiated from numeric data, which comprises only numbers, doesn't appear within quotes, and can be used in mathematical operations.

That's not to say that you can do some arithmetic-style operations on strings. In fact strings can be measured, added, sliced up, compared, and converted to a number ... as we'll see.

String comparisons and additions

For a start a number of the operator symbols more usually associated with maths can be employed with strings: they are the plus sign and the so-called 'relational' operators.

You can't really *add* two strings, and you certainly can't expect to get 'THREE' if you have a statement including 'ONE' + 'TWO'. Instead you can use the plus sign to *concatenate*, which means ramming one string on to the end of another: 'FOOT' + 'BALL' produces 'FOOTBALL'. This direct-mode line illustrates the difference between addition and concatenation:

```
PRINT 111+222,"111"+"222"
 333          111222
```

The 'relational' operators are probably more useful. The simplest is the equals sign, and you will quickly get so used to it that you won't think twice about it:

```
10 INPUT "HAVING A GOOD DAY"; Y$
20 IF Y$="YES" THEN PRINT "IT WON'T
LAST"
30 IF Y$="NO" THEN PRINT "IT'LL
PROBABLY GET WORSE"
```

Lines 20 and 30 are examples of the most basic string comparison. The other familiar one is its direct opposite – '<>', the 'not equal to' sign. Line 30 could have been written as:

```
30 IF Y$<>"YES" THEN PRINT "IT'LL
PROBABLY GET WORSE"
```

The equal/not equal signs have obvious applications in checking question-and-answer inputs, and will also be used extensively if you're doing something like searching through a list. For instance, if you have a telephone-number list and you're adding a new name you will want to check first that there isn't an entry for that person: you could INPUT the new name as the string variable N$, then read through your list looking for a match to N$. If the program doesn't find one it could ask the user for the phone number.

Greater and less than a string

You can also use the two other relational operators with strings, the 'greater than' symbol > and the 'less than' mark <. The most obvious application here is in sorting lists of text, because a string starting with 'A' is considered 'less than' one beginning with 'B'.

Try this simple example, which prints any two character strings in alphabetical order:

```
10 INPUT "TYPE TWO STRINGS"; S1$, S2$
20 IF S1$>S2$ THEN PRINT S1$, S2$
30 IF S1$<S2$ THEN PRINT S2$, S1$
40 GOTO 10
```

Keep trying different combinations and you'll start spotting that (a) strings with fewer characters are 'greater than' long strings starting with the same characters: and (b) that strings starting with numbers are 'greater than' strings starting with characters.

LENgth of strings

LEN is a handy function. Used in the form *LEN (variable name)* it just figures out how many characters there are in the variable string. But that doesn't mean it's trivial – it is very useful for instance to make sure that someone entering something in response to an INPUT prompt is typing the right number of characters:

```
10 INPUT "TIME NOW"; TI$
20 IF LEN(TI$)<>6 THEN GOTO 10
```

The TI$ variable must always have six digits or you'll get a ?SYNTAX ERROR. Line 20 here checks that you've entered the correct number of digits.

LEN can also be handy to determine the end of a FOR/NEXT loop, for instance when fooling around with anagrams and letter-switches like the one illustrating the MID$ function. And one neat trick is to use LEN to centre text:

```
100 INPUT "TYPE SOMETHING";I$
120 PRINT TAB(20-LEN(I$)/2) I$
```

TAB is followed by a number or expression to determine the starting point along a line for a PRINT. Since the 64 has 40 characters per line, the mid-point is 20; and line 120 above works out the length of your text, halves it, and uses that to figure out where to start printing the string you've entered as I$.

Be wary about LEN if your strings are full of non-printing characters like spaces: they're all counted, and this could muck up your calculations if it's only the printable characters you want to use.

Segmenting strings – LEFT$, RIGHT$ and MID$

These three are handy methods of slicing lumps out of a string. With LEFT$ and RIGHT$ you specify how many characters you want, and you'll get that many starting from the left or the right of the string respectively; MID$ plucks characters out of the middle of the string, and you indicate the position where you want to start from as well as the number of characters you'd like.

To summarise that:

```
10 A$ = "SEGMENTATION"
20 PRINT LEFT$(A$,2)
30 PRINT MID$(A$,7,4)
40 PRINT RIGHT$(A$,2)

RUN
SE
TATI
ON
```

This somewhat pointless exercise shows how they work – LEFT$ has found the two characters on the left of the string variable indicated, RIGHT$ did the same for the two on the right, MID$ started at the seventh position and read four characters starting there.

Once you've got them you can use those selected characters:

```
10 A$ = "SEGMENTATION"
20 B$ = LEFT$(A$,1) + MID$(A$,7,3) +
RIGHT$(A$,3)
30 PRINT B$

RUN
STATION
```

Here we're picking out one character from the left, three from the middle starting at the seventh, and the three at the rightmost end. The plus signs concatenate the lot of them and we get a whole new word in B$.

These functions can provide a few quite entertaining games. Here's MID$ at work with LEN and a FOR/NEXT loop to spell your name backwards:

```
10 INPUT "YOUR NAME"; N$
20 FOR A=1 TO LEN(N$)
30 PRINT MID$(N$, LEN(N$)+1-A,1);
40 NEXT A

RUN
YOUR NAME? DENNIS
SINNED

READY
```

(Note that you need the semi-colon on line 30 to persuade the computer to print all the characters on the same line.) Here's another one:

```
10 INPUT "YOUR NAME"; N$
20 N$=N$ "... PRINTED SLOWLY"
30 FOR A=1 TO LEN(N$)
40 PRINT MID$(N$, A,1);
50 FOR T=1TO999: NEXT T
60 NEXT A
```

LEFT$ is useful with INPUT statements and other responses the user might type in. Take the classic 'yes/no' question; this check on the INPUT will cope with 'YES' *and* 'Y' as valid responses ...

```
100 INPUT "OK TO PROCEED"; A$
110 IF LEFT$(A$)<>"Y' THEN 100
120 END
```

If you're just checking for A$ = 'Y', a diligent user who typed 'YES' would find the program just cycling around until both got bored.

Converting strings to numbers (and back again)

It's often necessary and/or useful to convert numeric data into a character string. The 64's Basic has a clever STR$ function to do precisely that: it takes a number and turns it into a character string comprising the same digits. Two examples to try in direct mode:

```
PRINT STR$(99)
 99

V=1024: PRINT STR$(V)
 1024
```

Note that you're getting a blank space in front of the number. As with PRINT, this is there to cope with a minus sign for negative numbers – as you can see from this:

```
PRINT STR$(1024): PRINT STR$(-1024)
 1024
-1024
```

If you don't want the space to be PRINTed you can remove it with MID$ or LEFT$ – MID$ combined with LEN is probably the most efficacious method:

```
10 INPUT "TYPE A NUMBER"; N
20 N$=STR$(N)
30 PRINT MID$(N$,2,LEN(N$)-1) 40 GOTO
10
```

And why should you want to turn numbers into strings anyway? Well, one important consideration is that it's generally easier to format character strings neatly on the screen or on a printer. For example, PRINTing a list of numbers will line them up against the left-hand margin; but if you want them aligned more conventionally in a column, using STR$ and an adaptation of the centering routine is much easier than the fiddling around that would be necessary if using numeric data:

```
100 INPUT "NUMBER"; N
120 N$=STR$(N)
130 PRINT TAB(10-LEN(N$)) N$
```

This routine aligns numbers to the right, against the 10th column of the screen. Of course you could have INPUT the numbers as strings in the first place, but I'm assuming you have other purposes for them as numeric data.

STR$ also lets you use MID$, LEFT$ and RIGHT$ to slice lumps out of a number. Another pretty-PRINT example: if you want to PRINT the result of some calculation on the screen you *could* do it like this:

```
100 INPUT N1, N2
110 ANSWER = N1+N2
120 PRINT "THE RESULT IS" ANSWER "."
```

But you'd get a space after the number and before that full stop, because that's how numeric variables are printed. And you would also get a space before the number for a minus sign – which you can cope with by *not* putting a space at the end of that first string of text ... except that if ANSWER *is* negative you then won't have any space at all between the words and the number. This approach looks even more ugly if it's a currency calculation and you want to include a dollar or pound sign.

But to have it come up neatly without leading blanks causing an unsightly gap, you can use STR$ with the MID$/LEN$ trick to suppress the space allowed for a minus sign. This means the number can start right after a currency symbol; and STR$ numbers don't have that space following them, so the full stop will be in the right place too.

```
100 INPUT N1, N2
110 ANSWER$ = STR$(N1+N2)
120 PRINT "THE RESULT IS #"
MID$(ANSWER$,2,LEN(ANSWER$)-1) "."
```

When you're working with data that mixes text and numbers STR$ comes into its own. Take addresses; you will use addresses as strings, but you may want to sort them by house number or by postcode – so those you'll be keeping as numeric variables. STR$ lets you PRINT a combination of the two without requiring you to hold the whole lot permanently as strings.

On the other hand, if you wanted to do that you could. The VAL function is the corollary of STR$ in that it converts a string to a number. It only works on numeric digits, though, because its role is to permit you to use string numbers in calculations:

```
PRINT VAL("2379")
 2379

PRINT VAL("2379")/0.2
 11895

PRINT INT(VAL("2379")/0.2E3)
 11
```

If you had your postcodes tucked away with the rest of the addresses as string variables, you could use VAL to get at them for mathematical operations like some kind of sort. And here's a practical example: if you are storing information about birthdates, it wouldn't necessarily be easy to detect that '11159' meant 1 November 1959 rather than 11 January 1959. Putting a leading zero in there – '011159' – would make it clearer, but then the computer won't let you store numbers that start with a zero. So if you have to do it like that, you could use '011159' as a character string; and have VAL turn it into a useable number when you want to do some calculation with it.

There are a couple of things to watch with VAL. It will come back with 0 if the first character in the string is not a number, a sign, or a blank; so VAL('C12') will produce 0. But VAL('12C') will yield 12 – in other words, provided the string starts with an acceptable character VAL will return a value. It'll stop converting the string to numbers though as soon as it hits a non-qualifying character: VAL('12C12') will pick up only the first number, and you'll get '12' as the value.

ASC and CHR$

It's often useful to be able to work in terms of the ASCII value of a character – ASCII is the American Standard Code for Information Interchange, a widely-used set of numeric codes that correspond to the characters available on your computer. The ASCII codes for the 64's characters are listed in Appendix 5.

The ASC function lets you use the ASCII values of those characters directly, and that's its most important use. In particular, ASC can let you get at some of the non-printing characters – like Return and DELete:

```
10 GET A$: IF A$="" THEN 10
20 PRINT A$, ASC(A$)
30 IF ASC(A$)=13 THEN PRINT "RETURN"
40 IF ASC(A$)=20 THEN PRINT "DELETE"
50 GOTO 10
```

This program will print the ASCII values of any of the keys on the keyboard, including the colour-change, Shift and CBM combinations.

The two forms of ASC are ...

ASC("*character*") which comes back with the ASCII code of the character
ASC(*string variable*) which comes back with the ASCII code of the first (ie the leftmost) character in the string

The corollary of ASC is CHR$, and in practice it's rather more useful – CHR$ is followed by an ASCII code number and generates the equivalent character. This makes it really useful for incorporating non-printable characters like Return in PRINT statements ...

```
PRINT "TOP" + CHR$(13) + "LINE SPACE"
```

It can also be used instead of ASC to check for Return in response to a GET:

```
10 GET A$: IF A$="" THEN 10
20 IF A$=CHR$(13) THEN PRINT "RETURN"
30 REM program continues ...
```

CHR$(34) is another useful one, since it enables you to put quote marks inside a string. If you attempt to include quotes in a string simply by typing the character, the 64 will interpret your efforts as the end of the quotes that signal the start of the string. But this kind of thing is acceptable:

```
10 INPUT "NAME"; N$
20 PRINT CHR$(34)+N$+CHR$(34)+" ...
WHAT KIND OF NAME IS THAT?"
```

A third use of CHR$ is to improve the readability of programs. There are character codes for all the screen controls and video effects; so you can use CHR$ codes instead of filling character strings with the unhelpful and often illegible hieroglyphics produced when you switch reversed printing on or change the colour of text. For instance ...

```
10 PRINT CHR$(147)
20 PRINT CHR$(14) CHR$(18) CHR$(28)
"HELLO THERE" CHR$(144)
```

CHR$(147) is the clear-screen code; CHR$(14) switches to lower-case, CHR$(18) puts the reverse-video on, CHR$(28) changes the text colour to red. CHR$(144) leaves the cursor and subsequent text in black.

That brings us to the most important use of CHR$ – it's the *only* way of specifying some of the controls on the 64, notably those for printer formatting.

CHR$(10), for instance, forces a line feed after printing: in other words the paper automatically rolls up one line. CHR$(13) is generally preferable, and as with PRINTing to the screen it forces a Return after printing: the paper automatically rolls up *and* the print head goes back to the lefthand edge. When CHR$(14) is used every following it will be printed double-width; CHR$(15) switches back to normal-width printing ...

CHAPTER 12
Handling Numbers

Largely because Basic originated in the groves of academe, the language is loaded with number-handling facilities. Which is great if you want to do complex calculations, and it's ok too if you like playing around with numbers. It's not so great for the innumerate or those of us who haven't had a particularly mathematical education.

The reference summary in Appendix 1 lists the number-handling facilities in Basic, and your 64's handbook will indicate how they operate. But even if you are intimidated by maths (like me!), read on: this Chapter highlights some of the more generally useful attributes of Basic's ways with numbers.

Random numbers

Here at least is one numeric function you needn't be scared of. Computers are fast rule-following idiots. Because they follow rules, they're completely predictable; there wouldn't be much point in writing programs for them to obey if they weren't.

On the other hand, sometimes you'll want the computer to do something *un*predictable – particularly in games, where being predictable is a positive liability. To provide at least a measure of chance, Basic includes a RND statement which generates numbers randomly ...

Or rather it doesn't. It generates 'pseudo'-random numbers, which means the computer continues to operate predictably and within a set of rules; it just does so within quite a large range of possible outcomes. RND works on what is called a 'seed' number and applies to it a preset algorithm which can actually give the same result every time ...

```
10 PRINT RND(-1): GOTO 10
RUN
```

To make the number produced as random as possible, the starting point for the algorithm – the seed number in brackets after the RND – should be chosen carefully. Basically you've got four options:

Use a negative number. That way you'll get the same random number generated every time – not much help in running a game, but it can be very useful in checking through a program that uses a random number generator. Once you've sorted out the program you can change the RND statement

Use a positive number. A bit better, since once the first random number is generated it's taken as the seed value for the next one. So it's unlikely that values will be repeated:

```
10 PRINT RND(1): GOTO 10
RUN
```

Use zero. This is much better at producing random numbers, since the seed is taken to be the current contents of the jiffy clock – which is constantly changing, so the seed number is unlikely to be the same for any two runs unless you're resetting the clock every time:

```
10 PRINT RND(0): GOTO 10
RUN
```

Use a mixture of values. The seed can be any numeric constant, variable, or composite expression; so if you want to go over the top, you can complicate the randomising process quite cheerfully:

```
10 PRINT
RND(TI*(RND(0)*RND(0)*RND(1)))
```

Not that it's necessary, but it's fun.

RUNning these examples will show you that RND produces numbers in the range 0 to .999999999 inclusive. And providing you're not using a minus value as the seed, all the possible random numbers have an equal chance of turning up. So if you want your program to make decisions on the basis of that chance you could just split the range accordingly:

```
10 INPUT "HEADS OR TAILS"; A$
20 IF RND(0) > .5 THEN B$="HEADS"
30 IF RND(0) < .5 THEN B$="TAILS"
40 PRINT B$
50 IF A$=B$ THEN PRINT "YOU WIN"
60 IF A$<>B$ THEN PRINT "YOU LOSE"
70 INPUT "AGAIN?"; Z$: IF Z$<>"Y" THEN
END
80 GOTO 10
```

More likely you'll want to convert the random number you've produced into an integer. There's not much point in using INT(RND(X)), since that's going to produce only 0; and unless you want to restrict yourself to two choices (0 and 1) there's very little mileage out of adding 0.5 to round up the number as in INT(RND(X) + 0.5). What you need to do is multiply the whole lot by something and tack on an extra value:

random number = *N1* * **RND(0)** + *N2*

Or, to ensure against fractional results:

random number = **INT**(*N1* * **RND(0)** + *N2*)

Here's a little program that uses this to simulate rolling a pair of dice:

```
10 DA = INT(6 * RND(0)) + 1: DB = INT(6 *
RND(0)) + 1
20 PRINT DA, DB
30 PRINT "ANY KEY FOR NEXT THROW"
40 GET K$: IF K$="" THEN 40
50 GOTO 10
```

The two dice in line 10 produce values in the range 1 to 6 because we're multiplying RND(0) by the number of possible results (dice have six sides – and 6*RND(0) produces a number in the range 0 to 5.99 recurring), chopping off the fraction with INT (giving a possible range of 0 to 5), and adding 1 to ensure that the end result is somewhere between 1 and 6.

Card shuffling

If you're producing card games, one way to ensure that you have a well-shuffled pack is to DIM an array with 52 positions and then generate random numbers in the range 1 to 52 to put into them. Of course you'll have to check that the same numbers aren't used twice; and doing it this way can mean a very long delay towards the end of the deck, with your program generated random numbers like crazy only to find that most of them have already been used.

A better way of shuffling cards is to emulate real life – set out the deck first and *then* shuffle them. This sets up a 52-position array and fills it with numbers in sequence:

```
10 DIM PACK%(52): PACK%(0)=-1
20 FOR CARD=1 TO 52: PACK%(CARD) = CARD:
NEXT
```

Now we can shuffle them. Line 110 here generates a random number in the range 1 to 52; that's uses to select one of the cards in PACK%, which we're putting temporarily into a variable called HH so that we don't lose it. Then a card from the pack is selected by line 130 and placed somewhere else in the pack (in position PACK%(RCARD) in fact), with the card originally there – the one in HH – replacing it. This exchange is done for all 52 cards:

```
100 FOR SHUFFLE = 1 TO 52
110 RCARD = INT (52 * RND(0)) + 1
120 HH = PACK%(RCARD)
130 PACK%(RCARD) = PACK%(SHUFFLE)
140 PACK%(SHUFFLE) = HH
150 NEXT
```

If you now want to deal a poker hand, you can just select the first five cards:

```
1000 FOR DEAL = 1 TO 5
1010 PRINT PACK%(DEAL)
1020 NEXT
```

Or for two players:

```
1000 PRINT "PLAYER 1", "PLAYER 2"
1010 FOR DEAL = 1 TO 10 STEP 2
1020 PRINT PACK%(DEAL), PACK%(DEAL+1)
1030 NEXT
```

You could make it more realistic, of course, by assigning suits and adjusting the values of the card. DIM an array to hold the suits, and add a couple of new sections:

```
10 DIM PACK%(52), PACK$(52):
PACK%(0)=-1

300 FOR SUIT=1 TO 52
310 IF PACK%(SUIT) < 14 THEN
PACK$(SUIT) = "<diamonds>"
320 IF PACK%(SUIT) > 14 AND PACK%(SUIT)
< 27 THEN PACK$(SUIT) = "<clubs>"
330 IF PACK%(SUIT) > 26 AND PACK%(SUIT)
< 40 THEN PACK$(SUIT) = "<spades>"
340 IF PACK%(SUIT) > 39 THEN
PACK$(SUIT) = "<hearts>"
350 NEXT

400 FOR NUM=1 TO 52
410 IF PACK$(NUM) = "<clubs>" THEN
PACK%(NUM) = PACK%(NUM)-13
420 IF PACK$(NUM) = "<spades>" THEN
PACK%(NUM) = PACK%(NUM)-26
430 IF PACK$(NUM) = "<hearts>" THEN
PACK%(NUM) = PACK%(NUM)-39
440 NEXT
```

The dealing section should also be altered:

```
1020 PRINT PACK%(DEAL) PACK$(DEAL),
PACK%(DEAL+1) PACK$(DEAL+1)
```

Statistical simulation with RND

You might be able to use this piece of information in simulation programs of some kind – flight simulators, say, or 'run your own country' games. If you know (or can assume) the *average* probability of something happening, you can use RND with LOG to come up with a reasonable figure for the *actual* event. The formula is ...

– *average* * **LOG (1-RND(0))**

So you're simulating the operation of a company and you know the average receivables period – how long it takes your customers to pay the bills you send them. You want to give the players some actual figures so that they can run their companies on a month-to-month basis:

```
10 INPUT "AVERAGE RECEIVABLES PERIOD";
AR
20 INPUT "NUMBER OF BILLS"; M
30 FOR C=1 TO M
40 PAYMENT = INT(-AR*LOG(1-RND(0)))
50 PRINT "BILL NO" C ": PAID IN " PAYMENT
"DAYS"
60 NEXT
```

Other uses for this: how often a jet will crash, how frequently you'll meet a really *simpatico* lover, when the nasties will attack your starbase ...

Last word on RND

Try this little one-liner – it produces a scrolling background of stars (or whatever you put into the quotes):

```
10 PRINT SPC (RND(1)*39)"*": FOR T=1 TO
40: NEXT: GOTO 10
```

Insert another FOR/NEXT loop at the front if you want to limit the number of stars.

ABSOLUTE values

The ABS function determines the absolute value of a number – which means it tells you what the number is, irrespective of the sign it carries. In other words it leaves positive numbers as they are and lops the minus sign off negative numbers.

Which, you might think, is a bit pointless. But here are some potentially useful applications for ABS:

- **Financial calculations** sometimes have expenditures or losses expressed as a minus quantity.

That makes quite good sense if you have a long column of incoming and outgoing money and you just want to find the overall balance by totting up the whole lot:

```
10 CLR: DIM BALANCE(8)
20 FOR C=1 TO 8 30 INPUT "ENTRY (MINUS
FOR EXPENSES)"; E
40 BALANCE(C)=E: NEXT
100 FOR C=1 TO 8: SUM=SUM+BALANCE(C):
NEXT
120 PRINT "BALANCE =" SUM
```

Here we've DIMensioned an array called BALANCE into which you read eight items that could be income or outgoings. Line 100 goes through them and works out the nett result.

If however you wanted to do your accounts a different way, with the more conventional formal layout of income in one column and expenses in another, you might want to list the outgoings without the minus sign. ABS will do that for you – it won't contribute much to the calculation but it will allow for neater formatting.

So here's another method of doing your balance-sheet calculation in a way that stills allows you to enter the data without having to separate income items from expenditure first: I've added a bit more flexibility by allowing you to define the approximate size of the operation (line 20).

```
10 CLR
20 INPUT "MAX. NO. OF ITEMS?"; M
30 DIM INCOME(M), OUTGOING(M)
40 FOR C=1 TO M
50 INPUT "ENTRY (MINUS FOR EXPENSES)";
E
60 IF E<0 THEN OUTGOING(C) = ABS(E)
70 IF E>0 THEN INCOME(C) = E
80 NEXT
100 FOR C=1 TO M: S=S+INCOME(C):
T=T+OUTGOING(C): NEXT
120 PRINT "INCOME =" S, "OUTGOINGS =" T
130 PRINT "BALANCE =" S-T
```

- The **distance between two numbers** is ABS(n-m).

Why should you want to calculate that? Well, the classic example is usually given as finding out how much older one person is than another:

```
10 INPUT "FIRST PERSON'S AGE"; A1
20 INPUT "SECOND PERSON'S AGE"; A2
30 PRINT "DIFFERENCE IS " ABS(A1-A2)
```

If line 30 just PRINTed A1-A2, it would come out a minus number if the second person was older. Using ABS means it always comes out as a sensibly positive number ...

- **Preventing program crashes with square root and logarithm calculations** is another helpful role for ABS.

The SQR and LOG functions demand a positive number to work on; otherwise the program will die with a ?SYNTAX ERROR. Using ABS ensures that they always get an acceptable number to work on.

- ABS can also **simplify statements** in some cases – for instance where you're testing for a range of values from plus to minus.

Instead of using something like ...

```
100 IF ZA > -3 AND ZA < 3 THEN ...
```

... you can write:

```
100 IF ZA < ABS(3) THEN ...
```

- **Toggling between two values** is another neat one. You might have a program that flips a variable between different values depending on which iteration is running – the first time around a FOR loop the variable has one value, the next time it has the alternative value, the third circuit has it reset to the first one, and so on.

The obvious way to do this would probably be a couple of lines like this, typically in a subroutine:

```
100 IF V = 1970 THEN V = 1980: RETURN
110 120 IF V = 1980 THEN V = 1970: RETURN
```

But here's a version with ABS that takes only one line:

```
100 V = ABS(V-(1980+1970)): RETURN
```

Adding the two possible values and subtracting them from the current version of V ensures that the remainder will be the alternative one; and using ABS ensures that it won't have a minus sign.

Reading the signs

The SGN function is the corollary of ABS. While ABS discards any sign so you can look at the number, SGN junks the number and keeps the sign. There are three possible results:

- **SGN(***expression or value***)** = **1** if the number is positive
- **SGN(***expression or value***)** = **0** if the number is zero
- **SGN(***expression or value***)** = **–1** if the number is negative

And why should you want to know the sign of a number? Here's an alternative to the income/expenditure program:

```
10 CLR
20 INPUT "MAX. NO. OF ITEMS?"; M
30 DIM INCOME(M), OUTGOING(M)
40 FOR C=1 TO M
50 INPUT "ENTRY (MINUS FOR EXPENSES)";
E
60 IF SGN(E) < 0 THEN OUTGOING(C) =
ABS(E)
70 IF SGN(E) > 0 THEN INCOME(C) = E
80 NEXT
100 FOR C=1 TO M: S=S+INCOME(C):
T=T+OUTGOING(C): NEXT
120 PRINT "INCOME =" S, "OUTGOINGS =" T
130 PRINT "BALANCE =" S-T
```

Lines 60 and 70 now use the entirely logical basis of the sign of your input to make the decision on which array to put the entry into. Using SGN to make decisions like this can be taken a stage further:

```
60 ON (SGN(E)+2) GOSUB 1000, 2000, 2000
```

Adding 2 to SGN(E) turns it into an integer acceptable to the ON statement – if E is negative, SGN(E) ends up with -1 and adding 2 makes the result 1; if it's zero, adding 2 gives 2; if it's positive SGN(E) is +1 and you get 3. The first option could take you to a subroutine that sets up the array for expenditures; if SGN(E)+2 evaluates as 2 or 3 you'll go to a subroutine that handles income.

SGN can also be used with INT and ABS to give you fault-free rounding. This little number rounds to as many decimal places as you want:

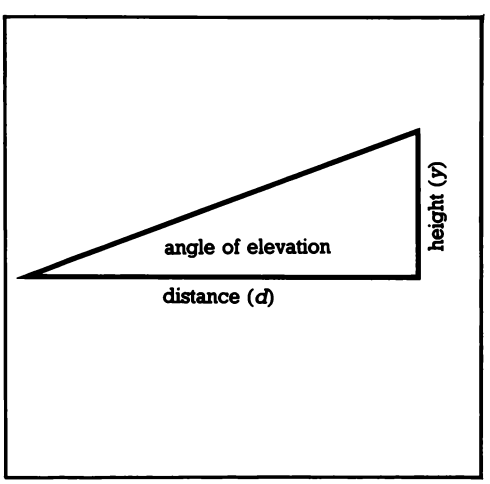
```
10 INPUT "NUMBER"; N
20 INPUT "HOW MANY DECIMAL PLACES"; D
10 INPUT NUMBER; N
20 INPUT HOW MANY DECIMAL PLACES; D
30 DP = 10↑D
40 M = (ABS(N) * DP) +.5
50 PRINT SGN(N) * INT(M)/DP
60 GOTO 10
```

Trigonometry

Basic is packed with trigonometrical functions – more than you'd expect, certainly, and more than the average user uses. It would probably have been a good idea for the people who wrote the 64's Basic to omit some of them and put in a few helpful extras of more general relevance ...

Still, that's what we're stuck with. Trigonometry you either love or hate, or maybe you just can't get on top of it – like me. So you might never use the trig functions. But there are a couple of helpful wrinkles available with them even if you do loathe trigonometry.

Here's one of them. The ATN (ArcTaNgent) function tells you what angle would be formed by a given distance and height. Or to put it another way, ATN tells you how many degrees from the vertical you'll have to raise the muzzle of your laser cannon if you know how high the invading nasties are and how far away their ship is:



ATN tells you what that angle is by dividing the *y* value – the height – by the *d* value (distance). You get the result in radians, so divide by *pi* and multiply by 180 to get the answer in degrees.

This knowledge you could use in a game. Every press of the f1 key moves the gun up one degree, say, and f3 moves it down one degree (use GET and check for CHR$(133) and CHR$(134) respectively. When the angle of the gun matches the position of the alien, hit F for fire.

Make it a bit more difficult by having the alien dodge around a bit, getting nearer to you and lower every few jiffies, so that the player has to move the gun around.

Here's a simplified version of that scenario:

```
10    REM Arctan alien - initialise
20 CLR
30 D=100000: R=80000: H=50000:
TI$="000000": T=1000
40 POKE 53280,1: POKE 53281,1

1000    REM getting closer
1010 FOR C=1 TO 50: PRINT "SCANNING
...": NEXT
1020 PRINT "<clr>" TAB(8) "<pur>ALIEN
CRAFT APPROACHING!"
1030 PRINT TAB(2) "<cud><blk><rvs>
DISTANCE:<rvo>" TAB(12) D "MILLION
MILES"
1040 PRINT TAB(4) "<blk><rvs> HEIGHT:
<rvo>" TAB(12) H "MILLION MILES"

2000    REM decrement range
2010 H=H-500
2020 D=INT(D-(TI*10)): IF SGN(D)=-1
THEN GOSUB 6000: GOTO 5020
2030 IF D<R THEN GOTO 3000
2040 IF D>R THEN PRINT" ... OUT OF RANGE
...": FOR C=1 TO 300: NEXT: GOTO 2000

3000    REM in range!
3010 PRINT TAB(8) "<ora><cud x 2>
ALIEN WITHIN RANGE!": FOR C=1 TO 40:
NEXT
3020 INPUT "<cud><blu> ANGLE OF FIRE";
F
3030 P=INT(ATN(H/D)/π*180)
3040 IF F<P THEN PRINT "<rvs><brn> TOO
LOW!!!": GOTO 1030
3050 IF F>P THEN PRINT "<rvs><gr2> TOO
HIGH!!!": GOTO 1030
3060 PRINT "<rvs><blu> ON TARGET -
PRESS 'F' TO FIRE!"
3070 GET F$: IF F$="" THEN 3090
3080 GOSUB 6000

5000    REM messages
5010 PRINT "<clr><cud x 8><blk> WELL
DONE.": GOTO 5030
5020 PRINT "<clr><cud x 8><blk>
SORRY, THEY GOT YOU."
5030 PRINT "<cud> ANOTHER?"
5040 GET A$: IF A$="" THEN 5040
5050 IF A$="Y" THEN 10
5060 END

6000    REM explosion
6010 PRINT "<clr>"
6020 FOR C=1 TO 254: POKE 53280,C: POKE
53281,C: NEXT
6030 POKE 53280,1: POKE 53281,1: NEXT
6040 RETURN
```

The nasties appear at 100,000 million miles and they get within range at 80,000 million miles (decided by the variable R set in line 30). Their height is reduced by 500 miles on every scan (line 2010) and the TI clock is used to decrement their range on a rather less predictable basis (line 2020). If you haven't zapped them by the time they reach you – determined by line 2020, where they've got to you if the distance D gets to be a minus figure – you're assumed to be zonked yourself.

Once they are in range (distance D = range R) you're asked for an angle of fire. Line 3030 uses ATN to calculate the right angle and that's compared with your attempt in 3040; if you're wrong, the program loops back to give another scan with both height and distance decremented again.

If you've got the right angle – and because D is being altered by the TI clock it's not easy to predict – you can fire.

Logarithms

When I was a kid I stumbled through lessons on logarithms with a near-total lack of comprehension and a blind devotion to rule-following: I dutifully looked up the logs in a book, added them together, found the anti-log in another column, and for no good reason that I could see I'd added two numbers together.

While hopefully I have a somewhat clearer understanding I don't intend to try explaining logs here, either – though I will attempt a definition: the log of a number is the exponent to which a base must be raised to get that number. The log to base ten of 100000 is 5; reversing that, PRINT 10 ↑ 5 (10 to the power of 5) will produce 100000.

The LOG function in Basic is a bit more esoteric: it has an *implied* base, the constant 'e', which is 2.718 and a bit. This is called a *natural* logarithm, and when you see a log written without a specific base it's a natural log.

Using the base 'e' results in several rather special mathematical properties. Here are two applications for Basic's LOG function – figure out how they work if you want, or just try using them blindly:

- **Pretty printing** is provided on many versions of Basic by a 'PRINT AT' and/or a 'PRINT USING' statement. Commodore Basic doesn't have these as standard, which among other things makes it difficult to align columns of numbers neatly around the decimal point ...

```
    2.55
  712
34883.3
    0.72
```

There *are* ways to do this using character strings, but it can get very complex. For instance, you could use STR$ to turn numbers in string variables, have LEN find out their length, and PRINT them with a TAB position calculated from LEN:

```
10 INPUT "COLUMN"; C
20 INPUT "NUMBER"; N
30 N$=STR$(N)
40 NL=LEN(N$)
50 PRINT TAB(C-NL) RIGHT$(N$,NL)
60 GOTO 20
```

But not even that will align numbers around the decimal point – you'll get everything butted up against the right-hand position you've specified:

```
 2.55
 712
34883.3
 0.72
```

What you'd *really* like is for all the decimal points to be below each other. You can do this with LOG quite simply:

```
10 INPUT "COLUMN"; C
20 INPUT "NUMBER"; N
30 L=LOG(10)
40 P=LOG(N)/L
50 PRINT TAB(C-P);N
60 GOTO 20
```

This version has the inestimable merit of leaving the numbers as numbers rather than as strings; so you can use them subsequently in calculations without having an extra VAL to convert them back again.

Sadly, there is a problem which complicates things a bit – LOG won't work with numbers that are zero or negative. Adding a line to cope with that rather reduces the compactness of the routine:

```
25 IF N=0 OR N<0 THEN P=1: GOTO 50
```

- **Compound interest calculations** can also be done with LOG – or rather, you can find the number of years required for money to grow at a specified rate.

This is the formula:

years = * **LOG** (*growth factor required*) / (<*compound interest rate*>/ **100**)

Or, to put it another way ...

```
100 YEARS = LOG(GROWTH)/(RATE/100)
```

Here's a little 'what-if' program using that:

```
10 INPUT "INVESTMENT"; MONEY
20 INPUT "GROWTH"; GROWTH
30 INPUT "RATE"; RATE
100 YEARS = LOG(GROWTH)/(RATE/100)
110 YEARS = INT(YEARS*10)/10
120 PRINT MONEY "WILL INCREASE BY "
GROWTH "TIMES AT " RATE "% IN " YEARS
"YEARS"
```

Line 110 is just a neatener. The LOG formula will produce a lengthy fraction; so 110 multiplies it by 10 (which shifts the decimal point), takes the integer (which lops off everything after the decimal point) and then divides by 10 (to restore the point to its original position). The result will be a number of years with a single decimal place.

Defining your own functions

The SIN, COS, TAN and ATN functions all produce an answer in radians – which probably won't be as useful to you as an answer in degrees. The Arctan Alien program included a method of converting radians to degrees, but if you're going to do a lot of trig it may be as well to define your own functions for trigonometry.

Basic includes a statement that allows you to do this. It is useful to set up formulae that you don't want to have to repeat all the time, and it looks like this:

DEF FN *variable name* (*numeric value*) = *rule*

Subsequently you can use more or less like any other function, in the form ...

FN *variable name* (*numeric value*)

The variable name you're using to identify the function can be of one or two alphabetic characters (so you can DEFine up to 676 functions with different letter combinations!). The value bit is another numeric variable name that may be used in the rule that actually defines what the function will do. This bracketed name is a dummy variable, in fact, and that's why 'X' is generally used for it.

To see how DEF FN works, here are the redefinitions for SIN, COS and TAN:

```
10 DEF FN SI(X) = SIN(X*π/180)
20 DEF FN CO(X) = COS(X*π/180)
30 DEF FN TA(X) = TAN(X*π/180)
```

Thereafter you can use something like this in your program:

```
100 INPUT "ANGLE"; A
120 S = FN SI(A):  C = FN CO(A):  T = FN
TA(A)
130 PRINT "SINE OF ANGLE " A "IS " S
140 PRINT "COSINE OF ANGLE " A "IS " C
150 PRINT "TANGENT OF ANGLE " A "IS " T
```

When you come to use an FN function, you include a value (or an expression) in the brackets in place of the numeric variable name that appeared in the DEF FN statement. This number is then utilised by the 64 to work out the rule that you defined.

Here are two more trig functions that aren't provided in Basic as it stands and which you might need – for arc sine and arc cosine:

```
40 DEF FN AS (X) = ATN (X/SQR(1-
X*X))*180/π
50 DEF FN AC (X) = (π/2-ATN(X/SQR(1-
X*X))) * 180/π
```

Note that all of these defined functions demand input in *degrees* – and not degrees and minutes (or seconds). One or two additional DEF FN statements can be added to convert them to degrees and fractions of a degree; alternatively you could do it with three INPUTs, with the resulting number being assembled into one variable before it's PRINTed:

```
10 DEF FN SI (X) = SIN (X*π/180)
20 DEF FN CO (X) = COS (X*π/180)
30 DEF FN TA (X) = TAN (X*π/180)
40 DEF FN AS (X) = ATN (X/SQR(1-
X*X))*180/π
50 DEF FN AC (X) = (π/2-ATN(X/SQR(1-
X*X))) * 180/π
100 INPUT "DEGREES, MINUTES, SECONDS";
D, M, S
110 A = D + (M / 60) + (S / 60 / 60)
120 S = FN SI (A):  C = FN CO (A):  T = FN
TA (A): A1 = FN AS (A):  A2 = FN AC (A)
130 PRINT "SINE OF ANGLE " A "IS " S
140 PRINT "COSINE OF ANGLE " A "IS " C
150 PRINT "TANGENT OF ANGLE " A "IS " T
160 PRINT "ARC SINE OF ANGLE " A "IS " A1
170 PRINT "ARC COSINE OF ANGLE " A "IS "
A2
```

DEF FN can be extremely handy, even in non-mathematical work – to work out print positions, for instance. Or there's this one:

```
10 DEF FN R(X) = INT(((X*100)+0.5)/
100)
100 INPUT "NUMBER"; N
110 PRINT FN R (N)
```

This defines a rounding function that works conventionally enough – the number is multiplied by 100 to shift the decimal point two places to the left, 0.5 is added to ensure that it will be rounded up rather than down, INT lops off the unwanted decimal places, and the result is divided by 10 to restore two of them.

To give yourself the option of specifying the number of places you want to work with, you might DEFine another function and incorporate it:

```
10 DEF FN R(X) = INT(((X*FNP(M))+0.5)/
FNP(M))
20 DEF FN P(M) = 10↑M
100 INPUT "NUMBER"; N
110 INPUT "DECIMAL PLACES"; M
120 PRINT FN R (N)
```

There are a couple of limitations with DEF FN, though: for a start you can't define *string* functions, which is a bit of a shame.

And you can't define integer functions either, at least not by giving the function's variable name a percent symbol to force all uses of it to be an integer: doing that will produce a ?SYNTAX ERROR. On the other hand it's easy enough to include the INT function as part of the rule that you're setting up with a DEF FN statement, so it's no great loss.

The dummy variable in a DEF FN can be used with some totally different meaning elsewhere in the program. For instance, here the same name is apparently being used twice:

```
10 R=12.33
20 DEF FNA (R) = 2 * π R
300 INPUT "RADIUS"; R1
310 IF R1<>0 THEN R=R1
320 PRINT "AREA = " FNA(R)
```

R is defined as having one value in line 10, and even though the DEF FN *also* uses R it doesn't have any affect on the function until line 320 – and then only if you entered something other than '0' at the INPUT prompt.

The main thing to watch about DEF FN is where you have the function definition statements. The very best place is first thing, right at the start of your program: this way there's no danger that you'll try using the functions before they have been DEFined.

CHAPTER 13
Handling Memory

One of the really good things about the 64 is the amount of memory you get. An eight-bit microprocessor like the 64's 6510 can directly address a maximum of 64K bytes, since 65,536 is the largest possible number that you can express in the eight binary digits an eight-bit micro works with. The 64 has as much memory as it sensibly can have.

And one of the really irritating things about it is the way you can't use all that memory. What's worse, it can be a complicated business getting at the memory you *can* use.

How the 64's memory is organised

Memory consists of individual 'cells' in which information is stored. Those cells are one byte in size, which means each memory location may store whatever can be encoded in eight binary digits – could be a letter, a number, an instruction, or whatever.

Instructions are the important ones. Most of what the 64 does – everything from running your programs to keeping the cursor blinking – is controlled by software. In other words, the thing is loaded with little programs and subroutines even before you start putting anything in.

You can't do without those routines, since they effectively fit between the outside world (what you see and what you do) and the streams of electrons that are pulsing around the 64's innards. They drive the computer for you and do all the things that you can take for granted. And they live in the 64's memory ready to be used: which means they take up part of the 65,536 addressable memory locations, the 64K bytes that the thing has.

When you turn on the 64 it actually begins to run a big program that is permanently there inside the computer – it is not lost when you turn off the power, and it cannot be changed. This program isn't written in Basic; instead it is coded in a programming language called 'machine language' that is much closer to the way the 64 actually works. Every microprocessor works with and understands only its own type of instructions, a machine language, and this is the only one the processor understands.

That makes it pretty damn incomprehensible to the beginner – much more so than Basic, for instance, which does at least use some recognisable English-like words.

That big machine-language program handles all the 64's start-up activities: it checks out everything inside, puts the opening messages on the screen, and finally tells you it's READY. Subsequently it drives the cassette, transfers information to and from tape and disk, checks the keyboard continually to see if you've pressed anything, puts information on to the screen and keeps it there ...

It does a lot more, too. And when you type in Basic commands, they are in fact being translated by this machine-language program – it's *interpreting* them, one at a time. The 'interpretation' allows us to use a computer language that is significantly nearer to English; the nice thing about a 'high-level' language like Basic is the fact that you don't have to worry much about what's going on inside the 64. You don't have to concern yourself with anything really *machine*-oriented.

Including the way memory is organised ... Unless you want to use your 64 to the full. That will require some appreciation of what's going on in the memory (and where).

The machine-language program in the 64 is permanent because it's in ROM, 'read-only memory'. ROM is a type of memory (a type of memory chip, in fact) which doesn't lose its contents when the power is switched off; the machine code program is put into ROM at the factory, and you can't do much about it.

The 64 has 20K bytes of machine-language program in ROM, which means that you 'lose' 20K bytes of addressable memory in the sense that you can't use it for your own programs. In fact however the 64's memory is organised so that you lose only 16K of it; the other 4K of ROM is rather cleverly interchanged automatically with a 4K area of user memory. The 64 has the capability of referencing 65,536 individual memory locations: but it doesn't much care where it finds them, and they don't have to correspond to particular electronic components.

'User' memory, the memory you get for your own programs, is called 'RAM' – for 'random-access memory'. That's a slightly silly term, in fact, for there's nothing particularly random about how you access the memory locations in RAM.

So you've got a computer with 48K bytes for you and 16K for the system itself? No, because 6K bytes is required by the 64 for keeping variable information about what's happening when you run a Basic program you've loaded – things like the list of variables you've defined and pointers to which line should be executed next.

That brings you down to 42K bytes – and another 4K bytes is also lost to your Basic programs. That can be used for programs written in languages other than Basic, but you just can't get at it from a Basic program.

So the 64 has only 38K bytes of its claimed 64K for your Basic. And if you want to plug in some kind of extension cartridge, a Programmer's Aid or somesuch, it will tell the 64 that it's living in up to 8K bytes of your user-program memory – cutting that back to 30KB.

Memory in the 64

addresses	function
0 to 1023	1K RAM used by Basic and the Kernal to set system operating parameters and as temporary data storage areas
1024 to 2047	1K RAM used to hold the character codes currently displayed on a map of the screen. Also has pointers to locations holding the data for sprites
2048 to 40959	just about 38K of RAM for user's programs in Basic
40960 to 49151	8K ROM for Basic interpreter; or 8K RAM for machine-code – typically for user programs that don't utilise Basic, or for copying Basic itself for examination and modification
49152 to 53247	4K of unused RAM – inaccessible for Basic, but a great place to put machine-code subroutines called by SYS or USR
53248 to 54271	about 1K of ROM for sprite and video control
54272 to 55295	about 1K of ROM for the SID sound generator
55296 to 56319	1K RAM for the screen colour 'map'
56320 to 65535	8K ROM for the Kernal (the 64's operating system); or 8K of RAM if you're not intending to use the Kernal

Getting at memory

The 64 has a number of commands that allow you to get at specific memory locations. And while that the flip side of being able to observe and interfere with the way the computer works is the fact that you also have the power to make it *not* work, you shouldn't be put off: you can't interfere with the *electronics* of the 64, and next time you switch off and on again everything will be back to normal.

The two most useful such commands are PEEK and POKE: PEEK allows you to see what's in a particular memory location, POKE puts something into it.

PEEK is a function rather than a command, so it has to be used with another command. For instance, PRINT PEEK(53280) prints the contents of location 24576 – that happens to be the location which decides what colour the screen's border is. The results of a PEEK are a decimal number rather than binary, even though that memory location is storing the number as a sequence of eight binary digits.

PEEK can also be used like any constant or variable, with a label being assigned to it for instance ...

```
100 BRDR=PEEK(53280)
```

And it can be used in loops:

```
110 FOR I TO PEEK(53280)
```

Or in compound expressions:

```
120 PRINT (PEEK(53280)/4)+1/.67
```

PEEK tells you what's in a particular location – the POKE command lets you alter the contents.

```
POKE 53280,10
```

In effect that means 'put the value 10 into memory location 53280'. Try it and you'll get a different border colour.

You can PEEK at most locations in the 64's memory and POKE new values into them – except that you can't POKE into ROM areas, which seems reasonable enough since it's *read-only* memory.

Setting up the display

The classic use of these two commands is in setting up the display. The picture on your TV set is a reflection of two different areas of memory in the 64: one says which character is in each position, the other specifies what colour each character is.

All characters have a unique numeric code to distinguish them, and so do all colours. Since the screen is a reflection of memory, you can PEEK at specific locations in the two areas to see what character code and what colour code are being displayed at any one position: and you can POKE in new values to change the character on view and its colour.

If you turn the 64 on, press the CLR/HOME key to clear the screen, and type ...

```
PRINT PEEK(1024)
```

... the 64 should respond with '16'. Location 1024 is the start of the 'screen memory', the memory 'map' containing the codes for any characters currently on the screen. The top lefthand corner is location 1024, the next character position along is 1025, and so on: the bottom right corner is location number 2023 (there are 1,000 possible positions on the screen – 25 lines by 40 characters a line).

What you've just done with this statement is to PEEK into the location corresponding to the top left corner: and in there is character code 16 – which is the code for that 'P' from the word 'PEEK' which is there right now.

Press the CLR again and type ...

```
POKE 1024,55
```

As soon as you hit Return the 'P' of 'POKE' should change to a '5' – because the code for a 5 is '55': and you've just put code 55 into the top lefthand location of the screen memory.

The colour memory works in much the same way: the value in each location decides what colour anything displayed there will be. It starts at location 55296, again corresponding to the top left corner (bottom right is 1,000 positions away at location 56295).

To see how it works, try typing ...

```
POKE 55296,9
```

The '5' should have changed to brown, for '9' is the code for that colour. You can have any of the 16 colours in these locations, numbering them 0 to 15.

It can be tedious to lay characters on the screen and set their colour by specifying particular addresses for all the colour and character position locations. These two formulae may help:

character code position = 1024 + y * 40 + x
colour code position = 55296 + y * 40 + x

Here **x** is the horizontal position along a line and **y** the vertical position in terms of number of lines from the top. This program illustrates how the formulae work:

```
10 CLR
20 FOR X=0 TO 39
30 FOR Y=0 TO 23
40 P=Y*40+X
50 POKE 1024+P, CH
60 POKE 55296+P, COL
70 CH=CH + 1: IF CH>255 THEN CHR=0
80 COL=COL+1: IF COL>15 THEN COL=0
90 NEXT Y
100 NEXT X
```

Here we're clearing all variables to start, setting up a pair of FOR/NEXT loops, and using variable P for the common part of the formulae in line 40. Lines 50 and 60 POKE the character and colour memory locations respectively with the current value of CH and COL – zero the first time round thanks to the opening CLR, incremented by one in lines 70 and 80. The resetting to zero happens when CH (the character code) hits 255 and the colour code COL tops this.

The blanks you appear to get when you RUN this are there because the character is printed is the same colour as the background.

Bit masking

Locations like 1024-2023 and 55296-56295 are easy enough: you can just POKE into them a new decimal value that corresponds to some pre-defined set of options – character codes in the one case, colour codes in the other.

But most of the critical locations in the 64, those whose contents you might want to alter, operate in *binary*; it's the bits themselves that are important, and to change something you will be altering specific bits. For instance, location 53269 tells the 64 which of eight sprites are active – as follows:

```
bit 0    ...   sprite no. 0
bit 1    ...   sprite no. 1
bit 2    ...   sprite no. 2
bit 3    ...   sprite no. 3
bit 4    ...   sprite no. 4
bit 5    ...   sprite no. 5
bit 6    ...   sprite no. 6
bit 7    ...   sprite no. 7
```

The bits in an eight-bit byte or 'word' are numbered 0 to 7, with 0 being the lowest and 7 the highest. When the relevant bit is set to 0 or 'off' it means that sprite is inactive; when it's 'on' (set to 1) it means the sprite is to be used. So if sprites 0, 2 and 3 are required, location 53269 would look like this:

bit number	7	6	5	4	3	2	1	0
decimal	128	64	32	16	8	4	2	1

Using the 'decimal equivalents' line you can see that the values required to set those three sprites are ...

bit number	7	6	5	4	3	2	1	0
decimal	0	0	0	0	1	1	0	1

Which translates into this POKE statement ...

```
POKE 53269, 13
```

... because it's 1 + 4 + 8.

Decimal numbers with POKEs can be quite easy to use like this. On the other hand, frequently you will have to alter some bits in a location *without affecting others* – particularly when you can't be sure what exactly the current contents of the location are.

For this you're provided with the AND and OR statements. They let you compare the contents of a memory location with a number that is called a 'mask': the bit positions in the location and the mask are compared, and what happens next depends on the result of the comparison.

location 56369	0	0	0	0	0	0	0	0
mask value 13	0	0	0	0	1	1	0	1
result	0	0	0	0	0	0	0	0

So AND lets you check out the contents of a location without altering them at all. OR does the same kind of thing – except that the comparison is regarded as 'true' if there is a 1 at any bit position in the location *or* in the mask:

On the other hand, try this:

```
POKE 56369, 13
PRINT PEEK (56369) AND 13
```

Now you'll get the response '13' – meaning that a match had been found in the bit positions that tot up to decimal 13:

location 56369	0	0	0	0	1	1	0	1
mask value 13	0	0	0	0	1	1	0	1
result	0	0	0	0	1	1	0	1

And if your commands had been ...

```
POKE 56369, 20
PRINT PEEK (56369) AND 13
```

... you'd get '4', because a match had been found in one of the bit positions – bit 3 is decimal 4.

location 56369	0	0	0	1	0	1	0	0
mask value 13	0	0	0	0	1	1	0	1
result	0	0	0	0	0	1	0	0

For AND it works like this: if there is a 1 in the location at any bit position where the mask has a 1, then the comparison is regarded as 'true'. So, taking location 56369 again, you could try this:

```
PRINT PEEK (56369) AND 13
```

If you've POKEd nothing into 56369 yet, you'll get the response '0' – meaning that no match had been found:

```
POKE 56369, 20
PRINT PEEK (56369) OR 13
```

This time you get '29', because a match had been found in four bit positions – bits 0, 2, 3 and 4. Summing them gives you decimal 29:

location 56369	0	0	0	1	0	1	0	0
mask value 13	0	0	0	0	1	1	0	1
result	0	0	0	1	1	1	0	1

You can now use this information to alter selected bits, by combining a POKE and a PEEK. Say you've got some bits set to activate sprites: and now you want to turn on two more – sprites 5 and 7. If you simply POKE 56369, 160 that will indeed turn on those two (sprite 5 is set by bit number 5, which has the decimal value 32: sprite 7 is on when bit 7 is set – decimal value 128: and 128 + 32 = 160). But it will also turn *off* any other sprites, since decimal 160 means all other bits in location 53269 are 0.

Now, if you were sure that sprites 0, 2 and 3 were on you could POKE 53269 with 173 – that's 160 (for sprites 5 and 7) plus 13 (for sprites 0, 2 and 3). But it's neater, cleaner and safer to use a mask:

```
POKE 53269, PEEK (53269) OR 160
```

The PEEK/OR combination will find a 1 in bits 0, 2, and 3 (because that's what is in location 53269 already) and bits 5 and 7 (because they're in the mask). This command POKEs that result into 53269, giving a new value that has all those bits set – and if you PRINT PEEK (53269) you will see that it now has the value 173.

POKEing into ROM

If you do want to see what's in the ROM areas of the 64 and then change the machine code (to fix bugs, say) you *can* do so – assuming you're competent!

The things you might want to look at and modify are Basic itself and the Kernal. The procedure is simply to copy the ROM locations into the RAM that can occupy the same addresses: if you POKE to what starts off as a ROM location, the 64 will automatically put the data into the RAM at those locations. On the other hand, if you PEEK at one of those addresses the 64 will reflect the value in the ROM rather than the RAM.

Fortunately the 64 comes with the very handy location number 1. Among other things (specifically control of cassette deck) this manages which of the alternative memory areas are being used – bit 0 for

the Basic ROM, bit 1 for the Kernal ROM. By setting the correct bit to zero you can switch out the ROM and the alternative RAM at those addresses will become available.

So you have to alter address number 1 to indicate that you want to use the RAM. The problem is that any attempt to do this from Basic is liable to crash the machine, so you'll almost certainly have to do it from machine code.

For **Basic** you need locations 40960 to 49151:

```
FOR B= 40960 TO 49151: POKE B, PEEK (B):
NEXT
```

POKEing 1 with value 54 turns on the RAM area for Basic. To reset to normal, POKE 1 with 55.

For the **Kernal** you need locations 57344 to 65535:

```
FOR B= 57344 TO 65535: POKE B, PEEK (B):
NEXT
```

POKEing 1 with value 53 turns on the RAM area for the Kernal, or you can use value 52 for the Kernal *and* Basic. To reset to normal, POKE 1 with 55 again.

The WAIT command: pausing the program

And this one must be the most tricky Basic command in the 64. At its simplest, WAIT causes the program to pause until the contents of a specified memory location reach a given value. It takes the form ...

WAIT *location, mask for AND, optional mask for OR*

The computer PEEKs at the location and does an AND comparison with the first mask; if the second one is present it ORs the result with that too.

A statement like ...

```
WAIT 653, 4
```

... will compare the contents of location 653 with the binary mask 00000100 and allow the program to proceed only when there's a 1 in the same position in 653. Bit 2 of location 653 changes to 1 when the CTRL key is pressed; so this statement causes the program to WAIT until CTRL is keyed.

The optional second mask complicates things if it's used. Deep breath: the program will WAIT until the location has a 0 at one or more of the bit positions where *both* masks have a 1, or until the location has a 1 at one or more of the bit positions where the first mask has a 1 and the second has a 0.

Since the contents of locations don't alter in an arbitrary way, WAIT is most often used in input/output operations (including those involving the keyboard) where external events affect them. Typing something is just such an 'external event'; so is using the cassette unit –

```
WAIT 1, 16, 16
```

This WAITs until a button is pressed on the cassette deck (indicated in bit 4 of location 1). It's useful if you have a program that reads in some data from tape midway through its execution; if you include this just before an OPEN statement that gets at the data file, you won't have the 'PRESS PLAY ON TAPE' message messing up any nice display you're currently running.

An alternative is bit 5 of location 1, which tests whether the cassette motor is running:

```
WAIT 1, 32, 32
```

Using machine-code subroutines with USR

Where machine code would be a fitting solution to a particular problem, you don't necessarily have to write an entire machine-code program to handle it: you may be able to call from Basic to a subroutine in machine code. The USR command lets you do that, and it can also let you go to the subroutine with a particular value for it to work on.

The procedure for using USR in a program is as follows:

- set up your subroutine, generally by READing DATA and POKEing it into specific memory locations. The best option is probably the patch of memory between 49152 and 53247, since they aren't used by the 64 and they are not affected by Basic
- make sure that the routine terminates with a machine-code RTS statement (decimal 4). This is equivalent to Basic's RETURN and will get you back into your program after the machine code has been executed
- once the memory locations have been POKEd, set up locations 785 and 786 – these show the starting address for the next USR the program will come across, and the best way to set them is with two formulae statements as follows:

  POKE 785, *location* **- 256 * INT(***location***/256)**
  POKE 786, (*location***/256)**

- then have one or more USR statements in the form **USR (***parameter***)**, where the parameter is a numeric constant or variable, or a compound expression that evaluates to a number.

USR is a function rather than a command, so it must be used with another Basic statement – as in

```
100 PRINT USR(A%)
```

Once you've set up locations 785 and 786 with its start address, you can call the USR routine as often as you like. Those pointers won't be affected by commands like RUN, CLR, NEW and LOAD which reset variables and wipe out any existing DEF FN statements; they'll hold the start location of the routine until you POKE a different one into them or until you switch off.

The preferable alternative: SYS calls

The SYS command does much the same thing as USR, and in general it's the better option – it's a command, for a start, which simplifies things. And you can set up a machine-code routine that uses any number of variable parameters (rather than one) and text as an alternative to numbers (USR's parameters must be numeric). Above all, you don't have to specify the start location somewhere else: it's included as part of the statement.

The procedure is:

- set up your subroutine, generally by READing DATA and POKEing it into specific memory locations – again, go for memory between 49152 and 53247 and make sure that the routine terminates with a machine-code RTS statement

- once the memory locations have been POKEd, you can if required set up some locations holding parameters and variables for the routine to work with. (You don't need to set any pointers equivalent to USR's use of 785 and 786)

- then have one or more SYS commands in the form **SYS** *start location of routine.*

You can skip the first of these steps if you want to initiate a subroutine already provided in the 64 – though there's only one common application of that:

```
SYS 64738
```

The location is the beginning address for the 64's start-up routine, so a SYS to it will persuade the thing to think that the power has just been turned off and then on again. It clears memory, kills all variables, and restarts with the 'BASIC 2.0' and 'BYTES FREE' messages.

As with USR, any machine code you do write is unaffected by commands like RUN, CLR, NEW and LOAD which reset variables and wipe out any existing DEF FN statements; they'll hold the start location of the routine until you POKE a different one into them or until you switch off.

Saving and loading blocks of memory

It's often handy to be able to save chunks of memory on disk and tape and subsequently load it back again – complete ready-to-go graphics displays or text-entry forms, say, or blocks of other data. If you've got some machine code in your Basic program you'll also want to be able to save and load it.

You can do this with a few SYS calls to existing Kernal routines. The procedure isn't too complicated – but there are a couple of points to watch for. First, memory locations are hexadecimal addresses taking two bytes; and the disk/tape operations will save and load only single bytes. So you'll have to convert them:

```
10 INPUT "START ADDRESS"; SA
20 INPUT "END ADDRESS"; EA
30 EA = EA + 1
40 SHI = INT (SA/256)
50 SLO = SA-SHI*256
60 EHI = INT (EA/256)
70 ELO = EA-EHI*256
```

SA is the variable that holds the start address of your memory to be saved, EA is the end address. The end must be one byte more than the last byte to be saved, so line 30 does that. SHI and EHI are the high-byte parts of the address, SLO and ELO the low-bytes.

So you've now got two values for the start and two for the end.

```
100 SYS 57812 "filename", device
number, optional secondary address

160 SYS 65496
```

And that actually starts the SAVE going.

To reload the SAVEd memory, don't bother with LOAD; that requires some juggling if the loaded memory locations aren't to overwrite the program you currently have in memory. Instead you can by-pass the LOAD command with a SYS. If you saved the memory block as 'filename',1,1 it will go back into the locations it came from ...

```
1000 SYS 57812
1010 POKE 780, 0
1020 SYS 65493
```

Otherwise you'll have to specify a start and end destination for it and put those values into 781 and 782:

```
1000 SYS 57812
1010 POKE 781, DLO
1020 POKE 782, DHI
1030 POKE 780, 0
1040 SYS 65493
```

This does everything that a conventional SAVE command will do – except that it doesn't start the SAVE operation. In other words, it reads the filename, device number, and the secondary address if used (leave it out altogether for disk. There are a couple of restrictions with tape saves – in particular, you can't save locations higher than address 32767. If you want to do that, copy those locations into RAM at a lower point first. A

secondary address of 1 when the device number specifies cassette (1) will force the saved memory to be reloaded subsequently in the place it came from – which can be higher than 32767.)

```
110 POKE 780,2
120 POKE 2, SLO
130 POKE 3, SHI
140 POKE 781, ELO
150 POKE 782, EHI
```

Locations 780 to 782 provide storage for the registers that handle SAVEs, indicating the start and end of the operation. Then you just need one more SYS:

CHAPTER 14
Graphics

The 64 has a pretty flexible way with graphics. You can settle for the standard set of characters available at the keyboard; you can design replacement symbols for your own programs; you can create dot-by-dot pictures; you can have moving sprites.

The high-resolution display

The 64 has two alternative modes of displaying something – low- and high-resolution. 'Resolution' in this case means the number of individual points on the screen where you can place something.

Low resolution is what you get when you switch on; there's a maximum of 1,000 places on the screen where you can put a character or a graphic symbol. The characters and graphics you can use are those provided on the keyboard (in both of its incarnations – upper-case-only plus lots of graphics and the alternative upper/lower-case plus a few graphics) and with the CHR$ function.

In high resolution, though, you can work with the individual dots that make up the character shapes in low-res mode. Since each low-res character is created on a mini-grid eight dots high and eight dots wide, this gives you a display area of 320 x 200 points to play with – eight dots per character in the horizontal plane and 40 characters per line provides a width of 320 dots: eight dots vertically by 25 rows gives 200 dots.

You will probably use a high-res screen for one of three purposes:

- **designing special characters and symbols** to mix into the normal collection of characters, such that subsequently they can be incorporated into (low-res) displays of text and the like
- **plotting trigonometrical shapes and graphs** where a figure can be drawn on to the screen
- **creating full-scale pictures** that will probably be used as the background for action involving sprites.

User-defined characters

You can create your own shapes and use them as easily as any of the pre-defined characters – meaning you can put them on to the screen by typing at the keyboard, and you can use them in programs for displays. They might be fancier lettering, or scientific symbols, or foreign alphabets: they'll more likely be special shapes like space invaders or racing cars for games.

What you have to do is get into the memory locations where the 64 stores its characters. You'll be replacing some of what's in there already with your own shapes. Now, on the screen each character is built up in an 8x8 grid; and you get the shape because some of the squares are empty and others are filled.

The 64 *stores* the shape of the character in its memory more or less in the same way. Each character gets a block of eight consecutive memory locations, one for each row of the character's shape: and since there are eight bits per location, one bit can correspond to each position along the row of the shape.

The '@' sign happens to be the first character in the 64's upper-case character set. Which means it occupies the first eight locations in the character set memory. In those locations you get a binary patterns corresponding to the presence of filled and empty squares:

bit no	*binary pattern*	*decimal equivalent (PEEK)*
0	. . o o o o . .	60
1	. o o . . o o .	128
2	. o o . o o o .	110
3	. o o . o o o .	110
4	. o o	96
5	. o o . . . o .	98
6	. . o o o o . .	60
7	. . . .	0

From that you might suppose the easiest way to put your own custom-designed characters on to the screen is simply to place different values into one or more of the locations where the 64's standard characters normally reside: junk those characters you won't want to use a lot, replace the relevant bytes with a new POKEd decimal value corresponding to a new bit-pattern shape, don't alter any characters you want to retain.

If you were to do that for the first eight locations, you should be able to type an '@' or include the '@' symbol in programs – and what you'd actually get would be the newly defined character. Sounds easy?

Well, there's a couple of problems you have to get around first. For starters the character set definitions are in read-only memory; so while you can read it by PEEKing, you can't alter it with POKEs.

The solution is to take a copy of the character set locations from ROM (which you can do by PEEKing) and drop it into RAM somewhere with a few POKEs. Once it's there you can work on it.

But surely the 64 goes to the original ROM locations when it decides it wants to know how to display a particular character? Not quite: it doesn't go *directly* to the memory containing characters when it has something to print. Instead it calls on location 53272 first.

Among other things, location 53272 decides where the computer is going to look in its memory for the characters it'll put on to the screen. So you can alter 53272 to indicate a different area of memory in which to find them – like your amended copy in RAM somewhere.

And where will that be? Well, you've got a choice; but for a variety of reasons (like you don't want to mess up anything else in the computer) the best bets are 8192, 12288 or 14436 as the start locations for a new character set.

The 64 can access and use a total of 256 characters, which means that you need enough memory to cope with 256 x 8 character-defining bytes. So a full character set takes up 2,048 locations. That's a lot of characters, and it can cause a problem: the 64 will automatically load a program into spare memory – and it will carry on taking up memory until it hits the upper limit. This means your program could well overwrite your character set (or *vice versa*, depending which grabs the memory last).

At switch-on, the 64 automatically makes location 40960 the highest point in memory that a program and its associated workspace can reach. The lowest point, the start position for the statements in a Basic program, is 2049.

You change either (or both) of these to protect your special characters. Locations 43 and 44 decide where the 'bottom' of Basic memory is; locations 52 and 56 indicate the 'top' of Basic.

If you're putting characters at 8192 onwards, you'll have to relocate the start of Basic beyond 8192 + 2048 – meaning a start of 10240 and the loss of 8K's worth of Basic RAM. Similarly you'll need to set a higher start point for Basic if your characters are going to be at 12288 or 14436 onwards (though in the latter case you might choose to leave the start address of Basic alone and change the end location).

And you still haven't finished yet. Before you can take a copy of the contents of ROM, you have to turn off the 64's input-output control at locations 1 and 56334.

Sounds complicated? Well, if you apply yourself systematically to the process it won't be too hard. Here's a summary of the steps involved in programming for user-defined graphics:

1. Decide which characters in the standard set you want to retain and which you want to replace

2. Arm yourself with a pencil, a rubber, and some squared graph paper (or one of the purpose-built character designer pads now on sale). Design your new shape in an 8x8 grid, and for each row note the decimal equivalent of the binary pattern you've created

3. Start programming. First switch off locations 56334 and 1 with a couple of POKEs, as follows:

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

4. Now take a copy of the standard character set stored at 56334 on (I've decided to put the copy at 14336 onwards):

```
30 FOR CH=0 TO 2047
40 POKE CH+14336, PEEK(53248+CH)
50 NEXT CH
```

That transfers 2,048 bits containing 256 characters.

5. Switch locations 56334 and 1 back on again such that the ROM area is no longer being used:

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

This returns the 64 to its original state – using the standard character set, but with a duplicate sitting in 14336 on.

6. Now you can modify the character by putting new data into the relevant locations. For the purposes of this demo I'm only redefining two characters – the '@' symbol and the equals sign. I've decided I don't particularly want these in my new character set; instead I'll go for a lower-case 'e' with a French *grave* accent (to prove that there's a serious use for redefined characters) and a less than frightening space invader type (definitely trivial, but quite fun).

For them I need the DATA I've produced by my graph-paper calculations. Here's the French letter:

bit no	*binary pattern*	*decimal equivalent*
0	. o	64
1	. . o	32
2	. . o o o . . .	56
3	. o . . . o . .	68
4	. o o o o o . .	124
5	. o	64
6	. . o o o o . .	60
7	. . . .	0

And using that as DATA gives ...

```
1000 DATA 64,32,56,68,124,64,60,0
```

This is the shape for the alien:

bit no	*binary pattern*	*decimal equivalent*
0	o . o o o o . 0	189
1	o . o . . o . o	165
2	o . o o o o . o	189
3	o o o o o o o o	255
4	. . o o o o . .	60
5	. o o o o o o .	126
6	. o . o o . o .	90
7	. o . . . . o .	66

And the DATA for that:

```
1010 DATA 189,165,189,255,60,126,
90,66
```

7. We can now READ that DATA:

```
100 FOR M1=14336 TO 14336+7
110 READ D1: POKE M1,D1
120 NEXT M1
```

The DATA in line 1000, which contains the eight rows that make up the accented 'e' shape, is read one item at a time into a variable D1 and POKEd into successive locations from 14336 on. Now for a second collection of POKEs to handle the replacement for the up-arrow character; reference to the screen POKEs chart in the Appendix will tell you this character is number 30 in the set, so the next bit starts with a line that identifies it ...

```
200 L=14336+(30*8)
210 FOR M2=L TO L+7
220 READ D2: POKE M2,D2
230 NEXT M2
```

8. We need a line to reset 53272 to point to a new character set starting at 14336, and this one accomplishes that:

```
300 POKE 53272,(PEEK(53272) AND 240)
OR 14
```

To show how easy it is to incorporate customised characters into programs, try tagging a PRINT line on to the program:

```
3000
PRINT"<clr>@@@@@@@@@@@@@@@@"
```

The high-resolution screen

If you want the whole screen to play with on a dot-by-dot basis, many of the same considerations apply. You do need a lot more memory for yourself, though – a display 320 dots wide and 200 high means a total of 64,000; and since each byte can handle eight of them, you require an 8,000 byte lump of RAM.

And you get no choice at all about where to locate this. High-res drawing uses the VIC chip, and the VIC chip can't cope with addresses above location 16384 in a standard memory configuration (you could kill Basic and let the VIC chip expand its horizons, but that's tricky).

This means there are only two places to put the high-res display – starting at locations 0 and 8192. And because the low memory addresses are used by the 64, you're left with 8192 to 16191.

The steps involved are as follows:

1. Move the bottom on Basic past location 16193. These lines will leave it at 16384 (you can only move in 256-byte blocks, so this is the nearest you can get to 16193):

```
10 POKE 642,64: POKE 44,64: POKE 46,64:
POKE 48,64: POKE 50,64
```

2. Now locate the high-res screen at 8192:

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

3. And tell the 64 to switch to high-resolution mode by setting bit 5 or location 53265:

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

4. At this point the screen will be full of miscellaneous rubbish that was lying around in those memory locations, so you need to clear it:

```
40 FOR T=8192 TO (8192+7999)
50 POKE T,0
60 NEXT
```

5. You'd think the screen is now addressable in terms of conventional x,y co-ordinates – with the top left corner being x=0, y=0 and bottom right x=199, y=319. But no; the 64 actually does it by stacking blocks of eight bytes next to each other.

So byte 0 references 0,0 to 0,7: and byte 1 does 1,0 to 1,7. Byte 7 then handles 7,0 to 7,7. And byte 8 starts a new stack next to the first one, so it references 0,8 to 0,15. Byte 15 at the foot of that second pile does 7,8 to 7,15 ... and then there's another stack of eight bytes, up to bytes 312 to 319 in the top right corner (co-ordinates 0,312 to 7,319). Then another row of eight-byte stacks starts at 8,0.

And so on. Translating this into useable screen co-ordinates is fortunately easy enough using this formula, ideally in a subroutine:

```
1000 BYTE=8192+INT(Y/8)*320+INT
(X/8)*8+(Y AND 7)
1010 POKE BYTE,PEEK(BYTE) OR 2↑(7-(X
AND 7))
```

Sprites

Sprites are one of the features that makes the 64 rather special among home computers. Sometimes called 'Moveable Object Blocks' or MOBs in Commodore bumf, sprites are graphic mini-pictures that can be defined separately from other graphics characters and moved around the screen with special POKEs from a Basic programs.

Sprites can collide with each other; and they can pass in front of, through, or behind each other. They can also bump into or pass in front of, through, or behind anything else you've put on to the screen.

So the normal use for sprites is in an action game. For the background you'd define a shape using custom graphics; for anything that moves on the screen you'd use sprites.

You can have up to eight sprites ready to use, though there are restrictions on how many different sprite *shapes* you can actually have on the screen at once – normally (which means without any fancy programming) you're limited to three. You can still have eight sprites on the screen, but not eight *different* sprites.

Designing a sprite itself involves these stages:

- draw the sprite and find the decimal equivalents, much as for a custom-designed special character
- store that data and tell the 64 where it can find the stored definition for a particular sprite
- define the sprite's colour

There are two more steps to **using** a sprite:

- indicate where the sprite should be placed on the display
- switch the sprite on so that it *is* displayed.

Designing sprites

Sprites are created from 'pixels' (on/off dot positions) in a grid of 21 rows with 24 pixels per row. Since each pixel is a single bit, that means one sprite takes up 63 bytes of memory (21x24=504 and 504/8=63).

The drawing stage again needs graph paper or a special sprite-designing pad; either way you want a 21x24 grid on which to work, with the 24 columns divided into three groups of eight.

This should give you a pattern of possible dot positions like this:

```
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
........ ........ ........
```

You can now start filling in the points that you want to appear as solid parts of the picture. Here's a Phoenix (sort of):

```
ooo..... ........ ..... ooo
.oooo... ..ooooo. ...oooo.
..ooooo. oo..ooo. .ooooo..
.oooooo. ooooooo. .oooooo.
..ooooo. o.ooooo. .ooooo..
.oooooo. ..oooo.. .oooooo.
.ooooooo ..oooo.. ooooooo.
....oooo .oooooo. oooo....
....oooo ooooo..o oooo....
....oo.o oooooooo o.oo....
...ooo.o oooooooo o.ooo...
..oooo.o oooooooo o.oooo..
.ooooo.. oooooooo ..ooooo.
..oooo.. oooooooo ..oooo..
..ooo... oooooooo ...ooo..
..oo.... .oooooo. ....oo..
..o..... .oooooo. .....o..
........ oooooooo ........
.......o ooo..ooo o.......
.....ooo ooo..oo. ooo.....
....oo.o oo....oo o.oo....
```

This translates into the following binary pattern:

```
11100000 00000000 00000111
01111000 00111110 00011110
00111110 11001110 01111100
01111110 11111110 01111110
00111110 10111110 01111100
01111110 00111100 01111110
01111111 00111100 11111110
00001111 01111110 11110000
00001111 11111001 11110000
00001101 11111111 10110000
00011101 11111111 10111000
00111101 11111111 10111100
01111100 11111111 00111110
00111100 11111111 00111100
00111000 11111111 00011100
00110000 01111110 00001100
00100000 01111110 00000100
00000000 11111111 00000000
00000001 11100111 10000000
00000111 11100110 11100000
00001101 11000011 10110000
```

And that in turn provides this conversion into decimal of each eight-bit group:

```
11100000 = 224;  00000000 =   0;  00000111 =   7
01111000 = 120;  00111110 =  62;  00011110 =  30
00111110 =  62;  11001110 = 206;  01111100 = 124
01111110 = 126;  11111110 = 254;  01111110 = 126
00111110 =  62;  10111110 = 190;  01111100 = 124
01111110 = 126;  00111100 =  60;  01111110 = 126
01111111 = 127;  00111100 =  60;  11111110 = 254
00001111 =  15;  01111110 = 126;  11110000 = 240
00001111 =  15;  11111001 = 249;  11110000 = 240
00001101 =  13;  11111111 = 255;  10110000 = 176
00011101 =  29;  11111111 = 255;  10111000 = 184
00111101 =  61;  11111111 = 255;  10111100 = 188
01111100 = 124;  11111111 = 255;  00111110 =  62
00111100 =  60;  11111111 = 255;  00111100 =  60
00111000 =  56;  11111111 = 255;  00011100 =  28
00110000 =  48;  01111110 = 126;  00001100 =  12
00100000 =  64;  01111110 = 126;  00000100 =   4
00000000 =   0;  11111111 = 255;  00000000 =   0
00000001 =   1;  11100111 = 231;  10000000 = 128
00000111 =   7;  11100110 = 230;  11100000 = 224
00001101 =  13;  11000011 = 195;  10110000 = 176
```

Or, to put that another way:

```
row 1  ...   224; 0; 7
row 2  ...   120; 62; 30
row 3  ...   62; 206; 124
row 4  ...   126; 254; 126
row 5  ...   62; 190; 124
row 6  ...   126; 60; 126
row 7  ...   127; 60; 254
row 8  ...   15; 126; 240
row 9  ...   15; 249; 240
row 10 ...   13; 255; 176
row 11 ...   29; 255; 184
row 12 ...   61; 255; 188
row 13 ...   124; 255; 62
row 14 ...   60; 255; 60
row 15 ...   56; 255; 28
row 16 ...   48; 126; 12
row 17 ...   64; 126; 4
row 18 ...   0; 255; 0
row 19 ...   1; 231; 128
row 20 ...   7; 230; 224
row 21 ...   13; 195; 176
```

Storing the sprite data

Now you've got a binary picture for a sprite, you need somewhere in RAM to store it. You need 63 bytes per sprite definition, but in fact you have to reserve 64 owing to the computer's preference for multiples of eight – the last location in the sprite data block is never used.

Clearly you should store the sprite data away from the character set, any background graphics you've defined, and your program itself. One good spot starts at location 12,800; for a variety of reasons another good option if you have only a few sprites to define is the cassette buffer, that area of memory normally used to store stuff coming from or going to the cassette unit. (You can't then use it for cassette I/O without losing the sprites, but presumably once your program is running you wouldn't need to use the cassette.)

The cassette buffer occupies memory locations 828 to 1023, so there's room there for three sprite definitions – one in the 64 bytes starting at 828, another at 896, the third at 960.

Here's how the Phoenix might be stored there. First let's get in the DATA for it:

```
1001 DATA 224; 0; 7
1002 DATA 120; 62; 30
1003 DATA 62; 206; 124
1004 DATA 126; 254; 126
1005 DATA 62; 190; 124
1006 DATA 126; 60; 126
1007 DATA 127; 60; 254
1008 DATA 15; 126; 240
1009 DATA 15; 249; 240
1010 DATA 13; 255; 176
1011 DATA 29; 255; 184
1012 DATA 61; 255; 188
1013 DATA 124; 255; 62
1014 DATA 60; 255; 60
1015 DATA 56; 255; 28
1016 DATA 48; 126; 12
1017 DATA 64; 126; 4
1018 DATA 0; 255; 0
1019 DATA 1; 231; 128
1020 DATA 7; 230; 224
1021 DATA 13; 195; 176
```

(With a lot of DATA like this I generally try to make the DATA lines relate to their function – 21 lines of three groups of numbers are easier to check at a glance if the program lines echo their position.)

Now for some statements to READ the DATA:

```
20 FOR SD=0 TO 62: READ SP
30 POKE 828+SD, SP
40 NEXT SP
```

That provides for one sprite: how about putting three into the cassette buffer? You could do that with some more DATA lines (or perhaps a reREAD of the Phoenix DATA, if you want three sprites the same, though as we'll see that's rather pointless) and a slightly different READ section:

```
10 FOR SV=1 TO 3
20 FOR SP=0 TO 62: READ SD
30 POKE 828*SV+SP, SD
40 NEXT SP
50 NEXT SV
```

Here I've got SV as the sprite location variable, SD the sprite data, and SP the individual sprite bytes.

Activating and colouring sprites

Before you can use a particular sprite you have to set a pointer to indicate the one you want. This you do by POKEing up to eight locations (one for each of up to eight possible sprites) with one of the sprite block numbers.

The sprite pointers start at location 2040 for sprite 0 and run to 2047 for sprite number 7. The target location indicators that you POKE into these are the

start location of the relevant sprite DATA divided by 64; so this would put a Phoenix at your program's disposal as sprite 0 ...

```
100 POKE 2040,(832/64)
```

Once it's there, the same DATA can be used for more than one sprite. This would give it eight:

```
100 FOR SN=0 TO 7
110 POKE 2040+SN,(832/64)
120 NEXT SN
```

At this point you can decide a colour for the sprite. If you don't do this, the 64 will assume default settings as follows:

```
sprite 0    ...   white
sprite 1    ...   red
sprite 2    ...   cyan
sprite 3    ...   purple
sprite 4    ...   green
sprite 5    ...   blue
sprite 6    ...   yellow
sprite 7    ...   medium grey
```

Sprite colours are controlled by the eight locations running from 53287 on, so to select a colour for sprite 0 you POKE 53287 with one of the colour code numbers 0 to 15 – for instance:

```
130 POKE 53287,8
```

That gives you an orange Phoenix. For multiple sprites you have to assign the colours individually:

```
130 POKE 53287,9: REM sprite 0
140 POKE 53288,0: REM sprite 1
150 POKE 53289,3: REM sprite 2
160 POKE 53290,12: REM sprite 3
170 POKE 53291,5: REM sprite 4
180 POKE 53292,13: REM sprite 5
190 POKE 53293,7: REM sprite 6
200 POKE 53294,1: REM sprite 7
```

Displaying sprites

Sprites then have to be turned on before they can be seen (and subsequently off again when you want them to disappear). That is done by setting individual bits of location 53269 – bit 0 corresponds to sprite 0, bit 7 to sprite 7.

This can get a mite complicated if you have eight sprites turning on and off at different times. One solution is use this handy table:

sprite no.	*bit setting for ON*
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000
SUM	

You can then work out the binary number for the sprites you want at a particular time, convert that to decimal, and either POKE 53269 directly with the value or (preferably) use OR. Say you want to turn on sprites 1, 5 and 7:

sprite no	*bit setting for ON*
1	00000010
5	00100000
7	10000000
SUM	10100010 = decimal 162

The settings you can do by ...

```
POKE 53269,PEEK(53269) OR 162
```

A slightly easier method is simply to use the relevant powers of 2:

POKE 53269, PEEK (53269) OR 2 ↑ *sprite number*

You can then string together several of those to enable more than one sprite:

```
POKE 53269,PEEK(53269) OR 2↑1+2↑5+
2↑7
```

The AND comparison effectively disables sprites in the same way. To enable *all* sprites you can of course POKE 53269, 255: and POKE 56369, 0 disables them all (you can't clear sprites from the screen by pressing the CLR key – you have to POKE 53269 if they aren't to be displayed).

Even if you have enabled a sprite you might not be able to see it. It could be the same colour as the background; or it could be positioned off the screen.

It isn't possible to enlarge a sprite by adding more DATA, of course. But you can stretch it horizontally or vertically – or both. Once again this is done by setting a single bit corresponding to each sprite in particular locations. For the horizontal stretch, it's location 53277: for vertical expansion you use 53271.

sprite no.	*bit setting for expansion*
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000
SUM	

POKE the appropriate location with the decimal equivalent of the binary sum you end up with, or use OR:

```
400 POKE 53277,PEEK(53277) OR 16
```

That creates a wide fat version of sprite 4. This would double its length too:

```
400 POKE 53277, PEEK (53277) OR 16
410 POKE 53271, PEEK (53271) OR 16
```

Placing sprites

You can place sprites on a grid that is 255 vertical co-ordinates by 344 horizontally; this is larger than the screen, so that sprites can appear to move smoothly on to the visible area – the screen is a kind of 'window' placed on top of the sprite area, covering its centre section from positions 50 to 299 vertically and 24 to 320 horizontally.

The position of a sprite is determined by where you put its top lefthand corner. So to have a sprite appear in full, that corner must be located between co-ordinates 50 and 299 vertically and 24 to 320 in the horizontal plane; otherwise you'll see only part of it.

The sprite's position is defined by three registers per sprite. The vertical position gets one location, the horizontal needs two. Why? Because the largest number one byte-sized location can hold is 255, and that's the maximum number for the vertical: so you need only one location for it. But the largest horizontal position is 344, and you need *nine* bits for that; so the 64 gives you one whole location (for the positions 0 to 255) plus one bit of another location, 53264, for anything larger.

Any or all sprites can also overlap anything else on the screen. If you LIST the program with the sprites still there, you'll see that all the sprites appear on top of the listing; the default is that sprites get priority over everything else on the screen.

You can change that by altering location 53275. Again each bit relates consecutively to each sprite – bit 0 to sprite 0, bit 7 to sprite 7. If a bit is set to 1, the corresponding sprite loses its priority and appears *behind* whatever is on the screen. Once again you might need the helpful table:

sprite no.	*bit setting for behind display*
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

SUM

The 64 provides a mechanism for detecting when a sprite collides with another or with the background – whereupon it might bounce off, explode, disappear or simply pass in front or behind depending on your programming.

The two locations to check are 53278 for sprite-with-sprite bumps and 53279 for sprite-with-background collisions. Either way the format is the same: if the contents of the location are the same as the binary indicator for the sprite, a collision involving that sprite has occurred.

The AND operator is the best way to check this. For instance:

```
500 IF (PEEK (53278) AND 4) = 1 THEN
PRINT "SPRITE 3 COLLISION WITH SPRITE"
550 IF (PEEK (53279) AND 32) = 1 THEN
PRINT "SPRITE 5 COLLISION WITH
BACKGROUND"
```

The horizontal and vertical locations for each sprite go in pairs from 53248, as follows:

sprite no.	*horizontal position*	*vertical*
0	53248 plus bit 0 of 53249	53249
1	53250 plus bit 0 of 53251	53251
2	53252 plus bit 0 of 53253	53253
3	53254 plus bit 0 of 53255	53255
4	53256 plus bit 0 of 53257	53257
5	53258 plus bit 0 of 53259	53259
6	53260 plus bit 0 of 53261	53261
7	53262 plus bit 0 of 53263	53263

Let's try using this.

```
300 POKE 53269,255
310 POKE 53281,15
320 SH=53248: SV=53248
```

That switches on all eight sprites, colours the background light grey, and sets up some variables for the horizontal and variable positions of sprites. Now to distribute eight sprites around the screen:

```
330 POKE SH,17: POKE SV,55
340 POKE SH+2,19: POKE SV,53
350 POKE SH+4,100: POKE SV,100
360 POKE SH+6,109: POKE SV,153
370 POKE SH+8,160: POKE SV,180
380 POKE SH+10,190: POKE SV,20
390 POKE SH+12,210: POKE SV,40
400 POKE SH+14,245: POKE SV,253
```

Sprite priorities and collisions

When sprites overlap the one with the lowest priority gets obscured. Priority is determined by sprite number; sprite 0 gets the absolute priority.

CHAPTER 15
Sound

On your 64 you have some of the best sound-producing facilities available with any of the home computers currently on the market. The range of possibilities is genuine enough for some suppliers to have produced commendably good speech synthesisers using the 64's sound generators to imitate the human voice. And while the sound generation on the 64 is notoriously tedious to program, the results of doing so have produced at least one commercially-available package that demonstrates the 64's capabilities by emulating professional-grade purpose-built music synthesisers ...

That kind of thing does take a good deal of programming, and unless you're enthusiastic about the prospect you might do better to shell out for one of the (many) sound synthesis packages now being sold for the 64. But even if you set your sights lower you should be able to produce impressive music and sound effects.

There are two main problems with sound on the 64. First, the 64's Basic includes none of the helpful and meaningful commands with which you can create and play sounds on other computers. Instead you have to do it all with POKEs to particular memory locations. And if that wasn't bad enough, you can't use PEEK to check the current setting of those locations – they're in ROM, so trying that *always* returns the value 0.

The other fly in the ointment is the sheer capability of the 64's sound functions. In fact it helps to have a theoretical understanding of how sound works, since the 64 can synthesise most aspects of sound. That all means you get a *lot* of options to play with and a *lot* of different parameters to set: in other words, 64 sound is quite complicated.

Don't despair, though ...

The basics of sound on the 64

Beginners start here. Sound is vibration: what we hear is actually a collection of pressure variations in the air, a pulsating effect commonly called vibration. That broad definition will explain why the 64's sound generators work with the parameters that they do.

Sound on the 64 is produced by the SID chip. This can produce up to three notes at once; these operate independently – which means they have to be set up independently – and they are called **'voices'**.

For each voice you have to specify a **waveform**. This gives the sound a characteristic tone: it's actually made up of three elements – *frequency*, equivalent to the musical term 'pitch' and specifying the rate of change in the waveform's shape: *amplitude*, or 'loudness' (how far from the mid-point a waveform reaches): and *harmonic content* or 'timbre', which can be thought of as mini-waveforms set up by each frequency variation (timbre is what decides whether a sound sounds full and rich or bright and zingy).

The 64 comes with four waveform options, each of which produces a different style of sound. The *sawtooth* form is good for rich sounds like horns and pianos. *Triangle* waves are much reedier, like a flute. *Pulse* waves are a bit special, since for them you can alter the pulse width and so produce a quite extraordinary variety of sounds from clarinets to oboes via saxaphones and – depending on your skill at setting the waveform – the human voice.

The fourth waveform is described as *noise* and is used for untuned pitchless sounds like cymbals, explosions, surf and the like.

You get some **modulation** effects if you want them, too: these can be used to affect the tone and timbre of the sound produced by the waveforms.

You also have to specify an *envelope*. The sound envelope is a kind of description of what happens to the sound over time, and it comes in four phases – an opening **attack** which determines how quickly the sound reaches its maximum volume: a **decay**

phase during which it fades slightly until it reaches its **sustain** level: and a final **release** indicating how quickly it dies away to silence.

The initial letters of these phases – ADSR – are often used to describe a sound. Here are some examples of the ADSR envelopes for three specific sounds:

sound	attack	decay	sustain	release
handclap	fast	fast	nil	fast
drum	fast	medium	nil	medium
organ	fast	nil	long	fast

Two other key parameters are required – **volume** and **pitch**. The volume can't be specified for each sound individually, which is a bit of a shame. The pitch decides exactly what note is played – where in the scale, what octave (the SID chip has a total range of nine octaves, a couple more than a piano).

Procedures for programming sound

The steps involved in programming the 64 for sound go like this:

- you always have to set the **volume** before you'll be able to hear anything
- you must select which of the three **voices** you'll be using. Since all three of them can be in operation at once, you can play three-note chords ... but no more than three
- for each voice you have to select one of the four different **waveforms**. These determine the basic type of sound produced; and in practice most people think of them as approximating to different musical instruments
- for each voice you must set a **frequency range** (the pitch)
- for each voice you have to specify an envelope. On the 64 this involves two pairs of settings, one for **attack/decay** and the other for **sustain/release**
- you'll need a **delay loop** while the note plays at the sustain level for a specified amount of time, and you will also have to put delay loops into a series of notes if you want gaps between them
- if you want to use them you can set **modulation** effects, where two tone generators can be combined to produce one complex sound that alters the standard type of sound produced by the waveform selected
- there's also a **filter**, a kind of master tone control. This is one of the cleverest aspects of the 64's sound generation, and if you can get on top of it you'll find that this is really what makes the 64 so much more powerful than its competitors when it comes to producing sound

Make the 64 sing

In brief, any sound-producing program or routine needs to include statements that ...

- set the overall volume
- specify a waveform for each voice
- set the ADSR envelope for each voice
- delay the sustain phase for the required time
- set a frequency range for each voice
- play the note(s)
- kill the sound

SID sound in summary

These are the memory locations you can POKE to affect sound, shown here with the 'ready-reckoner' reference that I use in my sound programs to simply the POKEs:

voice	*location*	*ready-reckoner*	*controls*
1	54272	SID	low frequency
	54273	SID+1	high frequency
	54274	SID+2	low pulse width
	54275	SID+3	higher pulse width
	54276	SID+4	waveform
	54277	SID+5	attack/decay
	54278	SID+6	sustain/release
2	54279	SID+7	low frequency
	54280	SID+8	high frequency
	54281	SID+9	lower pulse width
	54282	SID+10	higher pulse width
	54283	SID+11	waveform
	54284	SID+12	attack/decay
	54285	SID+13	sustain/release
3	54286	SID+14	low frequency
	54287	SID+15	high frequency
	54288	SID+16	lower pulse width
	54289	SID+17	higher pulse width
	54290	SID+18	waveform
	54291	SID+19	attack/decay
	54292	SID+20	sustain/release
all	54296	SID+24	master volume

Possible values for SID POKEs

This is the range of values that you can use for these parameters by POKEing the appropriate locations:

- **Volume**: 0 is no sound at all, 15 is maximum volume

- **Low and high frequency of note**: you need both to specify a particular note to be played – see the table for all possible notes in the 64's range, but for each of the high and low frequency settings the possible range is the usual 0 to 255. (If you want to know the precise value of the frequency in Hertz, it's equal to the numbers in these locations multiplied by 0.0596.)

- **Low and high pulse width**: pulse widths are set only if the pulse waveform is specified. Possible values for the low pulse are 0 to 255 again, for high it's 0 to 15.

- **Waveform**: the theoretical range of possible values is 0 to 255. In practice it makes sense to stick with just four:

17 ... *triangle wave*
33 ... *sawtooth wave*
65 ... *pulse wave* (define your own by setting the pulse width)
129 ... *noise*

- **Attack/decay**: combined in one POKE according to the formula **attack x 16 + decay**. Both 'attack and 'decay' are in the range 0 to 15. A low attack value means that the sound quickly achieves its maximum volume; a low decay means it loses volume quickly. Here are the figures Commodore gives (timings in seconds):

value	*Attack rate*	*Decay rate*
0	0.002	0.006
1	0.008	0.024
2	0.016	0.048
3	0.024	0.072
4	0.038	0.114
5	0.056	0.168
6	0.068	0.204
7	0.080	0.240
8	0.1	0.300
9	0.25	0.750
10	0.5	1.5
11	0.8	2.4
12	1.0	3.0
13	3.0	9.0
14	5.0	15.0
15	8.0	24.0

- **Sustain/release**: again combined, and again the formula is **Sustain x 16 + Release**. Sustain interrupts the Decay to prolongs the note at its current volume, Release decides how long it's prolonged for before allowing it to decay again. Sustain can be any value from 0 (minimum) to 128 (a permanent sustain). The Release value can be 0 to 15, which means the Release happens after the time indicated here:

value	*Release rate*
0	0.006
1	0.024
2	0.048
3	0.072
4	0.114
5	0.114
6	0.168
7	0.204
8	0.3
9	0.75
10	1.5
11	2.4
12	3.0
13	9.0
14	15.0
15	24.0

So let's start programming. Since all the SID chip sound locations run quite neatly from 54272, one obvious simplification is to begin a program with an assignment ...

```
10 SID=54272
```

Or you might prefer to make life even easier by identifying the individual voices:

```
10 V1=54272: V2=54279: V3=54286:
VOL=54296
```

Actually there's a lazier way to do it:

```
10 V1=54272: V2=V1+7: V3=V2+7:
VOL=54296
```

Here I've also nominated VOL as the variable for the master volume control. Now you can POKE a setting into VOL to switch the sound on or off, and you can use the starting point of V1, V2 or V3 to get at the different parameters for each voice:

Voice plus **0** sets *low frequency of note*
Voice plus **1** sets *high frequency of note*
Voice plus **2** sets *lower pulse width*
Voice plus **3** sets *higher pulse width*
Voice plus **4** sets *waveform*
Voice plus **5** sets *attack/decay*
Voice plus **6** sets *sustain/release*

You can make life easier still for yourself by giving these names too:

```
20 LF=0: REM low frequency of note
30 HF=1: REM high frequency of note
40 LP=2: REM lower pulse width
50 HP=3: REM higher pulse width
60 WF=4: REM waveform
70 AD=5: REM attack/decay
80 SR=6: REM sustain/release
```

And before we get going we might as well make sure there's no garbage hanging around in the key locations that might adversely affect our musical ambitions:

```
90 FOR WIPE = V1 TO VOL: POKE WIPE, 0:
NEXT
```

From here on you'd better have an idea of what you want to do. This program acts as a taster, allowing you to try different parameters:

```
100 N = 0: REM set up voices
110 INPUT "WHICH VOICE"; V
120 IF V = 1 THEN V = V1
130 IF V = 2 THEN V = V2
140 IF V = 3 THEN V = V3
150 IF V > 3 OR V < 1 THEN PRINT "MUST BE 1,
2 OR 3": GOTO 110
160 N = N+1

200 REM set up note
210 INPUT "FREQUENCY RANGE FOR NOTE";
F1, F2
220 POKE V+LF, F1: POKE V+HF, F2

300 REM set Attack/Decay
310 INPUT "ATTACK-0 TO 15"; A
320 INPUT "DECAY-0 TO 15"; D
330 POKE V+AD, A*16+D

400 REM set Sustain/Release
410 INPUT "SUSTAIN-0 TO 15"; S
420 INPUT "RELEASE-0 TO 15"; R
430 POKE V+SR, S*16+R

500 REM set up waveform
510 INPUT "WHICH WAVEFORM-T, S, N, P";
W$
520 IF W$ = "P" THEN W = 65: GOTO 600
530 IF W$ = "T" THEN W = 17: REM triangle
540 IF W$ = "S" THEN W = 33: REM sawtooth
550 IF W$ = "N" THEN W = 129: REM noise
660 GOTO 700

600 REM set up Pulse waveform
610 INPUT "LOW PULSE WIDTH"; P1
620 INPUT "HIGH PULSE WIDTH"; P2
630 POKE V+LP, P1: POKE V+HP, P2

700 REM another voice?
710 WD(N) = W
720 IF N = 3 THEN PRINT "THREE VOICES
SPECIFIED": GOTO 1000
730 INPUT "ANOTHER VOICE-Y OR N"; A$
740 IF A$ = "Y" THEN 110

1000 REM play note
1010 INPUT "DURATION"; T
1020 INPUT "VOLUME"; Z
1030 POKE VOL, Z
1040 POKE V1, WD(1): POKE V2, WD(2):
POKE V3, WD(3)
1050 FOR M = 1 TO T: NEXT M
1060 POKE VOL, 0: POKE V, 0
```

If you RUN this it will make the noises specified for the length of time given in line 1010; use it to experiment with different effects.

Music

Well, music of a kind. The example following blows a Kentucky-Derby fanfare, and it illustrates how composing music is more complicated than merely making sounds because you need different durations and 'rests' as well. The easiest way to handle that is to READ a sequence of DATA statements:

```
10 V1=54272: V2=V1+7: V3=V2+7:
VOL=54296
20 LF=0: REM low frequency of note
30 HF=1: REM high frequency of note
40 LP=2: REM lower pulse width
50 HP=3: REM higher pulse width
60 WF=4: REM waveform
70 AD=5: REM attack/decay
80 SR=6: REM sustain/release
90 FOR WIPE = V1 TO VOL: POKE WIPE, 0:
NEXT

100 INPUT "NUMBER OF CHORUSES"; N
110 FOR C = 1 TO N

200 POKE V1 + AD, 85: POKE V1 + SR, 85
210 POKE V2 + AD, 85: POKE V2 + SR, 85
220 POKE VOL, 15
230 POKE V1 + WF, 33: POKE V2 + WF, 17

300 FOR R = 1 TO 6: REM six notes in the
repeated chorus
310 READ A1, A2, B1, B2
320 POKE V1 + HF, A1: POKE V1 + LF, A2
330 POKE V2 + HF, B1: POKE V2 + LF, B2
340 IF A1 = 50 THEN FOR D = 1 TO 150: NEXT:
REM pause
350 FOR E = 1 TO 100: NEXT: REM playing
speed

400 DATA 25, 30, 8, 29
410 DATA 33, 135, 25, 30
420 DATA 42, 62, 1, 5
430 DATA 50, 55, 55, 199
440 DATA 42, 62, 31, 65
450 DATA 31, 65, 50, 60

500 NEXT R
510 POKE V1 + WF, 33
520 POKE V2 + WF, 0
530 FOR W = 1 TO 500: NEXT
540 RESTORE
550 NEXT C
```

```
600 FOR WIPE = V1 TO VOL: POKE WIPE, 0:
NEXT
610 END
```

Another way to code music with READs is to organise the DATA in groups of three numbers – high and low frequencies to define the note plus a value for a timing loop to decide its duration. For instance:

```
2000 READ H: READ L: READ DUR
2010 POKE V+HF, H: POKE V+LF, L
2020 FOR T = 1 TO DUR: NEXT

3000 DATA 43, 52, 1000
```

Assuming you have a voice number, volume, waveform and ADSR specified somewhere, this should play an E (the 43, 52 in the DATA) for a thousand cycles.

And you could decide that 1000 cycles corresponds to a whole note – in which case 500 would be a half note, 250 a quarter, 375 a dotted quarter, and 125 an eighth.

CHAPTER 16
Extensions to Basic

It is possible to add commands to the 64: certain areas of memory are not available for use by your programs, having been set aside specifically for machine-code routines and programs – which could be the instructions that handle additional Basic commands.

A number of suppliers have stepped in to fill the (many) gaps between Commodore's Basic and the 64's capabilities – including Commodore itself. This Chapter looks at a number of extra-cost additions to a 64 that may help you make more of the computer ... or at least to make more of it more easily.

Most of the products of offer that add helpful extras to the standard Basic come in the form of disk or cartridge; they generally add a whole collection of different types of command. Some of the more limited ones – those that handle only sprite creation, in particular – are available (at a much lower price) on tape.

And what kind of additional commands can you look for? What follows is a summary of the commands on offer, divided by function.

Programmer's Aids

Under this heading you'll find a number of facilities that simplify the writing and debugging of Basic programs. These are often provided under the title 'Programmer's Aid' or 'Programmer's Toolkit'.

- **AUTO**: a new Direct-mode command that provides automatic line numbering while you're typing in a program.

You can specify the increment between lines, and the start number too:

```
AUTO 1000,10
```

When you press Return the number '1000' will appear on the screen waiting for you to type in some Basic statements for it. Complete the line, press Return, and the next number in sequence will appear – '1010', since you asked for an increment of 10.

- **RENUMBER**: renumbers your program, ideally altering all GOTOs, ONs, THENs, GOSUBs and so on.

You'll specify the new starting line number and the increment; you should also be able to renumber only a specified range of the original lines – either a block between two line numbers, or lines up to or from a given line number.

- **HELP**: when a program error occurs this command will print out the line where things failed, usually with the errant statement highlighted in reverse video.

It's generally a Direct-mode command, so you when you RUN a program and it crashes you can type 'HELP' to pick up the duff line; some Programmer's Aids let you enter 'HELP ON' right at the start of your programming so that *any* error subsequently encountered during a run will immediately be displayed. (Note that the statement at which the program failed doesn't necessarily mean that the error is actually *there*, though.)

- **DELETE**: deletes a given range of line numbers.

You can delete *within* a specified range, from the start of the program *up to* a specified line number, or *from* that line to the end.

- **delete forwards**: the INST/DEL key deletes the character immediately to the left of the cursor – this facility usually selects RUN/STOP as an alternative that deletes to the *right* of the cursor.

- **FIND**: searches a program for a text string specified.

You may be told the line number of the first occurrence or of all occurrences; or (more likely) the line(s) with the first or all occurrences will be printed on the screen for you to see.

- **REPLACE or CHANGE**: a Direct-mode command that finds the first or all occurrences of a specified character string in your program and automatically replaces it/them with a new string also given in the command.
- **SHRINK**: removes all unnecessary spaces and REMs from a program so that it takes less memory (the main benefit) and perhaps runs faster (less likely).
- **MERGE or APPEND**: a Direct-mode command that loads another program file into the memory without destroying what's there.

This is very useful for subroutines and 'modular' programming; the line numbers from the loaded file will be interleaved with or tagged on to those of the program you're currently working on – so you'd better make sure there's no duplication!

Programmer's Aids vary in the way they do handle any inadvertent duplication. The best solution is probably for the original line number to be retained so that a merged line with the same number gets discarded; ideally the package should tell you what line number was duplicated so that you can do something about it. But the cruder Aids don't do that, and some will just overwrite the original lines with the merged lines willy-nilly.

- **TRACE**: shows the last few line numbers executed in a panel on the screen – usually the last six lines, usually in the top right corner of the display. The **OFF** command turns off the trace.

Some version of TRACE allow you to specify how quickly the TRACE is done – in other words, how quickly the program actually runs. If you're not using STEP (or if it's not provided in your Programmer's Aid!) this may be the only way to see what's going on – otherwise some of the TRACEd line numbers will whistle past too quickly.

- **STEP**: executes a program one line at a time after you press a key.

Combined with TRACE, this is a really useful facility in debugging since you can see exactly which line is causing problems for you – especially with jumps like GOTOs and GOSUBs that you might otherwise find tricky to track.

- **abbreviated Basic commands**: a single key – usually one of the function keys – will produce a string of characters that are all or part of one of the more common Basic commands or functions. So you press only one key to put 'LEFT$(' on to the screen; or, in Direct mode, to 'RUN' or 'LIST' a program (in the latter cases the string includes a terminating Return so that the command is executed immediately you press the key)
- **assign text to function keys**: generally by a KEY command, which lets you allocate character strings to the eight function keys. You will probably be restricted in how much text you can have per key – 10 characters seems to be the norm.

But this is useful if there's a statement or two that you use frequently; for instance ...

```
KEY<f1>="FOR T=1 TO"
```

That could be half a delay loop you put into programs regularly.

- **list variables**: usually called DUMP or VAR DUMP, this Direct-mode command provides a helpful list of the names and current values of all non-array variables that your program uses.
- **escape quotes mode**: it can sometimes be pretty annoying to have to type a closing pair of quotes and then backspace to change something that appeared in quotes mode when you actually wanted a different non-quotes character. A simple but useful facility, generally provided as a CTRL-plus-number-key command, will let you do just that.
- **scroll up**: when you LIST a program or run the cursor down through it, successive lines appear from the bottom of the screen. A few Programmer's Aids however let you move the cursor *up* through a listing too, so that preceding lines appear at the *top* of the screen – very handy!
- **LINES**: a Direct-mode command that calculates the number of lines in your program.
- **UNNEW or RESCUE**: recovers a program after you've inadvertently typed 'NEW'.
- **KILL**: returns the 64 to its normal state by removing the fix that enables the 64 to utilise the new commands (which otherwise it would reject with SYNTAX ERRORs). You probably won't get back the use of the memory that the new commands have taken up, though; for that you'll have to press RUN/STOP and RESTORE.

More Basic programming commands

This list suggests a good many of the more pressing omissions in standard Commodore 64 Basic. Most of them are possible to do with a succession of commands; but simplified, speedier and less error-prone programming will result from using these. And the deterrent effect of complicated statement combinations must put many people off attempting the generally desirable effects that these produce ...

- **AVERAGE**: a new function that calculates the average value of elements in a specified numeric array.

Whitby Computers uses it like this in its Softchip 64 and David's Basic cartridges:

```
PRINT AVG(A(0))
```

That averages the whole array. This averages part of the array:

```
PRINT AVG (A(4) TO A(9))
```

- **BELL or CHINKLE**: makes a simple bell-ringing sound, handy for instance as an audible indication that the program has finished some lengthy task like READing DATA or entering items into an array and is now waiting for you to do something. Your program will probably have to use a separate volume-setting POKE before you'll be able to *hear* the 'bell'. A few packages also allow use of CHR$(7) for the same effect: that's a standard ASCII character code that sounds a chime, but it's not provided in the standard Commodore version of ASCII.

- **BLANK**: a new function that checks a string to see if it's blank – if it is composed entirely of spaces, or if it is of zero length. You can then use that information in the program to make decisions:

```
IF BLANK (A$) THEN PRINT "NOTHING
THERE"
```

- **BREAK**: kills the effect of Run/Stop and Restore so that it can't be keyed accidentally during a program run. Whitby Computers uses it as ...

```
BREAK OFF
```

- **BORDER**: specifies the colour of the display border in a single command – no POKEs necessary!

- **CENTRE or DISP**: displays a centered message on the screen, optionally on a specified line:

```
DISP "WARNING!",20
```

- **CLOCK**: displays a clock at a specified screen position, as in ...

```
CLOCK ON 20,0
```

This shows a clock on the top line of the screen (row 0) at position number 20. The clock will stay there until you use a 'CLOCK END' command. It gets the starting time either from the TI$ clock or (in Whitby's case) from some value you've previously PRINTed at that same point on the screen. The 'clock' may actually be a rather more limited 'stopwatch', with a restricted range – for instance it may go only up to 9 minutes and 59 seconds before resetting itself to zero and restarting.

- **COL$**: a new function that changes the cursor colour to one of the specified colour-numbers without you having to PRINT a CTRL or CBM combination. Since it *is* a function, you can also use it in other expressions.

- **CURSOR**: places the cursor at a given x/y position on the screen, with 'x' sensibly being in the range 0 to 24 for rows and 'y' 0 to 39 for position along the rows.

- **DATE$**: automatically converts a numeric date into a string, which can be a handy function in programs that want to use calculations based on a date and text-handling using it. You're unlikely to get a simple translation: 'DATE$(21-01-84)' won't be converted to '210184'. Instead you'll get something like the conversion formula used by Whitby: *(day + 31) * (month + 372) * (year - 32)*.

- **DATE INPUT**: checks that a date is completely valid, including tests for numbers of days and months (checks for leap years and corresponding 29 February dates, too). You can use it as a replacement for INPUT.

- **DISABLE**: kills the effect of Run/Stop so that it can't be keyed accidentally during a program run. Whitby Computers uses it as ...

```
DISABLE OFF
```

A 'DISABLE ON' command restores the normal state of affairs. Either way the automatic LOAD using Shifted Run/Stop will still work.

- **EXECUTE**: a neat piece of shorthand, and a handy alternative to GOTOs and GOSUBs.

This one allows you to specify a collection of Basic commands as a single character string, and subsequently execute them:

```
10 A$ = "FOR T=1 TO 1000: NEXT: GOTO 150:
REM DELAY"
100 EXEC A$
150 PRINT "DELAY LOOP COMPLETE"
```

This can also be used to provide a 'computed GOTO'. Normally GOTO will not work if the line number is given as a variable – in other words, if your program tries to GO TO a different line number depending on the result of a calculation that you're holding in a variable. EXEC does it like this:

```
EXEC "GOTO" + STR$(N)
```

... where N is your variable.

- **GOTO or GOSUB with a labelled destination**: GOTO and GOSUB normally require a line number. It's less of an effort (less liable to mistakes) if you can jump to a *labelled* line rather than a numbered one.

Whitby Computers offers a couple of extra commands and a special form of the standard REM:

```
10 GOSUB "Delay routine"
1000 REM [Delay routine]
1010 FOR T=1 TO 1000: NEXT: RETURN
```

The label is enclosed in square brackets after the REM.

- **INK**: specifies the colour of the cursor and text in a single command.

- **INPUT CHECK**: Whitby Computers has a 'generalised INPUT' command GENIN.

This allows you to pre-specify the maximum length and acceptable characters for an INPUT:

```
GENIN(12)"0123456789";A$
```

Here the INPUT replacement demands a string of 12 characters that can be composed only of the numerals 0 to 9 or spaces. Whitby also has a similar command for checking numerical input where the numbers are accepted as a string variable. The maximum size of the integer part of the input can be specified; so can positive numbers only. For instance:

```
NUMIN4,+;A$
```

Here the program will accept only positive numbers up to 9999.99.

- **MAX**: a new function that finds the highest value of elements in a specified numeric array.

Whitby Computers uses it like this:

```
PRINTMAX(A(0))
```

That prints the highest value in the whole array. This finds the highest value in part of the array:

```
PRINTMIN(A(4)TOA(9))
```

- **MIN**: a new function that finds the lowest value of elements in a specified numeric array.

Whitby Computers uses it like this in its Softchip 64 and David's Basic cartridges:

```
PRINTMIN(A(0))
```

That prints the smallest value in the whole array. This finds the lowest value in part of the array:

```
PRINTMIN(A(4)TOA(9))
```

- **NOKEY**: makes selected keys inoperative. The new command is followed by a character string containing the single letter denoting an individual key that you want to kill.

- **ON ERROR GOTO** or **ON ERROR GOSUB**: spots errors during a program run. Instead of things grinding to a halt with the 64 stopping the program and printing an error message, the program will go to a specified line number for you to take some evasive action – usually on the basis of what's in a new reserved variable, typically called ER. That will hold a number identifying the error; so depending on the value of ER you can tell the user that something is wrong and hopefully try to correct it without leaving the program.

- **ON KEY GOTO and ON KEY GOSUB**: Switches program flow when a key is pressed.

Depending how this command is presented, it could be !**any** key; more likely you'll specify a key – or a series of keys, with different destinations for the GOTO/GOSUB according to which of them is actually detected. For instance:

```
10 PRINT "YES, NO OR MAYBE"
20 ON KEY "YNM" GOTO 1000, 200, 280
```

Note that you don't need an INPUT variable to check; line 20 will take the program to line 2000 if 'Y' was typed, to 200 if 'N' was entered, or to 280 if it was 'M' that the user pressed.

- **PAD or SPACES**: fills out a string variable with spaces to the right of text until the specified number of characters is reached:

```
PRINT PAD$("HI",9)+"THERE"
```

That PRINTs 'HI' followed by seven spaces (because the specified size of the string is nine characters you get 9-LEN('HI') spaces tacked on to the text) followed by 'THERE'. This of course is a trivial example; the padding is more likely to be useful when you are taking INPUT or READing DATA that's to go into a file containing records with fields of a specific length.

- **PAPER**: specifies the colour of the display background in a single command.

- **PAUSE**: lets the program pause for a specified number of seconds, optionally with a preceding message – it replaces for FOR/NEXT delay loop.

- **PUSH**: PUSH lets you fool the 64 into thinking that there's a subroutine call so that the next RETURN encountered in the program will jump to a specified line number.

It works because the 64 has an area of memory called the 'Stack'; in this it keep a reference of line numbers for GOSUB calls (among other things). The PUSH command adds a fake GOSUB call to the Stack, so when RETURN turns up the 64 will obligingly jump to the line number specified in your PUSH statement. One use of this is in calculating a line for an equivalent to GOTO (which normally works only with explicit line numbers):

```
INPUT "LINE NUMBER";N:PUSH N:RETURN
```

- **POP**: another Stack command – this cancels the line number that the 64 remembered from the last GOSUB, so that you can GO to a subroutine without having to RETURN from it (ordinarily this would result in an OUT OF MEMORY error). As

such it works rather like a conventional GOTO in practice, but there are some clever tricks you can do if you're repeatedly calling subroutines and don't want to RETURN in some circumstances.

- **specified RESTORE**: the RESTORE command in its conventional form allows re-READing of any DATA. An updated version would enable you to specify a start position for the RESTORE:

```
10 DATA 1,2,3
20 DATA 4,5,6
100 READ A,B,C,D,E,F
110 RESTORE 20
120 READ G,H,I
```

With the standard RESTORE, line 120 would READ '1', '2' and '3'. This way it READs '4', '5' and '6'.

- **SAVE and LOAD SCREEN**: saves a complete copy of whatever's on the screen and reloads it as required. You can save any text and colour information this way, which makes it particular useful for designing (say) game backgrounds and data entry forms.

- **SCAN**: this command searches a specified string for a given set of characters and comes back with the character position where the match is made.

You will also have to specify the start position – normally that's '0', meaning 'scan from the very first character in the string', but doesn't have to be. For instance:

```
10 A$="HELLO DAD": P=2: SCAN A$,P FOR
"HE"
```

The string will be searched from the third character for the characters 'HE'. It won't find them. Here's a successful scan:

```
10 A$="HELLO DAD": P=0: SCAN A$,P FOR
"DA"
```

This time the search will start from the front of the string and the variable P will come out of the scan with the value 6 (since that's the start of the searched-for string). And you can also scan for the first character position that is *not* the string specified. This example strips off any leading spaces:

```
10 A$="<spc><spc>HELLO DAD": P=0:
SCAN A$,P FOR NOT "<spc>"
20 PRINT MID$(A$,P)
```

- **SEARCH array**: searches a one-dimensional string array for a nominated collection of characters. You can usually use 'wild-card' characters in this.

- **SORT array**: sorts all or part of a one-dimensional string or numeric array into increasing order.

- **SPC$**: gives a string of a specified number of spaces – easy enough to do in standard Basic, of course, but any help is welcome ...

- **SWAP or CHAIN**: loads in another program without losing all the variables set up by the first (as normally happens when your program includes a LOAD command). In other words, program one can define some variables to be used by program two: that way you can have in effect a single chained program that's larger than the maximum memory size available for it.

- **SUM**: a new function that calculates the total value of elements in a specified numeric array.

Whitby Computers uses it like this in its Softchip 64 and David's Basic cartridges:

```
PRINT SUM(A(0))
```

That sums the whole array. This sums part of the array:

```
PRINT AVG(A(4) TO A(9))
```

- **compress numbers for economical storage**: Whitby's SHR$ ('shrink') function clever squashes a number into a smaller number of string characters. A single character only is needed for numbers up to 127; you can get a ten-digit number into just four string characters. An XPD (expand) function is provided to 'decompress' numbers again when you want to use them.

- **decimal alignment for numbers**: A special form of the PRINT command specifies a position at which decimal points should be printed. Numbers will then be arranged appropriately around it, so that the points always appear in a column.

- **handling lightpens, joysticks and potentiometer-based add-ons like paddles**: single commands or functions to read the values of the position – no PEEKs necessary!

- **number systems converter**: some offerings come with a mini-calculator for converting between binary, decimal and hexadecimal numbers. A few have this provided as a function operating on numeric variables.

- **extended user-defined functions and subroutine calls**: Whitby's ROUT and PER statements are more or less the equivalent of DEF FN and the FN function, but they allow you to define and use a function with multiple arguments and multiple statement lines – and you can define string functions too. ROUT (for ROUTine) defines the function, PER performs it. In effect PER works like a GOSUB call to a subroutine the first command of which is ROUT; then when PER is executed the variables specified in that command are assigned appropriately to fit those demanded by the ROUTine.

Sprites and user-defined graphics

There's a good deal of scope for added commands in this area, which otherwise involves so many PEEKs, POKEs, IF/THENs and GOSUBs that it can easily drive the average person nuts – while taking a *very* long time to complete even a simple graphic. And the more you have to type in, the more likely the chance of mistakes.

- **LINE**: draws a line (or possibly erases a line) between two specified points.
- **ROTATE**: turns a shape on the screen around it x or y axes, usually by 45 degrees at a time or by a specified number of degrees – some packages just turn until you take your finger off the key.
- **DRAW**: lets you use the cursor as a 'pencil', with cursor controls or (more likely) joystick to 'draw' on the screen.
- **MOVE**: moves the cursor to a specified x,y position.
- **BOX**: draws a box of specified x and y dimensions.
- **CIRCLE**: draws a circle of specified radius. You may also get an **ARC** command for partial circles.
- **user-defined commands**: Whitby's Softchip lets you define your very own commands in addition to the collection of commands and functions you get in the standard product. You have to specify the command's name and a start address for the machine code that handles it (Softchip can also include a monitor for you to write that code).
- **PLOT**: plots one or more points on a graph – possibly connecting them with solid or dashed lines, or some other character. This command can generally be used to draw triangles as well.
- **FILL or PAINT**: position the cursor anywhere in a bounded figure to fill it with a specified colour and/or pattern.
- **COPY**: takes a copy of the character set and puts it into RAM for you to amend with some user-defined characters. You may also get a character editor for this.
- **sprite designer**: generally works by clearing the screen and putting on to it an enlarged grid on which you can design the shape and specify colour.
- **save and load sprite data**: a single command like 'SSPRITE' is usually provided to SAVE the definition of a sprite on disk or tape. Something like 'LSPRITE' is provided to LOAD those definitions subsequently.
- **SPRITE ON/OFF**: a single command can turn one, some or all sprites on and later off again. You may be able to use a number of functions with this command – setting sprite colour, turning expanded and/on multi-colour modes on and off, specifying the sprite position, specifying transparent mode.
- **collision detect**: Whitby's Softchip includes two functions that can be used to detect easily whether a nominated sprite has hit the background (HITBACK) or another sprite (HITSPR):

```
2000 IF HITSPR(3) THEN PRINT "SPRITE
3 HIT BACKGROUND"
```

- **SCR WAIT**: an unusual one, this, and only provided in BC Basic as far as I know. It synchronises movement on the screen with the TV set's blanking periods – they happen very quickly, but without this synchronisation you can get a noticeable flicker particularly on fast-moving sprite displays.

Structured programming

Structured programming is more a philosophy than a set of rules, but in essence it takes the view that programs should be constructed with logic and clarity. Some of the standard forms of Basic's commands on the 64 do not encourage this; but there are a number of options that can improve the sanity and comprehensibility of your programming – while making it more versatile at the same time.

- **CONDITIONAL IF:** when the program comes across a 'conditional' IF the statements after it are evaluated in the normal way. If they are 'true' the rest of the statements following are executed. Otherwise trhe program will skip ahead looking for an **ELSE** or an **ELSE IF** statement. An ELSE IF is treated like the opening conditional IF and branch is made depending on whether the statements following *that* are true or false; an ELSE gets executed only if all preceding conditional IFs and ELSE IFs are false. The whole lot is terminated by a conditional **END** statement.

Sound

If anything, setting up the 64's sound generators are even more complex than working with graphics. Anything that helps is greatly appreciated, though truth to tell I haven't yet seen a general-purpose cartridge that matches the capabilities provided in highly specific (and relatively costly) synthesiser packages like MusiCalc.

- **ENVELOPE**: specifies the ADSR envelope for a particular voice.
- **WAVE**: specifies the waveform similarly.

- **MUSIC**: allows you to define a string of notes. You can usually specify duration, octave and other data for each note.

- **PLAY**: actually plays the notes you are setting up – or can play a previously-created MUSIC file.

Assembly language tools

Most Commodore computers apart from the 64 come with a ready-provided machine-code monitor built in; that makes it easier to enter, view, edit, save and load machine-code programs.

There are a number of quite excellent machine-code programming tools available for the 64, particularly in terms of assemblers (which are one step up from monitors in that you can use mnemonic commands rather than actual raw .exadecimal numbers for machine-code programming).

A few add-on cartridges with other functions do however include a monitor for direct access to the hex notation of machine code. The kind of functions you can expect from these should include these:

- **display memory** – *between* given locations, *from* a location, or *up to* a location

- **display registers** – examine the contents of the 6510's internal registers as they were when the monitor was entered

- **save** a block of memory between two specified addresses

- **load** a previously-saved block of memory into a specified start location

- **start executing** machine-code at a given address (equivalent to Basic's SYS command)

- **return to Basic**

Disk handling

The 64's use of the disk drive isn't as easy as it might be, a major impediment to its more widespread use. Some of the extra commands noted here are brand new, but most are abridged versions of the standard set; quite apart from the increased simplicity, again it's the case that the less you have to type in the fewer mistakes you're likely to make.

- **abbreviated commands**: the OPEN and PRINT sequences required for disk handling can be tedious, to say the least. Larger Commodore computers have shorthand forms of the disk commands, and similar minimum-keystroke effects are provided by some add-ons.

- **disk DIRectory**: the conventional method of loading a disk directory and then LISTing it effectively wipes whatever was in memory. The 1541 now comes with a 'wedge' that allows you to take a look at what's on the disk without the LOAD '$' and LIST sequence – and without destroying the program in memory. This kind of command is also provided in add-on cartridges that offer disk-handling commands; in some cases it's extended so that you can check the disk directory for files that match a partial name plus wild-card.

- **reading the disk error channel**: a quick and easy method of finding out what the problem is when the disk isn't happy

- **COLLECT**: tidies up a disk by removing dead space btween files and relocating what's on the disk to make the best use of the space available.

- **CONCAT**: joins together (concatenates) two sequential files, perhaps allowing you to give a new name to the enlarged file thus created.

Printer control

There are a couple of awkwardnesses about the Commodore PRINT# statement, quite apart from the built-in problem of not being able easily to use a non-Commodore printer ...

- **PRINT# with TAB**: the standard version of PRINT# cannot include a TAB.

- **SCREEN COPY**: prints an exact copy of what's on the screen.

- **select printer**: a command like Whitby's PCTRL will set the device number and characteristics of a printer you want to use. It should include a specification of device number that defaults to the standard setting of 4; the number of line feeds associated with an end-of-line Return (some printers expect one, some expect none); and a 'printer type' indicator, which in Whitby's case is one of three – 0 for standard Commodore printers (meaning one that takes Commodore's ASCII character set): 1 for printers taking everyone else's version of the ASCII character code correspondences: and 2 for Commodore's own tractor-feed printer.

CHAPTER 17
Writing Faster Programs

... And not 'writing programs faster', I'm afraid. This Chapter should show you a few ways of squeezing more out of the 64 – using less memory and accelerating the RUNning speed of your programs.

But for starters there's the major drawback that you're almost certainly writing programs in Basic.

All programming languages score by letting you write programs in something that more or less resembles English, using single-word commands for what would otherwise be quite lengthy and detailed operations. The alternative is the string of 0s and 1s that actually corresponds to the way the computer works.

And because of this, Basic is in general a pretty desirable alternative: it makes the computer generally more accessible.

But the problem with any programming language is that sooner or later the computer has to translate your programs into the 0s and 1s that the computer needs. This inevitably means a time penalty while the translation is done; and because there has to be some kind of converter program running in there, it cuts down the amount of memory available to you.

The problem is exacerbated on home computers like the 64 because their Basic is almost always *interpreted* – after each line you enter by hitting Return, the computer has decode everything in the line into binary before it can execute it and move on to the next one.

So before going further, it's worth observing that the best ways to make more of the 64 – and in particular the most effective ways of making programs run faster – is to use either machine code or a Basic compiler.

On the other hand, machine code is nothing like English: it is a whole new set of concepts and terms to learn, and if you aren't particularly numerate you might well find it a struggle.

And Basic compilers are undeniably desirable – you write the program and put it through a 'compiler' that turns it into binary before RUNning it, so when you use it in earnest the computer doesn't have to waste any time translating the lines of code: so the final program ends up a good deal shorter as well as running much more swiftly. But compilers are expensive.

But don't despair. Provided you don't set your sights too high, there are ways of making Basic programs run faster and use less memory. The major caveat is that several of these suggestions conflict with the kind of advice you should hear about writing 'better' Basic – especially the one that says you should make your programs easy to read for any subsequent amendments that you (or someone else) might want to do. Life can be a real sod sometimes.

Incidentally, you may think that with 64K of memory on the Commodore 64 (well, 38K anyhow) you might not need to bother too much about how much memory you're taking up. Well, quite apart from the garbage collection problem (of which more later) there is a direct link between how much memory your program takes and how fast it can RUN: Basic reads *every* byte of a program, even though it may contain non-operative things like REMs and blank spaces. Since reading all those bytes takes time, a longer program will inevitably run more slowly.

Chop out those REMarks

For instance, the simplest way to cut down the size and run-time of your program is to chop out all REMarks.

On a line by itself a REM will take up at least four bytes anyhow, more if it's on a high line number: if you want some explanatory text with it, that takes yet more memory. And each REMark has to be laboriously decoded by Basic before it's discarded as a non-executing line, so it slows things down.

You can do without REMs and still retain some distinction between different sections of the program by starting them on identifiable line numbers – initialisation at line 10, say, with the main parts of the program as multipliers of 100 and subroutines and DATA numbered in the 1000s. You have up to 63,999 line numbers available to you!

Go for low line numbers

But while you may have that many line numbers on offer, the bug in the unguent is the fact that the longer a line number is the more space it occupies in memory (one byte per figure).

The trade-off between readability and space is something you'll have to make for yourself. If you're serious about minimising the memory requirement and run-time speed of any program, though, write a legible and clear version *first* and save it. Then you can get stuck into the code to produce a slimmed-down go-faster equivalent without losing the elegance and clarity of the original.

Reposition REMs

If you're loathe to chop out all REMs – and you should be – it's worth considering where they fall in the program and how often they are executing. The cardinal rule here is *don't put REMs inside loops*; it's almost invariably unnecessary, and while the program is running around the loop it will wasting time decoding the REM and deciding that it's a non-actionable statement before proceeding.

Cut out spaces

Blank spaces in the lines of a program also take up room in memory, one byte per space. Again, you'll have to make your own decisions about how much readability you need: but don't of course try to economise too much by chopping out blanks within quote marks that you need for spacing the display or printed output.

Removing spaces probably won't make more than a one or two per cent difference to the RUNNING speed of a program.

Save on quotes

Lines that end with a pair of closing quotes do not actually require them; omitting the terminating quotes saves one byte. You can't of course omit them if they *aren't* at the very end of a line.

Use multiple instructions per line

The 64's Basic allows up to 80 characters per a line of program code, and you can use that length to its full by putting several successive statements on it separated by colons.

That takes less memory, because you don't need to store so many line numbers: each line number takes five bytes, a colon occupies only one. And the program runs faster, because the computer has only the one line to decode rather than several. Be wary of illegal combinations, though – like one of the lines you're cramming into one may be the target of a GOTO or GOSUB somewhere else in the program.

Abbreviate where possible

To make the most of the 80-character line you can reduce most commands, statements and functions by using abbreviations. Normally these are the first letter of the command plus the second letter when you're holding the shift key down. When you subsequently LIST a program produced using the abbreviations, you'll see that the 64 has helpfully translated the abbreviations back into the original terms: and on the display the line then may well be over 80 characters, though inside memory it's still within the limits.

GOTO and GOSUB

When Basic comes across a GOTO or GOSUB it goes and looks for the line number specified in the statement. If that number happens to be *lower* than the current line number (the one containing the GO statement), the search will start from the beginning of the program. If it's higher, the 64 begins looking from the current line number.

If the destination is a long way away it'll take time for the 64 to get there; so there is some incentive to place subroutines and jumped-to lines as close as possible to the calling line.

But the real penalty comes from RETURN. When the program has to get back from a subroutine, it will start searching for the point at which it left by going back to the start of the program. Now, this might not make so much difference on short programs: but the delay can be dramatic on longish ones.

The policy usually recommended is to put all frequently-used GOSUBs at the beginning of your program to get the best mix of speed and efficiency. Apart from anything else it takes less time for Basic to read a low (short) line number than a high (long) number, so it should be able to decode the GOSUB more quickly.

On the other hand, this may not be true. That's because Basic stores line numbers as hexadecimal numbers in two bytes, a 'high'-byte and a 'low'-byte; and when it looks for a line Basic reads the high-byte first to see if the destination is later in the program than the current line. Lines have to be numbered in increments greater than 256 for the high-byte to change from one line to the next; so a lot of low-numbered lines will actually be a drag on program execution, since the high-byte can't be used as the instant identifier – all the low numbers will have the same high-byte value.

In other words, there's an argument for spacing your line numbers in increments of 300 or more. That itself poses new problems – if you want all lines to have a different high-byte you can only have 250 of them before you hit the 63,999 limit; increasing the size of line numbers also causes new delays, because Basic takes longer to decode them. And of course they occupy more memory.

You just can't win, can you?

Look after variables

During the execution of a program Basic stores all the names of variables and all numeric values for future reference as it comes across them. That means they're stored in the sequence in which they were found.

That means it makes sense to put all variables the program will be using frequently right at the beginning of the list Basic sets up. That way Basic won't have to search too far before it find the name and the current value. You can do this by making sure Basic comes across them right at the start of the program by setting them to zero as part of the initialisation procedure.

The fact that Basic maintains this list in the first place suggests that it takes up memory; and indeed it does. So cut down on memory occupancy and the length of the variables list by re-using variable names whenever you can – use 'DELAY' or 'COUNT' for any and all FOR loops, for example.

You can save a little memory space by using short variable names like 'C' rather than 'COUNT'; but there's very little saving on run time. Basic finds a variable with four characters (or even eight or ten) almost as quickly as it can reach one with a single-character name.

The garbage collector cometh

Whenever you assign something inside quote marks to a string variable, Basic takes a copy of it. And if you redefine the variable subsequently, Basic copies that as well ... but it doesn't overwrite the first string associated with the variable name. When a lot of string variables are being redefined, this means that a lot of unwanted and unnecessary strings can accumulate in memory; and eventually there may be so many of them that they take up all the free space and you'll get an ?OUT OF MEMORY message.

Basic does includes an automatic garbage collection routine which is called in as soon as the string variable storage area starts to get so full that it is approaching the adjacent memory space for the actual program. The garbage collector wipes out all the early versions of duplicated strings, leaving the active string variables in there alone.

The problem is that the garbage collector suspends your program while it's sifting through the rubbish: things just seem to go dead for you until it's finished. So it is obviously important to minimise the amount of garbage the program is generating – to cut down as much as possible on the use of strings, in fact.

Use variables not constants

In many cases you can save a lot of time and some memory by using a variable name for a constant or value rather than the value itself. This is particularly true with longish decimals (say four or more digits). A number like 83.2264 take up eight bytes every time it appears – six digits plus the decimal point and the (implied) sign; and it will also have to be converted by Basic into floating-point arithmetic before it's used.

So it's preferable to give frequently used constants or values a name as part of the initialisation routine (so that the variable name is high up in Basic's list of variables) and then use that whenever you need the number.

On the other hand, for short numbers (those with only one or two digits) and integers it's normally better to avoid the use of variable names. The program will probably fun faster if it's dealing with '6' or '32' rather than having to go to its list of names to find out what value a particular label has been given.

Integer variables – ugh!

Simple integer variables are a real time-waster, because Basic always converts integers to floating-point numbers.

That means if you come across the expression ...

```
A%=B%+C%+D%+E%
```

... Basic would find B% and convert that to floating point: then do the same for each of the others: then add them together: then convert the result to integer again: and then assign the value to A%.

It's quicker to use INT, as in ...

```
A%=INT(B+C+D+E)
```

Integer variables – hooray!

On the other hand, integer variables do score if you're using arrays. That's because each element in an integer array takes only two bytes of memory – compared with five bytes per item in a standard numeric (floating-point) array.

So if you're setting up an array that doesn't need the decimal point, try using DIM Z%(25,4) rather than the apparently simpler DIM Z(25,4). You'll be saving 300 bytes in this case!

Avoid arithmetic functions

Functions like exponentiation and COS simplify the writing of calculating programs, but they are pretty sluggish in operation. That is especially true of the exponentiation operator is particularly sluggish (quite apart from its occasional habit of giving answers that are 0.000001 out).

It's generally much quicker to raise a known number to a particular power by repeated multi-

plications. This applies also to SIN, COS, TAN and the other trig functions.

On the other hand, if you're using a variable in the operation the ready-made functions *are* quicker. That's because if you are asking the program to run through a series of calculations involving variables it would have to keep checking the list of variable names for the current value, and that diminishes the benefits of the direct-calculation approach.

Eschew IF

A FOR-NEXT loop can often replace several IF statements – especially in setting up screen layouts. Of course that two-statement combination (FOR/NEXT) takes much less space than a number of IFs. And use GOSUBs whenever there's a lengthier repetition of code.

Multiple IFs can often be replaced with a single ON-GOTO, particularly in testing for keyboard input.

Put your largest loop inside the nest

If you are using nested loops, ensure that the *innermost* one has the greatest number of iterations. Take a look as these two illustrations:

```
100 FOR A=1 TO 1000
200 FOR B=1 TO 10
300 NEXT: NEXT

1000 FOR A=1 TO 10
1010 FOR B=1 TO 1000
1020 NEXT: NEXT
```

In the first case the B loop is executed 1,000 times. In this latter the inner loop is only done 10 times, so the program doesn't have to skip back to line 1000 so often.

Omit the NEXT variable

You can gain a very slight increase in the speed of FOR/NEXT loops by omitting the variable on the NEXT – using NEXT on its own allows the loop to complete about 25 per cent quicker than one with NEXT *variable*.

Economical PRINTing

To put characters in particular positions on the screen you *can* use PRINT statements with spaces and cursor control commands. But they can very easily soak up memory; the SPC and TAB functions are much more economical when you're PRINTing spaces, simply because they take up fewer bytes – and you can save a couple of bytes per command by abbreviating them.

Minimise text on displays

Simplifying your screen layouts can obviously cut down on memory and run-time delays. All those fancy formatting features, colour changes and verbose messages can take up space and time.

Obvious candidates for display diets: abbreviate messages ('SCORE' instead of 'TOTAL SCORE' perhaps, or simply 'TOT') and use GET rather than INPUT (forces you and the user to enter only one key rather than a lumpy character string).

Use READ, DATA, and DIM

If you have a lot of data to be pushed around, the most efficient way to do it is with a READ/DATA construction and/or an array. This obviously takes up less space than a series of repeated assignment statements.

Take DATA from tape or disk

The INPUT# and GET# statements allow you to read information from cassette or disk, and their most effective use in this context is with DATA. With them you can write a DATA-using program as normal, but without READs and DATA statements; instead all the DATA will be on a separate program on tape or disk.

READ statements are replaced by something like ...

```
100 OPEN1,1,0,"DATA"
110 INPUT#1,A$
```

A series of INPUT# statements like this can read in the DATA a line at a time. The result is a significant reduction in the memory your program takes up, since all that DATA no longer has to be part of it.

Segment programs

You can use programs that are larger than the 64's available memory by separating them into discrete sections, storing them in sequence on tape or disk, and loading them in one by one. The first section ends with a LOAD command that restarts the tape player, say: section two is then read into memory, where it actually overwrites part one.

This is called 'chaining' or 'overlaying', and clearly it only works for programs that *can* be split into discrete chunks. But for several types of program it's a good solution, and it does have some merit in that it forces you to write more or less modular programs.

Kill the screen

When you SAVE or LOAD programs on the 64, the screen goes blank. This can be a bit disconcerting, but it's worth noting that any program will run around five per cent faster if the screen is blank at the time. You can blank the screen by ...

```
POKE 56325,PEEK(56325) AND 239
```

Return to normal after the program has done its business with ...

```
POKE 56325,PEEK(56325) OR 16
```

Last word on loops

Most of the techniques suggested here save memory rather than increase operating speed radically. The major brake on your programs will be loops, and it's worth summarising the three key points – those that will have the most significant impact on execution speed:

- Avoid IF/THEN in favour of FOR/NEXT
- Use variables rather than constants within loops
- Don't put REMs within loops

Section 3

Extra hardware

CHAPTER 18
What's a File?

Anything that you direct to one of the 64's externally-connected devices is a 'file'. That includes anything you save on tape or disk, which makes a kind of sense; a file is effectively a collection of related items that can be treated as an entity.

That's a somewhat pompous way of putting it, but it happens to be quite an accurate definition. If you want to save or load anything, whether it's a program or something for a program to work on, the 64 demands that you call it a file: and there are a number of concepts and commands specific to file-handling that you'd better get used to if you want to utilise cassette or disk storage.

So as far as the 64 is concerned a file can contain a *program*: or a bunch of *records* that a program will use (rather as it might use DATA, which in a way represent a collection of 'records' on which the program code can get to work).

There are some other important features of files on the 64:

- **All files must have a name** – otherwise you wouldn't be able to distinguish one from another
- **A file must be OPENed before you can get at it** – though in practice you can sometimes ignore this requirement, since the 64 will do it automatically for you. That applies especially to cassette operations, though even there you might find a use for the OPEN command
- **A file must be CLOSEd after you've finished with it** – again, this can sometimes be ignored since the 64 might do it automatically

The OPEN command

In fact the OPEN and CLOSE commands in Commodore Basic don't relate specifically to *files*; they're provided more in order to notify the 64 that you're going to be using a particular *device* – it's the peripheral device that you 'open' and 'close'.

Since the cassette and disk drive qualify as 'devices', and since your activity with cassette and disk is most likely to involve the movement of files one way or the other, it makes sense to address the subject here. The **OPEN** command is used in this format:

OPEN *file reference number, device number, secondary address, "file name, type, mode"*

For instance:

```
OPEN 15,8,15,"ADDRESSES,S,R"
```

OPEN can be used like this in direct mode, but it's more usually embedded in a program. You don't always need a full-scale OPEN like that one, with all the parameters and gismos tacked on to it – depending on what you're doing and which device you're sending to or receiving from, you may need only the file number and the device number. But let's run through the whole lot for the sake of completeness.

- **file reference number**: can be any number between 1 and 255, and you decide which you'll use. For the sake of simplicity you're advised to make the file number the same as the secondary address number described below, but you can pick any number you want. Each OPEN operation that hasn't been closed should have its very own file number, though: you can have up to 10 different unCLOSEd OPENs in action at once, but if you try using one file number with a second OPEN you'll get a ?FILE OPEN ERROR message.

You may see the file number termed '**logical** file number'; effectively it's an alternative way of identifying a file when your OPEN also includes a file name, but then as you'll see there are several ways of using OPEN *without* a file name – and while the command doesn't have to have a named file, it must *always* have that file number specified with it.

In the example we're opening channel 15.

- **device number**: this one specifies which device you're sending to or receiving from. You've got to use the appropriate number here – see the table. The example refers to device 8, the disk drive.

Device numbers

no	device	activity	connection
0	keyboard	send only	[integral]
1	cassette	send and receive	cassette port
2	RS232	send and receive	user port
3	screen	receive only	video or TV port
4	printer	receive only	serial IEEE port
5	second printer	receive only	serial IEEE port
6	plotter	receive only	serial IEEE port
7	[not allocated]	send and receive	serial IEEE port
8	disk drive	send and receive	serial IEEE port
9	second disk drive	send and receive	serial IEEE port
10-255	[not allocated]	send and receive	serial IEEE port

The common 'secondary addresses' and what they mean with different devices

device:	cassette	printer	plotter	disk
device no:	1	4 or 5	6	8 or 9
address no:				
0	read file only	upper-case and graphics	print ASCII	read from directory
1	write file only	[not used]	plot x,y data	write to directory
2	write file with "end of tape" indicator when CLOSEd	[not used]	select colour	[not used]
3	[not used]	[not used]	select character size	[not used]
4	[not used]	[not used]	rotate characters	[not used]
5	[not used]	[not used]	select scribe line mode	[not used]
6	[not used]	[not used]	upper/lower case mode	[not used]
7	[not used]	lower-case and graphics	select character reset and clear	[not used]
8-14	[not used]	[not used]	[not used]	[not used]
15	[not used]	[not used]	[not used]	select Command channel

- **secondary address**: this unhelpfully-named parameter isn't mandatory, but it can be used to specify a particular type of action depending on which device you've referenced. The example is using disk secondary address 15.

  file name: a unique alphanumeric reference of up to 16 characters. If you use more than 16, the 64 will just ignore the extra. In theory any character can be used in a file name ... apart from the double quotes mark, comma, question mark, asterisk or colon.

  type: for disk drives only you can specify what kind of file it is. There are four options – **S** for *sequential* files, **P** for *program* files, **U** for *user* files and **R** for *relative* files. If you don't include this parameter the 64 will assume that you're reading or writing a program file.

- **mode**: if you do however have the file type indicator, you can also restrict activities on the file by specifying an 'access mode'. This can be either **R** for *read* or **W** for *write* – meaning respectively that your program can *only* read from or write to the file in question.

There's also an undocumented oddity, mode **M**, the discovery of which should earn one Mike Todd a bouquet of thanks: tacking this one on to a sequential file name may allow you to get back most of an inadvertently corrupted program from disk.

The example then OPENs a sequential read-only file called ADDRESSES on the disk.

Types of file

type:	*identified in OPEN command as*	*identified in directory with suffix*	*means*
program	P	PRG	saved from the user program RAM space and therefore almost always a Basic or machine-code program. Cassette or disk
sequential or serial	S	SEQ	data file containing information that can only be put there with PRINT# commands and which can only be read by INPUT# or GET# commands. It's 'sequential' because to get to a data item anywhere in the file the program first has to read and discard everything that precedes it. Cassette or disk
relative	R	REL	data file containing information that can only be put there with PRINT# commands and which can only be read by INPUT# or GET# commands – as for sequential files. The key difference is that the program reading from or writing to the files does *not* have to work serially: it can go straight to the required record. Disk only
user	U	USR	quirky and little-used type of sequential file. Disk only
–	–	DEL	warning message appended by the 64 to a file name when that file has been deleted from disk without being CLOSEd properly. Disk only

The CLOSE command

The OPEN command sets up an area of buffer memory associated with the device referenced. What happens then is that information passing to or from the file on that device gets put into the buffer until the next read or write command (the next INPUT# or GET#, or the next PRINT#) is executed: or until the CLOSE command is detected.

This means that if you OPEN a file and don't subsequently CLOSE it, the last bit of data won't be transferred. Any cassette file will be incomplete, a new disk file will be empty, an existing disk file won't have all the updates.

The CLOSE command takes the format ...

CLOSE *file reference number*

If you try to OPEN a file with the same file reference number without having used CLOSE first, you'll get a ?FILE OPEN ERROR.

And there's another problem: the 64 will 'lose' an OPENed file if it isn't CLOSEd before a variety of Direct-mode actions – including the addition or editing of new program lines and the use of RUN, NEW, CLR or LOAD. Once the OPEN has been 'lost' like this, all file reference and device number associations are cleared and you'll lose anything sitting in the temporary transfer buffer. You won't ever be able to CLOSE that file, either.

Commands for files

The two most common commands associated with files are SAVE and LOAD. Which, reasonably enough, save and load programs (but not data files, those with the SEQ and REL suffixes). We'll look more closely at these when discussing disk and tape operations in the two chapters following, along with the VERIFY command (checks the accuracy of a SAVE).

Three other commands are provided to read and write data items on files. GET# and INPUT# are obviously versions of the familiar GET and INPUT for reading data; PRINT# sends something to a file. In all three cases the file reference number must follow the hash mark.

Here's a simple illustration that sets up a club membership fees system on tape:

```
10    REM set up membership records in
CLUB file
20 CLR
30 OPEN 1,1,1,"CLUB"

100 INPUT "NAME";N$
200 INPUT "MEMBERSHIP NUMBER";MN
300 INPUT "MEMBERSHIP CATEGORY";C$
500 INPUT "FEE PAID (Y OR N)";YN$
500 INPUT "AMOUNT PAID";AP
```

```
600 PRINT "CHECK DETAILS:"
610 PRINT "NAME:" TAB(20) N$
620 PRINT "MEMBERSHIP NUMBER:" TAB(20)
MN
630 PRINT "MEMBERSHIP CATEGORY:"
TAB(20) C$
640 PRINT "FEE PAID (Y OR N):" TAB(20)
YN$
650 PRINT "AMOUNT PAID:" TAB(20) AP
660 PRINT: PRINT "ALL OK?"
670 GET K$: IF K$ = "" THEN 670
680 IF K$ = "N" THEN 40

700 PRINT#1, N$
710 PRINT#1, MN
720 PRINT#1, C$
730 PRINT#1, YN$
740 PRINT#1, AP

800 C = C + 1
810 INPUT "ANOTHER"; Z$
820 GET Z$: IF Z$ = "" THEN 820
830 IF Z$ = "Y" THEN PRINT "NUMBER OF
ENTRIES SO FAR TODAY:", C; GOTO 40

900 PRINT "MEMBER RECORDS ADDED THIS
RUN:" C
910 CLOSE 1
920 END
```

Here line 20 kicks off with a tidy CLR just to reset all variables to zero. Then we're OPENing a cassette file called CLUB with reference number 1 and the secondary address 1 (this adds an end-of-file marker when you eventually CLOSE the file, so that subsequent reads don't go past the finish of this file and try to get further data from anything else that follows it on the tape).

The INPUTs in lines 100 to 500 request a bunch of member information. I started at 100 to give some space for fancy screen formatting statements after line 50. And I've numbered in 100s here to leave some room for the kind of input verification checks that you'd want to include. For instance you might want some tests here to make sure the membership membership number has the right number of digits: that the category entry is a valid one: that the fee is one of the standard rates you operate for the club ... and so on.

Lines 600 to 680 let you confirm that the information typed in is what you intended. If it isn't, line 680 will whip you back to 40 to re-enter the whole lot.

Otherwise lines 700 to 740 write the member's entry to tape. Doing that with separate PRINT# statements isn't the only method, and it certainly isn't the neatest – you could use a single line command like this:

```
700 IF K$ = "Y" THEN PRINT#1, N$;
CHR$(13); MN; CHR$(13); C$; CHR$(13);
YN$; CHR$(13); AP; CHR$(13)
```

Or you could simplify things by defining CHR$(13) as a shorter variable in the set-up phase:

```
50 R$ = CHR$(13)

700 IF K$ = "Y" THEN PRINT#1, N$; R$; MN;
R$; C$; R$; YN$; R$; AP; R$
```

Why do you need CHR$(13) in there anyhow? Because Return is the only delimiter that INPUT# recognises as separating one item from another, and INPUT# is one obvious possibility for reading them back. The separate PRINT# statements automatically put a Return character after each item written to tape; you'll have to do it yourself if you use the single-line approach.

Having created a membership file, let's have a program to reference it:

```
10    REM read CLUB file
20 CLR
30 OPEN 1, 1, 0, "CLUB"

100   REM menu of options
110 PRINT "<clr>OPTIONS"
120 PRINT "<cud>1." TAB(5)
"<rvs>BROWSE<rvo>"
130 PRINT "2." TAB(5) "<rvs>FIND
MEMBER BY NUMBER<rvo>"
140 PRINT "3." TAB(5) "<rvs>FIND
MEMBER BY NAME<rvo>"
150 PRINT "4." TAB(5)
"<rvs>EXIT<rvo>"
160 INPUT "WHICH ONE"; E
170 ON E GOTO 200, 300, 400, 500

200   REM browse
210 GOSUB 600
220 GOSUB 700
230 INPUT "ANOTHER"; A$
240 IF A$ = "Y" THEN 210
250 GOTO 100

300   REM search for number
310 INPUT "FIND WHICH NUMBER"; WN
320 GOSUB 600
330 IF MN <> WN THEN GOTO 320
340 GOSUB 700
350 INPUT "ANOTHER"; A$
360 IF A$ = "Y" THEN 320
370 GOTO 100

400   REM search for name
410 INPUT "FIND WHICH NAME"; WN$
420 GOSUB 600
430 IF N$ <> WN$ THEN GOTO 420
440 GOSUB 700
450 INPUT "ANOTHER"; A$
460 IF A$ = "Y" THEN 420
470 GOTO 100

500   REM exit from program
510 CLOSE 1
520 END

600   REM INPUT# subroutine
610 INPUT#1, N$, MN, C$, YN$, AP
620 IF (ST AND 64) <> 64 THEN RETURN
630 PRINT "END OF FILE – NO MORE ENTRIES"
640 PRINT "REWIND TAPE IF YOU WISH TO
CONTINUE"
650 GOTO 100
```

```
700    REM display details subroutine
710 PRINT "NAME:" TAB(20) N$
720 PRINT "MEMBERSHIP NUMBER:" TAB(20)
MN
730 PRINT "MEMBERSHIP CATEGORY:"
TAB(20) C$
740 PRINT "FEE PAID (Y OR N):" TAB(20)
YN$
750 PRINT "AMOUNT PAID:" TAB(20) AP
760 PRINT "ANY KEY TO CONTINUE"
770 GET K$: IF K$ = "" THEN 770
780 RETURN
```

This looks more complicated than it is. For a start we're OPENing the file CLUB as a read-only tape file and then printing a list of options. If option 4 is selected the program jumps to an exit subroutine that closes the file and includes an END to prevent any running over into the other routines down there; otherwise the three 'find a record' options all use the routine at 600 to read one member's worth of data and the other subroutine at 700 to display it.

The 'browse' option simply starts reading and displaying member records from the start of the file. Or rather, it will read from whatever point the tape happens to be started at; so if you want to adapt this program for yourself you'd be well advised to include some kind of warning message to the user – like a PRINT 'REWIND TAPE!!!' statement somewhere.

The 'search for member number' and 'search for member name' options work similarly – you tell the program what you're looking for, it uses the routine at 600 to read one record, it compares the relevant variable with your entry, and displays the record via the routine at 700 if it finds a match. If it doesn't find what you specified it goes back to try again on the next record.

Testing for end-of-file

A word about lines 620 to 640 in the file-read example. It makes sense to include some kind of test for the end-of-file mark after an INPUT# – otherwise you'll get an error message (rather unhelpfully, it's ?STRING TOO LONG). This is because the 64 automatically puts the null character CHR$(0) on to the end of a tape file; and you don't want that to be picked up, since the program will stop dead with 'READY' on the screen.

Covering that situation is what these lines do. They test the system STatus variable, which among other things can tell you when you've hit the end-of-file mark put there by the secondary address part of your OPEN 1,1,1 in the file creation program.

Bit 6 of ST is set if that end-of-file indicator is encountered, and so you could simply test for ST=64; I've used the ST AND 64 option because I'm testing for *not* equal to 64, and since there are several other bits that could conceivably be set in the ST variable during a cassette operation this is the only safe way to make a decision based solely on the setting of bit 6.

More about PRINT#

PRINT# writes things more or less as the straight PRINT does, except that PRINT without the hash sign writes only to the screen ... and except for a few rather special characteristics (mostly restrictions) in PRINT#.

There are basically four things to watch.

- **PRINT# doesn't recognise the line length limitation of the screen or printer.** You can separate data items by commas, and when they're written to tape or disk they will have 10 spaces between them ... and the total number of spaces and items can exceed the 80 character restriction. Don't go overboard with this, though; lots of commas means lots of space characters, and that it turn can mean a pretty big file (takes much space on cassette or disk, takes longer to read through).

- **TAB doesn't tab.** The disk or tape file won't recognise the specific character positions you get on the screen, where a PRINT TAB command starts the next lot of printing at the character position you nominate; with PRINT#, the TAB function simply works like SPC – and you get the specified number of spaces *after* the preceding item and *before* the next.

- **PRINT# items will automatically be followed by a Return character unless you have a semicolon or a comma in the line.** The **comma** is required anyhow after any of the three # commands; but when used subsequently in a PRINT# statement between items it will give you ten spaces there (and don't forget that positive numbers have an extra space automatically tacked on before the digits, so you may find yourself with *11* spaces when you read something back). The **semicolon** means you get *no* space between items.

- **PRINT# doesn't recognise screen control codes as screen control codes.** Instead it just records them as ordinary quotes-mode text – reversed characters or symbols.

More about INPUT#

INPUT# works almost exactly like INPUT without the hash. And most of the same restrictions apply:

- *The normal rules about variable types apply.* If the INPUT# variable you've given is numeric, you'll get an error message (?FILE DATA ERROR) if the next item read from tape or disk is not a number – and vice versa.

- **Similarly the line length restriction is still enforced.** You'll also get a ?STRING TOO LONG ERROR if your INPUT# attempts to read something longer than 88 characters from the data file

- **Quote marks won't be included if they open the data item unless you use one of the tricks mentioned in Chapter 8.** Quotes anywhere else in a string will be included, though.

- **Leading spaces are ignored** (apart from the one space automatically put in front of positive numbers to hold an invisible plus sign). Spaces between numbers are also ignored.

- **Commas are handled in the same way as for INPUT** – if they're inside quote marks they are just treated as part of the string: otherwise they separate sub-items and the PRINT# statement also uses commas to request a group of items from the file. Which means that you can read a bunch of data with one INPUT# that has been set up by one PRINT# followed by items separated by commas – though it's generally safer to use Return as the separator for items on a file. (Commas shouldn't be used in numbers, of course – if they are, the comma and everything following it will be discarded ... and the program won't tell you it's done that!)

The GET# command

This too works rather like its hashless progenitor – it reads a single character from the nominated file reference number.

GET# has one major advantage over INPUT#: since it can be used to read every character in a file, your program can test the input character by character. This makes it particularly useful when there's a number of decisions possible on the basis of character type, when there is any danger of scrambled input, or when the data is being received one character at a time anyhow. In particular, GET# is great for communications applications using the RS232 facility in the 64 – they fulfill all three criteria. More about this in Chapter 28.

But for normal file-reading work the GET# command can be a bit awkward, though, and in general I find INPUT# a lot easier to use. For a start, GET# will read any and every character you can put into a file. That includes spaces, Return, DELete, and so on. This means you will have to test for characters like Return that are being used to separate data items.

Numeric variables can be used with GET#; but you'd better be sure that you will be GETting a number. If the item being read starts with anything other than a digit, a plus or minus sign, a point, or the exponent sign E, they'll all be accepted (though that character will be ignored – including the minus signs). If it starts with a colon or a comma that too will be discarded and you'll get ?EXTRA IGNORED; at least the program won't die, though, and the GET# will pick up the next character.

The real bugbear is when the GET# finds a data item starting with anything else, because then the program will stop short with a ?SYNTAX ERROR and return to Direct mode. Nasty. To cover that, you're best off reading items with GET# as string variables in any case and have subroutines to figure out whether in fact the data is numeric.

All in all, though, the INPUT# command looks a lot more tolerant ...

CHAPTER 19
Cassettes

When you buy a 64, you'd be daft not to take a cassette unit as well. Switching off the computer means that everything that was it in its memory is erased – which includes any program you're working on. And you can't have more than one program at a time sitting in the 64's memory; if you want to work on a second program, you'll have to get rid of the first one somehow.

The other argument is not so trivial, either: while disk drives are so expensive and cartridges still pretty costly, the cheap and simple distribution mechanism for software will continue to be the cassette tape. If you want to buy games, utilities, or just about anything else in the way of software for your 64 you'll find the widest choice available on tape.

You don't have to buy Commodore's own tape unit. It is possible to use your own portable tape player ... but it's not easy. This is because you'll need a special connector to match the audio-oriented characteristics of the tape player to those of the computer.

You can buy such converters, but you'll probably have to search through the small ads in the microcomputer magazines. The people who sell them aren't large and sophisticated marketing organisations, so you're likely to get a bit of a amateurish products with few if any instructions or advice in return for what will admittedly be a modest outlay – ten quid or so at most.

It's simpler to buy Commodore's own tape unit. This isn't cheap, it won't let you play audio tapes, and it's slower than the average cassette player in moving the tape past the read/write heads. But it is designed specifically for use with Commodore computers, so it comes with the right sort of plug. It doesn't need a separate power lead, either – it gets its electricity direct from the 64. It is also pretty sturdy and reliable too, which means it isn't likely to break down and it isn't likely to give you any errors when you store a program on tape or read it back into the 64's memory.

There are two other considerations if the computer manufacturer is also responsible for the tape unit. For one, commands to sense the operation of the tape unit and to start and stop it can be incorporated into the design of the computer. In fact Commodore didn't go that far, and it's not easy to control the cassette unit from a program – but bits 3, 4 and 5 of addresses 0 and 1 can be PEEKed and POKEd to affect tape operation.

The other plus point for the Commodore unit is less equivocal: a computer manufacturer who is also responsible for the tape unit can incorporate automatic error-detection and correction procedures that would not otherwise be feasible.

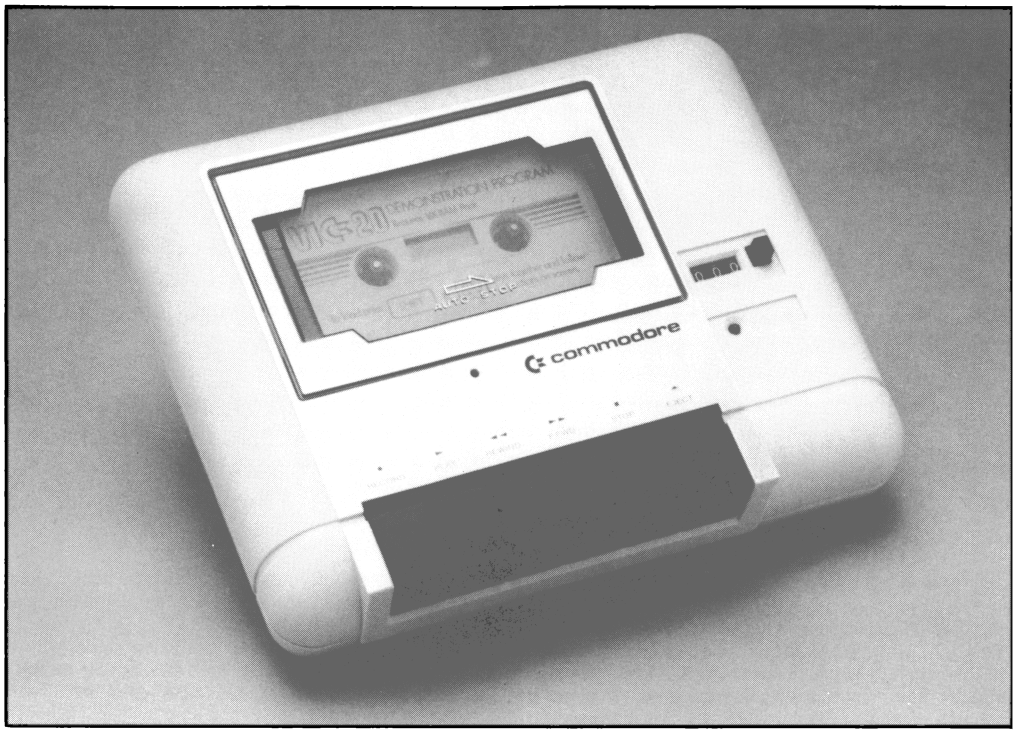
Just for a change, Commodore deserves a round of applause here: it's the only home computer manufacturer I know of that has taken the opportunity to build in this kind of security. And the result is a pretty high degree of reliability when reading something from tape.

That incidentally is the reason for the relatively slow speed of the tape unit: information is actually passed to and from it at about 1,200 bits a second – but everything is actually done twice just to make sure the reading and/or writing is correct: and when you add an allowance for the automatic error detection and correction (the computer can figure out what's wrong if there are minor errors in the passage of data) the overall performance is reduced to a mere 30 or so bits a second.

The fast-loading 'turbo' tapes get their extra speed by tweaking the Kernal routines in the 64 to by-pass the duplication of effort. In theory this should mean more likelihood of load errors; in practice though the Commodore tape units do seem inherently sound anyhow, and my experience suggests that all the effort Commodore apparently put into ensuring high-reliability data transfer wasn't really necessary. Highly commendable, though.

There are two tape players from Commodore that you might come across – Commodore calls them 'Datasettes' and sometimes refers to them as the 'C2N' as well. The current model is the flashy one with rounded edges; you might however find the older version still on sale.

The new one is reportedly not much different internally, it doesn't have any extra goodies, and they both plug in exactly the same way:



The newer Commodore cassette unit is the one with the go-faster styling

There's at least one German-built plug-compatible alternative to Commodore's cassette units. It feels a bit less solid, but appears to perform identically; it's rather cheaper, too, so I wouldn't recommend against it – especially as Commodore seems peculiarly prone to running out of stocks of its own tape players at regular intervals.

Listening to a tape

Because computer cassettes and computer cassette players are identical in concept and design to their audio equivalents, it's possible to hear a program by playing the tape on a normal tape recorder.

The first thing you should hear is around nine seconds of continuous tone. This is a 'leader' tone that the 64 uses to synchronise the reading operations. Then you should hear three seconds or so of buzz; this is a 'header' or 'title' 192 bytes in length which identifies the type of file, where in memory it should be placed, and the name of the file. After a brief pause that should be repeated – Commodore computers automatically store everything twice on tape for security purposes.

That intro is common to all types of file. Thereafter you may get a longish buzz with a brief stop and then another long buzz: that's a program, repeated as before. Or you may hear a series of stop-start three-second buzzes: in that case you're listening to a data file, stored again as even-length 192-byte blocks each of which is repeated before the next one starts.

Cassette SAVEs

You store programs on tape by typing 'SAVE' while you've got something in memory. When you press Return the 64 will tell you to PRESS RECORD & PLAY ON TAPE. You push down those two keys on the tape unit *at the same time*; the cassette should start turning, the tape counter will go round, and the 64 will let you know that it's OK and you're SAVING something (if that *doesn't* happen the cassette unit probably isn't plugged in correctly).

This will SAVE whatever is currently in memory, but it won't give it any kind of identifying name. That you do simply by adding a name in quotes after the SAVE command; the program name can be up to 128 characters long, including spaces and symbols.

The SAVE command doesn't erase a program from memory – it just takes a *copy* of it on the tape. So if you're developing a program and you want to take a 'work-in-progress' copy to ensure that a Murphy's-Law electrical blackout doesn't kill all of your work, you can continue working after the SAVE without having to reload the program.

Subsequently you can SAVE updated versions of the program under the same name, if you want: because each SAVE operates on a different part of the tape it doesn't matter that the names are duplicated. On the other hand using identical names is a bit daft – it will be damned difficult subsequently to locate the most up-to-date version of your opus.

Incidentally, be careful when you press buttons on the cassette unit. The 64 doesn't know the difference between PLAY, FAST FORWARD, REWIND and RECORD. If you're saving something and you're told to 'PRESS RECORD AND PLAY ON TAPE' the 64 won't have the sense to spot that you've inadvertently hit the REWIND key; nor will it check that both PLAY *and* RECORD are down ...

'Write-protection'

If you want to SAVE something and you find it's impossible to press RECord and PLAY, the cassette may not be sitting in the tape unit properly. Hit EJECT and try it again.

Or the cassette itself may be 'write-protected'. All cassettes have a couple of slots on their back side that ordinarily contain a couple of plastic tabs. If you remove those tabs with a small screwdriver you will prevent any recording at all on that tape – they become 'write-protected'.

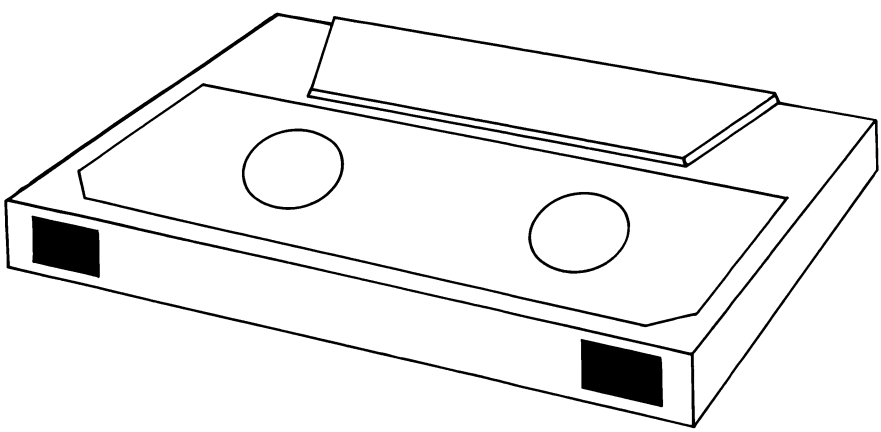
Most commercially-sold tapes have the tabs removed so prevent you inadvertently overwriting their contents, for when you (or someone else) pushes down RECord and PLAY with a write-protected tape in there the RECord head simply won't engage.

If you *do* want to record on to a write-protected tape, you can just put a piece of sticky tape over the holes.

The tape counter

The tape counter lets you know how far into the tape you are. It is good policy to reset it to 000 every time you rewind a tape; there's a little reset button next to the counter that does this.

The counter doesn't measure anything particularly – it doesn't correspond to millimetres or tenths of an inch or whatever. But it does let you know where on that particularly cassette unit your SAVEs start and end. If your first one starts at 000 and finishes at 015, you can take a note of that on the cassette's liner card alongside the name of the program; then you can hit the Fast Forward key to wind the counter on for a bit, say to 020. Press the Stop button and the next program you put on to the tape can then start there; the liner card then provides a sensible, easy way of cataloguing what's on the tape.



Most cassettes have two write-protect tabs on the rear. When these are removed the record head won't engage

Be warned – different tape units wind the counter on a different rates. The counter on the new C2N goes at a different rate to that on the older version, and both are different from that West German look-alike. So while all three should happily read tapes created on any other, you shouldn't expect a note of where a program starts on the tape to be appropriate for anything other than the model on which the SAVE was done.

Verifying a SAVE

Once you've SAVEd something, it is a good idea to check that whatever's now on the tape is the same as what you still have sitting in the 64's memory. To do this, REWind the tape until it comes to the end (at which point it should stop itself and the REWIND button should spring back up – if it doesn't, hit STOP yourself) or until the counter shows you've reached a couple of counts before the start of the program you want to check.

Then you just type VERIFY followed by RETURN. The 64 should tell you to PRESS PLAY ON TAPE; do that, and it should respond with 'OK' and 'SEARCHING'. When it finds a program it will come back with 'FOUND ...' followed by the name of the program, or it will just say 'FOUND' if it finds a program that you didn't name. Then it'll start trying to verify it – which means it will go through the program on the tape comparing it character by character with whatever is currently in memory.

If the 64 can't find anything at all you'll just see the SEARCHING message until the tape runs out. Either you're using a tape with nothing on it or the 64 isn't trying hard enough: press RUN/STOP and RESTORE, rewind the tape, and try the VERIFY again.

When a verification does match the taped program against that in memory the 64 will say 'OK' and 'READY'. Or you'll get a ?VERIFY ERROR message; that might be because the program the 64 found simply isn't the one you want to check, in which case type 'VERIFY' again – if the PLAY button is still down the 64 will just restart the tape and carry on looking for something.

On the other hand, it might be because for some reason the copy on the tape isn't a perfect match for the program in memory. Trying rewinding the tape and saving the program again.

If you don't ask the 64 to VERIFY a particular program name, it will just use the first program it comes across for the verification. Or you can have the 64 look for a specific program to verify: as with SAVE you just type 'VERIFY' followed by the program name – in which case the screen will list every program the operation comes across until it reaches the one specified. (If it finds unnamed programs you'll get 'FOUND' followed by nothing.)

LOADing from tape in Direct mode

To read something into memory from tape you use the LOAD command. In Direct mode the simplest form of the command is simply to type 'LOAD' followed by Return – you can type this out in full, or you can hold down a Shift key and press RUN/STOP.

This will produce the PRESS PLAY ON TAPE message, and pressing the tape unit's PLAY key will start the 64 searching. The screen will go blank – in other words, it goes completely blue – until the 64 finds a program on the tape. It will then flash the message FOUND followed by the program name (assuming you gave it a name when you SAVEd the program). The screen blanks because of a rather silly little technical problem with the VIC chip that the designers probably should have thought of ...

The screen will then go blank again almost immediately while the 64 loads this program.

When LOAD is used without a specified program name the 64 will simply load the first program it finds on the tape. You can tell it to load a particular program by giving a program name in quotes (this way you can't use the RUN/STOP trick – you have to type an identifiable name.

In fact you don't need to give the full program name: you can just specify the first couple of characters, enough to identify the program you want. The 64 will then search the tape for the first program it comes across whose name starts with characters you entered; this will then be LOADed.

Or you can use what are called 'wild cards'. Specifying a file name with one or more characters followed by an asterisk or a question mark will start the 64 looking for the first program that fits the match, as follows:

- the asterisk means 'any character and any number of them – including zero' so LOAD "BB*" will locate the first program on the tape that starts with the characters 'BB', including a program the full name of which is simply 'BB'.
- the question mark indicates 'any single character' and any number of question marks can be used. LOAD "B??" will locate the first program on tape that starts with the character 'B' followed by any two other characters.

LOAD and go

An adaptation of the LOAD command permits an immediate RUN. For this you need the format ...

LOAD "*program name***", 1, 1**

The first '1' there specifies device number 1, the cassette unit; the second is the trigger that tells the 64 to RUN as soon as it's LOADed the program.

Tape directories

There isn't a neat, quick and obvious way of finding out what's on a tape – other than by handwriting the tape's contents on to the cassette inlay card.

A crude alternative though is rewind and try to LOAD a program name that you know the 64 *won't* find on the tape. Then you can take a note of the all the program names it *does* find while it's searching.

LOADing from within a program

LOAD can also be used from within a program. The simplest way to utilise this is to set up a short 'tape contents' program something like this:

```
10 CLR
20 PRINT "<clr>PROGRAMS ON THIS TAPE
ARE:"
30 PRINT "<cud><rvs>1." TAB(10)
"PROGRAM ONE"
40 PRINT "<rvs>2." TAB(10) "PROGRAM
TWO"
50 PRINT "<rvs>3." TAB(10) "PROGRAM
THREE"
60 PRINT "<rvs>4." TAB(10) "PROGRAM
FOUR"

100 INPUT "WHICH NUMBER DO YOU WANT"; N
110 LOAD N
```

Used in Direct mode the LOAD command wipes everything clean – all variables are reset to zero, all DEF FN functions are killed, all arrays are cleared, and any OPEN files are 'forgotten'.

But when LOAD is from within a program you *won't* lose all those settings: variables retain their last values, all DEF FN functions are still present, arrays are still in operation, any OPEN files remain open. You also get an automatic RUN when the new program is LOADed.

The implications of this are obvious: if you want the second program to use the same variable names but not the same variables values, the program to be LOADed should start with a CLR. On the other hand, you might want those values – you can separate long programs into discrete lumps, save them as separate programs, and then LOAD each section one at a time to perform specific operations with a common set of variables and the same OPENed files.

This is called 'chaining'.

The only real problem with chaining lies in the way the 64 allocates memory to a program – and more specifically to the arrays and variables the program sets up. What happens when you RUN a program is that the code itself is dropped into the lowest point of the memory area available to you; and immediately following that are two chunks of memory assigned to hold current values for the variables and arrays. The bigger the program, the further along in memory those variable and array areas will be.

This means that if the first program – the one which includes the LOAD command – is larger than the second, there shouldn't be a problem: the new program will overwrite some of the program that's already there, and so it will fit into memory below the existing variables with space to spare. (That space will actually still have some of the original program's code in it, so your second program would do well to include an END statement so that it doesn't inadvertently run on into the left-over garbage when it's being executed.)

On the other hand, the drawbacks are obvious if the second program is larger than the first; when the 64 puts it into memory it won't adjust the start of the variables area. So the longer program will overwrite some of the variables.

You've got very little choice here: you've got to force the 64 to rethink the location of the variable and arrays areas. There are two ways of doing this, one of which is pretty tedious – before the second program is loaded you must find out where those areas are and read the contents of that memory (probably by converting it into DATA statements). Now the LOAD can proceed; and then you can drop the temporarily stored contents of the original variables and arrays into memory immediately above the newly-resident code.

The easier way is simply to decide how big your biggest program is (you can chain together any number of programs) and set everything to point to the maximum possible memory occupancy. This involves some POKEs to the relevant pointers:

How the 64 stores a program in memory

boundary pointer addresses

Start of Basic program 43 and 44
Start of program variables 45 and 46
Start of arrays 49 and 50

First you'll have to determine the largest size of program you need. One method of doing this is quite clever: program files all start with a header block, the first byte of which decides respectively the file type (1 for program files, 4 for sequential files). The next two give the start address in memory from which the program was SAVEd, the next pair give the end address.

You can then write a short program something like this to check the memory occupancy of programs on tape:

```
10 INPUT "PROGRAM NAME"; PN$
20 OPEN 1, 1, 0, PN$
30 GET#1, A
40 IF A <> 1 THEN PRINT "SORRY, NOT A
PROGRAM FILE"
100 GET #1, B1
110 GET #1, B2
```

```
120 GET #1, C1
130 GET #1, C2
140 CLOSE 1
200 PRINT "BOTTOM OF PROGRAM MEMORY = "
B1 + (B2*256)
210 PRINT "TOP OF PROGRAM MEMORY = " C1 +
(C2*256)
300 GOTO 10
```

Do that for all the programs on your tape and you should end up with a list of memory requirements.

Alternatively there's the simpler way – you can LOAD each program in turn and PEEK at locations 45 and 46. Between them these indicate the start of variables for the program currently in memory. Type this in Direct mode ...

```
PRINT PEEK(45) + PEEK(46) * 256
```

... and take a note of the result. Do this for each program you want to chain. Now pick out the highest start point for the variables and use this for two calculations, again in Direct mode:

```
PRINT INT(number/256)
PRINT number-256 * INT(number/256)
```

Again, note the results – you're about to use them to POKE the relevant pointers. That you can do by including this bunch of instructions at the start of the first program, the one containing your first LOAD:

```
10 POKE 45, second result
20 POKE 46, first result
30 POKE 47, second result
40 POKE 48, first result
50 POKE 49, second result
60 POKE 50, first result
```

Resist the temptation to use variables in order to do the necessary calculations within the program – these POKEs *must* appear before any variable or arrays are defined, since they decide where the 64 will store variables and arrays.

They need appear only once, at the start of the first program in a chain.

Data files

The SAVE, LOAD and VERIFY commands all work with *programs*. The only other kind of file that you can set up on tape is a *sequential* (or *serial*) data file. To create a sequential file the command format for the OPEN is ...

OPEN *file reference number*, **1**, *secondary address*, *"file name"*

The file number is as has been described: you'll see that most programmers using it just opt for '1' so they don't have to work their imaginations or their fingers too hard. The device number is '1' for cassette; the secondary address is '0' for read-only files, '1' for write files, '2' for write files that will get an end-of-file mark appended automatically to them when you subsequently use the CLOSE command.

The OPEN command will also check that you've pressed the right buttons on the cassette unit, and if you haven't it'll tell you to do so.

Once the file has been OPENed for writing, PRINT# statements can be used to put something into it. The CLOSE must then be used – otherwise the last PRINT# items won't get on to the file (and if you specified an end-of-file mark in the OPEN that won't get on either).

One other helpful attribute of the end-of-file mark – it will stop the 64 looking for a program to VERIFY or LOAD. So you can use it as the last thing currently on the cassette to prevent the machine grinding all the way to the end of the tape as it looks for something.

For reading the file GET# and/or INPUT# can be used as described in Chapter 18.

If you want to check the creation of a data file to make sure that everything's going to appear on tape as you want it, there's one fairly obvious method – convert all your PRINT# statements to PRINTs so that the data appears on the screen. A bit tedious, though, all that stripping of hash characters: the simpler way is merely to change the device number in the OPEN statement from 1 (the cassette) to 3 (the screen).

In much the same way you can also mimic the read operation to see that it's working by switching the device from tape (device 1) to the keyboard (device 0). INPUT# will work the same way, with the program now expecting input to be typed rather than encountered on tape. You won't get the question-mark prompt that you'd expect with a normal INPUT, though: instead there's just a blinking cursor. So it might not be too easy to do this.

Checking the STatus variable

Cassette read and write operations are generally reliable on the 64, but in any case Commodore provided the built-in ST variable which can be checked from time to time to see what if anything has gone wrong.

It's main use is probably to check for an end-of-file mark (ST = 64, or more reliably ST AND 64 = 64). But in addition an INPUT# or GET# operation should normally include a subroutine to check on the status of ST – ST is normally zero, so you jump to an error-handling routine if ST > 0.

The most common cause of ST-detected problems is reading a program file as a sequential file, in which case ST is likely to hold the values 4 or 8 (test for ST AND 4 = 4 and ST AND 8 = 8). The other likely hiccup will come when the automatic error-correction facilities can't cope, in which case ST may well have the value 16 ('unrecoverable read error') or 32 ('checksum error').

Tape problems and caring for cassettes

- **Buying tapes**: you may come across special premium-price tapes that are supposed to be special 'data' cassettes for computers, but in fact any decent-quality audio cassettes will do.

- **Buy short tapes**: C15s or C30s are preferable, for longer tapes are thinner and they are more prone to stretch with use: that can cause read errors since the data will no longer be in precisely where the 64 expects it. Besides, with shorter cassettes it doesn't take so long to fast-forward to the program you want to load.

- **Tension your tapes**: run a new tape through a tape player by fast-forwarding it and then rewinding it. This shouldn't be necessary, but it evens out the tension on the tape and will help prevent SAVE errors: you can use the 64's cassette player or any audio cassette deck for this.

- **Wind cassettes past the leader tape before a SAVE**: a fully rewound cassette may have quite a lot of non-recordable leader tape at the start. Now, when it starts a SAVE the 64 allows about nine seconds of non-SAVE time while it synchronises things: but many proprietary tapes have ten seconds or more of leader. You can get leaderless tapes in some computer shops.

- **Store your cassettes sensibly**: keep them away from heat, dust and damp – the kind of plastic storage racks you can buy in record and hi-fi shops are fine.

- **Keep tapes away from strong magnetic fields**: that can screw up anything that's recorded on the tape. This prohibition includes your TV, by the way; televisions sets produce quite a strong magnetic field. The tape player itself shouldn't be too near the TV for the same reasons.

- **Clean the record/playback heads from time to time** by using one of the proprietary tape cleaners that you get get from your local record store. These are generally cassettes containing a mildly abrasive tape that scrubs the accumulated debris off the parts which actually touch the tape.

- **Demagnetise the heads occasionally, too**: rubbing against a surface like the tape does cause a little magnetism to build up on the heads, and that can lead to errors. To demagnetise them you buy a little cassette-like thingy that runs off a couple of watch batteries: this you just insert, press PLAY, and leave for a few seconds. Alternatively I've seen hand-held gismos that you plug into a wall socket and point towards the tape unit heads (keep all pre-recorded tapes well away unless you want to wipe them too). Again, a record shop should be able to sell you either kind of demagnetiser.

CHAPTER 20
Disk

Any computer needs a relatively inexpensive and certainly a convenient method of storing programs and information; and all computers need programs to operate, they also need an inexpensive and convenient medium on which those programs can be supplied and sold in the first place.

The cassette, which looks like (indeed is often exactly the same as) an audio tape cassette, happens to be a cheap and easy way of storing programs and files. So it's a natural for the starter 'hobby' computers.

But disks are generally better suited to 'professional' computing. They're faster in use, they are more flexible (literally, in the case of floppies), they permit much better and much more versatile means of storing files.

Tape has one major drawback: it's a 'serial' medium – to read a record at the wrong end of the tape you have to pass over everything in between, checking to make sure that it's not what you want.

Disks can hold sequential and program files, just like cassettes. Their virtues there come down to speed of access: not only do you have to spend less time waiting for SAVEs, VERIFY operations and LOADs – it's also feasible to chain several programs with an overall performance speed that the cassette equivalent just can't match.

How disks work

Disks work rather like LP and 45rpm records. They are coated with a magnetisable skin on to which information can be encoded, just as with computer tape. The key difference is that tape is pulled past the bit that gets information to the surface and reads from it, the read/write head; by contrast a disk spins around its centre (at about 300 rpm) and the the head is positioned on a movable arm which can move over the disk from the edge to the centre and back again.

So the disk can be turned until it's in the right position, and the head can then be moved until it's over the track containing the required information.

The disk then might have to turn through no more than one revolution to get the right record or the start of the specified program directly under the head. It's a bit like finding the start of a song on a record.

That in itself is a much quicker approach than the serial medium of tape; disks also work more quickly and more reliably because the head can be engineered specifically to fine tolerances since it's operating with digital signals – cassette read/write heads derive from the analogue technology of audio signals, and the same level of performance and reliability just isn't possible.

But disks also take things a stage further. The information is encoded on 'tracks' which are much like the grooves on a record (except that the grooves spiral outwards from the centre of a record, the computer disk's tracks are concentric rings). In addition, the tracks are divided into wedges called 'sectors'; one track in one sector is a 'block'.

You can't usually *see* the blocks, tracks or sectors, because they're not *physically* there. Instead they're delineated by the magnetic encoding placed there by a read (a SAVE) operation; that indicates the boundaries, along with very precise mechanical positioning of the read/write head (such that it knows where to expect the sectors).

So if the read/write head can operate with sufficient precision to find the sectors and tracks, it can move quickly to the right block of data more or less every time. And it doesn't have to pass through everything else on the disk to get where it wants to go. Even if you're storing sequential files on disk, the contents of which have to be read serially, the files themselves can be located *anywhere* on the disk's surface – there's no need for any equivalent of rewinding a cassette to ensure that the read starts at the beginning of the tape and is therefore guaranteed to pick up anything on the tape.

This ability to move to a particular part of the disk demands quite precise engineering of the 'disk drive' – the unit which spins the disk and also incorporates the read/write head.

And it also requires some considerable cleverness both in the electronics and the operating system to tell the head where to go for a particular block. In this case of Commodore's disk drive, said cleverness resides inside the drive as a separate microprocessor with its very own control software (8K bytes of it).

Floppy disks

There are several types of disk for computers. But the least expensive and the simplest are floppy disks – also called 'diskettes' (originally an IBM term, in fact) or (less formally) 'floppies'.

These are thin, flat, flexible discs of coated plastic; they come in a more or less rigid square envelope of cardboard or plastic, and you *never* remove them from the envelope – which is just as well, for a floppy disk is *very* floppy.

The cheapest floppy disks are currently the so-called 'mini-floppies' measuring 5.25in diameter. These are the disks that the Commodore disk drives use; and they actually use the simplest and least expensive variety of floppy, too. You shouldn't have to pay more than about £2 for a disk and by shopping around you should be able to find boxes of ten for around £16 or less.

Into the envelope are cut a central circular hole that exposes a rim of disk (the hub of the disk drive fits into that to spin the disk) and a longish slot showing more of the disk's surface (this is where the read/write head actually makes contact with the disk to transfer information to and from the computer).

Near the central ring is a smaller hole, the 'index hole', which provides a reference point for the read/write head to enable the controlling software to find the right sector.

There is also a squarish notch in one edge of the envelope. This is the 'write-protect' notch; if it's covered you can't 'write' on to the disk, which means you can't overwrite anything on it or add new files or programs.

The 1541 disk drive

The original disk drive sold with the Vic and 64 was called the 1540. The present model is the 1541, a mildly reworked version that looks and works more or less identically but which incorporates some extra features that allow it to be used with the Commodore 64 as well as the Vic.

Cutting corners

If you're really trying to cut costs with your single-sided 64, you may be tempted to 'flip' your floppies and use the underside. For that you'll need to cut another write-protect notch on the other edge of the disk – you can buy special tools to do this, but some care with scissors or a scalpel on the kitchen table will have the same effect.

In theory you'll then be able to double the storage capacity of the disks you buy and half your diskette costs.

In practice it may not work satisfactorily. There are a couple of brands already sold as 'flippies', with machine-cut notches; and in my experience they are prone to errors in reading and writing on *both* sides, particularly after prolonged use.

Why? Well, one technical argument is that the pressure pad opposite the read/write head starts to accumulate magnetic particles anyway since it's pressing directly on the disk surface: and that can both destroy any information stored on that side while causing enough deterioration on its surface to prevent satisfactory writing to it.

This corner-cutting doesn't seem worth the effort or the risk.

There are at least two independently-manufactured disk drives from the States which appear to emulate the 1541 pretty comprehensively (meaning they react identically to commands and perform much the same as the 1541 – and both of them store more than the 1541's 150KB or so).

You can also use other Commodore disks, notably the twin-drive set-ups sold for the 700 and 8000 lines; they offer more capacity per disk, possibly faster access, and probably greater reliability. You can't plug them straight in, though; you'll need to buy an IEEE adapter.

Let's stick with the 1541.

Disks should be inserted gingerly with the write protect notch on the left and the label facing upwards. Push the disk in as far as it will comfortably go, then flip down the latch to lock it into place. The disk won't start spinning THE 1541

The 1541 is completely software-controlled, which means a whole new set of commands and operating procedures for the user. They're provided in what's called an operating system – operating systems are chunks of more or less invisible software that sit between the user's programs and the hardware; all

Buying floppies

There are dozens of brands of floppy disk on the market. In fact there are only a handful of manufacturers of the disk surfaces themselves; they tend to supply in bulk to the people who put the disks into envelopes and check them out.

But you do have a very wide choice of suppliers, particularly as you'll probably be buying at the retail end of the chain – in a volume business like this the major suppliers won't sell direct. Most retailers stock disks from several different suppliers.

So which ones do you buy?

For a start you want disks for a Commodore 64. Most home computer retailers will know what disks will work in a 64 – their suppliers give them reference tables and charts to identify the disk requirements of particular machines – but if in doubt ask for single-sided single-density 35 or 48 tpi disks ('tpi' stands for 'tracks per inch'; 35 tpi is the layout 64 disks use, 48 tpi disks are more common and will work as well).

Then you can start shopping around. The differences in price between various disks and various retailers generally come down to a number of factors:

- **Quantity:** buy in bulk and your disks will be cheaper. Disks are usually sold boxed in tens, with discounts starting at five or ten boxes per order. So anticipate your needs over the next six months or so if you can – provided you don't store them in daft places like the boiler room, floppy disks will keep for a long while (the boxes are usually shrink-wrapped or sealed in cellophane anyway).

Group purchase is another obvious option; it also works for other consumables like paper and ribbons for printers. To my mind this ought to be one of the *raisons d'etre* for user groups and clubs. And if there are other 64 users in your area, why not suggest a bulk purchase to them?

- **Type of retailer:** 64 dealers and other computer shops stock consumables like floppies to be able to meet all the needs of their customers. But they don't expect to make a huge amount of money out of the sale of the odd box of disks here and there.

There are however specialist consumables suppliers, generally working from catalogues with a lot of choice in them. Orders are taken by telephone or mail and despatched by post or by van. They *are* geared up for volume sales, they buy in bulk themselves, and they can usually do the best prices on things like disks. Some insist on cash with order, many will take credit card orders; or most will bill you retrospectively if that's what you want, and some run account systems for customers.

- **Packaging:** the cheapest disks come in the flimsiest boxes, they have the smallest and plainest labels, their write-protect tabs are the least sticky and cover the minimum area. The dearer disks will have a more solid box, maybe good enough to use regularly as your disk file; and some will actually arrive in a purpose-designed flip-top stand-up plastic disk file rather than a cardboard carton. The labels will be bigger, may be colour-coded, could have helpful panels for identification, date, operating system and so on. The write-protect tabs will be big and super-glued.

All of that may be cosmetic ... but then it may not. A decent-sized label that actually helps you organise your work is no bad thing; a tiny sticker that flakes off within hours and doesn't easily cover the write-protect notch can drive you nuts; a solid box, especially a plastic case, may be useful storage for you.

- **Other extras in the disk's manufacturing:** you may be paying more for additional goodies on the disk.

Chief among these is a 'hub reinforcement'. The disk drive is constantly ramming a metal hub in and out of the hole at the centre of the disk, and if the disk is marginally off-centre the result *could* be deterioration around the lip of the hole – seen at its worst as a wavy edge, with the result that the disk no longer seats correctly on the hub. That could cause misreads. So some disks come with extra reinforcement around that lip, generally in the form of a white ring (their manufacturers like the additional benefit to be obvious!), that in theory minimises the strains put on to it by the disk drive. But on the other hand, if you insert your disks correctly (which means gently and with care) they should centre themselves automatically anyway ...

Jacket liners that do some cleaning of the disks as they turn inside the envelope are however a good thing. A lightly abrasive coating on the inside there can pick up some of the contamination that would otherwise accumulate on the read/write heads and effectively impair their operation. These self-cleaning coatings can't pick up everything, though they are handy ... but not vital, particularly not if you get into the habit of running a head-cleaner diskette through your drives every so often.

- **Specification:** finally, and most difficult to assess, there's the question of how rigorous the manufacturing has been. Obviously all manufacturers want to minimise their overheads, which translates into using the least possible coating for the disk and the fewest possible checks on it. A thicker coating means more regularity and thus more reliable data storage; more rigorous testing means more discards.

A common method of manufacture is to test all disks for the highest quality required (96tpi 80-track double-density on both sides); failures are then dropped down to testing at successively lower grades until they find an acceptable level. Or until the they flunk out even as single-sided single-density 35-track disks ...

Nearly all disks will proclaimed themselves to be 'Certified 100 per cent Error Free', but different manufacturers use different methods to figure that out. Usually it means writing to and reading from every possible bit position on the disk; sometimes though that's done once only, other quality control processes may involve several repetitions. It's widely reported that one of the best-known medium-price brands is actually made up of rejects from one of the higher-priced names, having been retested successfully at a lower grade of verification.

There's not much you can do to check out the claims for quality. A certain amount of writing-off-to-experience will probably have to go on; sure, you can complain if you get persistent errors and failures from a box of cheap disks – and you may even get your money back. But you may have lost valuable data. And anyhow the cause might have been your disk drives, not the disks themselves. Or it might have been a dud batch, with the norm for that brand being acceptable.

computers have them, even the 64. It's called the Kernal in the case of the 64, but one of the clever aspects of the 64's design is the fact that you need never know it's there.

The operating system that handles the disk mates up with the Kernal to provide all the disk management that's needed. You'll see the kind of things it does as this chapter goes on. The point to make here is that efficient use of the disk demands a certain amount of additional knowledge – it's hard just to plug in and go.

Getting started

Floppies don't usually come set up for a particular computer – they're mass-produced for sale to a variety of users with a variety of machines. They have to be *formatted* specifically for your 64, which in practice means the arrangement of tracks and sectors has to be laid on to the disk along with some control information whereby the 64 will subsequently recognise the disk as one of its own.

This introduces an alternative use of the OPEN command, the use of which I've hitherto restricted to files. Many of the operations specific to the 1541, like disk formatting, utilise the OPEN statement like this:

OPEN *file reference number*, **8, 15**

Here there's nothing in quotes at the end of the statement. Device number 8 is the disk unit, secondary address 15 indicates access to the disk's own operating system. The '15' isn't mandatory, but you probably ought to treat it as such since the alternatives lead you into constrictions and complications by the score.

Once you've OPENed a link to the disk drive you can use a form of PRINT#:

PRINT# *file reference number, "command"*

Alternatively there's a handy shorthand form that dispenses with the need for a separate PRINT#:

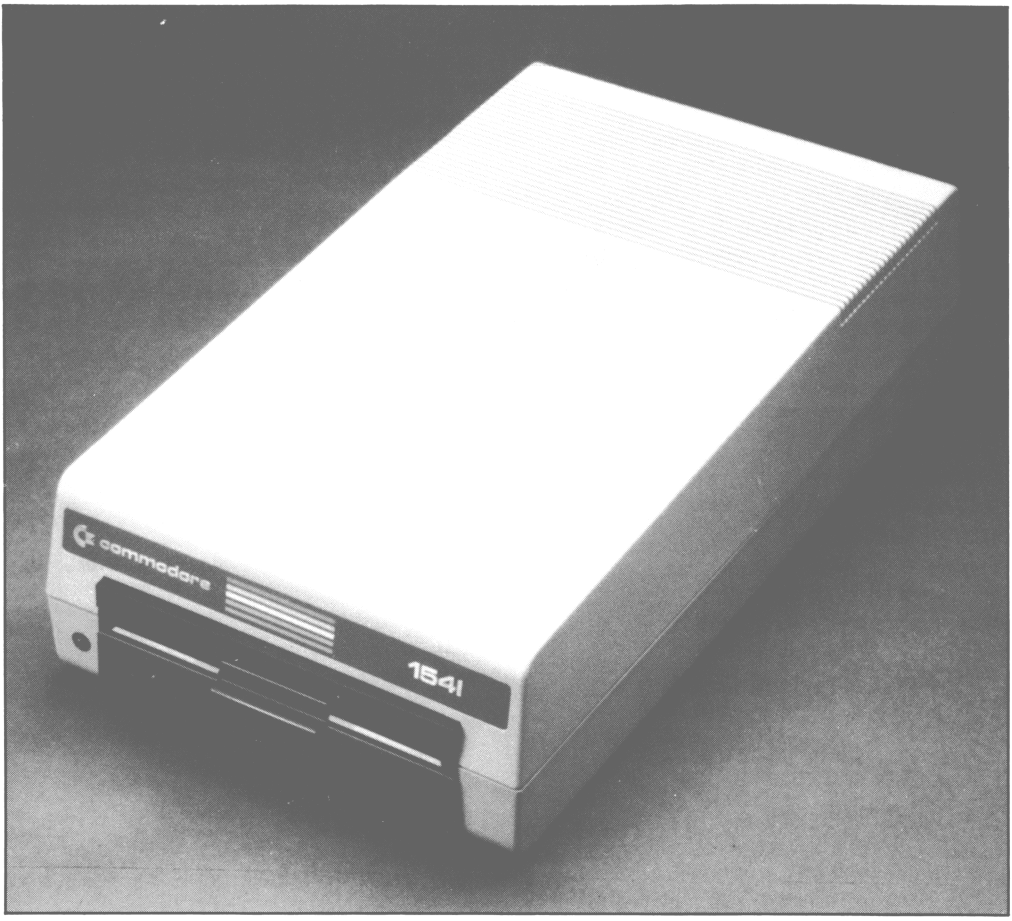
OPEN *file reference number*, **8, 15,** *"command"*

For the 1541 the format command is NEW. It's generally used in Direct mode, and it takes this form:

NEW 0:*disk name, identifier*

The word 'NEW' can be abbreviated to 'N'.

The '0' in there denotes which disk drive you're referring to – in theory several disks can be 'daisy-chained' on to a 64, and while there's a perfectly good method of referencing them in the form of the device number (device 8 is the first disk drive, 9 the second, and so on) Commodore elected to complicate things by calling the first disk 'drive number 0' and the second 'drive 1'.


commodore
1541

In theory this drive identifier can be omitted; in practice you might find that the command won't be executed if you do this – certainly on the SX-64 you need it there. The colon is very definitely mandatory.

The 'disk name' is a name of your choice, up to 16 characters long. This too is optional in theory ... and in practice the command might not work if you do leave it out. That also applies to the 'identifier' suffix, one or two characters that you tag on to the end of the disk's name. Both the name and the ID suffix appear at the head of the directory of files on the disk; they're also used internally by the 64 and the 1541 to check which disk they're using.

So to format a new disk for a miscellaneous collection of work-in-progress projects the command would look like this:

```
NEW 0:WIP,01
```

Here's an example of the shorthand form, formatting a disk destined to hold word processor texts:

```
N0:WP TEXTS,AA
```

Program files on disk: the SAVE command

To save, load and verify programs on disk the same commands apply as for cassette operations – except that you have to specify the device number (8) after the program name.

Here's an illustration of the SAVE command:

```
SAVE "ACCOUNTS",8
```

To SAVE a new file under the same name as an existing program file on disk you precede the file name with the characters '@0:' –

```
SAVE "@0:ACCOUNTS",8
```

That tells the 1541 to replace any existing file of the same name with whatever's currently in memory.

It's a good idea to tack the '@0:' on to the front of *any* SAVE to disk, in fact, just to be on the safe side – if you omit it and there happens to be an identically named program on the disk already, the SAVE will appear to proceed quite happily ... except that the existing program *won't* be replaced.

And you won't be told about that; the red light on the disk will start blinking, which is supposed to indicate some kind of disk error, but since it seems to blink most of the time anyhow that is not a particularly reliable indicator.

On the other hand there is a major problem with the save-and-replace command: it seems that the SAVEd program may overwrite existing records on a disk if it's a bit larger than the original. The only solution is the laborious one – delete the original and SAVE the replacement conventionally.

Here one positive tip, though: if you want to prevent the accidental (or deliberate) deletion or amendment of a program on disk, SAVE it with a Shifted Space as the first character of the name – in other words, type SAVE and the opening quotes: then hold down a Shift key and press the space bar.

The curious result is that the program in question will appear in the disk directory with *two* leading quotes marks rather than one; and it will be impossible to delete or replace. Trying to delete 'PROGRAMNAME' produces a FILE NOT FOUND message; an attempt to delete "PROGRAMNAME' will give an ILLEGAL QUANTITY error.

You can get at it only by including the Shifted Space in the name.

VERIFY and LOAD

VERIFY works similarly to SAVE:

```
VERIFY "PROGRAMONE",8
```

And so does LOAD:

```
LOAD "ACCOUNTS",8
```

Again, wild cards can be used. So can two other special characters:

```
LOAD "*",8
```

This will LOAD the first program on the disk. And the dollar sign will LOAD the disk directory:

```
LOAD "$",8
```

Which is a bit odd, because the directory is a list of what's currently on the disk ... and LOAD loads *programs*. Once the directory is in the 64's memory, you'll have to LIST it just as you would a program in order to see the directory.

More than a bit odd, actually – more like darned irritating, because the other program-like effect is that the LOADed directory will overwrite anything currently in memory. This means that while you're working on a program you can't take a look at the disk directory without first saving your work-in-progress.

The solution is the so-called 64-WEDGE supplied with 1541s and the SX. This is a small machine-code program that you can LOAD from disk and RUN as your first action of the day. It locates itself well away from everything else in memory and effectively adds a couple of commands to Basic; they let you read and view the disk directory whenever you want without having to overwrite the program at present in memory.

Detecting errors

Commands and Basic statements that produce an error will also produce an error message – not necessarily a particularly *helpful* message, but a message of some kind nonetheless. You'll be able to detect that something is wrong.

Errors at the disk are not handled so well. Sure, for some of the more obvious problems you will find that the 64 itself traps things like SYNTAX ERRORs; and if you ask the disk to LOAD a program that isn't there you will be told. But for the disk drive's *own* commands the only visible indication of an error is the blinking light – and that blinks a lot anyhow.

But the disk drive itself does have a collection of error messages, and in most cases these are commendably explicit. The problem is that you'll have to write a subroutine into your program if you want to *see* the message when a problem crops up. The messages don't appear automatically: you have to go and get them.

If you OPEN the disk's 'command channel' and subsequently use INPUT# to read four variables from it you'll get a summary of the situation. The variables can have any names, but they must in the right order:

- **a numeric variable** which holds the error number (a full list of error numbers is included in the 1541 manual)
- **a string variable** containing the message
- **a numeric variable** giving the track number at which the error occurred
- **a numeric variable** giving the sector number at which the error occurred

It's good policy to include a subroutine like this that you can call whenever your program OPENs a file or reads from or writes to it – here I'm using fairly explicit variable names:

```
1000 OPEN 15,8,15
1010 INPUT#15,EN,EM$,ET,ES
1020 IF EN<20 THEN RETURN
1030 PRINT "ERROR NUMBER:" EN
1040 PRINT EM$
1050 PRINT "TRACK:" ET,"SECTOR:" ES
1060 RETURN
```

Note that error numbers don't start below 20, so line 1020 just jumps back with no further action taken.

More disk commands

So far we've covered a couple of the 1541's own commands – NEW for formatting, and the special form of OPEN for the command channel to check for errors.

The rest of them pretty much follow the format of NEW. In particular, they can all be abbreviated to their initial letter: and they can all be used as part of an OPEN command (the PRINT# style given here assumes a preceding 'OPEN 15, 8, 15' somewhere).

- **COPY:** The command has the format **PRINT#15, "COPY 0: NEWFILE = OLDFILE"**. It makes a second copy of the file OLDFILE and names it NEWFILE. If there's already a file with that new name, you won't be given any clue about the failure of the operation apart from the blinking light.

COPY can be used to concatenate several files, meaning you can create one new file name containing copies of several existing files:

```
PRINT#15,"COPY 0:NEWFILE=FILEONE,
FILETWO,FILETHREE"
```

The contents of the copied files will appear in NEWFILE in that order. Note that COPY can't be used to take a complete back-up copy of a disk; there isn't any built-in command for that.

- **INITIALISE:** The 1541 manual advises you to "initialise" a disk every time you load it for any reason. This is a precaution rather than a necessity. The format for the initialise command is **PRINT#15, "INITIALIZE"** (note the US spelling if you're going to use the command in full). It's only really necessary after a disk drive error; then the 1541 may get the sulks and decline to handle disk commands as it should.
- **SCRATCH:** This one deletes a file from the disk via the format **PRINT#15, "SCRATCH 0: FILE1"**.

You wipe out several files at once by separating the names with commas. That's the theory anyhow, though in fact you might find SCRATCH tends to get weary after deleting two or three and gives up.

If you find that the save-and-replace procedure using SAVE "": ..." is wiping out files you wanted to keep, a neat and potentially useful little trick is to get into the habit of appending a delete-and-save statement at the end of your program. For instance:

```
9999 END
10000 OPEN 15,8,15,"S0:OLDPROGRAM":
CLOSE 15:SAVE "NEWPROGRAM",8
```

Then when you want to SAVE an updated version of the program you can just RUN 10000.

- **RENAME:** Predictably enough, this renames a file. As with COPY the new name comes first, which may not be the natural way to do it (you'd probably prefer to rename OLDFILE as NEWFILE); but then that's the way computer people have conventionally done things. The format is **PRINT#15, "RENAME 0: NEWFILE = OLDFILE"**. Again, you won't get any warning other than the blinking light if the new file name already exists.

- **VALIDATE:** A housekeeper that will delete any files on the disk that haven't been properly CLOSEd and generally rearrange things to make the most efficient use of the space available. In particular, after a lot of SAVEs and SCRATCHing the disk will become cluttered with unused data blocks that aren't actually large enough by themselves to hold new files; VALIDATE re-orders the files on disk to optimise the space available. The format is **PRINT#15,"VALIDATE"**.

Sequential files on disk

Data files on disk work much as with tape: you OPEN a file and use PRINT# to put something into it, to read it you OPEN the file again and GET# or INPUT# from it.

The principal difference is that the secondary address is not now used to indicate the type of file (read or write). Instead you get the fuller form of the filename with two suffixes:

OPEN *file reference number*, **8, 15,** *"file name, file type, read/write indicator"*

The 'file type' will be a single character – most often 'S' for sequential files or 'R' for relative files. 'File type' will be the single character 'R' for 'read', 'W' for write. Those commas are mandatory.

One proviso about the recommendation to use secondary address 15 as a matter of course applies here: if several sequential read files are to be OPEN at the same time, each must have a unique secondary address. Since the file reference number must also be unique it makes sense to use the same number for both:

```
OPEN 15,8,15,"FILENAME,S,W"
```

You can automatically create a new version of an existing file – to write a new file with the same name, in other words – by including the replace symbol '@' in the OPEN and subsequently using PRINT# to fill it:

```
OPEN 15,8,15,"@0:FILENAME,S,W"
```

If you want to *add* information to an existing file you should have an existing file OPEN for reading and writing at the same time. If the new material is already on the disk, the approach would be something like this:

```
10 OPEN 1,8,1,"@0:OLDFILE,S,W"
20 OPEN 2,8,2,"OLDFILE,S,R"
30 OPEN 3,8,3,"NEWFILE,S,R"
40 GET#3,NEW$
50 PRINT#1,NEW$
60 IF (ST AND 64)<>64 THEN 40
70 GET#2,OLD$
80 PRINT#1,OLD$
90 IF (ST AND 64)<>64 THEN 70
100 CLOSE 1:CLOSE 2:CLOSE 3
```

Here we're OPENing OLDFILE twice, once to create a new version of it as a sequential write file and once to read it. The GET# in line 40 starts by reading from the NEWFILE and putting data into the new version of OLDFILE; line 60 is a test for end-of-input. When the last item has been read from NEWFILE the program moves on to put the original contents of OLDFILE into the revised version of OLDFILE, so that they follow the data put there from NEWFILE.

This should work, too – it takes input from the keyboard and puts it at the start of the new OLDFILE:

```
10 OPEN 1,8,1,"@0:OLDFILE,S,W"
20 OPEN 2,8,2,"OLDFILE,S,R"
30 INPUT "NEW DATA";NEW$
50 PRINT#1,NEW$
60 INPUT "MORE";M$
70 IF M$="Y" THEN 30
80 GET#2,OLD$
90 PRINT#1,OLD$
100 IF (ST AND 64)<>64 THEN 70
110 CLOSE 1:CLOSE 2
```

Relative files

Sequential files are all well and good, but they can be slow and clumsy if you want to access a particular record quickly from a large file of data items. A relative file is one in which you can do just that.

The price you pay for this pinpoint accuracy is the set of rules and restrictions you have to use to construct the file. In summary:

1. You must have two OPEN statements – one for the disk drive's command channel ('OPEN 15, 8, 15' with no other parameters) and one for the file itself.

2. The OPEN for the file must have a particular file name format: the name has to start with a colon and be followed by a comma, the letter L, and another comma – with nothing else before the closing quotes.

3. The OPEN must terminate with a record size indication in the form "+ CHR$(*number*)". This is where it starts getting complicated; records in a relative file consist of a fixed number of characters (that's the principal restriction, for you might not fill up every record with data – yet *all* records have to be big enough to accommodate the largest amount of data you might put in there).

You calculate the size of the record by summing the size of each separate item (or 'field') that you'll have in the record, plus 2 characters for each field (one for a space between fields, one for a Return to delimit each field). In short ...

record size = sum of field sizes + (2 x number of fields)

How to handle floppy disks

Most problems with disks – which means most faults in writing information to or reading it from the disk, usually detected only when you try to read something – are due to duff disk handling. In other words, it's *your* responsibility rather than the disk's or the disk manufacturer's.

It's practically impossible to follow all the rules for looking after your floppy disks. Somehow they seem to proliferate across the desk, and somehow it seems to be much quicker to swap disks without putting the previously used one away properly ...

This however represents an ideal-world list of recommendations.

- **Take copies of all programs – as soon as you get them.** You certainly ought to take more than one copy of any program you buy and anything that comes with the computer when it first arrives. For business use, at least three copies are recommended: one to use on a day-to-day basis, one to get at quickly when you pour coffee over your working disk, and the original – which you ought to keep somewhere safe, well away from the computer. The problem here is that copying disks can be a real drag when you only have one disk drive, but don't shirk it.

- **Take backup copies of all working disks you have – frequently.** A good habit is end-of-the-week copying to duplicate all working disks that you've used during the day. And as soon as a disk is full, or when it contains information that you won't want to alter, take a copy of it and keep the duplicate somewhere safe away from the computer.

- **Use the 'write-protect' stickers.** These prevent any changes to the files on the disk; you can read the contents, but you can't save anything on to the disk – so you can't overwrite any file on it already. All your 'master' disks should get one – these are the originals of programs and your 'stable' working disks, the disks you'll be keeping well away from the computer.

- **Put related programs on to the same disk.** That way you won't get too confused about what's where and which version of a file is the most up-to-date.

- **Don't try to cram too much on to a disk.** Disks aren't really all that expensive, after all, and the cost of a run-on disk counts for little against the hair-tearing aggravation occasioned by your unthinkingly trying to save something on a disk that's already full.

- **Label your disks.** All boxes of disks come with a pack of sticky labels – they tend to get lost as soon as they arrive, so keep them somewhere handy. Use the labels sensibly: the kind of information you might put on the labels would be a summary of the disk's contents, the program that's used with it if it's full of data files, the date it was last copied, and so on. If there isn't room for all that on one label, stick a second on the disk somewhere: I tend to use a second label to keep a record of the backup copies taken. You could also write or stick labels on the disk sleeve, but sleeves have a habit of wandering away from the disk they originally belonged to. Those metallic-ink felt-tipped pens you see in stationers and art supply shops can be used to write indelibly straight on to the black disk jacket, too, though *don't* press too hard and *don't* try this with any other type of pen. And *don't* use pencil or biro to write the label if you've already put it on to the disk – use a felt-tipped pen, and go gently. Of course, it makes sense to write the label *before* you stick it on ...

- **Stack your disks sensibly.** The most common problems with disks appear to result from them snagging inside the jacket. Dust and other debris that gets in there could be responsible, but another major cause is pressure – such as might be caused by stacking too many disks too close together. That can result in the disk binding inside its envelope, and is particularly prevalent when disks are stacked horizontally; keep them upright.

Another restriction: the largest possible record is 255. So you have to be pretty careful about how big your records can be.

Here's a couple of OPEN statements at the start of a relative-file program that uses 50-character records:

```
10 OPEN 15,8,15
20 OPEN 10,8,10,":ADDRESSES,L," +
CHR$(50)
```

4. You can now tell the 1541 that you're going to put something into the file. This you do by passing a message along the command channel, the one OPENed in line 10, and it's done by a pretty complicated PRINT# statement:

```
100 PRINT#15,"P" CHR$(10) CHR$(1)
CHR$(0) CHR$(1)
```

The 'P' tells the 1541 to expect a 'Pointer' that will direct it to reference a particular field within a

- **Store your disks.** It's amazing how floppies proliferate. The cardboard boxes that floppy disks usually come in are fine for storage of disks you won't use often, like your 'master' copies, but they aren't great storage boxes for day-to-day use: they weren't designed for it, they tend to fall over when you'd rather they were upright, they don't let you flip easily through a number of floppies to find the one you want, they split apart because you're trying to stuff in too many disks. It's worth investing in plastic trays that will keep your disks stacked neatly at an accessible angle, perhaps with identifying dividers. Go for a tray that has a hinged plastic cover to keep out hair, flies and coffee.

- **Keep disks cool and dry.** 'Disks should be kept in a cool, dry place', says the disk manufacturers' blurb. In practice disks can put up with quite a range of temperatures – but not a lot of temperature variation; and they don't much like moisture either.

- **Don't leave disks out of their cardboard sleeves.** That's a difficult one for the frequent user, but do your best. The notes above about copying related programs and separating your disks logically should at least cut down the amount of disk swapping you have to do, which will help.

- **Don't touch the exposed brown surfaces.** Goes without saying, really, but even the small amount of contamination transferred to a disk from the thin film of grease and dirt on your fingertips can cause errors in reading and writing. This rule actually prevents you putting the disk in the wrong way round, since you'd have to hold it by the exposed slot area to get it back to front ...

- **Hold disks in the middle of the top edge or one of the sides.** Grip it (gently, now!) along an edge – but not the *bottom* edge, the one with the long exposed slot.

- **Don't bend the disks.** Not even to prove to your *amorati* that they really *are* floppy.

- **Don't leave disks in the sun or near heat.** They warp.

- **Keep disks away from magnets** – and that includes objects which generate a magnetic field, like TV sets and even ringing telephones. Actually I've never heard of anyone having a disk corrupted by magnetic fields, but it's better to be safe than sorry ...

- **Post disks sensibly, if you've got to send them somewhere.** Ideally this means using purpose-designed cardboard mailers rather than a couple of sheets of card in an ordinary envelope. If you really must economise, do it by sandwiching the floppy between two polystyrene ceiling tiles. Either way, to be doubly safe in providing a partial barrier against errant magnetic fields (bomb detectors?) wrap the disk in aluminium foil before you put it into the packaging.

- **Don't force the disk into or out of the drive.** They should slide in and out fairly easily – a bit notchily, perhaps, but certainly not so much so that you have to exert pressure. If the disk drive latch won't shut easily, the disk hasn't been inserted correctly; take it out and start again.

- **Keep disks well away from cigarette ash, drink, paper clips and pens.** Contamination on the disk surface can can prevent you reading from that part of the disk or saving a file on to it. Disks don't like paper clips, either, since they could slip inside the jacket; and keep floppies away from writing implements and anything else that might inadvertently scratch the disk's surface or drool ink on to it.

particular record within a particular file. That information is given by the four following CHR$ numbers: the first one is the file reference number of the OPEN command for your file – in this case we OPENed 10, 8, 10 for the file ADDRESSES, so that's the value in the first CHR$.

The next two CHR$ numbers identify the record within the file. As with any use of CHR$ neither of these can exceed 255; but since you can use two in combination (in fact you've got to) the record number thus referenced can be higher than 255. In fact you can go up to record number 65,535, which ought o be enough for anyone ...

For record numbers up to 255, though, the first of this pair of CHR$ numbers gives the record and the second should be zero. In the example CHR$(1) CHR$(0) references the first record in the file.

For record numbers higher than 255 you'll need some lightning arithmetic. The procedure is as follows:

- divide the record number by 256. Forget any fraction – the *integer* part of the result is what you use for the *second* of the two CHR$ numbers

- multiply this second number by 256 and subtract the result from the original record number. The answer is the *first* of the CHR$ pair.

Take record number 539, for instance. Dividing by 256 gives you 2 and a bit; the 2 goes into the second CHR$. Working out 539 – (256 x 2) provides 27, so in this case you'd end up with this pair of CHR$ numbers to point to that record:

```
CHR$(27) CHR$(2)
```

You could of course build that into a program:

```
100 INPUT "RECORD NUMBER"; RN
110 IF RN > 65535 THEN PRINT "TOO BIG!":
GOTO 100
120 IF RN < 256 THEN N1 = RN: N2 = 0: GOTO
200
130 N1 = INT(RN/256)
140 N2 = RN - (N1 * 256)
```

Then there's the last of the four CHR$ parameters. This indicates exactly which field (meaning which data item) you're after in the record:

```
200 INPUT "FIELD"; F
210 PRINT#15, "P" CHR$(10) CHR$(N1)
CHR$(N2) CHR$(F)
```

5. Now use PRINT# to put something into that field of that record in that file. For this example you might have something like the following:

```
300 INPUT "DATA"; N$
310 PRINT#10, LEFT$(N$, 15)

400 INPUT "ANOTHER"; Z$
410 IF Z$ = "Y" THEN 100
420 CLOSE 10: CLOSE 15
430 END
```

This puts a 15-character name into the specified field – the LEFT$ chops off any over-length INPUT, since I've decided that the field shall be that length.

6. Check the error channel and by-pass it. You think you've finished OPENing the relative file yet? No such luck. If you try to write something into this file now you'll get an error message. Why? Because relative files can be for reading *or* writing: and when you try writing to it for things, it will immediately come back with a RECORD NOT PRESENT error. Which is the truth, I guess, though it's not particularly helpful.

The solution is to include a routine that reads the error channel (in fact you ought to have one of these anyhow). In a record-writing program you can then ignore error number 50 and RETURN. It would be safest to have a GOSUB immediately after your OPEN and another immediately following PRINT# statements.

```
30 GOSUB 1000
220 GOSUB 1000
320 GOSUB 1000

1000 INPUT#15, EN, EM$, ET, ES
1010 IF EN < 20 THEN RETURN
1020 IF EN = 50 THEN RETURN
1030 PRINT "ERROR NUMBER:" EN
1040 PRINT EM$
1050 PRINT "TRACK:" ET, "SECTOR:" ES
1060 RETURN
```

Phew!

CHAPTER 21
Printers and Plotters

This chapter provides a general overview of the printer game and summarises the options before getting stuck into some specifics about printers with the 64.

Styles of printer

In theory there are several methods of getting information out of a small computer and on to paper. And most of them are indeed available one way or another in the form of purchaseable products. But in practice the prices of most of the printer technologies render those products frankly prohibitive for the buyer of a small computer.

So what does that leave you?

For the requirements of the small computer – the best balance of price, reliability, physical size, and print speed – there are basically only three types of printer a 64 owner is likely to consider. The **dot matrix printer** is what you will almost certainly use; you might however be attracted by the speed, quietness and physical compactness of a **non-impact** printer. If you want to do word processing, though, you might consider a **correspondence-quality** printer – that will probably be what's called a *daisywheel* printer, delivering a much better print quality but costing a lot more.

The matrix of possibilities

Dot matrix printers don't have typewriter-style pre-formed characters like the daisywheel printers. Instead there's a single print head that contains a collection of metal needles. By selecting a particular pattern of needles and pushing them against the ribbon on to the paper, the printer can produce a collection of dots that resembles a character – just in the same way as dots of light are used to build up a character on the 64's screen.

In technical terms this is surprisingly uncomplicated these days and the mechanisms used can be very compact and pretty long-lived. That all helps to keep the price low, and quite legible print is possible at quite fast speeds. It won't be as dark as the printing from a daisywheel unit, however – the pins simply do not strike the ribbon on to the paper as fiercely and the space between and around the dots tends to dilute the visual image anyway.

Non-impact printers work in much the same way, producing a character by putting a recognisable collection of dots on to the paper. They do it however without a ribbon. Some of them are *thermal* printers, in which case the print head contains an arrangement of tiny heating elements that effectively scorch a point on the paper. You might need a special extra-sensitive coated paper for this, though some thermal printers can use ordinary paper (the print image isn't as dark on the normal stuff, though, but that special paper is rather more expensive).

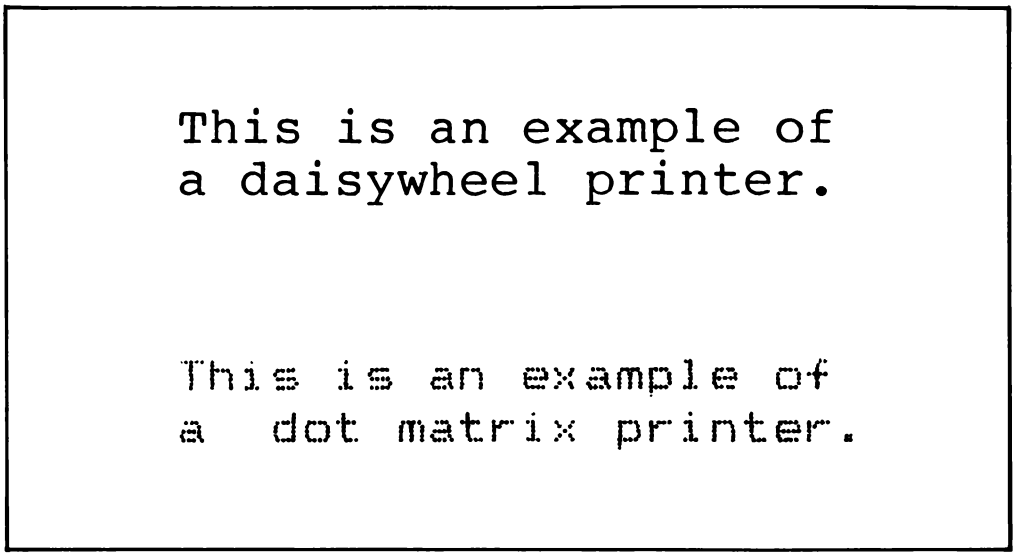
Others are *electrostatic*: the print head has in it electrodes which pass sparks to the paper, effectively scratching off a special coating to form a single dot. The printer of this type that you may well have seen is the Sinclair ZX Microprinter; the expensive silver-coated paper it uses is a characteristic requirement of electrostatic printers.

The non-impact printers score because they *are* non-impact – which means the print head doesn't actually have to pound the paper, so they're much quieter (near silent, in fact). And since the mechanical components are greatly reduced (nothing in the print head actually moves) they can be physically more compact.

On the other hand, they do generally need the special paper; and that puts them out of court for most 64 users ... with the exception of Softex's interface that links an inexpensive Sinclair printer quite effectively to the 64: more about this later.

On most dot matrix and non-impact printers the characters to be printed are formed from a 5x7 matrix – which means a pattern of possible dot positions arranged as seven rows of five. It's a case of the more dots the better, since more dot positions gives the opportunity for a crisper character to be formed, and 7x9 is the more acceptable compromise between quality and cost; but below the £300 mark you'll find very few 7x9 printers.

In the print head itself you find enough of those pins or heaters to provide one column of dots (so in a 9x9 matrix, which tends to be used by the more expensive higher-quality printers, you'd get nine pins in the head). The printer will print all the dot positions in the first column, move on to do the second, and so on until all the nine rows have been printed.



A 7x9 or 9x9 matrix is good enough to depict true 'descenders', that part of letters like 'g' and 'y' that fall below the line. You won't get that on a cheap 5x7 printer because there aren't enough dot positions to allow it. Instead the characters with descenders sit above the line, making a 'p' for instance look more or less the same as 'P'.

There are restrictions on the 7x9 pattern too, though. It cannot normally manage French accents and the other idiosyncracies of European languages, and neither can it cope with 'true' underlining where the underscore is printed at the same time as the character a couple of dots below it (as opposed to 'fake' underlining, with the underscore printed as a separate line immediately below the characters).

A 9x9 matrix should be able to manage the underscoring but probably won't do the accents. The ability to print sub- and superscripts is rare, too, though it seems increasingly to be available on more modern printers.

Handling graphics

There is an added complication with the 64 in that the screen can display graphic characters. When you're looking to print the output from a program there shouldn't be a problem with high-resolution graphics: the dot-by-dot printers can work just like the dot-by-dot method of displaying graphs and the like on the screen.

The problem comes with displaying low-resolution graphics, the characters actually produced from single keys on the 64's keyboard. And it's not just the graphic shapes that you're using in a fancy display or an actual picture: the colour and cursor control codes in a listing appear as graphic symbols in a character string. To take a copy of your program on paper you're going to need to print them somehow (unless your programs never use them, which isn't very likely).

Most printers can't reproduce these on paper – you just get a blank space where the graphic shape would have been, or you may get some eccentric characters that the printer manufacturer has put in there for that code. Since printers are designed and built to appeal to the widest possible market, even a high-definition multi-dot printer may not be able actually to print everything you can see on the screen of a particular computer like a 64.

With the printer you should get a manual that lists all the characters the unit can do, perhaps as a table of CHR$ equivalents so that you can compare the 64's own graphics-generating CHR$ codes against the printer's. For most of the common characters – numbers, the QWERTY alphabetics on the keyboard, standard punctuation and arithmetic marks – there won't be a problem (both the computer and the printer will follow a standard set of code-to-character correspondences known as the ASCII symbols). For 'special' symbols that weren't anticipated by the designers of the ASCII set, the 64 tries to send its own 'special' codes down the cable to the printer; and so the printer has to be set up specially to recognise them as well as the standard ASCII characters.

Some printers have been prepared to do just that for the 64's full character set – notably Commodore's own, of course, but also the Seikosha range and others. Several printers have the facility to use what's called a 'down-line loaded character set', where the user can tell the printer exactly how to treat the special character codes arriving from the 64.

Still, most of those special 64 characters aren't touched at all by the majority of packages; so if you are using an off-the-shelf word processor or spreadsheet package you won't need to print them anyhow. But if you're primarily interested in producing 'hard-copy' listings of your own work you've got three options.

- **Buy a Commodore printer** (or a Commodore-compatible one). This is the easy way out, though obviously it means you don't get much chance to shop around for the best/cheapest/most versatile printer.

- **Buy a printer that can be programmed to handle 64 graphics.** Most low-cost matrix and thermal printers these days can be set up for 'alternative' characters; it will usually involve some programming that at best is laborious (lots of DATA statements defining character shapes) and at worst is really tricky (you've got to find out exactly *how* the printer responds to input from the 64). It is however feasible, as the accompanying program from Chris Durham demonstrates – it sets up the popular and widely available Epson FX80 to handle the Commodore graphics, is altogether a clever and effective piece of coding from a bright bloke, and originally appeared in *Commodore User* magazine.

- **Convert the graphics control characters to some kind of recognisable text that the printer can handle.** This won't let you print the *results* of running a program that uses graphics; but it *will* let you LIST programs in an easy-to-understand form which is probably preferable in any case to the usual mish-mash of hieroglyphics (hands up anyone who can remember *all* the screen control code symbols that can be listed in a character string).

There are two ways of doing the option-three conversion. The easy one is to buy a plug-in converter like the Tripler – this lets you run a non-IEEE printer from the 64 (more about that later) but also gives you the option of selecting 'code conversion' to convert graphics symbols automatically into the kind of English mnemonics that I've used throughout the book (CUD for 'cursor down', RED for 'change colour to red', and so on).

Alternatively you can try writing a program that does this conversion for your LISTings *before* you start printing, but that can be a bit laborious. You've got to write the code converter first, for a start. Then you've got to run it every time you want to LIST another program; and to produce a printable file your program could set the 'target' program up as a SEQ data file to be read character-by-character by the converter. It would check each character to see if it was one of the control graphics symbols, replace it with a three-character string from a matrix or a DATA statement, then send the string to the printer ... and do it all again for the next one.

Possible, certainly. But also a drag. Save up for a code-converting interface if you don't want to use one of the Official Commodore printers ...

Matrix printing capabilities

The more you pay for a printer, the more you get. At the £200 or so charged for the Commodore-compatible printers you won't get a lot of print options built into the unit (though as we'll see you can do quite a lot from software). But in fact the capabilities of the average £300 matrix printer's capabilities are quite extensive, so let's summarise them.

There are likely to be four line widths:

- **normal:** generally 10 characters per inch (which means 80 characters per line maximum on the normal 8in wide paper)

- **enlarged:** typically 5 characters per inch (40 per line). Useful for headings and emphasis

- **condensed:** 16.5 characters per inch, usually (132 per line). Many business accounting and spreadsheet calculator packages require that you can produce 130 or more characters on a line; this option enables you to cram a full 132-character line on to 8in paper. It's not especially legible, but for draft work and other printing that need not be too fancy it is a handy way of economising on paper.

- **condensed and enlarged:** some printers can do this, producing about 8.25 characters per inch (66 per line). The result approaches letter-quality printing, though it's by no means a substitute for a 'real' word processing printer.

Then there may well be three printing styles which can be used in combination with any of the line widths:

- **normal:** a single strike per dot

- **double:** two strikes per dot, so it produces a denser print at the cost of slowing things down a bit

- **emphasised:** shadow printing – two strikes per dot, with the second one marginally to the right of the first (usually about 0.118mm away) such that the two dots appear to merge into one. The result is a 'bold' effect, though printing the same character twice obviously means that the total throughput of the printer is slowed considerably.

Overstrike printing is another print style on some printers – it will allow you to print characters with a diagonal slash through them. This may be useful in

somewhat esoteric circumstances, for instance to indicate deleted text in word processing while leaving the original still legible.

In addition you might look for a multi-dot print that really does resemble **correspondence-quality**. Letter-quality printers are not cheap: they're also comparatively slow, undeniably noisy, and heavy on consumables like ribbons and printwheels. The quite recent development *'enhanced'* dot matrix printers means you only one printer that can be switched between fast dotty printing and slower correspondence-quality print. In the latter mode the print head moves slightly to one side after printing a dot and puts a second dot on to the paper; these printers can put their dots so close together that the result looks almost like a solid character.

The printing takes longer in the near-letter-quality mode, of course, and in this style the printer is usually at least 25 per cent slower than in its conventional mode.

Few if any of these printers can actually produce print as good as that from a daisywheel – after all, the characters are still built up from dots, and that will never be as crisp as a fully-formed character. But such printers are always faster and generally cheaper than daisywheels: and the print is good enough for many purposes.

Daisywheel printers

For ordinary office work the matrix printer is fine. It is relatively cheap, fast, pretty robust, fairly quiet, compact, and easy to use. The printing is legible. Ribbons are inexpensive and easily replaced (though even today some printers do require you to cover your fingers in ink before you can do that). The print element itself lasts a good long while – 30 million character impressions is not uncommon – and it too is probably replaced quite easily.

But no-one would claim that matrix printers produce a pleasant, highly readable typeface. They don't. They produce a pattern of dots which more or less resembles an alphanumeric character; and when you have a lot of them together – as you would in the kind of document for which you'd utilise your word processor – the result is at best unpleasant to read, at worst impossible.

The very best print quality comes from a preformed type element like the IBM Selectric's golfball. But adapting that technology to computer-driven printing has been a bit troublesome – very slow, very noisy, relatively unreliable.

Of the leading alternatives, the so-called daisywheel mechanisms are the preferred option to balance performance, cost and reliability against print quality. Daisywheel printers produce a good-quality print image because they use fully-formed characters, rather as a typewriter does (indeed, some typewriters these days use a daisywheel element).

The problem is that the mechanisms have to be relatively complicated: they involve many more moving parts than a matrix printer. Moving parts inevitably wear, so there is a greater risk of failure. And the mechanic bits tend to bump up the price, especially as the manufacturing has to be much more precise.

The mechanical element also means that the print mechanisms can't really go very quickly – the fastest speed of daisywheels is about 80 characters a second right now, and that's for a price of £2,400 or more. Most daisywheel printers can do between 35 and 45 characters per second for around £1,500, with 20cps units selling at just over £1,000 and 10-15cps printers at prices around £500.

Still, they represent the best option we've got. And the good news is that there's a whole new breed of (mostly Japanese) daisywheel printers coming along at prices below £500: they appear to exhibit no compromises on quality or reliability, and a slower print speed – 12 characters a second is the norm – is their only obvious deficiency.

The other option is to use a typewriter. For a while now enterprising people have been selling purpose-built converters that allow you to utilise a standard electric typewriter as a computer printer; but that often involves some messy soldering, and it's really not too satisfactory a solution. More good news: many of the modern low-cost electric typewriters (which tend to use daisywheels these days) come with a ready-to-use computer-connection socket. The typewriter works as a typewriter when you you to type; plug in the computer and it works as a printer.

Two problems – first, you'll still pay something over £350 for the typewriter/printer (might not be so painful if you do *need* a typewriter anyhow). And second, the standard interface provided isn't usually the IEEE connection of the 64; so you'll probably have to allow another few quid (maybe more than few) for an interface converter of some kind.

In all these printers the 'daisywheel' itself is the removable print element, the equivalent of a Selectric's golfball; it's a flat disc with radiating spokes, between 96 and 120 or thereabouts depending on what characters the printwheel can produce. On the end of each spoke is a fully-formed character, and it does look a bit like a stylised flower. There's also a variation used by one manufacturer, NEC, in which the spokes have been bent up from the hub to resemble a cup or thimble.

Either way you plug the printwheel into the printer. What happens in operation is that the printwheel is turned so that the right character faces the ribbon, and then that particular stalk is pushed forward pressing the ribbon on to the paper to leave the impression of the character.

The daisywheels are easy to slot in and out; and they are available in a variety of typestyles. You probably won't need many fancy typefaces, but it might be useful to have 'specials' like printwheels

with scientific symbols or European-language characters on them in place of the conventional QWERTY/ASCII characters.

Daisywheels come in various character sizes. 10- and 12-pitch are common, 15 is found occasionally ('pitch' indicates the number of characters printed per inch). You can use a 10-pitch printwheel if the printer is set up for 12-pitch; the characters will just be closer together, and some people prefer that. But the printer should have on it a pitch selection switch that has to be set to the appropriate pitch for the daisywheel so that the mechanism moves the right amount each time a character is printed.

There is one obvious drawback as far the average 64 user is concerned: daisywheel printers can only print the characters provided on the daisywheel. Which means no Commodore graphics. And you won't be able to print pictures and graphs made up of dots placed very close together, since the 'dot' on the daisywheel – the full stop – takes up a whole character space.

Paper and how printers handle it

The two standard methods of using paper in printers are *friction feed* and *tractor* (or 'pin') *feed* – the 'feed' being the way the paper is 'fed' into the printer and past the print element.

Friction feed works just like a typewriter. There's a rubber-covered roller called a platen which the paper wraps around; it's pressed against the platen by rubber-covered wheels. The platen turns, and because the paper is pressed against it the paper rolls with it.

Obviously the paper has to be pressed tightly against the platen, otherwise you'll misalign as it slips. But then it doesn't want to be *too* tight – or the paper won't be able to roll through. Look for some kind of adjusting lever to control the pressure.

Another common problem with friction feeds is ink on the platen being transferred to the back of your prized letters (caused by inadvertently trying to print something when there isn't any paper in the printer).

The other frequent irritant is too small a gap between the paper and the print head and/or the ribbon; or, in the case of daisywheels, between the paper and the little plastic shield through which the daisywheel's stalk strikes the ribbon. This can drag back the paper, preventing it rolling smoothly; and it can result in mucky ink being transferred on to the sheet. The answer may be one of the possible adjustment levers on most printers, but occasionally some work with a screwdriver may be called for to realign the errant components.

The friction-feed approach is ideal for single sheets, and especially for use with standard letter-headed stationery. That's because the tractor/pin feed mechanism requires your stationery to have sprocket holes down the edge; these engage in rubber or plastic teeth in the paper feed mechanism, and when these 'pins' are turned they drag the paper through.

The result should be precise alignment every time ... provided you've got the tension right. There should be some kind of adjustable bar that presses against the paper as it goes through to maintain the proper tension: too slack, and the printing will be slightly out of line.

The sprocket holders are also adjustable, to cope with different widths of paper; and as well as remembering to lock the holders down (there's normally a simple push-down lock) you have to get the positioning correct. It may *look* right, but after a few sheets have gone through the slack may have become so great that the paper starts slipping out of the pins.

Paper for tractor feeds is 'continuous' or 'fanfold' stationery, long strips of paper perforated horizontally at the equivalent of a page's length. It comes in arm-stretching boxes of 500, 1,000 or 2,000 sheets; it's 'continuous' because it is basically a long sheet of paper perforated at the page ends and folded in a concertina fashion into the box. There's normally a perforation down the edges too, so you can tear off the bit with the sprocket holes.

The paper is fed into the printer and the holes along its long edges engage in sprocket teeth that pull it through the printer when the thing's running. This means that you can't easily use single sheets of paper: there isn't a 'friction feed' such as you get on a typewriter, where the paper is pressed against a rubber roller and moves through the printer because the the roller is moved. If you want to do a single-sheet printer you'll have to use continuous stationery and neatly trim the edges of one sheet.

Continuous stationery is generally lighter in weight than conventional typewriter stationery. It comes in various lengths and widths; for word processing draft work the most popular size is about 8.5in wide and tearing into sheets 11in deep – a bit shorter than the A4 you'll use for correspondence, but wide enough to get 65 or so characters on to a line. Business accounts packages often demand stationery capable of printing a full 130 characters a line, or thereabouts – which means you have to have a printer that can manage the width, as well as paper on to which it'll fit.

For a fairly modest price premium you can have continuous stationery ready-printed with a letterhead, which gets over the somewhat utilitarian look of the plain stuff. It's also possible to buy heavyweight continuous stationery sized so that when you tear along the perforation you end up with an A4 sheet; the tear lines are barely noticeable, so this (somewhat expensive) type of paper can be used with a tractor feed for correspondence-quality work.

But most printers destined primarily for word processing – most daisywheels, in other words – come

with a platen and the friction-feed approach as standard. You can generally get a tractor feed attachment for extra cost, in which case the add-on device is attached by clamps to the rods that drive the platen and a set of gears and pulleys turn the sprocket pins. The platen is not used as a friction feed device, and is usually disengaged.

Matrix printers tend to have a tractor feed built in, and many do not have a platen at all – so you don't get the chance to use single sheets.

As for carbon copies, it's possible (but costly) to buy multi-part continuous stationery – usually up to two copies in addition to your top original. These may utilise interleaved carbons, in which case separating the sheets can be a real drag, unless your print volumes justify the very considerable cost of a device that does it automatically for you. There's a pricey alternative in the form of interleaved sheets with built-in 'automatic' carbons.

Daisywheel printers hit the paper hard enough to give you four or five legible carbon copies, and some matrix printers can manage two or three. A few matrix printers cannot produce even one, though; and even with those printers than claim multiple copies you can run into problems with the resulting thickness of paper causing slippage and misalignment.

Printer controls

Whatever kind of printer you have, you may find a problem in the way it copes with automatic line feeds. In some packages and with some computers 'Return' just indicates the end of the line and does *not* automatically mean 'that's the end of the line, now move the paper up and start printing on the next line'. So most printers have a switch that can be set to 'automatic line feed'. Every time the printer detects a Return it shifts up one line as well.

But you will have to set it back again if the next program you use incorporates a line feed automatically in its 'returns' – otherwise you'll get *double* line spacing.

That automatic line feed switch isn't necessarily very accessible, either. It could be an obvious push-button on a control panel: or it could be buried in the printer as a tiny switch on the printed circuit board inside there, the kind of switch that needs a delicate touch with a pencil (or anorexic fingers!) to move it.

As for the other controls, all printers will have an on/off power switch. Beyond that the simplest printers (like Commodore's own) don't have many controls, the fanciest have dozens. Typically they come in three groups.

- **Main panel:** usually on the front edge of the printer.

Ideally this will have a *select* button or switch to tell the computer the printer is ready to receive material to be printed (along with an indicator light to show you when 'select' is on) and a *deselect* or *reset* control to halt the printing temporarily (while you adjust the paper, say). Using the reset shouldn't mean any loss of data; when you press 'select' again the printing ought to resume exactly from the point where it left off, even if this was in the middle of the word. Try before you buy ...

You will probably have a *form feed* control there and maybe the *line feed* too. These adjust the paper positioning accurately. Form feed rolls the paper up one whole page, which is useful if you want to eject a single sheet or to get to the next perforation of continuous stationery in order to tear off the wadge you've just done. Line feed rolls the paper up one line at a time, which allows you to adjust the paper so that printing starts in the right place. Both of these will probably repeat if you keep your finger on the button.

A more expensive printer may also have a 'backwards' line feed to roll the paper *down* one line; and there might also be upwards and downwards *form alignment* controls that shift the paper a fraction at a time to enable you to get the positioning dead right.

Typically these will work only when the printer is not 'selected', because they roll the paper up – and you might accidentally touch one of them while you're printing something.

Commodore's printers have a self-test switch, but on several models the form feed button is also used for testing the printer. With the power off you place your finger on the control button: when you switch on the printer will automatically start printing every character it's capable of (handy for checking just what it can and cannot do). A more comprehensive self-test involves doing this in different line lengths or different patterns, so that the same characters aren't printed in the same position every time.

The main panel should also have a *'power on' light* and an *'error' indicator*. That latter may be accompanied by a beep from the printer. The errors in question are likely to be your reaching the end of the paper, but clever printers can also detect paper jams and the increased tension that means a one-time ribbon is about to run out. Very few printers however give you *separate* indicator lights for these three conditions.

- **DIP switches for communications:** generally buried inside somewhere, often just under the front cover

'DIP' switches (you may see them referred to as 'DIL' switches too) are tiny, so small in fact that you generally need a screwdriver or a pencil point to shift them. If you have a printer with multiple operating options you will almost certainly have to get at them ... you'll probably find the printer's manual drifting completely from mere terseness into smog-like opacity.

There are usually two banks of DIP switches, and the smaller group normally sets the printer's communications options. The 64 uses a method of sending stuff to the printer that follows a version of the specification called IEEE; just about no-one else uses that, though, and there are two more common printer interfaces. So depending on what options you get, the DIP switch bank may include a single switch that allows you to select *Centronics* communications (could be called 'parallel' in the manual) and several that select different transmission speeds for *RS232* communications (which could well be called 'serial'). The available options for RS232 speeds will probably be from 300 to 9600 baud, the baud rate corresponding roughly to 'bits per second'; divide by ten and you get an approximate count of characters per second.

Most printers are obviously sold to work with a variety of computers, and elsewhere on this panel of switches you may be invited to select other RS232 options (full- or half-duplex, parity or no parity) and a variety of what are called 'handshaking' protocols to suit various computers. To utilise these properly you'd better know what the 64 wants – and what the program you'll be using expects from its printer, too.

- **DIP switches for printer functions:** again likely to located inside somewhere, often just under the front cover

Again depending on how much the printer is capable of doing, there could be another bank of small switches to be set. These will be on/off selections used as the 'default' conditions (which means that they will be used unless you override them somehow) for functions like ...

- *form length* – a standard length for the paper you're using (which the printer needs to know if it's going to give you a correct amount of paper when you press the 'form feed' button or when your word processing program tells it to start a new sheet)

- *skip over perforations* – yes or no? How many lines to jump?

- *automatic line feed* – should the printer assume that a 'carriage return' from the computer automatically assumes a line feed as well? See the comments above.

- *line spacing* – how much space between lines? The usual options are six or eight lines to the inch

- *character pitch* – how many characters to the inch? 10 or 12 are the normal alternatives

- *alternate fonts* – combinations of three switches are generally used to select which of several built-in character code sets are to be used. These generally provide for different styles of and characters in European languages (including 'English' and 'American', the difference generally being the availability of pound and dollar signs)

Loading paper

The design of printers has improved dramatically in the past few years. But it can still be difficult to position the printer such that it takes up as little room as possible, and to load the paper simply and quickly.

An arrangement where the fresh continuous stationery is below the printer and the output stacks up behind or below is the best option, but sometimes that is just not possible. Various purpose-built printer stands are available for popular makes, but they can add £100 to the cost of the installation.

Using an existing desk or table usually means the paper has to feed up at the front and pass *under* the printer before it can be loaded at the back: but some printers do not have enough clearance underneath for this, and the wonderfully thoughtful non-slip rubber feet tend to get in the way.

Try loading some paper before you buy a printer, too. Actually getting continuous stationery into the printer can be very fiddly; and to insert single sheets such that they line up accurately may well involve wrist and eye
contortions that threaten dislocation of either.

Print speed

The rate at which a printer can print is one of the reasons why printers span a wide range of prices. Below £300 or so you're unlikely to find a printer that will go faster than 100 characters per second. The top speeds for matrix printers are well above 300cps, but in practice the range 80 to 180cps seems to be a good compromise between performance and price for most people.

The quoted speed of a printer is something of a chimera, rather like the MPG figures cited for cars. The actual throughput will be affected by many other attributes, not least of which is the amount of work that the printer itself has to do in decoding the commands and data arriving from the computer as material to be output. Incomplete lines, tabbing, underlining and emboldening will all add to the overall printing time.

On the other hand the printer may include facilities that speed up the throughput. *Logic-seeking* is one of them, and these days most reasonably-priced printers include this facility: it means the printer is able to skip over blank spaces to get to the next character that needs to be printed. In other words, logic-seeking printers do not have to take as long thinking about the gaps as they do about positions which contain characters. This speeds up the throughput.

Bi-directional printing also helps. Actually the correct term is 'boustrophedon', which means 'like an ox ploughing' – going first in one direction, then turning and coming back. So the printer does not print only from left to right, returning to the left-most

position each time it starts a new line. Instead it can print continuously, starting at whichever end of the line happens to be nearest.

Not all printers can do this; it's rare in daisywheel printers, for instance. But again it speeds up the throughput since the printer does not have to waste any time getting to that left-hand position.

But there are ways of improving the speed of printing; you may be able to cut down on how much time the computer spends waiting for printing to complete before you can use it again. The printer's overall speed (and its price) will also be increased if it includes a decent-sized *print buffer*, a small chunk of memory within the printer. The use of a print buffer means that the 64 can squirt a lot of material to be printed at the maximum possible speed down the cable to the printer and forget it; the printer then chugs away at its own best rate, printing whatever is in its own little memory rather than taking information to be output directly from the 64 itself.

When the buffer is full, the computer obviously has to stop sending characters until there's room for them; and at that point it might slow down again to the printer's speed – since the printer can't take information to be printed from its own memory any faster than it can handle print data from the computer.

But in any case you should be able to get use of the 64 again much sooner. And if the buffer is big enough, it's possible that *everything* you want to print can be stuffed into it almost immediately. The printer can then chug away at its own speed while you get back to the screen and keyboard.

Many low-cost printers have a small print buffer, typically one that holds a single line's worth of information to print – say 200 characters tops. Most however have a buffer capable of holding at least a thousand characters, though. And more expensive units may have a buffer capable of holding up to 6,000 or more characters.

You can help things along dramatically by buying a separate print buffer. Essentially these are little black boxes (or rather, not so little in some cases) that fit somewhere between computer and printer. The box contains a small microprocessor and some pretty basic control software ... along with a chunk of memory. Generally that's between 8K bytes and 64KB, with prices rising commensurately from around £60 to something over £300.

These units work exactly like the buffer built into a printer. They fill up with
information the computer wants to be printed, and they do so at the fastest speed the cable can get it to them. Thereafter they feed it on in the dribs and drabs that the printer can cope with. And if the printer itself has a sizable buffer, those can be substantially sized dribs.

Using one of these buffers won't affect the operation of the computer in any way ... except to minimise the time the 64 spends tied up on printing. Because these boxes contain more memory than the printer's own buffer is likely to have, you'll get the computer back that much sooner.

Some buffers can plug straight into the printer (most of the popular Epson line of matrix printers have a slot that can take a small circuit board, and there are a number of printer buffers supplied as circuit boards that fit into it) or into the 64 (where the buffer is again a circuit board, this time fitting one of the 64's expansion slots). But the 'in-line' buffer, the black box that fits somewhere along the length of the cable, is the most common approach – they are the simplest to fit and can be used with any printer (or any computer, for that matter).

Reliability, maintenance and consumables

Being inherently mechanical, printers do tend to wear out and break down. There's not a lot you can do about that, apart from the obvious precautions like not leaving the thing switched on all night.

At least the two consumable components, the print head and the ribbon, are easy enough to look after. On dot matrix printers the pins that are fired forward against the ribbon do deteriorate. They wear, get blunt and can become slightly deformed under heavy use; the electrical connections to them can sometimes fail, too, so you might find your printing missing a dot or two per character.

Most manufacturers provide at least a promise of longevity and sometimes a formal warranty on the print head, usually expressed in millions of characters and/or length of service. Print heads are not too expensive, so if you do have a warranted printer you should be able to get a replacement without quibbles. These days matrix printers often utilise a print head that is very easy to replace – they just plug in and out – and if that's the case it is well worth while buying a spare head for DIY repairs.

The equivalent on the daisywheel is the print wheel itself. The daisywheels themselves aren't as robust as one would like, and replacing them quite can be quite an irritating expense.

That's particularly true when you find yourself having to discard an entire printwheel because a single character has snapped off. Paying more for your wheels may well buy you better quality, for instance with extra strength on the characters used most frequently – like the full stop. Top quality in printing and in longevity come from metal-plated daisywheels.

The cheapest printers use a spool ribbon rather like a portable typewriter's. These are inexpensive, but they can be a bit messy – generally you have to thread the ribbon around a couple of guide posts and through the print head. That can transfer ink to your fingers, and it may also leave some on or around the platen to smudge your paper.

In fact you are more likely to look at printers which use some kind of cartridge ribbon. These are easy

(and usually clean) to replace. The only problem is that there is very little standardisation in types of ribbon cartridge; each manufacturer seems to have its own format. This limits your sources of supply, and that in turn means less opportunity to shop around for the best prices and delivery.

Still, if you have one of the more popular makes of printer (Diablo, Qume, Juki, SCM, TEC among daisywheels: Star, Epson, Seikosha, Mannesmann Tally and others among the matrix printers) you will find that there is a market emerging among the independent ribbon manufacturers for alternatives to the standard offering.

Naturally those independents will be cheaper. But beware of short lengths. Most ribbons can only be used once, so the length of ribbon determines its life: an apparently economical ribbon might be so in expensive because there simply isn't as much of it.

Your disk supplier and/or your dealer should be able to tell you whether it is worth shopping around for ribbons on a particular printer – and where to do it.

A few printer manufacturers insist on your buying ribbons from them on pain of losing your warranty protection. That is quite rare, though, especially among the lower-cost printers, and in any case it is probably illegal now.

You can also save money on some times of ribbon by re-inking them. If you have a lot of printers, it may be worth buying a specialised re-inker or finding a supplier who can do that for you. There is also a small-scale re-inker on the market for certain types of ribbon, and demonstrably this does enable ribbons to be used three or four times before the fabric itself deteriorates. The average impoverished 64 user is more likely to get stuck into a fun evening spent applying Indian Ink from a rubber-stamp pad to a used ribbon (and to your fingers, favourite trousers, least happily selected shirt etc); there again, it's generally better to believe the manufacturers when they describe their product as 'one-time' ribbons.

Commodore's printers

Commodore in its time has sold three printers for the 64 and you may still see all three of some in the dealers' stores.

They print up to 80 characters a line. Those characters can be anything you're able to put on to the screen – upper- and lower-case letters, the 64's standard graphics, your own user-defined graphics. They can be reversed to come out as white shapes on a dense black background; and they can be printed double- width as well – handy for headings and emphasis.

The Commodore printers use continuous stationery only, with built-in tractor feeds; that means that you can't easily use single sheets of paper – there is no typewriter-style platen.

The 1515 and 1525 are the two older printers. They were both variants of the same unit; Commodore bought them from a Japanese manufacturer called Seikosha which is well known for cheap, simple, and reliable printers. A number of suppliers can sell you a Seikosha-branded equivalent of these.

The 1515 used only paper each sheet of which is 8.5 inches wide and 11in deep – not a particularly standard size of paper, since most computer stationery is wider to allow for the sprocket holes. The 1515 was slow (30 characters a second) and noisy; the 1525 was no faster and looked much the same as the 1515 – all the styling elegance of a doughnut – but it printed rather larger and more legible characters (10 per inch rather than 12) on wider paper – one sheet measures 8.5 x 11 ins *after* the perforated strip with the sprocket holes is torn off. Now that *is* a standard size, and it's much more widely available.

One of the plus points of the newer MPS-801, Commodore's current printer offering for its small computers, is the way it takes a standard size of paper. It's also much sleeker, more modern-looking in fact; it's quieter, which is a blessing (when printing the 1515/1525 were fine competition for a bunch of cats raking their claws across a sheet of tin); and it's faster at 50 cps.

It is also the same price, retailing in Britain at a recommended £230. This is quite competitive, though for the money you might expect a few more of the features mentioned above: against that you'd have to include a special interface adapter in the cost for a non-Commodore printer, so £230 starts looking quite good if you want to do the graphics.

The MPS-801 comes with a power lead for the mains (Commodore characteristically doesn't include a plug) and a second lead to connect it to the 64. If you don't have a disk unit on the 64, this cable goes into the serial port (the round socket farthest from the large expansion slot). If you do have a disk, that'll be plugged into this socket and the printer then connects to the spare socket at the back of the disk.

The ribbons on the 1515/1525 could be a bit of a pig to fit – it came as a funny little cassette arrangement consisting of two pulleys in plastic cases with a continuous loop of inked ribbon running between them. The cassettes slipped into plastic lugs inside the printer, with the front edge of the ribbon passing the other side of the print head: actually getting the cassettes to seat themselves properly on their lugs could take some time.

The 801 ribbon is a much saner slot-in plastic cartridge with a kind of ink pad – so you don't have to replace the whole cartridge when it runs out, just the ink pad (and at £6.90 per cartridge this is just as well). It too can be awkward to fit on to the retaining lugs, but at least you don't end up with inky fingers and loops of ribbon all over the place.

Once that's in you can inserting some paper. This too is reasonable straightforward once you've got

the knack, though again it can be a bit fiddly. You have to drag the tractor feed clips and the central rubber support along a hexagonal bar to get them more or less into position to take the paper; the paper itself is inserted under a hinged plastic flap which separates incoming from outgoing (the manual, to Commodore's usual pitiful standard for peripherals' documentation, doesn't mention that this flap comes away in your hand to simplify things).

Since paper feed is from the back you'd expect the rear of the printer to be free of obstruction. No such luck. That's where the two serial ports and the device selection switch are located – *two* ports so that you can daisy-chain two 801 on the same 64 (Lord knows why you should want to do that, but one of the serial ports can take a lead to the second printer while the other goes to the 64 or disk drive).

The device selection switch has three positions. One sets for self-testing, with the printer running continuously through its entire repetoire of characters until you switch off. If you have the power switched on and your ribbon and some paper in, you can move that switch to the 'T' position – whereupon the printer will screech into life.

If you don't get something like the sample, the first place to look is your ribbon – is it seating properly? The next thing to check is the whereabouts of your dealer's phone number: call and tell them what's wrong. You might need your guarantee card and the printer's serial number before you'll get any action or advice.

The other two select the device number of the printer – 4 for the first printer, 5 for the second (like most Commodore peripherals the MPS-801 has a degree of built-in 'intelligence' to decode incoming control information like device number). If you plug in a second printer you have to give them different device numbers unless you want the printing to be done on both printers at the same time.

The quality of the manual is one irritating aspect of the printer: another is the way it's not particularly easy to arrange the paper. You need a pile of the stuff to feed into the printer and another pile for the output; the sensible thing to do is to stack the incoming paper *under* the printer somehow and let the outgoing paper pile up just behind it.

This obviously requires some elevation for the printer. If you're handy with metal tube, you could try making some legs that replace the rubber feet on the underside of the printer: six inches or so should be about right. Alternatively, build or buy a platform – there's at least one manufacturer that does a moulded see-through perspex stand at about £18, but a wooden stand shouldn't be too difficult to make.

The 1520 plotter

The Commodore 1520 is a small four-colour plotter using small ballpoint pens to draw continuous lines. It's not particularly expensive, so it can be handy (and fun) for producing pictures, graphs and business-style graphics.

On the other hand, it's slow and rather limited. It uses roll paper 5.25in wide, which isn't a great width; and while it can do text characters (80 per line) it can't manage the Commodore graphics. Nor is it too suitable for high-resolution work, putting on to paper a high-res dot-by-dot colour display form instance (a graphic 'screen dump') – for that you're better off using a printer that produces dots just like the screen does.

The 1520 is device number 6, so the remarks following can apply to it once you've specified that device in an OPEN statement.

Printing ... at last!

Whatever printer you have, you actually get it to print something by first telling the 64 to open a link to it. The notes that follow assume you're using one of Commodore's printers; the documentation provided with a printer-plus-interface-adapater combo should tell you how how things differ for a non-Commodore set-up.

To start printing you kick off with an OPEN command followed by a file reference number and a device number ...

OPEN *file reference number*, **4**

The device number will usually be '4', the file reference number is anything in the range 1 to 255 as previously described.

So the command OPEN 1,4 lets the 64 know that a device called '4' is sitting on the end of an input-output channel called '1'. And to actually persuade the printer to print, you have a version of the PRINT# command followed again by the file reference number specified in the OPEN and with a comma coming before whatever it is you want printed.

This sequence will print a message in Direct mode:

```
OPEN 4,4
PRINT#4,"THE PRINTER LIVES!"
```

Or you can run it on one line:

```
OPEN 4,4: PRINT#4,"THE PRINTER LIVES!"
```

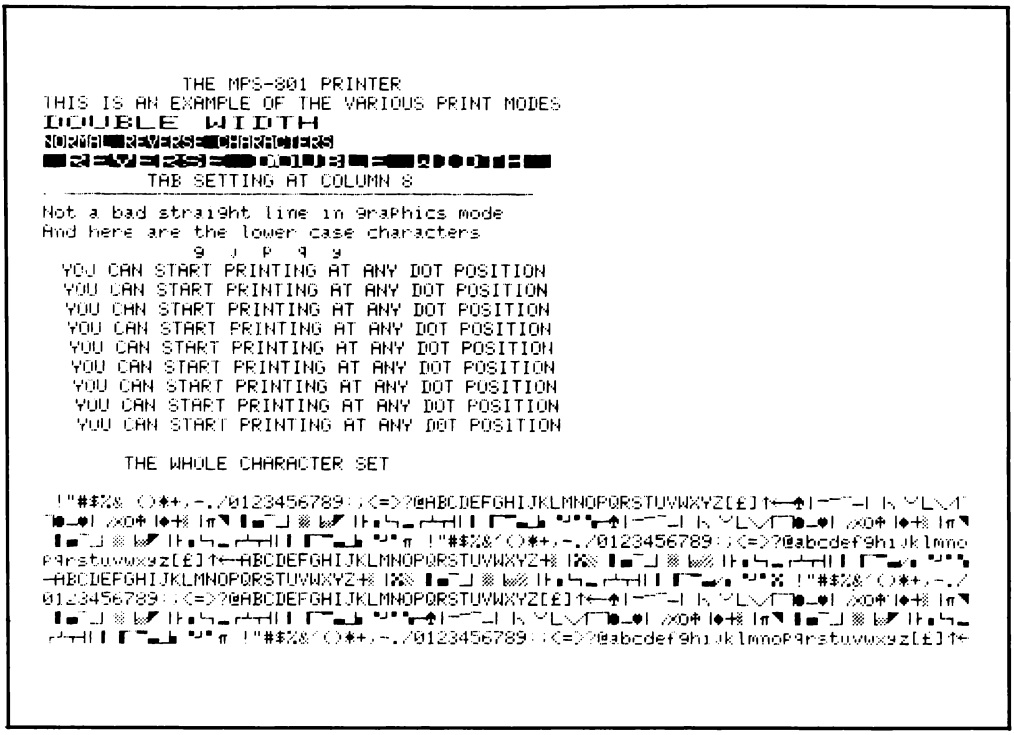
Or you can include it in a program:

```
10      OPEN 4,4
20      PRINT#4,"THE PRINTER LIVES!"
30      CLOSE 4
```

After OPENing the link you obviously have to CLOSE it (with that logical file number once again) before you can do anything else. Otherwise you're likely to get a curt FILE OPEN ERROR message.


commodore
matrix
printer
MPS 801





Selecting lower-case

There's one extra you can use with the OPEN command, a third parameter to specify whether printing should use the upper-case-plus-graphics character set or the upper-and-lower-case alternative.

The OPEN assumes that it's printing the upper-case-only set unless you tell it otherwise. To specify upper and lower case you append a '7', like this ...

```
OPEN 4,4,7
```

Printing will stay in the upper-and-lower-case character set until you reset it to the normal upper-case-and-graphics with a '0' as the third parameter:

```
OPEN 4,4,7
```

Selecting the lower-case print option with the OPEN is useful, but it's a blanket effect: it works for any and all text to be printed. If you want to be selective about what is and isn't to appear in lower case, you should use CHR$(17) in your PRINT# statement before material to be printed. For example:

```
30 PRINT#1, CHR$(17) "LOWER CASE!"
```

This overrides the OPEN statememnt – so CHR$(17) can give you lower-case characters even though you might have OPENed 4,4 or 4,4,0 for upper-case. Use CHR$(145) to cancel this:

```
30 PRINT#1, CHR$(17) "LOWER CASE!";
CHR$(145) "UPPER CASE!"
```

Ordinarily you'd expect CHR$ to produce an alpha-numeric character, but there are a number of CHR$ codes available for printer control: the panel lists them.

LISTing programs on the printer

The main reason for investing in a printer will probably be the appeal of a printed copy of programs – useful in checking out short programs, probably vital for anything longer.

The LIST command normally works by displaying your code on the screen, but there's a command – CMD – which switches to a specified device anything that would normally appear on the display. It has a simple format:

CMD *file reference number*

The file reference number must previously have been OPENed for the particular device. So this Direct-mode sequence is one you'll probably be using a lot:

```
OPEN 4,4
CMD 4
LIST
```

Specification of the MPS-801

- **Character printing**:

matrix 6x7
character set upper- and lower-case characters, numerals, symbols, Pet/Vic/64 block graphics
code CBM ASCII
height seven dots (total 2.82mm)
width normal width six dots (total 2.53mm)
characters per line 80 (normal width) or 40 (double width)
characters per inch 10
lines per inch six
line feed speed five per second

- **Graphics printing**:

vertical resolution seven dots per column
horizontal resolution width 480 columns
lines per inch nine
line feed speed 7.5 per second

- **Paper**

paper feed tractor
minimum width 4.5in (including sprocket holes)
maximum width 10in (including sprocket holes – 8.5in after sprocket holes removed)

- **Performance**

speed 50 characters per second
print head movement left to right, unidirectional

- **Physical**

height 115mm
depth 237mm
width 438mm
weight 4.8kg

That puts the result of the LIST command – the listing – on to the printer. With a Return after the CMD like this you'll get the word 'READY' printed before the LISTing; you can skip it by stringing the commands together on one line ...

```
OPEN 4,4: CMD 4: LIST
```

CMD can also be used within a program. This would cause the program to LIST itself:

```
200 OPEN 4,4: CMD 4: LIST
```

Once CMD is in operation, *all* output that would normally appear on the screen will now be 're-directed' to the printer. That includes the 'READY' message as mentioned; so will things like 'PRESS PLAY ON TAPE' and 'SEARCHING', the user's responses to an INPUT request, and all the information generated by PRINTs.

The CMD is switched off automatically and the screen will return to being the default output device as soon as the 64 detects a SYNTAX ERROR or an ILLEGAL QUANTITY ERROR. Run/Stop and Restore will also do that, of course. For a deliberate and orderly return to the screen you'll have to PRINT a blank line to the device by using a PRINT# statement like this:

```
OPEN 4,4: CMD 4: LIST
PRINT#4
```

Attaching non-Commodore printers

Commodore's printer isn't a bad piece of work. It looks reliable, prints quite quickly, isn't too noisy, and does handle Commodore graphics; nor is it an unreasonable price by comparison with the alternatives. But there are many other (and possibly better) printers in a well-filled marketplace.

The problem is that with very few exceptions you won't be able to plug them in directly; that's because most printers use either the RS232 or Centronics interface standard for the plug-and-socket connection to a computer while Commodore ploughs a lonely furrow in offering the IEEE alternative. (Not of course that there's any suggestion Commodore espouses IEEE simply because it means you're more likely to buy Commodore add-ons from Commodore rather than from anyone else ...)

So you'll need some kind of adapter to persuade a non-Commodore printer to run on the 64. You can make up the electrical connections and wire the plugs yourself: you'll probably have to write some conversion software too to send stuff to the printer. Or you can buy a ready-made interface – which may or may not come with the necessary conversion software. It will at least fit into one of the standard sockets on the 64 or Vic – usually the disk drive DIN-plug socket or the spare socket on the back of a disk drive, sometimes the expansion slot, just occasionally the user port.

For cut-price printing there's the Softex Printerface at a mere £22 or so. That money buys the necessary to plug a Sinclair Microprinter (shop around and you should find one of those for £45 or less) into a 64: designed and sold for the ZX81 and Spectrum, the Sinclair printer is a neat and cheap little box. But it uses special paper (because it's electrostatic – it works by scorching the silver coating off the paper in dot-shaped areas to reveal the blackness beneath) and it prints only 43 characters per line.

On the other hand, the Softex converter handles Commodore graphics rather well; and they appear on the Microprinter better than on the standard

CHR$ printer controls

You can PRINT# anything on the printer that you can PRINT on the screen – variables, character strings in quotes, calculations. You can also force the printer to do things that aren't so easy on the screen; and you can use CHR$ codes with PRINT# also to get some formatting and other facilities for which you'd probably use POKEs to achieve on the display.

All these extras you get by using a CHR$ expression with a code number after the PRINT#. Here's a summary of them – the examples assume that the printer has been opened with an OPEN 1,4 statement.

CHR$(8) ... switches to graphic mode for the printer. Line spacing changes to 9 lines per inch (so there's no space between lines) and you can define a shape within a 7x7 grid to have the printer print the individual dots you specify.

There are two ways you can do this: one uses DATA statements containing seven values per graphic, those values being the decimal numbers you get by converting the binary representation of the seven rows that make up a character (binary 1 in each position in the row that you want a dot and a binary 0 everywhere you want a space. You then READ the data and have the 64 print the result via a PRINT# statement with variables.

Alternatively you can do it with CHR$ codes. Treat the dot columns as seven positions with 0 being the top and 6 the bottom. For each dot to be printed in the vertical plane, note the decimal equivalent as if it were a seven-bit byte (dot 0=1, dot 6=64): sum these numbers, add 128, and use the result in a CHR$ expression. So this prints an oblong in ten columns of dots:

```
PRINT#1,CHR$(8);CHR$(255);
CHR$(193);CHR$(193);CHR$(193);
CHR$(193);CHR$(193);CHR$(193);
CHR$(193);CHR$(193);CHR$(255)
```

CHR$(10) ... forces a line feed after printing – so the paper automatically rolls up one line. But with most printers it doesn't also return the print head to the leftmost position, instead leaving it in the same position within the line that it finished on. So it can a bit dodgy: the next line you want printed will start from that position rather than the lefthand edge. Advice? Stick to CHR$(13).

CHR$(13) ... forces a carriage return after printing – the paper automatically rolls up one line and the print head goes back to the lefthand edge. So it's generally safer than CHR$(10).

CHR$(14) ... causes everything following it that's to be printed to appear double-width (so you get only 40 characters a line rather than 80). For example:

```
30 PRINT#1,CHR$(14)"DOUBLED!"
```

CHR$(15) ... cancels double-width or graphics modes – subsequent printing will be normal-width text.

CHR$(16) ... a kind of tab effect: this sets a start position for the next printing to be done. Exactly what that position is is determined by the first two digits of the next text string, which should be in the range 0 to 79. For example:

```
30 PRINT#1,CHR$(16)"20PRINT AT
POSITION 20"
```

That will start the print 21 spaces from the lefthand edge.

CHR$(17) ... switches to the upper-and-lower-case character set with some graphics, the same characters that you'd get on the keyboard by hitting the Commodore key while holding down SHIFT. For example:

```
30 PRINT#1,CHR$(17)"LOWER CASE!"
```

This overrides the OPEN statememnt – so CHR$(17) can give you lower-case characters even though you might have OPENed 4,4 or 4,4,0 for upper-case. Use CHR$(145) to cancel.

CHR$(18) ... everything following will be printed reversed. For example:

```
30 PRINT#1,CHR$(18)"REVERSED"
```

It is cancelled by CHR$(146); CHR$(10) and CHR$(13) also kill the effect automatically. The alternative method of printing reversed for one string only is to use the reverse-on character in the string:

Commodore 1515/1525 and MPS-801 because the characters are printed larger, so they're more legible.

Otherwise your non-standard printer will probably be a conventional dot-matrix plain-paper-and-inked-ribbon printer that offers rather more than the Commodore units (at a commensurate increase in price). You should expect a better quality print from them, with rather less wobble on the printed lines and true descenders too.

Most such printers have a Centronics interface. That's probably the most widely adopted standard for plugging in printers, and it got its name because

```
30 PRINT#1,"<rvs>ONE REVERSED LINE"
```

CHR$(26) ... a handy method of repeating any graphic data – prints the same line of vertical dots as many times as you specify. The format is ...

CHR$(8); CHR$(26); CHR$(***number of repetitions***); CHR$(*data to repeat***)**

So the example given for CHR$(8) could more economically be done like this:

```
PRINT#1, CHR$(8); CHR$(255);
CHR$(26); CHR$(8); CHR$(193);
CHR$(255)
```

It doesn't have to be your user-defined graphics that's repeated (but you have to have a CHR$(8) preceding it) so it's particularly useful for repeating things like barcharts.

CHR$(27) ... works with CHR$(16) to set a start position for the printing of user-defined graphics. The 64 assumes 480 positions per line (numbered 0 to 479) so you need two parameters to specify the start in terms of number of dots away from the lefthand margin (because 255 is the largest number you can have in a CHR$ expression). The format is ...

CHR$(27); CHR$(16); CHR$(***high number***); CHR$(*low number***)**

The printing position is then computed as **(256 x high number) + low number**.

CHR$(143) ... everything following will be printed in the normal character set of upper-case letters and graphics. That's how you start of, so this is used primarily to get back to normal from a CHR$(17) command.

CHR$(146) ... switches off the reversed print you got from a CHR$(18) to get back to normal.

CHR$ commands can be mixed in a PRINT£ statement. For example:

```
30 PRINT#1, CHR$(13) CHR$(14) CHR$(18)
CHR$(143) CHR$(16) "09EXAMPLE AT 9"
```

This will give a carriage return and then print the message in reversed expanded upper and lower case starting ten characters in from the left edge.

a printer manufacturer called Centronics came up with it first. The easiest (perhaps the only way) to use it is to buy one of the several ready-to-go printers that plug into the user port – should cost less than £30, come with an appropriate cable, and be supplied with the necessary software to convert your output to use it.

The Centronics interface is parallel, which means it uses eight wires so that the eight bits which make up a character all arrive at the same time at the printer. The other likely candidate is RS232, a serial interface – the bits arrive along a single wire one after the other. RS232 is probably the most common method of attaching one device to another in the computer business, and printer many printers therefore assume that you'll want to use RS232. Again, there are several independently-available Commodore-to-IEEE converters available at prices from £25 or so.

In both cases you can spend more to get more. The Centronics converter I use has the facility to recognise a secondary address on the OPEN command that automatically selects (or deselects) extras like automatic conversion into mnemonics of graphics codes, suppression of automatic line feed on Return, and so on.

There's another kind of add-on interface adapter you might consider: this converts the 64's version of IEEE into the variety employed by Commodore's up-market computers (the 64's IEEE is a simplified and not very standard implementation of the IEEE specification, the Pet/8000/700 version is the full-scale one – 64 IEEE is a serial interface, the 'real' one is parallel). With this kind of adapter you would be able to use the bigger and faster disk drives sold for those computers, and that's primarily what those IEEE converters are sold for; but you could also use them to plug one of Commodore's bigger printers into a 64.

Most non-Commodore printers don't have the ability to print the control graphics symbols from 64 listings automatically. If you don't have a code-converting interface adapter (and most of them don't do the translations into mnemonics for you) you could try writing a conversion program; the one here is Chris Durham's highly effective number for the £450 Epson FX-80, and that should give you a few clues about how it's done.

RS232 on the 64

The 64 actually has a kind of RS232 interface built in, but it's a slightly odd version of the standard RS232.

The key difference is the voltages that represent 0s and 1s on the connection lines. For 'true' RS232 they're +12V and -12V; for no particularly good reason Commodore opted for +5V and 0V. If you want to use an RS232 device on the 64 then you'll probably need a 'level converter', a box that applies the correct voltages to signals to and from the computer.

In practice connecting an RS232 printer to the 64 is more simply done by using an RS232-to-IEEE converter – that enables you to plug the RS232 printer in and use it exactly as you would the Commodore printers.

```
10 REM ****************************************
20 REM PROGRAM TO DOWNLOAD COMMODORE CHARACTER SET
30 REM TO AN EPSON FX-80 PRINTER - BY CHRIS DURHAM
40 REM ****************************************
45 POKE 52,152:POKE 56,152:CLR:REM RESERVE SPACE IN MEMORY FOR CHAR SET
47 POKE 53280,14:POKE 53281,6
50 PRINT"{CLR HOME}{CUR DN}{CUR DN}  SWOP CHAR SET INTO MAIN MEMORY"
60 CS=53248:CL=CS+512:ML=38913
70 PRINTCHR$(142):REM SWITCH TO UPPER CASE
80 POKE 56334,PEEK(56334) AND 254:REM TURN OFF KEYSCAN INTERRUPT TIMER
90 POKE 1,PEEK(1) AND 251:REM SWITCH IN CHAR SET
95 FOR A=0 TO 511:POKE ML+A,PEEK(CL+A):NEXT A:REM TRANSFER CHARS
100 ML=ML+512:FOR CH=1 TO 27
105 READ X:FOR A=0 TO 7
110 IF CH<25 THEN POKE ML+A,255-PEEK(CS+(X*8)+A):REM TURN INTO REVERSED CHARS
115 IF CH>=25 THEN POKE ML+A,PEEK(CS+(X*8)+A): REM CHARS NOT IN EPSON SET
120 NEXT A:ML=ML+8:NEXT CH
125 POKE 1,PEEK(1) OR 4:REM SWITCH IN I/O
130 POKE 56334,PEEK(56334) OR 1:REM TURN ON KEYSCAN INTERRUPT TIMER
135 PRINT"  CONVERT CHARS TO PRINTER FORMAT{CUR DN}"
137 DIM B1(8):FOR A=0 TO 7:B1(A+1)=2^A:NEXT A
140 PL=39700:MP=38913
145 FOR Y=PL TO PL+546:POKE Y,0:NEXT Y
150 FOR Y=PL TO PL+540 STEP 6
160 FOR A=7 TO 2 STEP -1
170 FOR B=0 TO 7
180 IF (PEEK(MP+B) AND B1(A)) THEN POKE(Y+7-A),PEEK(Y+7-A) OR B1(8-B)
190 NEXT B,A:MP=MP+8:NEXT Y
200 OPEN4,4
210 REM TRANSFER EXISTING EPSON CHAR SET TO USER AREA
215 PRINT#4,CHR$(27);"R";CHR$(0);:REM SELECT USA SET
220 PRINT#4,CHR$(27);":";CHR$(0);CHR$(0);CHR$(0);
225 PRINT"  NOW TRANSFER COMMODORE CHARS{CUR DN}"
227 FOR L=1 TO 2:READFC,LC
230 PRINT#4,CHR$(27);"&";CHR$(0);CHR$(FC);CHR$(LC);
235 FOR CH=0 TO 31:PRINT#4,CHR$(139);
240 FOR A=0 TO 4
250 PRINT#4,CHR$(PEEK(PL+(CH*6)+A));:PRINT#4,CHR$(0);
255 NEXT A:PRINT#4,CHR$(PEEK(PL+(CH*6)+5));
260 NEXT CH:PL=PL+(32*6):NEXT L
262 REM ALLOW ALL ASCII CODES (0-255) TO BE PRINTABLE
264 PRINT#4,CHR$(27);"I";CHR$(1);CHR$(27);"6";
266 PRINT"  NOW TRANSFER CONTROL/COLOUR CODES{CUR DN}"
268 REM ALSO INCLUDES CHARS NOT IN STANDARD EPSON SET
270 PL=40084:FOR CH=0 TO 26
280 READ CP
290 PRINT#4,CHR$(27);"&";CHR$(0);CHR$(CP);CHR$(CP);
300 PRINT#4,CHR$(139);
310 FOR A=0 TO 4
320 PRINT#4,CHR$(PEEK(PL+(CH*6)+A));:PRINT#4,CHR$(0);
325 NEXT A:PRINT#4,CHR$(PEEK(PL+(CH*6)+5));
330 NEXT CH
335 REM SWITCH TO USER DEFINED CHAR SET
340 PRINT#4,CHR$(27);"%";CHR$(1);CHR$(0);
350 PRINT#4,CHR$(27);"E";:REM SET EMPHASISED MODE
360 PRINT#4:CLOSE4
370 PRINT"{CLR HOME}{CUR DN}{CUR DN}{CUR DN}      CHAR SET TRANSFER COMPLETE":PR
INT
375 PRINT"      COMMODORE CHAR SET SELECTED{CUR DN}"
377 PRINT"****************************************{CUR DN}"
380 PRINT"TO SELECT EPSON CHAR SET, TYPE:"
390 PRINT"{CUR DN} {YEL}PRINT#4,CHR$(27);"CHR$(34)"%"CHR$(34)";CHR$(0);CHR$(0);{
WHT}{CUR DN}{CUR DN}"
400 PRINT"TO RE-SELECT COMMODORE CHAR SET, TYPE:"
410 PRINT"{CUR DN} {YEL}PRINT#4,CHR$(27);"CHR$(34)"%"CHR$(34)";CHR$(1);CHR$(0);{
WHT}{CUR DN}{CUR DN}"
420 PRINT"ENSURING STREAM 4 IS OPEN FOR PRINTER   OUTPUT.
430 END
1000 DATA80,5,28,95,92,30,31,94,65,85,86,87
1010 DATA88,89,90,91,18,70,83,19,81,17,66,29,28,31,94
1015 DATA192,223,160,191
1020 DATA144,5,28,159,156,30,31,158,129,149,150,151
1030 DATA152,153,154,155,18,146,147,19,145,17,157,29,92,95,255
```

Chris Durham's converter program allows the Epson FX-80 to produce the Commodore 64's graphics. This first appeared in Commodore User magazine.

What you might well need the 64's RS232 for is communications, particularly communications involving a modem that allows you to send and receive information over phone lines. Alternatively you could hook another computer directly on to a 64 for phoneless transfers.

Wiring for RS232

The procedure for using RS232 starts with the acquisition of a suitable connecting cable. This should be wired as shown.

The key requirement is that the device's *transmitted data* line should be connected to the 64's *received data* line – and *vice versa*.

Incidentally, if you're using the Commodore VIC-1011A RS232 interface (which is basically a voltage level converter that also allows you to use a conventional D-type RS232 plug at the 64's end of things) you may find that the plug has to be wired such that the connections to pins 2 and 3 are swapped. That may also apply to pins 4 and 5.

This is something you'll have to check for yourself, since it depends on what connections the device at the other end of the cable is expecting.

OPENing up for RS232

Now you can set up the interface, essentially by OPENing it rather as you would for any other external device:

OPEN *file reference number*, **2**, *secondary address*, *RS232 specification*

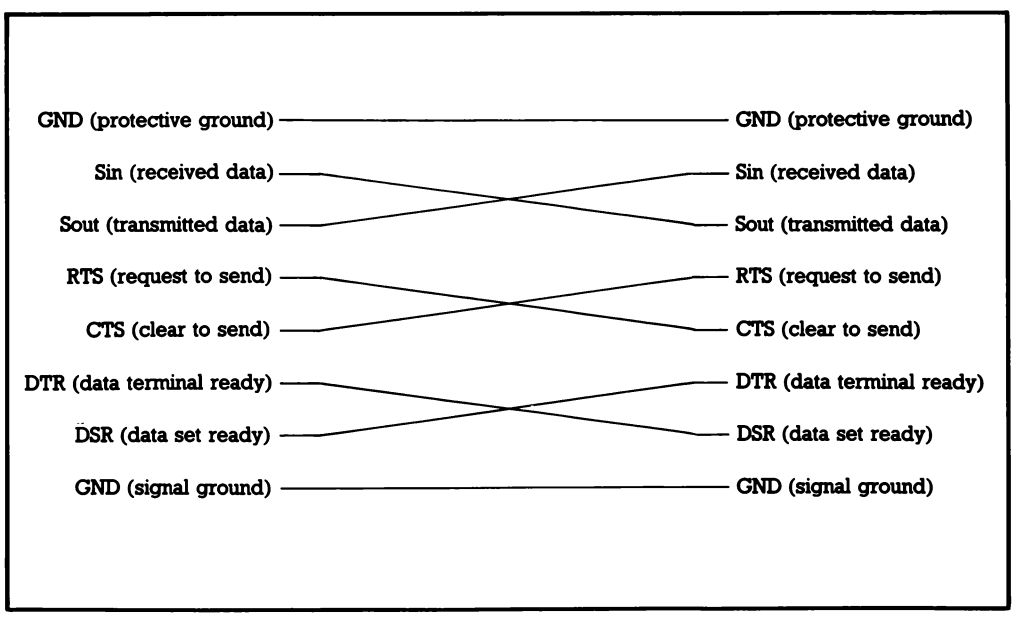
A program that OPENs an RS232 channel like this should have the OPEN command as its first statement. This is because the 64 allocates RS232 a couple of 256-byte areas at the top of Basic memory to serve as input and output buffers respectively; and if your program is cheerfully setting up variables and arrays *before* the OPEN, you may find to your chagrin that these are overwritten by the 64 when it executes this OPEN (it practices what happens is that the OPEN causes an automatic CLR).

So define your variables *after* the OPEN; that way nothing will get wiped.

The *file reference number* is the usual user-selected value in the range 0 to 255. It's important to note here though that values of 128 and above have a special meaning – they'll cause an automatic line feed after every Return. That could be useful (may even be essential) if you're trying to drive an RS232 printer that can't be set to give the automatic line feed itself. On the other hand it could be embarrassing if you're sending or receiving something that does have the built-in line feed: you'll get a *double* line space.

The device number for RS232 communications is 2. The *secondary address* component is pretty well undocumented; it's sometimes referred to as the 'communications channel', which doesn't exactly help since no-one seems to know what it's for. As a rule of thumb, putting '0' in there seems to work most of the time; '3' is used by Commodore in its own documentation for the 1011A to connect RS232 printers.

The RS232 specification is a string of CHR$ codes, up to four of them. The decimal values with these decide the parameters for RS232 transfers in terms of the bit settings of the CHR$ byte. (In practice the parameters will usually be determined by what the device on the other end of the link is expecting.)



Code one

bit 7 – sets number of stop bits transmitted with each word. When set to 1, two Stop bits are sent; when 0, one Stop bit is sent.

bits 6 and 5 – specify word length to be transmitted. These determine the number of data bits per word, as follows:

bit 6	bit 5	word length
0	0	8 bits
0	1	7 bits
1	0	6 bits
1	1	5 bits

bit 4 – not used.

bits 3, 2, 1 and 0 – specify baud rate. In theory these four bits allow enough combinations to denote 16 different possible speeds for the data transfer, but in practice speeds above 2400 baud aren't feasible with the 64. The combinations allowed are:

bit 3	bit 2	bit 1	bit 0	baud rate
0	0	0	1	50
0	0	1	0	75
0	0	1	1	110
0	1	0	0	134.5
0	1	0	1	150
0	1	1	0	300
0	1	1	1	600
1	0	0	0	1200
1	0	0	1	1800
1	0	1	0	2400

Of these, the common European rates are 0110 for 300 baud (low-speed data communications through a modem or acoustic coupler) and 1000 for 1200 baud or 1010 for 2400 baud to link to a printer.

A setting of 0000 for these bits indicates that the baud rate will be specified in the third and fourth of the CHR$ codes following the OPEN.

Code two

bits 7 to 5 – set parity options, as follows:

bit 7	bit 6	bit 5	parity
0	0	0	none
0	0	1	odd
0	1	1	even
1	0	1	mark
1	1	1	space

bit 4 – specifies full- or half-duplex. Set for half-duplex.

bits 3 to 1 – not used.

bit 0 – specifies handshaking protocol. Set for X-on/X-off.

Codes three and four

These two CHR$ codes are optional, but if code 1 is CHR$(0) a user-specified baud rate is provided in them. The values are calculated from this formula:

baud rate = frequency / (baud / 2) – 100

The *frequency* is the video picture frequency for your part of the world. In the States and other countries using the NTSC standard it's 1022730 Hz; for PA:-using lands like most of Europe it's 985250 Hz.

The two CHR$ codes provide a two-byte summary of this. Calculate the high byte first, though this is actually the second of the two code:

CHR$ baud-rate two = INT (baud rate/256)

The first of them, the low byte, is computed as ...

CHR$ baud-rate one = baud rate – (256 * baud-rate two)

Using RS232

Once you've got your OPEN straight, you can send and receive. To send you use CMD and PRINT£ commands (in both cases using the file reference number in the OPEN, of course).

To receive you can use either GET£ or INPUT£, though GET£ is generally preferable – INPUT£ may hang if no Return is received. Either way you should use the command as frequently as possible; otherwise the receive buffer may overflow as new information is stuffed into it.

At the end of the transmission you should of course CLOSE the file reference number. But make sure this just about the last statement in the program, for CLOSE performs another CLR and anything that's still in the buffer areas will be wiped.

Errors on the RS232 link can in theory be checked for by reading the STatus variable. There's a problem, though: a bug in ST with RS232 means that any attempt to check ST actually clears it, so by the time the 64 reports ST's status it will show no problems at all. Instead you can PEEK (663) to get the same information.

The error bits will be set as follows:

```
bit 7  ...   break detected
bit 6  ...   no DSR signal
bit 5  ...   [unused]
bit 4  ...   no CTS signal
bit 3  ...   receive buffer empty
bit 2  ...   receive buffer overflow
bit 1  ...   frame error
bit 0  ...   parity error
```

Section 4

The 64 in business

CHAPTER 22
Buying Business Software

There are a number of reasons why you might consider using the 64 in your business. It is after all a *computer*, just like any other computer: disks are available for it: and – more important – there's a good selection now of really rather fine business-oriented software for the 64.

The 64 makes a good business computer ...

For a start, let's consider the 64 as a serious business computer rather than a recreational tool for home use.

- On the plus side there is its **keyboard**. It may feel a bit lightweight; but it is pretty robust, and it is certainly good enough for business use – even for word processing and other tasks which involve a lot of keying and therefore rely on the availability of a comfortable and sensible keyboard.

- The substantial **memory** of the 64 is also in its favour. You will probably need it all if you are going to buy and use ready-to-run business packages, since the facilities and functions they offer will demand as much memory as possible. You probably can't run a reasonable general-purpose business system on a computer with less than 64KB. On the other hand, many of the more conventional business microcomputers have no more memory than the 64K in this 'home computer'.

- There is a good deal of mileage for business systems designers in the **colour** options of the 64. Colour changes on the border and/or the text can be used to highlight different types of message or different operations, for instance. And a sensible use of colour can make form-filling on the screen more or less idiot-proof. Since *any* program should aim to be as helpful towards the user as possible, this can be a major plus.

- The same applies to the availability of the **function keys**. These are large and unmissable – ideal for single-keystroke responses to select a particular course of action that otherwise might require a full-word multi-key input of the user.

... Or not

On the debit side there's the quality of the **display**. A TV screen produces a rather coarse and ill-defined image; its legibility might simply be too poor for concentrated use. The solution, as in many cases, is the judicious application of additional quantities of money: buy a purpose-built monitor – that will provide a much crisper image.

The display is also **limited** to 25 lines of 40 characters each. That really isn't enough: conventional business computers typically show double that, 25 lines of 80 characters. Depending on the function of the package, the supplier may have a way of getting around this restriction: with many word processors and most spreadsheet calculators, for instance, the screen is treated as a kind of window that lets you look on to a much larger workspace – you can scroll or 'pan' across a document or a spreadsheet, for instance, so that material on the left of the screen appears to disappear off the edge as you move the cursor to the right.

A couple of suppliers are now trying to promote '80-column converters' with their packages. These are hardware add-ons that stretch the screen for you; that way you do get 80 characters to a line, but they're pretty small.

Business accounting packages and file managers, both of which have yousetting up and amending records, generally take the other approach; the 'record form' is spread across two or more 'pages' and you get to see only page at a time.

There again, the cheapest and simplest packages adopt the cheapest and simplest approach – and you find yourself limited to one screenfull of information per record.

Another problem is the need for **disk storage**. It *is* possible to use cassettes for business work on the 64, but the limitations are legion. Cassettes much slower than disk, which means it simply takes longer to load the information you want to work with and save new data; that effectively rules out any quick look-up.

More important, cassettes are *serial* – which means the tape has to be rewound and read right from the beginning if you're searching for particular records. If you have a customer file on tape and you want to get the details for Zamanelski Plastics to print a

statement, you're going to have to read through all the preceding customer records before you find it.

And once you have found the record you'll be greatly restricted in what you can do with it. Since the records follow one another in a rigid sequence, adding new information to one record in the middle of a tape file is virtually impossible. Some space may have been left in the original format, but some extremely fancy programming would be required to amend only one record in a file. The more usual procedure with tape-based systems is much more tedious: you have to read the whole file into memory, make the changes, then save the file again on tape.

Not only are disks faster in operation, they can also operate in a much more flexible way. Records do *not* have to be stored on the disk in strict sequence; so it is possible to go directly to a record that you want to amend and change only that one.

Speed and ease of access thus militate in favour of your adding a disk drive to a basic 64 system; and in any case the best packages – those that can do most for you, in the best possible manner – are available *only* on disk. You may not even have the choice of using tape.

So you've almost certainly got to buy a disk; and that will probably cost you more than the 64 itself did. Add the price of a monitor and you're approaching the £750 mark.

And you may not be satisified with the 1541 disk drive that Commodore sells for the 64. This is a low-capacity disk by contrast with industry standards; and while it is much faster than cassette it is relatively slow in operation. It's also a single-drive unit, and there are important advantages in have two disks attached to a business computer – notably for when you take security back-up copies of your files.

One day disaster will strike, and in that eventuality you do need a fairly up-to-date copy of your key business files; taking a copy of the contents of a disk when you have twin drives is easy, for you can just copy whatever's on the disk in one drive over to the blank disk waiting in the other drive. But with a single-disk set-up the copying involves reading some of the disk's contents into memory, taking out the original disk and putting in a blank one, transferring the contents of memory to the fresh disk, then re-inserting the 'copy from' disk, and doing it al over again ... several times, perhaps.

There are ways around this, like you can buy and use one of the Commodore disk drives normally sold for the 700 and 8000 series computers. They are twin-disk units, and they have the additional merit of storing more informationm per disk and operating much faster than the 1541.

Against that, of course, there's the price tage – £800 or more per disk drive unit. There's the complication of actually plugging it in – you'll need an adapter of some kind (more expense) since the 64 wasn't designed to work with them. And there is always the danger that the software you intend to use will not run on the bigger, faster Commodore disks.

You haven't finished signing cheques yet, either: you'll need a printer. And very few printers plug straight into a 64. Those that do are intended for the home user, so they are built as cheaply as possible – and it shows. For business use you'll need a much better printer; and to use that you'll have to pay extra for a another adapter unit.

Your £200 Commodore 64 home computer could easily end up being a £800-plus business system even if you stick with the 1541.

Now read on

Don't despair, though: you're still saving money by comparison with purchase of a conventional business computer. Whether you'll save money in the long run is a matter for conjecture – shopping around for the different bits you need can be a drag, for instance, and it can certainly lead to much finger-pointing and protestations of innocence from all the suppliers if something goes wrong.

You can cover yourself a bit by looking for the software you want and then asking that supplier to advise on the hardware. The package purveyor may be able to furnish the hardware as well, in which case you should get the assurance that everything *will* work: besides, that way you have only focus for any complaints. At the least the software house should be able to recommend hardware – and probably a supplier too.

Buying Packages

The point of the pre-written software termed 'packages' is that in practice most organisations have roughly similar requirements to do roughly similar things – word processing, ledger accounting, debtor analysis, VAT analysis, maybe inventory control and some financial planning. So the same software can apply more or less to each.

The other side of this particular coin is that the package solution is inherently designed for a mass market. An attempt to be most things to most people will almost inevitably result in some deficiencies from your point of view in what the package does and how it does it.

But software is expensive to produce – even if you try to do it yourself. Designing, writing and testing programs is time-consuming and labour-intensive, and you certainly can't afford to make any mistakes if you're going to run your business from the software. The package option ought to carry some guarantee of performance and freedom from errors; and at least if it doesn't work you'll have someone from whom to seek redress.

Do-it-yourself packages

There are a few possible pitfalls lying in wait for the purchasers of some packages, especially those for accounting. A ledger package might allow little flexibility in the size and content of records, in the transaction processing procedures, in the design of reports and other printed output.

This is where you may find your business procedures differing in small but possibly significant aspects from the idealised (or at least generalised) approach of a package.

But a 'file manger', for instance, will typically impose few restrictions on what kind of files you design with it – and how you handle those files. And such packages will include sort-and-select functions, the ability to perform arithmetic on the contents of records (like running totals and VAT percentages), and a wide variety of user-definable output options for your reports, summaries, order documentation and so on.

So it is feasible to develop your own accounting systems by using an off-the-shelf file management package. In practice you are using the package as a kind of programming language – except that the sequences of commands you set up are geared specifically for handling files, and they are likely to be in plainer English than the rather more esoteric commands in a conventional programming language.

A newish breed of packages called '*program generators*' take the DIY approach a stage further. They give you a halfway house between a ready-to-use package and your own attempts at programming. They are easier to use and simpler to grasp than a programming language; they let you create working systems that might fit your requirements better than a general-purpose off-the-shelf all-things-to-all-people package.

A program generator lets you say what you'd like to do, and it does allow you to use your own language (or rather a language that you can understand quite easily) without demanding not too many brand-new terms and concepts. And then it does the conversion between a near-English language and the electronics. It will lead you quite gently through the business of specifying what you want the program to do – what you want to get out of the other end, and how you want to get it.

But it won't accommodate all the possibilities – you won't be able to do anything and everything. It too will make many assumptions about what you'll be trying to do with the program you generate as the end result; it will impose several restrictions on what you can do and how you can do it.

In general, most program generators are oriented towards commercial applications. That means they are geared to *files*. They generate programs that create files, add to and amend delete from files, search through and sort and select from files, and produce reports based on file search/select operations perhaps with some simple arithmetic being performed en route.

Many business activities can be seen in those terms – but not all. You wouldn't be able to produce a kind of 'what if' financial forecasting program with a program generator, because the bulk of computing activity in a spreadsheet involves setting up and controlling different types of formulae and movements of data; that's rather different to creating and manipulating information in files, where the actual number of operations you'd want to perform is considerably smaller.

But if you have an application that can be based on records in files, and if the actual day-to-day operation can be seen in terms of amending the files and extracting information from them, you could consider using a program generator.

Comparing packages

Whatever type of package you buy, there are broad criteria that you can apply to your choice.

But just how do you compare packages when they do virtually identical jobs in apparently similar ways? It's not particularly easy to make the choice in the first place, and it's certainly not very easy to assess just how things are working out in practice after you've committed yourself.

In most cases you'll probably end up trusting the supplier who you like the look of. And that may not be such a bad thing; confidence in the supplier may be highly subjective, but you've got to trust *someone*. You can cover yourself a bit, though, and here are some general guidelines.

- **Rule One: FIND OUT WHAT YOU NEED FIRST**: Start from the viewpoint that the package has to do something for you. Get to that stage *before* you look for a supplier. Look at your general application areas: analyse what goes on in them now, so that you will be able to compare the computerised solution offered with your existing procedures. Make sure that all the contenders do what you need; and make sure they can fit quite closely to your ways of doing things.

- **Rule Two: BE FLEXIBLE**: It is important to be clear about which of your current procedures could reasonably be modified – and which cannot. Most packages and indeed most small computers owe their existence simply to the fact that they work in *general* terms, are of *general* applicability, and *in general* are as good as any others; low-cost solutions demand some flexibility on the part of users as well as of the packages themselves.

You may be surprised at some of the variations between theory and practice when you're assessing the way the organisation does things prior to computerisation. And the system you go for will

need to cater for the *actual* conditions – don't forget, computers have no common sense to apply to unforeseen occurrences and on-the-spot improvisations.

- **Rule Three: TALK TO EXISTING USERS IF YOU CAN**: You won't need to do this if you're just buying a standard word processor or a spreadsheet package; few of them are actually dire. But even then it helps to hear someone else's opinion. And if you're thinking of acquiring a package that will be central – maybe even *essential* – to your organisation, that opinion could be vital.

By getting in touch with present users of existing systems it soon becomes obvious how reliable and effective the system is when used in anger. More important, you can also find out how the *supplier* performs. How much help and advice will a purchaser get?

- **Rule Four: BE CIRCUMSPECT ABOUT MATURITY**: All software has errors in it, but new software has most. The best guarantee of the reliability of a package in this respect is that it has been used by many companies to their satisfaction for a reasonable period.

But don't get too carried away. New packages and new computer systems can be very good: apart from anything else, they will include any recent legislation. They will also exhibit new facilities and new functions (based possibility on the shortcomings of their predecessors).

- **Rule Five: TRY BEFORE YOU BUY**: It goes without saying that you should try the package. But as well as looking for the functions you're after, you should pay special attention to what it's like in operation.

Is it easy to use? You should be led through the process by instructions and lists of options (called 'menus') displayed on the screen. On the other hand, the programs should get you where you need to be as quickly as possible: you should not need repeated trips through tedious menus.

Commands to perform common tasks should be straightforward, and should not require awkward combinations of keys. They should also be easy to learn and remember.

Check out the reports and documents produced by the system. A computer can generate prodigious amounts of paper, but quantity doesn't count. Does the printed output tell you things you really want to know, laid out in a way you can understand?

And what about operator errors? Mistakes will be made: so how does the system react to them? No system is totally idiot-proof. But at least it should alert the user to the mistake without destroying any of the work already done. And if the worst happens, there should be an assessment of just how much has actually been lost and how it can be recovered.

- **Rule Six: READ THE MANUAL**: Have a look at the manuals too; ask to borrow them for a couple of days if you're unsure, and go through them at your leisure. You'll need manuals to explain the facilities that the computer and its packages provide, how they can be used and how they should be operated.

- **Rule Seven: DON'T FORGET THE EFFORT OF SETTING UP**: Getting your files on to the computer in the first place is always a drag. How much will it cost? And how long will it take?

- **Rule Eight: CHECK OUT THE ERROR HANDLING**: Mistakes will be made: so how does the system react to them? If the computer can spot the obvious blunders and watch out for wrong inputs, it will be able to compensate for the faults. No system is totally idiot-proof. But at least it should alert the user to the mistake without destroying any of the work already done. And if the worst happens, there should be an assessment of just how much has actually been lost and how it can be recovered.

The best programs force you to give reasonable answers to their questions. So try to anticipate some of the mistakes your firm might make. Enter blatantly incorrect answers: if it asks for an item reference, press RETURN. Or try typing in alphabetics when you're asked for a VAT account number. Does it merely ask you to try again? Does it tell you why it didn't like your first attempt.

CHAPTER 23
Filing Systems

All organisations depend on information. They need it just to stay organised; critically, they also need it to operate. Take away the files and the filing cabinets, and you're left with a vacuum.

That certainly applies to businesses, for instance. There information is money. You have to know about your staff and your customers, your products and your debts; you need that information for marketing, for market assessment, for analysis of performance. You probably need it for personnel management. You may have to compile catalogues and indexes for source material.

Maintaining files of information has turned out to be one of the most valuable uses for a computer. You have three choices, in principle: you can use specially-written programs, you can buy very specific packages for a particular task, or you can look at general-purpose 'file management' or 'database management' packages.

The last of those options is the one we'll consider here.

The word 'computer' is something of a misnomer. In practice most computers don't do an awful lot of computing – not in the sense of performing mathematical calculations for you, anyway.

What computers are good at, and what they are generally used for in practice, is blindly following repetitive rules very quickly. The implications of this are simple: if you have a task that can quite easily be seen as a collection of basic rules to follow, and if it is one that involves a tedious amount of effort for people to do quickly and accurately, the computer is the obvious choice.

That particularly applies to the storage and use of information. Computers can store large amounts of information in a small space: they can store it accurately and in a consistent format: they can (in theory) let you keep it up to date easily: they can (in theory) let you get at it quickly and easily, presenting all the information you'd like in the format you want.

Nearly all of the tasks on which you might use a file manager or a database package *could* be handled by a purpose-written program. The difference is that using a 'conventional' programming language like Basic starts you on the problem from another angle.

The file managers assume that you want to set up records and then perform operations on them. So they give you the facilities to define a file, and they give you commands that in effect 'process' the file.

A programming language doesn't assume you'll want to set up a file at all, since not all programs will utilise files. It gives you instead a set of commands and rules that are more general in their application – and paradoxically much more specific in their effects, operating at a much lower level in terms of how they relate to the electrons buzzing around inside the computer.

You'd find it difficult to use a file manager package for tasks involving lots of computation on a limited amount of filed data. You'd find it virtually impossible to use such a package to handle readings being received directly from lab instruments or to control the operation of an automated production line. But nearly any programming language could be employed on those jobs.

On the other hand, if you *do* have the kind of problem that requires information to be stored in files for subsequent use, you won't beat the file manager as a way of getting up and running quickly with a computerised solution. The details of the program's operation will be much less susceptible to errors and unforseen pitfalls, since they are far more likely to be both predetermined (by the manufacturer) and tested in the field (by users).

And the range of possibilities is known; the manufacturer does not have to include all the commands and functions you can and should expect from a general-purpose programming language. So the type of commands provided by the file manager can be much more explicit, much more relevant to the task at hand; and the terminology used can be more explicit, too, closer to spoken English.

The same arguments of course apply to other packages with a specific function – word processors and spreadsheet calculators in particular. It *is* possible to use a programming language to produce programs that do the same jobs, and clearly someone did just that to write the package in the first place.

But the advantages of those packages lie in ease of use and the speed with which you start getting useful work out of them. In view of that, it's worth making those two the main criteria when you come to select any kind of package.

What's a filing system?

At its most basic, a filing system (or 'file manager', or 'data management system', or 'database manager') is a program that allows you to set up your own format for records in a file – and to set up different formats for different files, too.

Subsequently the package gives you the facility to type information into those blank record formats; and subsequently to pick out selected records or selected items from records, including summaries of all or part of the file using selected items. You will probably be able to decide just how such reports and summaries are to be presented on the page.

One of the attractions of the filing packages is that the principles involved are so familiar. Everyone knows what a 'file' is: it's a collection of related information, that's all. And files consist of 'records', which are fairly obviously individual items of a like nature that will be complete in themselves (holding all the relevant information you have on a particular customer, for example) in a form that allows you to compare the information therein with the same kind of information held in other records in the same file.

Computerised files aren't quite analogous to the office filing cabinet, though. In one of those folders or suspension files you can put any kind of information that you regard as relevant to the name on the file – documents, correspondence, photographs, bills, newspaper cuttings. The computerised system requires you to set up a much more rigid format for records, and in practice each of the file entries therefore have to look much the same.

A better analogy is with a card index, where the file consists of a bunch of individual cards (say personnel records) in more or less the same layout (name, followed by address, personnel number, date joined, current department, job history, etc) – but with the scope to add information that doesn't fit the rigid headings you've laid down, personal notes and managers' comments for instance.

Pros and cons

In short, filing packages are very flexible ways of handling the kind of information that you can keep in files. They do impose some strictures, of course.

There is normally a limit on the number of headings you can have in your record format and there may be a restriction on the amount of information you can enter against each heading.

And you usually have to get your record format correct right from the start, since you generally have to specify the format in considerable detail – how many characters you want to allow for each entry, which items on the record you'll want to use for 'sort and select' tasks, which items you may want to pluck out and use in calculations.

Once you start entering the information it's often very difficult or impossible to change the layout. So it's worth taking a good deal of care and no little time to cover all the possible options when you're setting up the file in the first place.

There will be other restrictions as well. Some packages have a maximum number of records that you can put into a file, though with many that's limited only by the capacity of your disks – and with some you can spread a file across more than one disk. Some also put a limit on the amount of information you can hold in any one record.

And with most of them, the size of the individual records relates directly to the maximum size of the file; all records have to be approximately the same size, because they are all laid out to follow your predefined format, so if you're using a long record format you will obviously take up more disk space per record.

The fancier packages do use clever techniques to minimise that. Instead of repeating all the headings on each record stored, they will use a kind of abbreviation form that clearly takes up less space; the full expanded form of the heading will be stored somewhere else within the file in a kind of glossary, and every time you want to print or display a record the program will go to the glossary to fetch the full heading.

What can you do?

So far so good. You've bought a filing package because you want to set up files. But you also want to get at those files, to *use* them. What kind of facilities will the package provide?

- **Setting up records**: For a start, the package will set up records. That is a fairly obvious point, but it is worth making. The computerised system will impose order on your record-keeping. It will force you to decide what kind of information you want to hold; and it will force (or at least *encourage*) you to fill in each record with the same kind of information in the same manner.

Some of the information might be entered automatically for you. When it's required, today's date is often inserted without your having to type it. If you have an entry in your supplier's record for 'outstanding balance', say, the program might well be able to fill that in all by adding the last invoice total you put in to a current balance and perhaps subtracting what you've just keyed into the 'payment received' line.

Most file managers require that all records are identical in format and the same length.

- **Maintaining files**: The filing system will also maintain records and files. New records can be added as they arise: when information changes, existing records can be amended simply, quickly, accurately and *rigorously*.

Both factors mean your file should be as up to date as you can make it – and since amendments are fast and easy to do, with a defined procedure to the job, the sheer effort of keeping the files up to date should be minimised.

- **Searching**: Once you have your information in there you'll want to get at it. This means the ability to search for a particular record and display or print it. Typically you will want get at a record by referencing the main subject on it, the supplier's name for example.

Or you might want to pick it out by using some kind of subsidiary information which doesn't happen to be at the top of the record layout – an abbreviated name-code or a customer number, perhaps. Perhaps you can't remember the name, so you'll want the package to find you the record for that sales prospect in Whitehaven or the job applicant to whom you wrote on 25 January 1982.

All the packages allow you to do that, though they will usually require that you nominate in your initial setting up of the format which bits of information on the layout you might want to use for searches. These are generally called 'key' fields, a 'field' being one of the entries in a record – files consist of records, records consist of fields.

In searching through a file like this the package will typically pick up all the records that appear to fit the information you've offered. You are likely to get a screen message telling you it's found four records with a Whitehaven address and do you want to see the first of them now?

- **Browsing**: When you have located one record, the program will probably let you look at the next one along in the file; most will let you go back to the previous one, too.

'Next' here means the next one in terms of your specification for the file. You might have put a company name as your main header, in which case the following record will be next in alphabetical sequence. But your main header might be a filing code or a number, in which case the sequence might be alphabetical or numeric (or a combination of those).

- **'Conditional' searching**: When you tell it to look for 'Whitehaven', you're giving it just one parameter for the search. In fact all but the very simplest filing packages will pick records on the basis of a combined set of different criteria. You will probably be able to say 'find all records for reports and books on marketing thermocouples in the Middle East that we ordered in the past two years and which have a publication date later than 1980'.

The programs do vary in just how many different criteria you can use to locate the precise record(s) you're after; and they also vary in whether they let you use what are called 'operators'. The thermocouples example strings together a collections of ANDs, and AND is one of the 'logical' operators. NOT is another; so some packages would let you add '... and which have not been taken out of the library by anyone from this department during 1983'.

OR is another useful operator, extending this particular search for instance to say '... that were produced by independent bodies within Europe or North America or Europe but not Switzerland'.

A few packages can also incorporate arithmetic in the searches. Usually this means the 'greater than' and 'less than' operators, for instance to pick out all suppliers whose outstanding orders total less than £1,000. Or you might have the program do the comparison between two items in the record itself. Say there's one entry for credit limit and another for current balance; you could tell the system to compare the two and give you a list of all customers who are running over their limit right now.

- **Calculation**: Actually performing arithmetic calculations on information held on a record is more likely to be of use to you when you want to produce reports or print something. In fact it's possible to use many filing packages to set up business accounting systems: rather than buy a ready to use package you can set up files from which the package can automatically produce invoices and statements with subtotals, VAT calculations, and totals.

Not all the filing packages on offer will permit this; but those that do will let you add, subtract, divide and multiply (maybe even do exponentiation and other fancy maths) on specific fields.

To provide the basis for an invoicing system, for instance, the program can pick up outstanding customer orders from a file, automatically apply VAT and discount rates, calculate a total due, and add a note with the calculated discount for prepayment.

You might also do some arithmetic on the file for business forecasting. You could have the program go through the records to pick out all unfilled

advance orders for the next year and total the lot. Or you could ask for subtotals by product group and month due.

Depending how clever the package is, your forecast could include information from more than one file – sales details from the customer records, purchases from another file for suppliers, production data from a third file of your manufacturing records. Some filing systems let you do this, totaling all the relevant fields and performing calculations on the results.

- **Sorting**: You will almost certainly be able to have the package resort all or part of the file. The records might be stored alphabetically by name: you might want them sorted into a 'date of last order' sequence, or by postcode, or whatever. You can do the resorting by using any key field you set up originally, and obviously you can combine sorting and selection to pick out only the relevant information.

- **Stored routines**: Some combinations of search criteria and arithmetic processes may be used frequently. A few packages allow you to predefine these and store them separately, in effect as mini-programs, so that you can call up the sorting parameters simply and quickly or insert a standard formula easily into records or reports whenever it's needed.

- **Output**: Once you have extracted the information you want, you can print or display it. The trick is that as a rule you can set up the layout of this output quite independently of the they way the records themselves look.

So your forecasting program can be arranged as a table with headings along the top and side and the information taken from the files inserted as numbers in the columns. Your invoice format would use the name and address fields from the record in one position and other data elsewhere on the page, adding some specific material like your standard payment terms. And so on.

The package will probably come with some predefined output formats anyhow. Many have a built-in label-printing function, for instance, a layout that suits sticky labels on those continuous sheets and just takes the name and address part from the records.

CHAPTER 24
Spreadsheet Calculators

The term 'spreadsheet calculator' may be intimidatingly clumsy term; but it happens to be one that should be part of the vocabulary for anyone interesting in using the 64 seriously in business.

A spreadsheet calculator is a load-and-go package that sets out a grid on the screen akin to the spreadsheet layouts used by bookkeepers and accountants: you can fill the grid with numbers and textual headings, and the package will perform calculations on them.

Those calculations will include the kind of totaling of horizontal rows and vertical columns that all financial calculation seems to involve. But you can also call for much more complex arithmetic to be performed on your entries – discounting an investment across 12 months by a variable rate, for instance: summing only certain entries: working out averages, standard deviations,

In fact an electronic spreadsheet (or 'worksheet') is exactly analogous to the way most business planning and other repetitive calculation is done. You would probably rule several columns down a sheet of paper and give them headings, one per time period say. The rows across would be for the variables and they would have headings in the left-hand column.

You'd enter a value along the rows for each slot and probably use them in calculations that produce results along the bottom (totals for each month, perhaps) and in a right-hand column (totals per item for the whole period). Some of the rows and columns might hold sub-totals, and there would probably be some annotation and comment dotted around the sheet.

To do an alternative set of calculations you would take a fresh sheet, put on the same headings and columns, enter a different set of figures, and repeat the calculations.

So you have a grid with slots in it for the insertion of particular values. Other slots provide for the results of calculations involving the values you enter. And the layout, the headings and the calculations will all be the same even when you are trying out different numbers.

The computerised spreadsheet calculator provides exactly that – with greater speed, more flexibility, more completeness, and probably better precision.

Today's advanced spreadsheet packages, even those on the 64, demand no technical computer knowledge; and ease of use is one of the major points in their favour. The other critical element in their design is the provision of really quite sophisticated facilities for the analysis and manipulation of data.

So the spreadsheet calculator enables the business user or accountant to do all those financial and statistical calculations that would normally require analysis sheets, a calculator or slide rule, and a pencil.

And those calculations can be performed more quickly and with greater precision, automatically calculating specific entries on the worksheet and presenting information accurately and comprehensively.

Letting the computer do the work

By contrast with non-computerised methods those calculations can also be considerably more complex, if that is required: the package will allow the use of formulae and conditional expressions to express complicated relationships between the data you put into the spreadsheet.

But probably the most important attribute of the spreadsheet calculator is the way it can be set up *independently* of the entries you put into it. Because the entries on the worksheet may be computed in terms of the *relationships* between them rather than the actual values you've entered, the sheet itself exists independently of the numbers on it.

That means the data on the sheet is easily altered, replaced or deleted; and all the affected values are recalculated *automatically* using the same rules and formulae that you had applied to a previous set of values.

So a given situation can be worked out with different sets of information – in particular to answer 'what if' questions. A simple example: you know what you have to add up to work out cost of sales, so you can set up a 'formula' that sums those entries. And by altering the actual numbers you put against those headings, you can see how the 'cost of sales' entry would change – for instance, you might check

out the bottom-line effect of different wage increases to see just what you can afford to offer. Or you could do it the other way around: with an adjustment to the formula you're using, you could try a target figure for cost of sales and see just what that means for the individual elements of the total cost.

Used like this, a spreadsheet calculator can take much of the guesswork out of forecasting and forward planning, enabling you to consider alternatives in exact detail and to compare actual performance with projections.

The Electronic Worksheet

When you use a spreadsheet package you specify the size of the worksheet to suit your own needs – the maximum will be something between 200 and 300 rows (with 256 the norm) by 60 to 80 columns (63 is a usual maximum), and it is rare to use all of that.

But until you hit those limits there is nothing rigid about the worksheet grid: one of the really nice aspects of spreadsheet calculating is that you can add or delete rows and columns anywhere in the grid as you find the need, and ordinarily this won't affect any of the relationships you have already set up. So if you are summing all the values in a column from row 4 to row 7, and you add a new row within that range, the totalling formula previously defined should change automatically to sum rows 4 to 8.

The screen cannot display the full worksheet, naturally: there isn't enough space on it. Instead you get a 'display window' generally showing about 20 rows by eight columns. The cursor keys will move the window around the worksheet. You always know where you are because the row and column identifiers are always present on the screen.

You can also ask for certain rows, columns or both to be 'fixed' so that they don't move when you are scrolling. This is handy for keeping headings on the screen. For instance, if you have a 24-month business plan and you want to take a look at the start of year two it would be useful to 'fix' column one (containing your entry headings) and the top couple of months (with your month headings) on the screen.

This is typically used for headings and titles, but it can also display a split screen for comparison of different figures. On some packages you can also split the screen into two horizontally or vertically divided sections that can be moved independently as two separate 'windows'.

As well as defining the overall shape of the grid, you also define the size of each block (or 'cell') within it. At start-up the package will give a default block size, usually of eight characters; this you can alter, generally to a maximum of 100 or more.

You won't be able to *see* 100 characters in each entry, because that would make the grid undisplayable. Instead up to 27 or so will be displayed on the

Spreadsheet commands

All spreadsheet calculators give the user a set of commands for setting up the worksheet, manipulating it, and housekeeping. The way they are used varies from one package to another, but commands are generally preceded by a slash and the initial letter only is typed – the spreadsheet calculator confirms the command by filling out the rest of the word. This is a pretty full list of spreadsheet commands to look for:

- **Text** mode (usually just quote marks preceding your entry into a cell) allows alphanumeric entries distinguished from formulae (starting with a cell identifiers or any alphabetic entry *not* preceded by quotes) or values (numerics not preceded by quotes). Text entries can usually spill over into adjacent unoccupied cells
- **Help** or **Menu** (or sometimes ?) will list your available options at any point
- **Edit** allows you to amend the contents of a block, using cursor controls, deletion, overtyping, and insertion
- **Insert** adds a new column or row
- **Move** shifts a row or column to a new location. Some packages allow a 'cut and paste' move, where you specify a rectangular area to be shifted
- **Automatic cursor advance** saves a keystroke – it means as soon as you've finished one entry and pressed Return, the cursor will move automatically to the next cell in whatever direction it last moved
- **Jump** or **Goto** (often =) lets you specify a particular cell to which the cursor then moves
- **Page** or **Screen shift** is a convenient method of skipping to the next screenfull of information horizontally or vertically. Otherwise you'd have to 'Jump' to a specified cell or use the cursor to 'scroll' the display
- **Home** takes the cursor to the top left corner of the spreadsheet

- **Format** specifies a format for one or more blocks, a whole row or column, or the entire worksheet. Options available will include resetting the column width: left- or right-justification: decimal, integer, exponent ('scientific' format) or financial layout for numbers

- **'Floating' currency signs** with financial-format numbers puts the currency symbol immediately before the number irrespective of how big the number is. A percent mark following a number may also be available to you. Either way the presence of the sign doesn't prevent the number being used in calculations

- **'Floating' commas in numbers** automatically puts a comma in numbers to indicate thousands, millions etc. Again this doesn't affect the number's use in calculations

- **Negatives in brackets** fits the standard style for ledger sheets

- **Global** sets options for the whole worksheet. Functions here may include the display of all formulae in the blocks instead of values; a 'border' command to set or suppress the display of row and column identification; and skipping over empty or protected blocks.

- **Copy** takes a one-to-one copy of one or more blocks, replicating either the actual contents or the formulae as required

- **Replicate** operates similarly except that it copies into a destination larger than the source, so it can be used for multiple copies

- **Search** or **Lookup** will find the first occurrence from the current cursor position of a specified text string or value. It's rare. Even rarer is a **search-and-replace** option that allows you to replace one entry with another as specified in the command

- **Sort** will arrange rows or columns alphabetically (backwards or forwards) or numerically (high to low, or *vice versa*)

- **Protect** prohibits any change to the content or format of one or more blocks.

An **Unprotect** command cancels this

- **Calendar** automatically calculates and enters dates and time periods

- **Blank** erases the contents of a block, column or row as specified, leaving you with an empty cell but not actually deleting it from the spreadsheet

- **Delete** removes a whole column or row, or maybe a whole file from disk

- **Load** selects a specified worksheet from disk

- **Save** writes all or part of the current worksheet to disk – either as its contents alone or with both values and formulae

- **Print** prints all or part of the worksheet. As with 'Save', you will be able to specify blocks for the start and end of an operation on a partial worksheet

- **Print format** allows a variety of printout options – suppressing row and column identifiers, putting commas into long numbers, automatically saving overruns to put on to a second sheet

- **Title** 'locks' columns, rows or both into position on the screen so that they remain static while the display window can move around the worksheet

- **Window** splits the display window into two. Each can usually have its own format settings and can be scrolled independently

- **Graphics** replaces values in a specified row or column by bars of asterisks in proportion. It's by no means a replacement for true graphical representation, though

- **Clear** clears all contents and format settings. Some spreadsheet packages have engaging names like **Zap** or **Kill** for this function

- **End** or **Quit** leaves the package and returns you to the operating system – preferably having first given you the option of saving the current worksheet

worksheet. But the *actual* contents of the block you're at will be shown in full somewhere on the bottom of the screen, and the spreadsheet calculations will certainly *use* the full contents of the block.

Different columns can also have different widths in some spreadsheet packages. And text can normally overrun into adjacent blocks if they are unused, which saves you having to force headings and comments into the block width you've defined.

Rows will be numbered sequentially and columns will be identified by letter – once you get past 'Z' the columns will be named 'AA', 'AB' and so on: then 'BA', 'BB' etc. An individual block is identified by column and row position, so 'G9' is the ninth block down the seventh column. When you're moving the cursor around the worksheet, the current block (the one that the cursor is at) will always be identified at the top or bottom of the screen on a line containing status and command information. And one of those commands available to you is a 'go to' instruction that will take you *directly* to a specified block .

Setting up the worksheet

The information you put into a block can therefore be *text*, useful for headings, descriptions and notes. In this case the block will be ignored for the purposes of calculation.

Most entries will be *numeric*, and for each block or row or column (or the whole worksheet, indeed) you can specify a format as integer, decimal, or exponent form (scientific notation). The numeric precision is likely to be of the order of 16 digits plus decimal point or exponent sign. A 'financial' format to two decimal places and rounded to the nearest cent or penny is also the norm.

Another formatting option you get is justification – whether the block's contents butt against the left or the right-hand side. Again, this can be set for the entire sheet or for individual columns, rows or blocks. Personally I find the most convenient format for business forecasting is to justify all the numeric entries (which are in the 'financial' format) to the right, with rows and columns containing titles and headings justified to the left.

In a block you can also set up a 'formula', a calculation that may involve other blocks. The results will then be entered automatically. For instance, BB9 could be specified as holding the total you get by adding the contents of blocks BB1 to BB8. In that way, BB9 will always carry that total – even when the actual values being summed are changed by you.

With these formulae the spreadsheet calculator can add, subtract, multiply or divide the values at two or more blocks. It will probably be able to find partial sums, work out minimum and maximum ranges, and calculate square roots. Some packages also have commands for 'conditional' operations involving comparisons.

Functions

The functions in spreadsheet calculator packages vary a good deal, too. Again the names may differ, but these are a full selection of what you can look for:

- **ABS** uses the absolute value of a specified item
- **COUNT** finds the number of non-blank entries in a list of values
- **ERROR** identifies all blocks with the function specified, and all blocks with formulae relating to them
- **EXP** uses the exponent of 'e'
- **REGR** performs linear regression on a specified range of items
- **PROJ** calculates the best estimate of dependent variables
- **DEPD** calculates the best estimate of independent variables
- **SLOPE** computes the slope of the linear equation from the results of a regression
- **OR** compares two expressions, results in 'true' if either is true
- **AND** compares two expressions, results in 'true' if both are true
- **NOT** compares two expressions, results in 'true' if both are false

Using a formula

In fact the formula can use specific values ('0.15' for a VAT calculation, for example): the contents of other blocks: constants (some packages allow you to define constants yourself, so you might set up one called 'V' having the value '0.15' to use in your VAT calculations): and a whole range of built-in functions.

Those may be combined in one formula by using arithmetic and relational operators, with brackets if necessary to ensure the correct order of calculation. As a result the formula in one block can be really quite complicated; and it can take a while to get it right. But thereafter it is extraordinarily gratifying to see just how easy it is to produce a correct value at the appropriate place in the worksheet from a complex calculation – and how easy it is to get to the 'what if' result simply by altering the values used in the computation.

- **IF** if expression 1 is true, use expression 2; otherwise use expression 3
- **INT** uses the integer portion of a value
- **LN** uses the natural logarithm of the value
- **LOG10** uses log to base 10
- **MAX** finds and uses the maximum value in a specified list of items
- **MIN** finds and uses the minimum value
- **SUM** sums the values in a list
- **NPV** calculates the net present value of a specified range of items at the given rate of discount
- **PI** uses the constant *pi* (generally to 16 significant digits)
- **SIN** calculates a sine value
- **COS** cosine value
- **TAN** tangent value
- **ASIN** arc sine value
- **ACOS** arc cosine value
- **ATAN** arc tangent value
- **SQRT** calculates the square root

Blocks, rows and columns can be moved around the worksheet, copied and repeated. All the contents of the blocks will be moved, and if that happens to involve a formula the calculation at the new location will operate on the equivalent blocks. Say you're copying that BB9 block to AZ12; it contains a formula which sums the values in the eight blocks immediately above it, so at the new location it will sum AZ4 to AZ11.

When you set up a formula in a block it's the value computed by the formula that appears on the worksheet. The formula itself will appear outside the worksheet in one of the information lines.

There it can be edited and amended. Your spreadsheet calculator package will include a set of editing commands that do not require you to retype the whole entry for a formula; they can also be used on text and on absolute numeric values.

Formulae in spreadsheets

Any block in a worksheet may hold a formula that can involve values, contents of other blocks, constants and functions. These may be combined by using operators and brackets. Most of the operators (which are basically linking signs) will be familiar, some will not. These are the common ones:

\+ addition
\- subtraction
* multiplication
/ division
↑ exponentiate (raise to the power of)

= is equal to
= is not equal to
< is less than
<= is less than or equal to
> is greater than
>= is greater than or equal to

Uses for spreadsheet packages

Spreadsheet calculating has applications within virtually any organisation. Classic uses include these:

- **financial forecasting and business planning, with alternative scenarios being considered**
- **detailed costing and consideration of business propositions**
- **rate-of-return calculations**
- **assessment of break-even points**
- **alternative pricing strategies**
- **manpower assignment and other resource scheduling**
- **calculating depreciation and loan write-off periods**
- **profit-and-loss statements and balance sheets**
- **preparation of budgets and comparison of actual performance against budgets**
- **computation and presentation of results for market research and other surveys**
- **analysis of laboratory experiments**

A decent spreadsheet calculator will check with you when you ask for major changes – clearing the worksheet, deleting blocks, leaving the program without storing the sheet you have been working on. You should also be able to protect specific blocks against alteration or erasure.

CHAPTER 25
Word Processing

One way and another all organisations use a lot of words. That's how people communicate, and there's a good deal of communication in the functions of a business.

Apart from routine correspondence and internal reporting, there are several other ways in which words will figure in the business. Your marketing might involve mailshots using standard letters. You'll certainly have a name-and-address list of customers. Your products might need operator guides, maintenance instructions, a user handbook. Your invoicing procedures will almost certainly involve chasing debts with a pretty standard reminder letter. Your staff might write reports, press releases, articles ... and so on.

Or maybe some of that doesn't actually happen. Regular mailshots, updates to product catalogues and user guides, press relations can all be immensely time-consuming. And even if it is done, the quality of the finished document might be somewhat variable – small errors, mistypings, ugly phrases, no uniformity of format.

If any of this strikes a chord, you should take a look at word processing – and there are some excellent word processors available for the 64.

'Word processing' sounds a grandiose concept. In fact it's quite simple: word processing is the automation of the business of creating, storing and revising text.

The idea of WP comes into its own as soon as you find yourself needing to type more than one page of text: and/or when you require more than one top copy of it: and/or when your typing involves different versions of the same basic text.

'Word processing' did make its immediate impact as a straight upgrade for using a typewriter. But its real benefits should never be seen simply in terms of automating the existing procedures (which is also the case with computers in general, as it happens).

It's true the immediate appeal and the immediate cost-justifications may come from doing the same job in the same way – only faster. But word processing is more than high-speed typewriting. There are four key elements to word processing:

- **originating** a document (which usually means deciding what you want to say, perhaps dictating or handwriting it, then having it typed
- **amending and correcting** a draft (maybe several times, to get it right; and maybe with comments from other people to be incorporated)
- **despatching and delivering** the final version
- **retaining and storing** copies.

So at the start someone produces a document and somehow it gets into the computer (which could be done in several ways, but it will probably involve typing at a keyboard). Subsequently it is revised with new material added, mistakes corrected, some sections shifted around. This could happen more or less immediately; or you might use a previously-created document to alter it.

Some of the alterations you require can be done automatically by the word processing program – changing all occurrences of a single word or a group of words to something else, for example. Say you decide that 'USA' looks better than 'U.S.A.'; a WP program can find and alter all occurrences of 'U.S.A.' to the preferred style.

You might create an entire new document just by calling up several existing bits of text and merging them together. Or you might just want to take some parts from the material you have stored and add new text.

And you can replicate some or all of the text for a series of different recipients – the classic 'personalised mailshot'.

Eventually you will want to arrange for the documents to get to the recipients. Typically this will mean fitting your text into a particular format – with most word processing systems, various alternative formats can be predefined quite separately from the text itself: so you'll be able to set up one particular pattern of tab stops, spacing, page length and so on for each of the different types of document you're likely to want to produce. Having done that, you would print out the document and pass it on.

This isn't necessarily your only option for despatch, since you might be able to send a copy of the document on disk rather than on paper; the recipient could then load the disk on a compatible computer and read your text. Or if you have a computer with multiple terminals attached you might be able to transmit it to the recipient's own screen. Again, this need not happen immediately – you can store all or part of the document until you need to send it.

Word processor functions

The basic text editing functions such as insert and delete, move and copy, are self-explanatory. So are the ability to save and load documents, and to delete documents from disk. The other features that will be found include:

- **Search and replace**: locates every occurrence in text of a specified character string. Can automatically delete or replace it

- **Block move**: moves a block of text that you've marked from one part of a document to another, or into another document. Other block operations are **copy** (moves it without erasing it from the original place: also called **cut and paste**) and **delete** (erases a chunk of text)

- **Standard paragraphs**: frequently-used phrases, forms and paragraphs can be stored for recall and insertion into a different piece of text. Sometimes called 'boilerplating'

- **Document creation**: also called 'boilerplating' by some, though it's not quite the same – it's the related capability of creating documents from standard text modules stored on disk

- **Mail-merge**: for personalising form letters with variable inserts automatically merged into a standard text

- **Centering**: centres a heading automatically between the left and right margins. You may also be able to centre longer chunks of text

- **Ruler**: usually a line of formatting information that in particular specifies tab (and indent) positions. Most word processors allow you to alter the default settings; 'outline' tabs will indent the first line of a paragraph, 'reverse indent' (or 'undent') indents all but the first line. Most packages let you associate a particular ruler with a particular document

- **Decimal tab**: automatically aligns figures in columns around a decimal point to improve presentation and legibility

- **Columnar functions**: insert, delete and move columns within text

- **Format storage**: user sets up alternative page layouts, with margin settings and tab positions, for recall and use as required

- **Line spacing**: your WP package should allow single-line spacing (the norm) but also double- and triple- line spacing (useful for proofreading and hand-editing) and maybe more precise increments too (like one-and-a-half spaces). Your printer control menu should let you select the actual line increment as well, for six or eight lines per inch to suit Pica and Elite typefaces – though this may be done actually on the printer

- **Automatic pagination**: divides a multi-page document into pages of a user-specified length, numbering each one and then renumbering as necessary as a result of corrections.

- **Headers** and **footers**: most word processors also provide automatic insertion of headers and/or footers (but not usually both). Either can include the page numbering

- **Proportional spacing**: different widths for different letters (eg 'm' takes more space than an 'i'). Makes long sections of text much easier to read

- **Justification**: makes all printed lines the same length by adjusting the gaps between words or between letters. Best used with proportional spacing for really good presentation, though generally unnecessary for normal correspondence

- **Line length** (position of right margin): should be selectable for printout, but for lines longer than maximum screen width the effect on the display may be cdd – some

WP versus typing

Being computerised means the ability to define and perform functions automatically and repeatedly. The other important aspect of WP is the fact that you store information and manipulate it *without reference to its final form.* Those two attributes set 'word processing' well apart from 'typing'.

There are other important characteristics. Because the whole process is computerised it is extremely easy to take copies: in fact WP is based on storing and accessing files.

Editing text on a screen is much simpler, too. You can do 'block moves' or 'cut-and-paste' to shift whole lumps of text around the document on the screen, for instance, and 'search-and-replace' operations are easy. All changes and corrections in fact are easy, immediate, and visible: you don't have to commit yourself to the final printed version until you're satisfied.

The word processor will also cope automatically with line and page ends. The user does not have to worry about overrunning a page limit or a right-hand margin.

And there is less temptation to compromise, to leave in inapt phrases and minor errors. You get fewer interruptions because there is less paper handling and less use of the correction fluid.

So in principle word processing will enable fast typists to go faster, it will greatly improve the productivity of inaccurate typists, it means poor typists can do useful work. And this is a key point: anyone can perform really quite well on a word processor, because the inherent editing facilities mean the system is considerably more tolerant of the untrained.

word processors cope with it by scrolling sideways to show what would be on the whole line if you could see it all at once

- **Forced page breaks**: the word processor will automatically start a new page after a specified number of lines, but there will also be a facility to force a new page. Word processors can't usually cope with 'widows' (one line of a new paragraph starting at the foot of a page) or 'orphans' (one or two words at the end of a paragraph falling at the top of a page)

- **Printer pause**: a control command inserted into text that temporarily stops the printer (for you to change to a different stationery or another daisywheel, for instance)

- **Arithmetic**: basic four-function arithmetic may be provided, normally operating on columns within the text. Simple addition is the most common such function; more sophisticated systems can hold subtotals and totals in memory, and do percentage calculations too. A few word processors allow you to temporarily suspend the text operations and do an instant calculation using the Sirius's own calculator function or one built into the package

- **Spelling checkers**: compares WP document against stored lexicon of words (to which the user can add) and highlights any word not recognised: this picks up mistakes or new words to be added to the dictionary

- **Word count**: rarely provided (number of characters filed is more usual) and then generally inaccurate (since it is likely to include format controls as well)

- **Back-ups**: a safety feature – most word processors don't delete the original if you save a document with the same name. Instead you'll find that they retain it under than file name but with the '.BAK' suffix; so you can get at the original again if you need to

Pros and cons

It is easy to get carried away with all the advantages of word processing. But in practice it's not really appropriate for any and all text. The average single-page letter probably won't gain much from computerised word processing, for instance.

You have to set up for word processing on a small office computer – nothing too taxing, to be sure, but it still takes a little time. You have to pick out the right disk and load the WP program from it, define the spacing and page length and line width you want for the letter, make the printer ready, key the letter, edit it and correct it.

By the time you've done that it would have been quicker to type the thing in the first place on an ordinary typewriter – and it might even look better, for a golfball type element generally produces a crisper, cleaner print impression than the daisywheel commonly used with word processing printers.

But if you require anything longer than a single A4 page, or if the individual document is being reproduced several times more or less in the same form, word processing can be a real boon.

What kind of benefits should you look for from word processing?

- *The most common WP jobs are standard letters and reports* – and particularly documents that go through *several drafts*. There you should save significantly on retyping and proofreading time

- *Personalising standard letters*, typically for reminders and mailings, makes good use of the editing, storage and computer-controlled aspects of WP

- You can keep detailed *mailing list* records and do special selections from the file by using several parameters: the word processor really comes into its own on such applications, and the more sophisticated systems allow you several selection criteria

- By using *standard paragraphs* and *standard formats* you can considerably improve the appearance, content, timeliness, accuracy and production time of basic documents like quotations and reports

- You could try assembling customer *newsletters*, trade press *articles* and *internal communications* that would formally have been tatty, hurried and/or not done at all

- *Price lists* will change quite frequently, but with the basic information on a word processor you'll be able to produce updates easily as required

- You can make sure that *user instructions* for your products keep pace with new facilities that are added and design faults that are corrected. You will also be able to remove deficiencies in the original explanation

- WP can make producing *long texts* like reports less of a chore. With the WP system you can do the writing in a modular fashion – which means as you think of each bit or as you complete a section

- Your *company image* will benefit in general when everything you commit to paper looks better than the usual mistyped oft-corrected work. The ability to try different layouts and formats will mean that the content and the form of your material, whatever it is, should be more closely matched to its function

Contrasts in word processors

Word processors somehow foster the taking of sides more than any other type of software: you might not argue long into the night about which is best, but I've certainly come across some pretty entrenched positions on the subject – the kind of fervently-held views that you just don't get over sales ledger packages.

That's probably because word processing is the most *subjective* software you will ever use. It's working on your own words, your own preferred mode of expression, your writing style. And because the end format is not predetermined, it even *looks* the way you want it to.

As a result, even people who get to try several word processors tend to prefer the first one they come across that suits their mode of thinking – and usually that's the first one they come across full stop. They just get used to it.

So how to evaluate word processors *per se*? Well, obviously there are differences between them. But for most of the word processors available on the 64 there is a general consensus among designers of WP systems in terms of fundamental operations like move, delete, insert. And most packages have much the same overall capability.

How they let you use it is another matter, though.

For a start word processing programs tend to divide fundamentally into two approaches. With one, the text is formatted on the screen as it will appear on the paper; Wordcraft and Vizawrite are the classic examples.

Packages like Simply Write, Paperclip and Quick Brown Fox have a different approach; they fill the screen with text and let you decide later how it will look when it is output.

The latter philosophy probably finds its real home among people whose business is the creation and manipulation of words – like writers and journalists. The 'what you see is what you get' approach is generally preferred by users who operate specifically in terms of the finished product: secretaries and typists, for instance, may be happier with a WP program that actually treats text as individual pages that are laid out on the screen as they will appear.

On the other hand, a preference for WP packages that lets you see the final form of the document as you are creating may well be an unfortunate hangover from experience with typewriters. But given the fact that WP is more than automated typing, making a conceptual break between *creating* and *outputting* a document can open up a good deal of extra flexibility.

The other principal distinction between word processors comes in actually *using* the package. One of the cardinal sins of any word processing software is to slow down or disrupt the train of thought; they should let you get on with the job, providing a minimum of interruption. In particular, if multiple keystrokes and complicated command sequences are required the net result can be intimidating and/or irritating. The principal operations in any 64 word processor should be done with single keystrokes.

On the 64, ease of use can be taken one step further by means of the function keys. They can and should be used as single-keystroke commands to speed up the more common operations – like search-and-replace.

Another feature of the 64 that should be used is the ability to change colour to indicate different operations. The use of colour and the function keys (whether they are used at all, but also how well they are used) can generally be taken as an effective indicator of the amount of attention and care that the package's designers have given to the implementation of the software upon the computer. This applies to any 64 package, in fact.

Spelling checkers

Some word processors come with an associated spelling checker – like Vizaspell with Vizawrite and Easyspell with Easyscript.

They don't actually check your spelling, in fact. They just go through a piece of text that you've nominated, comparing the words therein with their own lexicon (it's properly called a *lexicon* and not a *dictionary*, even though every spelling checker I've seen uses the latter term, because it's just a list of words). Words they don't recognise are highlighted or – more likely – picked out for you and presented for your decision on what should happen to them.

And they *might* indeed be misspellings; or they could just be words that aren't in the lexicon – maybe technical or esoteric terms, or place-names and other proper nouns. They might be terms specific to your business, or idiomatic words you like to use. Or they could just be words the compilers missed, or plurals or participles they didn't bother including.

Having found the unrecognised word, the better packages will give you a variety of options. These are the choices you should get in an ideal world:

- **View the word in its context** – a line of text containing the unrecognised word appears on the screen for you to check just what it was you meant

- **Correct the word.** You can usually do this immediately; the package will stock up a list of changes and at the end of the operation alter the original document accordingly

- **Look up the word in the lexicon.** In other words, the program will try to find one or more words it *does* recognise that seem to be an approximation to the potential mistake. You can then select one of the choice to make the correction

- **Add the word to the main 'dictionary'** – because it's spelling is ok, because it's obviously not in the main word-list already, and because it's common enough for you to be using it again.

- **Add the word to a special 'dictionary'** – meaning that the spelling is good and the word is something you'll be using only in very specific circumstances, like recipients' names for internal memos or technical terms in reports.

- **Ignore the word** – because you can see it *is* a valid spelling but it's not one you want to include in one of the word-lists

- **Mark the word**, which means put a special marking character in front of it so that you can check it out later. The program will give you the chance to specify the marking character; after the spelling check, the package will file away a new copy of your text that includes the marker immediately before these words. You can subsequently use the word processor's 'search for' facility to find the marker and figure out what's wrong with the word

- **Go back to the previous word** or **forward to the next word** because you've changed your mind about what you want done with them (some spelling checkers don't allow you to go back more than one word or forwards at all)

Spelling checkers sound great, and they can indeed be very useful – *provided* you appreciate their limitations, and *provided* it's a reasonable package that you're using.

The limitations are primarily that you can't rely on the spelling checker to pick up your use of a perfectly-spelt word in the wrong place; and that includes misspellings which happen to come out as a valid word – the program won't pick up 'ward' when you meant to type 'word'. It won't pick up duplications, either, and it won't correct your grammar.

Beyond that, there are four points to watch for.

- The **size** of the main dictionary. Basically it a case of the more words the better; but that's a tricky trade-off, since the more words you have the more likely the check is to find a correct spelling for a different meaning. Besides, the larger the lexicon the slower the check

- The **speed** of the check – bigger word-lists tend to mean slower overall operation, but if everything else about the package is slow you might prefer to proofread your document yourself

- How **prefixes**, **suffixes**, **plurals** and **participles** are handled. Some spelling checkers try to be too clever; instead of actually listing words in their 'dictionary' they anticipate constructions like plurals and participles. As a result it might think 'womans' (for 'women') and 'haved' (for 'had') are perfectly acceptable

- How **faithful** the correction is. Are underlined words and those that might be reversed for printing in bold form replaced correctly? How about capitals?

CHAPTER 26
Business Accounting

Keeping the accounts may be a statutory requirement, but probably more important is their function as a measure of a business's activity. They involve a lot of record keeping, much transfer of information from one file to another, and a good deal of producing reports.

All those things make accounting a natural case for computerisation – especially as accuracy and attention to detail are vital (computers are very accurate and very picky) and timeliness is desirable (computers are very quick).

Computer systems for business almost always start from that premise – you'll be offered computerisation of your sales, purchase and nominal ledgers almost as a matter of course. One way or another this means setting up files containing much of the basic information of your business –

Operating accounts

Sales
Selling cost
Cost of goods
R&D
Administrative expenses

Cash management accounts

Cash on hand
Investments
Debtors
Creditors

Inventory and manufacturing accounts

Raw materials and supplies on hand
Supplies on order
Work in process
Finished goods
Order backlog

Fixed asset accounts

Land
Buildings
Machinery and equipment
Depreciation

Once that lot exists you and your computer have the facility of automatically presenting the current financial condition of the business. From such information, **financial statements** may be produced:

Balance sheets
Profit and loss accounts
Budgets
Financial analyses

The financial statements may present the appropriate data by period (e.g. weekly, monthly), cumulative year-to-date, comparison with previous years data, and comparisons with budgeted or forecasted data. And you could use the data to calculate various **business ratios** – key measures of your business, like these:

- **Current ratio**
- **Debt/equity ratio**
- **Average collection period**
- **Gross profit margin**
- **Net profit margin**
- **Return on investment**
- **Book value per share**
- **Earnings per share**

The Sales Ledger (Accounts Receivable)

The sales ledger accounts will involve a detailed file of customer information and a number of other files – at the very least a journal of transaction activities.

Some form of interrogation into the system is necessary when the traditional ledger forms are removed as file media. A computerised sales ledger system must provide some way to look at accounts between reporting times.

But the regular and automatically generated reports will be the main way of using the sales ledger system. As well as the sales journal, the kind of reports you'll get are:

- **Trial balance** to make sure all receivables entries are in balance.

- **Statements** which should probably give you the option of an open-item type (listing all unpaid charges and unapplied payments or credits) or balance-forward only (which includes the balance

brought forward at the end of the previous month and a list of all debits and credits for the current month).

- **Aged debtors report** will provide an open-item aged trial balance. Items should be grouped by age and perhaps by whatever credit terms are applicable; all open charges and unapplied credits should be listed by aged category. Someone could follow up delinquent accounts straight from this report if phone numbers are provided as well. The items listed could have a date and reference number too: this puts all the information in front of whoever is reviewing the report. There might also be a linked facility that automatically produces dunning letters.

- **Over-Credit Limit Report** is usually a daily or weekly summary of those customer accounts that have exceeded their established credit limits.

The net result should be a reduction in losses from bad debts, faster collection of accounts, more up-to-date information on accounts, and more and better information for policy-making.

The Nominal Ledger

The nominal (or 'general') ledger is rarely the first accounting system to be computerised, but it is one of the most powerful business tools you can apply to the financial management of your company. Its purpose is to take all the operating information in your company and summarise it in two basic reports – the Profit and Loss Account and the Balance Sheet. From these you should be able to determine the current financial status of your company and establish proper controls for direction.

If you need year-to-date information, make sure the Nominal Ledger system is cumulative. Most, but not all, are. Supporting accounts may also be available in some of the more sophisticated systems – sales, cost of sales, cash at bank, and depreciation schedules are some possible examples.

The ability to run up multiple ledgers for different departments or companies is handy if you have several subsidiaries with separate accounts. This enables you to keep each set of books separately and still produce consolidated corporate financial statements.

Some systems provide the ability to designate period beginning and ending dates. This gives you the ability to have other than monthly periods. For instance, thirteen periods a year is common.

If you want to be able easily to use your files for future planning, a computerised nominal ledger system will probably be able to maintain historical comparisons and budget details.

Standard reports that may be available in a system include:

- **Accounts List**
- **Transaction Journal**
- **Trial Balance**
- **Profit and Loss Account**
- **Departmental Income Account**
- **Sales Schedule**
- **Cost of Sales Schedule**
- **Cash at Bank**
- **Depreciation Schedule**
- **Cash Disbursements Journal**
- **Cash Receipts Journal**

The Purchase Ledger (Accounts Payable)

The purchase ledger keeps track of cash requirements and pays suppliers when required. In fact cash transactions will probably be rare: manual entry of cheques is likely to be provided. Otherwise all posting will be automatic.

Invoices are generally logged as they arrive but a computerised system should allow no posting to be done until the invoice has been approved by whoever ordered the goods or services.

A good purchase ledger package will be able to list all payments that should be made by a particular settlement date – but the company accountant should be allowed to amend that summary before an 'automatic payments' list is printed and the ledger updated. Remittance lists and advice notes, cheques and credit transfers can then be produced.

Typically the system will set up quite extensive supplier records, perhaps allowing you to assign 'supplier type' codes that will allow you to group different payments. You can also include a priority code to identify important people you'd like to pay first. The supplier record can have a lot of detail on settlement terms, periods and discounts: it can note options to pay by cheque or credit transfer, including bank details too: a temporary 'no-pay' mark can withhold payments.

Among standard reports you will probably get a *purchase day book summary* automatically: otherwise you can call for one as required. You can also ask for *cost centre analyses*, *'unapproved invoices'* lists and *ledger balances* among other reports available.

Order processing

Maintaining the accounts might be the usual task for the computer, but the most valuable function of many companies has been the automation of sales order processing and invoicing.

This is another classic application, involving quite rigid procedures that must be followed if the order is to be processed satisfactorily. So the computer's aptitude for following rules is useful.

What's more, the receipt and despatch of orders is right at the sharp end of your company's operation. Computers operate quickly, so orders can be processed quickly. Fast service makes for satisfied customers, and that means repeat business. It also allows you to get the invoices out quicker and the statements out on time.

Last words

It makes sense to computerise your accounts one way or another. There's just too much information floating around for you to manage a business effectively without some means of summarising it in an up-to-date fashion; and there's just too much opportunity for your cashflow to start getting out of hand without some way of applying rigid controls to that.

A computerised solution won't get rid of your accountant. But even if you're still running a small business, it should cut down on the paperwork he or she has to do: and it should give you more hard facts with which to manage your day-to-day operations.

But with more and more products appearing on the market for the 64, there's no easy answer about *which* you should buy. The key criteria? Try to summarise what you want to do – in particular, figure out what kind of volumes are involved right now and in the forseeable future (how many supplier accounts? How many stock lines? How many purchase accounts? How many different entries of all types per month?).

And work out what kind of analyses would really be useful to you: if you want 'Expenses' broken down into phone costs, office cleaning, car hire and so on, you'll need a different level of flexibility from someone who's prepared to lump them all together under one heading.

Thirdly, be aware of the essential limitations of any packaged solution. Accounts packages in particular are designed for a 'standard' or 'typical' business – and there ain't no such animal. While the better packages may work quite hard to appear as flexible as is possible, Hard Lesson Number One is that *no* package is going to suit you perfectly.

The final proviso: there's only one thing worse than a muddled manual accounts system, and that's a muddled computerised system. Go into this with your eyes very wide open!

CHAPTER 27
The SX-64

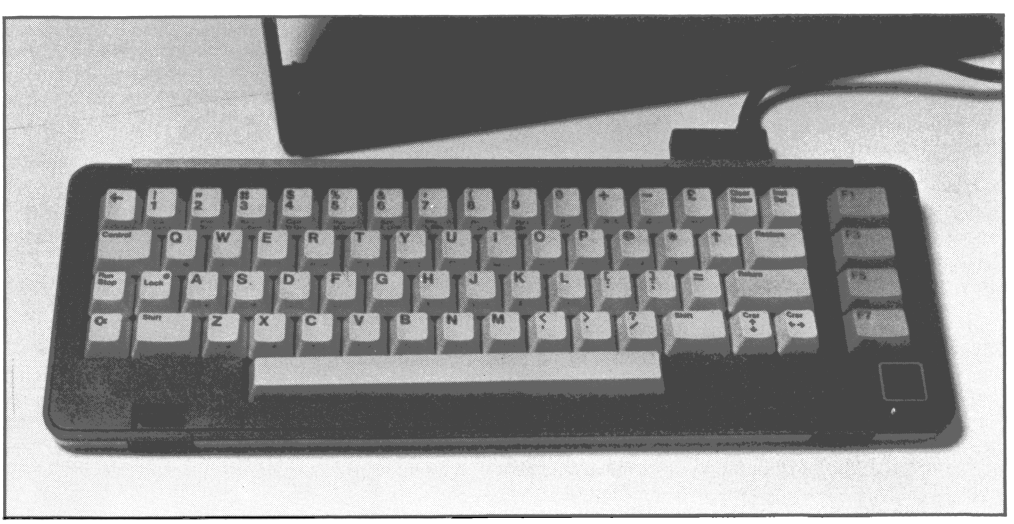
Commodore's announcement of a portable version of the 64 sounded like a good idea at the time. I've been using the thing for a while now, and frankly my reactions are mixed: this is the review I wrote for **Commodore User magazine at the time.**

Portable computers were all the rage last year. You couldn't really call yourself a computer manufacturer if you didn't have one – either the Tandy/Panasonic/ Epson style 'lap' computer with a built-in flat screen, or a 'real' computer with a conventional display, built-in disk drives, and a carrying handle.

Commodore went for the latter approach with the SX-64. Basically it's a box containing a 64, one floppy disk drive, and a five-inch display with a handle-cum-stand and a cable-connected keyboard that clips on to the front to double as a cover.

Sounds ok so far? Well, there are two ways of looking at the SX. You can consider it in isolation simply as another computer, in which case it's an expensive 64 with some extras and some omissions.

Or you can consider it as something to be used by someone – and then you'll be asking 'who', 'when', 'where' and 'why'. And to confuse the latter assessment there are a couple of bugs in the unguent. Like it costs £895. Like it has only one disk and the standard 6510 microprocessor from the 64, so it doesn't really compare too well as a business computer. Like it has joystick and cartridge ports but no cassette interface, so it doesn't look like a complete home computer system either.



And what's the SX-64? It's a 64 with a 1541 disk in a box, with a built-in screen and the electronics inside the cabinet rather than the keyboard.

Commodore has a different view: or more accurately, a number of different views. It's the "Commodore Portable SX-64 Colour Computer" says the manual, calling it the "Commodore Executive 64" elsewhere with a schizophrenic mien that as we'll see characterises the whole thing. "One of the best values in the home computer industry", it continues; but isn't something called the 'Executive 64' going to be more of a business computer?

Ah well, "the SX-64 is portable computing in the office, home, hotel room or at any location where mains power is available". So it's for everyone, y'see: that's what the PR agency says. But hang on, here's the brochure: "the Commodore SX-64 portable colour computer allows the businessman to truly take his work with him where he goes", it opens. So it's a business computer.

Or maybe not. Paragraph two: "if you're a busy executive keeping a diary, a scientist making notes about experiments, a salesman on a sales call, an on-site engineer, a reporter, an auditor, an accountant, or even a hobbyist who travels, this portable system is a must". Seven to one: must be a business computer.

Ah-ha, the price list: yes, at £895 it's a business computer. So, down to business.

Inside the box

A big box containing the usual quantity of protective polystyrene: embedded therein is an SX, a mains cable (no plug), a keyboard cable (three feet of it), handbook, cut-price flock-wallpaper handbag (for cables?), and some freebie software.

The thing is basically a deep and surprisingly heavy steel box – weighs 10.5kg, which isn't too bad: measures just over 400mm square by around 135mm (say 15x15x5ins), so for the average purchaser (whoever that is) it shouldn't scrape on the ground. Colour scheme muted elephant grey and matt black, with a few frilly bits in blue and silver.

Carry handle is a nice piece of work, thick and ribbed: you get a good grip on it and it doubles as a stand – ratchet swivel with simple but effective locking mechanism on each pivot. Doesn't really raise the screen high enough for crickfree viewing, though.

Keyboard unclips from the front (spring clips hold it on, feel less than 100 per cent mighty but keyboard never slipped off while we were carting the review model around) to reveal dinky little screen – 5in diagonal. Also there: one disk drive helpfully labelled 'drive 0', one disk-drive sized gap labelled 'storage', one spring-clipped panel labelled with the CBM logo in sexy silver (opens to reveal display controls and reset button. What's a reset button, daddy?).

After the multi-cable hassles of assembling an ordinary 64 system, it's magic to get up and running by making two connections – mains into back, keyboard to the front via solid 25-pin plugs and usefully long, flexible cable. The keyboard can stay attached when clipped back on for transit, but then the plug juts out a bit.

The keyboard is a restyled version of the 64's, with all 67 keys in the same layout. The keytops are smaller and the whole thing looks a lot more modern – very European, cleaner and clearer, with the keytop characters fashionably offset to the top left corner of the keys and the number keys printed with both the Control and CBM colours. 'Control' is spelt out, too, rather than being 'CTRL': so is 'Clear', but 'Insert' stays as 'Inst' for some reason. It's physically smaller than the 64 keyboard, but in terms of getting your fingers on to the keytops that's no problem.

No 'on' light to show you that the thing's working if the screen has blown, though. At least there's one on the Shift Lock.

The disk is obviously a repackaged 1541 for 5.25in 170KB floppies; comes complete with 1541's clunk-click latch and formatting characters. No improvement on 1541 speed. The 'storage' hole is a joke: may be useful for the leads in transit, but you can't fit disks into it unless they're out of their card overcoat (not recommended); the manual won't fit either, and it's too deep for pencils and too tight for buns (sandwiches might fit).

The display (reputedly a Sony screen) is a colour monitor – no watching Football Focus or Dallas on it. Displays the 64's 16 colours, 255 screen/border combinations, 25 40-character rows (but can scroll over 240 characters), bit-mapped for 320x200 pixels' worth of hi-res graphics. Text looks a bit dotty on it, and there's a slight but noticeable refresh flicker: but colours reproduce well, and hi-res graphics can look very crisp.

Start-up display is the Vic-style dark blue characters on white background with cyan border. The border isn't proportionally as large as on the 64 because the screen surround chops off the edges: gives the illusion of a near screen-size display area, but the sharp edge looks better anyhow.

Open the dinky little door and there's dinky little controls for volume (yes, the SX has its own speaker – pretty good one too, considerable loudness and little distortion), brightness, contrast, colour balance and vertical hold. Plus two fine-tuning screwdriver holes for contrast and brightness, which makes up for the limited turn range on the knobs.

And then there's the reset button. Tiny, discreet, unexplained. On most computers there's a reset button which does the same as Run/Stop and restore followed by NEW, but this isn't one. No-one at Commodore could tell us what it's for; only effect seems to be spinning the disk. For emergency hang-ups? Surely Commodore isn't admitting that the 1541 is prone to hang?



Atop there's the cartridge slot, clean and easy insertion through small flush-fitting spring-loaded doors. All the 64 cartridges we tried worked ok; they look a bit wierd sticking out of the top, but it's better than fiddling around the back.

Where you'll find a big finned heat disperser below a neat strip of standard 64 sockets – two joystick ports, DIN sockets for audio/video and serial, user port, mains lead, 1.5A fuse, chunky power switch.

Obvious absentee: cassette port. That's the major technical difference between the 64 and the SX. Not only have all the tape electronics and the edge connector been omitted, the Kernal's been modified to kill all reference to it. Like the SX gives 'ILLEGAL DEVICE NUMBER' when you try to read from or write to device 1, the cassette.

Instead there's a new meaning to the instant-load method of pressing Shifted Run/Stop: it produced LOAD":*",8 and RUN to load and run automatically the first thing the SX finds on disk. Note that they didn't amend the Kernal enough to make the disk rather than tape the default device – try typing LOAD "FILENAME" and you'll get the cheery illegal device message; you still have to laboriously spell it out as LOAD "FILENAME",8.

That kind of thing is a bit of a shame, since there has been some messing around in the Kernal –opening message promotes SX-64 BASIC V2.0 not COMMODORE BASIC V2, some tidying up to cursor handling and jumps has been done, and there's a whole new routine at 58579-58588 in what was formerly empty memory. It's only short, but we haven't yet been able to figure out what it does.

No other Kernal changes, and none to Basic. A real missed opportunity for simplified disk handling in particular; the SX is stuck with basic 2.0's protracted OPEN and PRINT# disk controls rather than the shorter and easier commands offered in Basic 4.0 from the 700 (if it still exists) and the 8000s.

True, there is the DOS Wedge on the TEST/DEMO disk supplied; sits on the cassette buffer, gives two-key disk directory (nonm-destructive) and file load. But it still has to be loaded; it could have been Kernalled. Ditto the disk copying programs supplied (one for single-disk copying, one for twin drives). There's still no quick 'n' easy formatting.

Freebies

That TEST/DEMO also includes a 'how to use' text display (which says very little other than to name the utilities it provides) and a few SX demos (which to our mind didn't really make the most of the SX's facilities).

The other freebies to sell you on the SX are three packages and a diskfull of six games. The games are so-so to good. The fruit machine has good graphics but some oddities (like you can hold a winning line: doesn't happen that way at our local) and anyhow fruit machine games are boring. The text-only mini-Adventure serves as an intro to the genre (120 rooms, 53 objects to collect; but again some inconsistencies). The 3D maze is ok; so is the 2D Patience. Average arcade action comes as a very ordinary 2D depthcharge game; much better is a travelling gunsight plus laser cannons to zap the Imperial fighters from space (really difficult at level 9).

The games may not be great; how about the packages, worth £210 according to Commodore? Well, there's Precision's well-known Easy Script word processor now branded by Commodore. Not at all a bad WP package, with many top-notch facilities including tabs, search-and-replace/delete, mail-merge for personalising form letters. It's by no means the easiest 64 word processor to use, though, and you'll find more facilities and more friendliness from other WP software on the 64 ... but then you'll pay extra for it.

Future Finance is a financial planning program producing cashflows and gross and net profitability forecasts from your input of up to 15 sales items, 15 purchase heads, and 30 nominal (expense) items. If you need more than that, you can split the model into different parts and later consolidate them. Various output report formats for different needs.

Problem? It's slow, awkward (especially on consolidations) and inflexible to use (for instance you can't edit a mistyped input – you have to do the whole entry again). One of the many decent spreadsheets for the 64 will be easier and better ... but then you'll pay extra again.

Anagram's Easy Stock is probably the best of the three in that it's as good as most inventory packages on the 64. Takes a while to define your stock file, but thereafter works well; book orders in and out, see stock-lows and slow movers highlighted, produce stock movement analyses (only two sales analysis codes though), do profitability assessments and stocktakes (including an override to alter stock quantity for pilferage and breakages without affecting values of sales and purchases), raise the price across the board by a percentage, and so on. Several good and useful reporting options using all or part of the file, too.

Major omission: no parts explosion for manufacturers, but then it's obviously aimed at retailers. Only significant complaints, then, are the thing's inflexibility (no tailoring is possible, like it won't cope with your giving selective discounts on selected lines to selected customers) and the use of lower-case letters in the display – may look nice on a big screen, but too small really for the mini-display.

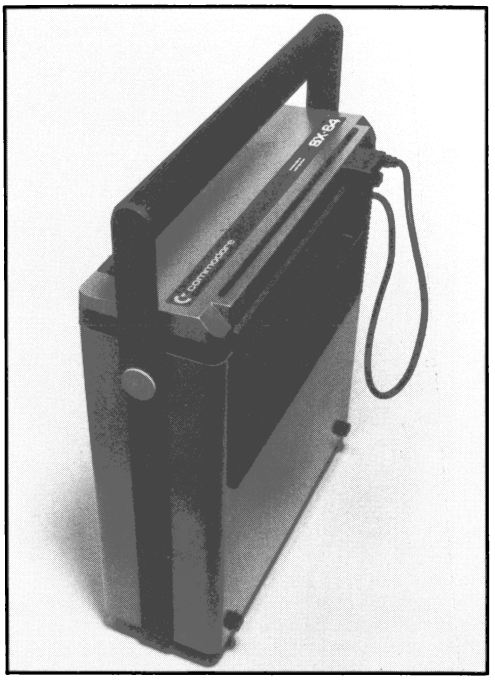
Manual labour

The SX handbook is a classic Commodore opus. Lie-flat spiral binding, 174 pages plus two blank ones labelled 'MEMO' and the useless bound-in 'Quick Reference Card' – all printed on that glossy paper which turns into superglue when you spill coffee on to it.

Inside there's a mish-mash of reprinted and/or lightly rewritten sections from the 64 handbook and the 1541 manual. Not very well mish-mashed, either: random access files and block read/write on page 27? In a section called 'Getting Started'?

Much hilarity too in 'Expanding your system'. Has anyone seen a 64 IEEE Interface Card in any Commodore dealers? Or the Z-80 cartridge with CP/M? Or the Speech Module? Or the Superexpander 64? Or the modem – "lets you use your telephone to connect your Executive 64 to other computers and information retrieval services"?

Or a prestel link or networking of the pet emulator? All promised in the brochure ...



Is this the portable computer for you?

The fashion for portables with a 'real' display (a TV-like CRT monitor capable of showing enough characters at a time to be useful) and a 'real' keyboard (with typewriter-style keys rather than the Spectrum/PC jr 'Chiclet' pushbuttons) was all started by the (largely self-styled) microcomputer industry guru Adam Osborne.

He sold a lot of Osborne 1s before he went bust. That computer was/is surprisingly unadventurous technically, with a conventional eight-bit micro in there along with two floppy disk drives and a tiny eye-straining screen in a bulky and ugly box (personal opinion – some people like it).

But it was/is portable (especially if you want to develop your biceps). It was/is complete – all you need is a printer. It had/has a proper keyboard, a proper screen, a proper disk operating system. And the price was/is really attractive, particularly as it includes a lot of free software.

Since the Osborne's launch, though, portables have come a long way. Displays are bigger and better, the virtually indestructible Sony micro-floppies are being used, 16-bit processors are now de rigeur, compatibility with the IBM PC (or more specifically with the huge amount of good-quality software available for the PC) is the norm, the styling has infinitely improved.

But who needs a portable computer anyway? It costs money to build something that's robust enough to stand being lugged around, and there are all kinds of technical and ergonomic compromises that have to be made in manufacturing a physically compact

unit. Are there that many people who need a computer to take with them whenever they leave the office or their home?

In fact it looks as though most portables for the business world aren't actually ported very much at all; they tend to sit on a desk more or less all their working lives. Not unlike a conventional non-portable computer, in fact.

Now, there is probably a case for arguing that some business people will want to take a computer home for out-of-hours work. travelling salesmen and suchlike use a portable for instant recording of visits and (hopefully) orders, perhaps with a phone link back to the head office computer to down-line the data collected. It's certainly useful for software developers to have a luggable computer: they tend to want to work wherever they happen to be. And there's at least one computer journalist makes a habit of getting up people's noses by sitting on the front row of press conferences clacking away at a portable computer.

What's more, there probably is also a market for a small, neat, compact and complete home computer – everything you need in one box with just one mains lead, as an alternative to the spaghetti of wiring that connects a plethora of small boxes to each other and to your TV set.

The appropriate comparisoncould be with those portable stereos with clip-on detachable speakers on the side of a compact unit containing amp, tuner and tape deck: you may not get as much in the way of hi-fi quality as with a component set-up, and you'll probably omit some things like a record deck. But in return you have an easy-to-use simply-connected unit that's physically small, which delivers pretty good quality and at a pretty low price.

So where does the SX-64 fit in?

On the plus side it's quite a nice piece of design work. It looks quite good, feels solid, is heavy but fairly comfortable to carry, has no irritating reflective surfaces.

There is the video-out connection for attaching a full-size monitor, but in any case the little display is pretty legible. The small size of the characters gets in the way of comprehension only on some 'curved' letters like capital 'S'.

Key bored

Then there's the keyboard. Styling is ok, if you accept that any portable with typewriter-style keys isn't going to leave a lot of room on the actual keyboard for considerate extras like space to rest your palm. The styling of the SX's keys (including dished keytops and slightly raked keyboard – just what ergonomicists recommend) is better than the standard 64's, and their labelling (including the extra colours accessed by the CBM key) is both clear and more helpful.

On the debit side, the keyboard is much too light: a solid plate of lead in it wouldn't have added much to the overall weight, and it would have made the keyboard feel a lot more solid (a psychological consideration, maybe) while stopping it sliding around at the slightest touch (entirely practical).

And the key action is awful – far too light, far too clacky, generally imprecise. Our spacebar felt like the spring had failed at one end, so it didn't register at all if you hit the left side; hope that doesn't happen too often on other keys.

Identity crisis?

Then there's the two real problems, the things that confuse the SX's identity comprehensively: cassette and disk. Providing a diskfull of games would point to a home-computer appeal ... or maybe not, since the games are at best average. Commodore could have given away some much better offerings if it wanted to impress the domestic punter: how about the International Soccer cartridge?

Nor does it provide any of the facilities that home users would have found so helpful for programming – no extra sprite and sound commands on disk or cartridge, for instance.

Since tape will be the cheap and obvious home-computer storage and program distribution methods for some time to come it looks as though the SX isn't a home computer. Not for the average buyer, anyhow, even if said average hacker can stump up the necessary readies.

The business person and the software developer are the other candidates among the SX's potential market. Both need disk, and indeed there is a disk ... but what a disk! It's the 1541, for a start; a temperamental little number, to say the least.

And only one drive. For serious work (notably to take back-up copies and for handling applications that require both a data file and a program disk available all the time) you really need two. You could attach another 1541 to the serial port and use it as device no. 9, but that rather removes one of the points about a complete portable system – and those 1541s are notoriously sensitive: you can't lug them around like you can the 64.

There is in theory a DX-64 too – 'SX' presumably meaning 'Single-disk Executive computer', with 'DX' being the 'Double-disk' sibling. The manual talks about it, and all the 1983 PR from Commodore suggested a twin-disk portable as the main product with the single-disk version being the el cheapo model for the impecunious. But the DX isn't here yet, and we hear that it never will be – reportedly Commodore has found it really difficult to get two 1541-style drives into the box.

Commodore presumably sees a business market for the SX, which is why it gives away three business-orientated packages with it. Well, the word processor and the financial planner are ok ... but there are probably better options available commercially.

Anagram's stock control package is a good piece of 64 software, but it's a curious choice of freebie: a file-orientated application like inventory really shows up the limitations of speed, capacity and overall flexibility that the 64-plus-1541 combo necessarily imposes. Besides, stock control isn't the most commonly required of business tasks for microcomputers; a sales ledger with VAT computation would surely have appealed to a wider range of people.

As a software developer's tool – and for a magazine's reviewers and programmers! – the SX-64 isn't all that bad, however. Particularly if what you need is literally a portable 64, a machine that you can take anywhere and use to develop or review software intended for the average £200 Commodore 64.

There are no 'toolkit' facilities built into the 64, though; and nor is there a freebie disk full of those little essentials that would make programming the thing a lot easier. Why not a cut-down Programmer's Aid on disk?

Maybe Commodore doesn't see a software development market. But with the proviso that cassette would be useful, the other deficiencies and drawbacks pale into irrelevance against the fact that the SX is an ideal homework machine for people working with 64s; for us the SX-64 would be ideal.

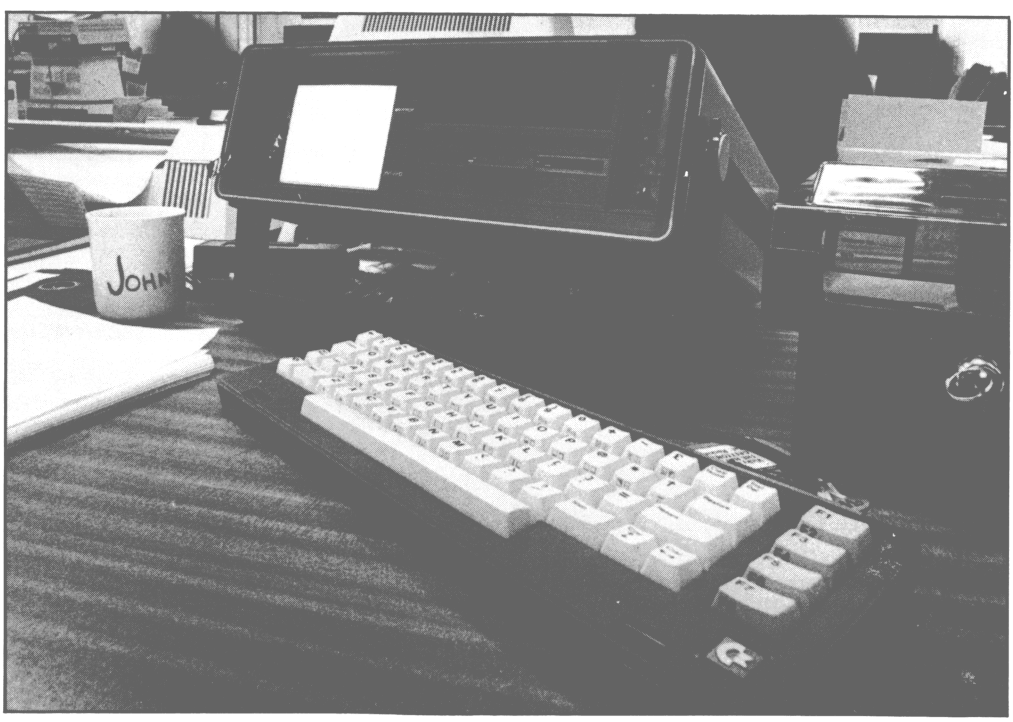
The REAL problem ...

But there you come up against the major problem – the price tag. At £895 the thing is just too expensive to be a home computer, but it's also too expensive to be a straight portable 64. Using your home TV you can have a 64 with 1541 for about £400; for not much more than half the SX's price you can get a decent monitor as well. The freebie software isn't worth the difference.

The word is that Commodore wants so few sales that the limited production facilities it has in Japan for the thing won't be embarrassingly overstretched. Seen in that light, the SX-64 is probably a market research and R&D exercise which will allow Commodore to get some idea of what the punters want and try out some packaging techniques to deliver that sometime in the future.

The SX is a neatly packaged and highly-priced 64 with a 1541, loaded with all the deficiencies implied by that and also blessed with most of the virtues of the combination. But as it stands, the SX-64 doesn't really look like a computer for anyone in particular.

But when the price drops to £499, it'll be a Good Buy for software writers and maybe for business users with a really tight budget: when the DX appears at £895 the business market will really open up for it: when there's a faster CPU and more functions in ROM and a rewrite of the DOS and higher-capacity microfloppies or maybe a 5MB mini-Winchester for less than £1,500, it'll sell a million. Shame about the keyboard. but anyhow, that's not the SX-64: that's the next-generation product for which the SX-64 looks like a suck-it-and-see exercise.



Section 5

Resource Directory

CHAPTER 28
Error Messages

When the 64 hits a problem, it puts a message on to the screen preceded by a question mark. Here's an alphabetical list of the error messages you might get, complete with possible explanations and what you can do about it.

Where a line number is involved, it indicates at which line in your program the error was detected. This doesn't necessarily mean the error itself is actually in that line – it may have been caused by something else in the program.

- BAD DATA ERROR IN LINE [*line number*]

The program was expecting numeric data and it got a character string. Correct the duff command(s) – normally it's a READ or INPUT# statement followed by a numeric rather than a string variable.

- BAD SUBSCRIPT ERROR IN LINE [*line number*]

You're trying to reference an element in an array that's outside the dimensions set up with a DIM statement by you – or, more likely, those assumed as defaults because you didn't DIMension an array. For the former you'll have to find and correct the DIM so that the array is larger, or change the array element number so that it's within the range; in the latter case you can just add a DIM to define the defaulting array.

- BREAK IN LINE [*line number*]

Not exactly an error message, just an indication that the program has been stopped and what line it had reached. This is usually because you wanted it to stop (you inserted a STOP statement in it to see how it was doing so far, or you hit RUN/STOP to halt it) or because you accidentally leaned on the RUN/STOP key. Either way you should be able to proceed from the BREAK point by typing 'CONT', unless ...

- CAN'T CONTINUE ERROR IN LINE [*line number*]

You've used a CONTinue command but the program has been deleted or perhaps amended such that program execution cannot proceed. The most likely cause, however, is that the computer has previously picked up another error and you haven't corrected it – it won't let you CONTinue until that has been fixed. Your best option is probably to try restarting with RUN and see if that throws up an uncorrected error.

You'll also get this message if you try to get going again after you've hit RUN/STOP *and* RESTART rather than RUN/STOP on its own.

- DEVICE NOT PRESENT

This usually appears when the 64 is executing a CMD, PRINT#, INPUT# or GET#. In theory it means the device you're trying to get at (usually printer or disk, occasionally cassette) isn't connected to one or both of the computer and the mains. That's easily remedied, of course.

In practice though you'll sometimes get this message for no apparent reason – in my own case it occasionally appears when I'm trying to read a file from disk while the printer happens to be switched on. The solution that usually works is to switch off everything you can and try again; alternatively OPEN a channel to the device you want to address and then re-attempt it.

This message will also be produced in some cases if you switch everything on in the wrong order, especially if you have two 1541 disk drives connected. You'll have to experiment to see which sequence the system prefers.

- DIVISION BY ZERO ERROR IN LINE [*line number*]

The computer won't allow you to divide by zero. You might have tried to do that by mistyping something, but it's more likely to occur within a FOR ... NEXT loop or as a result of filling an array with numbers that you then use in a division. The easiest solution is to put in a check for zero on any procedure that might just produce one in a division.

- EXTRA IGNORED

The 64 has been given too much in response to an INPUT prompt; alternatively, the input format isn't right – and though the user thought he/she was following all the rules, the 64 disagreed. In the latter case it's the programmer's fault ... Either way the unwanted and/or unrecognised items have been discarded.

- FILE ALREADY EXISTS

The computer won't let you set up two files on disk with the same name, but the only time it tells you that you're trying it is in the COPY command. Give up and rename the file.

- FILE NOT FOUND

You're trying to LOAD or VERIFY a file that the computer can't find on the tape or disk. It may not

be there; but on the other hand you may have mistyped the file name or misremembered what you called it. As a somewhat esoteric outside cause, with tape it could be that the 64 detected an aberrant end-of-tape marker before it got to the file.

- FILE NOT OPEN

You haven't used an OPEN command when the computer wants one from you – which is before a CMD, CLOSE, INPUT#, GET# or PRINT# statement. Or more likely you forgot that you'd CLOSEd the file you thought was open. So OPEN a file and retry.

- FILE OPEN

You used an OPEN command on a file that's already open. Either you don't need to re-OPEN it, or you need a different logical file number in the OPEN command. If in doubt, CLOSE the file and OPEN it again.

FORMULA TOO COMPLEX ERROR IN LINE [*line number*]

You've asked the computer to do too much – you used an expression that has too many brackets (probable) or too many functions (less likely). Split up the expression somehow; that's good policy anyhow, since it will make life easier for anyone who has to read through and understand the program in the future.

- ILLEGAL DEVICE NUMBER ERROR

As it says – you're attempting a SAVE or LOAD with a device number that can't be used. No 64s will accept device numbers 2 or 3 for these statements; the SX-64 won't like device number 1, and it will also give you this message if the device number is omitted altogether.

- ILLEGAL DIRECT

Most of the 64's commands can be used in 'direct' mode (ie they are executed as soon as you hit Return) ... except for these:

DATA
DEF FN
GET
GET#
INPUT
INPUT#

- ILLEGAL QUANTITY ERROR IN LINE [*line number*]

You have a variable that is outside the computer's range – usually meaning you're trying to POKE with a value less than 0 or above 255, occasionally signifies an attempt to assign a number greater than 32767 or less than -32768 to an integer variable.

- LOAD ERROR IN LINE [*line number*]

Something is wrong with an attempted LOAD from cassette – and in fact it's likely that the file you're trying to LOAD has been scrambled somehow, possibly beyond recovery. Leaving the tape near a reasonably strong magnetic field, such as you might get on top of a TV or near a loudspeaker, could cause this; so could extremes of temperature (like leaving the cassette in the sun).

Either of those is likely to be permanent. A temporary malfunction might just be due to the cassette player or its cable being too close to the 64's mains lead, though frankly that's unlikely. If repositioning the cassette unit doesn't help there's not much you do about this, except put it down to experience – and take more backup copies in future.

- MISSING FILE NAME

You're trying to SAVE a file to disk without giving it a file name – or rather you're attempting to file it as a pair of quotes with nothing between them.

- NEXT WITHOUT FOR ERROR IN LINE [*line number*]

The computer has found a NEXT statement in your program that is not associated with a preceding FOR. You might have missed out the FORE altogether, or a NEXT somewhere else in the program might have been tied to your FOR – that can happen if you aren't specific about which FOR variable you want executed NEXT. Check that your FORs and NEXTs pair off; and get into the habit of specifying which FOR each NEXT refers to.

- NOT INPUT FILE

You've OPENed a tape file for output only and you're now trying to read from it. Check your READ# command, but in practice the fault is more likely to be in an OPEN – if the third parameter of the OPEN statement isn't '0' or omitted altogether, you have opened a write-only file.

- NOT OUTPUT FILE

A tape file has been OPENed for input only and you're now trying to write to it. Check your READ# command, but as above the fault is more likely to be in an OPEN – if the third parameter of the OPEN statement isn't '1' or '2' you have opened a read-only file.

- OUT OF DATA ERROR IN LINE [*line number*]

The computer has run out of DATA items to READ – there must be enough entries in your DATA lines to fill all the variables in all the READ statements (though you *can* have more DATA items than you have READ variables). The simplest solution is to start counting and make sure there *is* sufficient DATA.

- OUT OF MEMORY ERROR

You may have run out of memory because your program is too big (which is just about possible) and/or it's creating too many new values for variables (which is considerably more likely, especially if you have big arrays). The answer is to simplify the program, or stop it producing so many new values for the computer to store.

You might also get this error even when PRINT FRE(0) shows you have a lot of memory left. In this case what's probably happening is that the stack is filling up with too many nested GOSUBS or FOR ... NEXT loops (the stack is an *aide memoire* for the computer that indicates where in the program it has to jump to and when). The solution? Again, simplify the program. That's not a bad maxim anyhow ...

- OVERFLOW ERROR IN LINE [*line number*]

You have a calculation which has produced a number too large for the computer – the largest number the 64 can handle is 1.70141884E+38 (that's 1.70141884 multiplied by 10 followed by 38 zeros). You will have to alter the program to avoid this, perhaps by changing the order in which your calculations are done. Do you really need numbers that big?

- REDIMENSIONED ARRAY ERROR IN LINE [*line number*]

The same array name has been used in more than one DIM statement; or you are now trying to DIM an array name to which the 64 has already assigned a default value since you started to use the array without DIMensioning it. Either way this error is likely to be the result of carelessness: for instance, have you got a DIM statement within a FOR/NEXT loop? Or have you missed out a RESTORE?

It helps to keep all your DIMs at the start of the program; that way they're much easier to check.

- REDO FROM START

Sounds awful but isn't. It just means the wrong kind of response has been given to an INPUT prompt – the program was expecting numeric and someone typed alphabetic, or *vice versa*. The message will continue to appear until the computer gets the appropriate kind of input.

- RETURN WITHOUT GOSUB

The 64 has found a RETURN that isn't linked to a preceding GOSUB – perhaps because you missed out the GOSUB or inadvertently added a RETURN (in which case you can add or delete as appropriate), but more likely because the sequence of execution caused the program to fall into a subroutine that you've put after the main body of the code. In this case, correct the program flow: a STOP inserted before the subroutine might help you find out why it's happening. An END there should prevent the program running on into the routine in the first place.

- STRING TOO LONG ERROR IN LINE [*line number*]

A string being read from tape or disk exceeds 88 characters; or, much more likely, you have too many characters in a string variable – the maximum allowed is 255. That may have happened because you tried to add two acceptable strings; if there's any danger of a concatenation producing an overlong string, it's worth inserting a test for length via the LEN function.

- SYNTAX ERROR

An illegal term or construction has been used. The probably cause is mistyping – common culprits are bad spelling, accidentally hitting one character twice while typing, and too many or too few brackets.

- TOO MANY FILES

You're trying to OPEN more than ten files or devices – no more than 10 different reference numbers can be in use at the same time. Sounds like a typo.

- TYPE MISMATCH ERROR IN LINE [*line number*]

Your program has tried to put the wrong type of value into a variable – string characters into a numeric variable (meaning a string enclosed in quotes being assigned to a numeric or integer variable name) or *vice versa* (meaning a string variable hasn't been given a CHR$ code or something in quotes). Change the errant command.

- UNDEF'D FUNCTION ERROR IN LINE [*line number*]

Your program is trying to use a function that you haven't yet defined via a DEF FN statement. So define your function – and it helps to put all your DEF FNs at the start of your program where you can spot them easily.

Sometimes this error happens because you weren't actually attempting to reference a user-defined function, you've just mistyped something that the computer has detected as a function.

- UNDEF'D STATEMENT ERROR IN LINE [*line number*]

You tried to RUN or GOTO or GOSUB a line number that doesn't exist. The target line has been omitted or you simply got the number wrong.

- VERIFY ERROR

The program you're trying to verify doesn't match what's in memory. There are many possible reasons for this; the best option is to SAVE it again and have another go at VERIFYing. If you are still getting a VERIFY ERROR on cassette, move the tape unit (it might be too near a magnetic field like that generated by the TV set) and/or try a different cassette.

CHAPTER 29
A package of POKEs

There are lots of potentially useful memory locations which can be addressed directly or from a 64 program by POKEs that change their value. Here's a short list of some of them and what you can do with them; to some extent this section will also serve as a partial memory map, with explanation and advice on several of the specific locations.

For starters, here's a little program I keep handy for PEEKing and POKEing:

```
10 INPUT "PEEK AT LOCATION"; P
20 IF P = 0 OR P > 65536 THEN END
30 PRINT "LOCATION" P "=" PEEK(P)
40 PRINT "NEW VALUE"
50 GET V$: IF V$="" THEN 50
60 IF V$=CHR$(13) THEN 90: REM exit if
Return
70 V=VAL(V$)
80 IF V>255 THEN 40: REM out of range
90 POKE P, V
100 INPUT "END"; E$
110 IF E$<>"Y" THEN 10
120 END
```

- **Location 1: memory configurator**

Decides which of the possible memory configurations in the 64 will actually be used.

Useful combinations for these first three bits are:

111 ... normal, with the KERNAL and 38K Basic
100 ... a 60KB memory with the character set but no KERNAL or Basic
001 ... 60K RAM with no KERNAL routines but with I/O

Bit pattern **110** is used for the C/PM-plus-Z80 cartridge, and different combinations of **111** and **110** with the other bits in location 1 are used for other plug-in goodies. But to fool around here you can assume that all other bits can start at 1, so the relevant POKEs would be ...

```
POKE 1,255      ... to get back to normal
POKE 1,253      ... to get 60KB RAM with
characters
POKE 1,249      ... to get 60KB RAM with
I/O
```

Location 1 also handles cassette control.

- **Location 19: INPUT query**

You don't always want a question mark to appear when an INPUT statement is executed: POKE 19,1 will suppress the question mark ...

```
100 POKE 19,1: INPUT "ENTER SOMETHING:
";Z$:
```

Note that you'll lose the automatic space between prompt and input *and* the automatic Return after the input, so include a space in the prompt and terminate the line with a colon.

- **Location 43: pointer to start of Basic program area**

Normally contains 1, and it's used with ...

- **Location 44: pointer to start of Basic program area**

... which normally contains 8. Between them these two point to location 2049. How? Because an eight-bit byte can't hold a number larger than 256, the computer cleverly uses *two* bytes to indicate

value	*set to 0*	*set to 1*
bit 0	switch out of Basic ROM	use Basic ROM
bit 1	switch out KERNAL ROM	use KERNAL ROM
bit 2	use character ROM	switch out character ROM
bit 3	no cassette data output	cassette data output
bit 4	cassette switch open	cassette switch closed
bit 5	cassette motor off	cassette motor on

Location 144 and the ST variable

Bit position in location 114	*Value of ST*	*Tape read*	*Tape load or verify*	*Disk*
n/a	0	none	none	none
0	1	none	none	timeout on write
1	2	none	none	timeout on read
2	4	short block	short block	none
3	8	long block	long block	none
4	16	unrecoverable error	any mismatch	none
5	32	checksum error	checksum error	none
7	128	end of tape	end of tape	device not present

addresses bigger than that – the first of a pair is called the 'high byte', the second is the 'low byte'. The decimal value of the address is then computed as ...

(value in high byte) + (value in low byte x 256)

In this case you can find the decimal address by ...

```
PRINT PEEK(43) + (PEEK(44) * 256)
```

... which is computed as ...

1 + (8 x 256) = 2049

- **Location 45: pointer to start of Basic variables**

Used with ...

- **Location 46: pointer to start of Basic variables**

... in a high/low byte combination. Variables are normally stored from 2091 onwards, but if you want a bit more room for a program you can take it by cutting down on the area reserved for variables. Garbage collection problems might result, though.

- **Location 51: pointer to end of Basic variables**

The high-byte of the pointer; normally 0, and used with ...

- **Location 52: pointer to end of Basic variables**

Between them this pair of locations tell you where variables for a Basic program will stop (the highest limit of memory that they can occupy). Normally contains 160, which indicates location 40960.

- **Location 55: pointer to end of Basic**

The high-byte of the pointer; normally 0, and used with ...

- **Location 56: pointer to end of Basic**

This tells you where Basic stops in memory. The end of Basic is usually the same as the end of Basic variables (40960). Normally this location contains 160, and if you get an OUT OF MEMORY error a POKE 56,160 may recover things for you.

- **Location 115: start of CHRGET**

This is the start of the machine-code routine CHRGET which is used to get characters from the keyboard, tape, disk or modem. It runs to location 138 and has been quite deliberately placed in RAM so that users can alter it.

- **Location 144: Status**

Holds the current status of a disk or tape operation – normally 0 until something starts happening. You don't need to check 144 directly, though, since you can use the reserved Basic word ST (typically in PRINT or IF statements).

- **Location 145: STOP and RVS key**

Normally 255, but POKE 145,127 acts as an instant RUN/STOP. Not that I've found any sensible use for it yet ...

- **Location 160: top of the jiffy clock**

- **Location 161: middle of the jiffy clock**

- **Location 162: bottom of the jiffy clock**

These three locations count time – they start at zero but as soon as you switch on the 64 they'll start counting. Every 1/60th of a second (a 'jiffy') the value of location 162 is incremented by one. When it hits 255, location 161 is incremented by one and 162 returns to zero; when 161 gets to 255, 160 is incremented and 161 goes back to zero, and when 160 hits 255 all three are cleared again.

The TI jiffy counter reads the values of locations 160-162, so you don't really need to reference these directly. But there's a quick way of resetting TI (and the string variable TI$, which can utilise it) by ...

```
100 FOR T=160 TO 162: POKE T,0: NEXT
```

(See also 788.) And you can time delays quite neatly with a WAIT combined with this reset:

```
100 FOR T=160 TO 162: POKE T,0: NEXT
110 WAIT 162, 120
```

That WAITs for 120 jiffies, about two seconds. It can be done rather more elegantly with TI$, too:

```
100 TI$="000000"
110 WAIT 162, 120
```

These locations can also be useful in precise timings, for reaction tests for instance. To convert their contents to something meaningful like seconds you could use something like this:

```
PRINT ((PEEK (162)/60)+(PEEK
(161)*256)+(PEEK (162)*256*256)
"seconds"
```

- **Location 174: I/O pointer**

Used with ...

- **Location 175: I/O pointer**

... in a high/low byte combination. If you alter locations 45 and 46 to store more Basic text, make sure you also change these two to the same values if you want to be able to save or load programs using disk or tape – otherwise you might lose part of the program.

- **Location 197: key currently pressed**

Normally 64, which means 'no key'. Key codes don't correspond to ASCII; see figure 29.2 for codes, or check them with ...

```
10 GET A$: IF A$="" THEN 10
20 PRINT PEEK (A$): GOTO 10
```

If two keys are pressed simultaneously, the one that appears here will be of a higher priority – see the table in figure 29.3.

The WAIT command is quite handy with 197 – for instance, you could WAIT 197,4,0 to stop everything until function key 1 is pressed.

PEEK (197) values for the 64's keys

Key	Value	Key	Value
A	10	6	19
B	28	7	24
C	20	8	27
D	18	9	32
E	14		
F	21	<backspace>	57
G	26	+	40
H	29	–	43
I	33	£	48
J	34	@	46
K	37	*	49
L	42	↑	54
M	36	:	45
N	39	;	50
O	38	=	53
P	41	'	47
Q	62	.	44
R	17	/	55
S	13		
T	22	f1	4
U	30	f3	5
V	31	f5	6
W	9	f7	3
X	23		
Y	25	<space>	60
Z	12	Return	1
		Stop	63
0	35	Home	51
1	56	Delete	0
2	59	CRSR down	7
3	8	CRSR right	2
4	11		
5	16	no key	64

- **Location 198: number of characters in keyboard buffer**

POKE 198,0 will clear out the keyboard buffer, thus avoiding any nasty surprises on your next INPUT or GET.

- **Location 199: reverse characters**

POKE 199,18 before a PRINT will switch to reversed characters and is easier than using RVS ON. Flip back to normal with POKE 199,0.

- **Location 204: cursor on or off**

Very handy when used with GET. Normally contains 0, meaning the cursor is blinking; but the cursor blink is suppressed while a GET statement is waiting for input. If you want to indicate that the user should in fact be entering something at that point, use POKE 204,1 on the same line as the GET ...

```
10 GET A$: POKE 204,0: IF A$="" THEN 10
20 PRINT A$
```

- **Location 211: Cursor position along line**

The value in this location indicates the current position of the cursor in terms of number of character positions along the line – just like TAB.

- **Location 214: Line location for cursor**

The value in this location indicates the current position of the cursor in terms of number of lines from the top of the screen. At least it does for values 0 to 25; but acceptable values for it go up to 255, and this little program produces some entertaining effects ...

```
10 PRINT "<clr>"
20 FOR P=0 TO 255: POKE 214,P: PRINT P
30 FOR T=1 TO 55: NEXT T: REM slow-down
timer
40 NEXT P
```

- **Location 251: free space!**

If you can squeeze a machine-code routine into four bytes, you can put it here – locations 251 to 254 aren't otherwise used, and there are advantages in being able to put a routine into Zero Page.

- **Location 631: Start of keyboard buffer queue**

The buffer is a bit of memory that watches what's going on at the keyboard and acts as a kind of 'type-ahead' store – you can enter a few more characters before the screen has finished displaying what you last typed and they won't be lost.

The buffer runs to location 640. You can POKE characters into the queue (start at 631, since it's a first-in first-out buffer) and the 64 will accept them as if you'd entered them from the keyboard. Change location 197 to show how many you've POKEd here, though.

- **Location 646: Current character colour**

This functions like a CTRL/colour key combination if POKEd with a value from 0 to 15.

- **Location 648: Bottom of screen memory**

Indicates the location holding the bottom left-hand corner of a 256-byte 'page' that contains the screen display – multiply by 256 to get the decimal value of the address. Normally holds 4, since the screen memory generally begins at location 1024; but you can use this to create alternative screens and flip between them, for instance for games or programs that use 'Help' information called up at any time by pressing a specified key.

- **Location 649: Size of keyboard buffer – and keyboard disabling**

Location 649 holds the number of characters that can be held in the 64's keyboard buffer – normally the buffer can hold 10 characters, but you can POKE 649 with any other number. POKE 649,0 prevents any input at all from the keyboard. POKE 649,10 enables it again.

- **Location 650: Keyboard repeat**

In its standard form, four of the 64's keys will repeat as long as you hold them down – INST/DEL for inserting and deleting, the two cursor movement keys, and the space bar. You can make *all* keys repeat by POKE 650,255: you can make *no* keys at all repeat by POKE 650,255: and you can return to normal with the four repeaters by POKE 650,0.

- **Location 653: Indicator for Shift, CTRL and CBM keys**

This location shows whether any of the three keys mentioned are currently being held down. The values are as follows:

none	0
Shift (or Shift lock)	1
CBM key	2
CBM and Shift	3
CTRL	4
CTRL and Shift	5
CTRL and CBM	6
CTRL and CBM and Shift	7

You can obviously test location 653 along with 197 to allow say Shift plus M to mean something different

from an unshifted M. You can also usefully employed WAIT with 653 – WAIT 653,4,0 stops everything until the Shift key is pressed.

- **Location 775: LIST killer**

POKE 775,200 to kill the LIST command – POKE 775,167 restores it again. This isn't as useful as it might sound, since the program has to be RUN before the ability to LIST it is disabled; and there's a much better way of preventing someone LISTing a program – include a REM followed by a Shifted L on the first line of the program. Any attempt to LIST will then crash with a ?SYNTAX ERROR. Tag this on to the end of a line containing some vital info so that any attempt to by-pass it by deleting the line also loses an essential part of the program. Not a perfect program protector, but it'll do ...

- **Location 788: STOP key**

To prevent an accidental crash of a program while it's RUNning you can use POKE 788,52 to kill the RUN/STOP key.

This also stops the TI/TI$ clock, incidentally: so be careful if you're running a timer using TI or TI$ – unless you *want* to stop the clock ...

To get them all back again you POKE 788,49. (But see location 808 for a better way to kill RUN/STOP.)

- **Location 792 and 793: disable RESTORE key**

If you shift the 64's screen to a new location you should disable the RESTORE key. That's because RESTORE will reset everything ... except location 648, the marker for the bottom of screen memory. So if you accidentally press it while using a new screen location, the display will be left in limbo. That can be avoided by ...

```
POKE 792,116: POKE 793,164
```

This doesn't actually disable the Restore key – it just sends all uses of the key to the READY routine, so you will get the screen back.

- **Location 808: RUN/STOP and RESTORE**

POKE 808,239 will kill RUN/STOP without affecting the clock – POKE 808,237 to get it back again.

And POKE 808,225 will kill the normal effect of typing RUN/STOP and RESTORE together – potentially useful if you want to remove the possibility of someone deliberately or accidentally exiting from your program. This one also kills STOP by itself; and LIST too (see 775 for a LIST-killer on its own). Once your program starts you won't be able to return to normal until it ends or unless you've included a POKE 808,237 in it.

Be wary with this one: POKEing 808 with other values might well screw things up for good until you next switch off and on.

- **Location 1024: Start of screen memory**

Locations 1024 to 2023 provide a 1,000-cell 'map' of the screen, 25 rows by 40 columns. 1024 is the top left position, 2023 the bottom right. To POKE a character into a particular position this formula is pretty useful:

position = (1024 + column number) + (40 x row number)

- **Location 2040: Start of sprite data pointers**

Locations 2040 to 2047 tell the 64 which block to look in for the relevant sprite data – 2040 refers to sprite number 0, 2041 to sprite 1, and so on.

- **Location 2048: Start of Basic program space**

Use locations 43 to 46 to make the space available for Basic programs smaller.

- **Location 53248: Start of sprite position data**

Locations 53248 to 53263 are a succession of pairs that give first the 'X' (or horizontal) position and then the 'Y' (vertical) position for each of the eight sprites. So 53248 itself has the X position for sprite 0, 53249 gives its Y position, 53250 is the X position for sprite 1 ... and so on.

- **Location 53264: X position past 256**

The screen 'grid' for sprites is 320 pixels wide; but the X position in alternate locations starting at 53248 is held in a single byte – and the largest number a byte can hold is 256. Location 53264 must be turned on whenever you want a sprite to cross the 256-pixel boundary, and it must be turned off again when it moves back over it.

- **Location 53265: Graphics control (VIC control register)**

Packed into this location are a whole bundle of 64 graphics controllers. From the bottom:

bits 0 - 2 *scrolling*: you can scroll over a display between 1 and 8 pixels per movement

bit 3 *select depth of display*: normally it's 25 lines, in which case this bit is set to 1. For 24 lines it's 0 (you lose half a line from the top and the bottom)

bit 4 *blank screen*: set to 1 if you want the display area the same colour as the border

bit 5 *bit map mode*: setting bit 5 to 1 switches it on, giving your program the opportunity to work on a grid 320 pixels wide by 200 high

bit 6 *extended colour mode*: set to 1 to switch it on – each character takes on a background colour in addition to the normal screen background

bit 7 register compare – used with 53266

Most of these are pretty complex and belong elsewhere in the book. But one quick and interesting effect is worth noting – a whole screenfull of text or graphics can be put on on to the screen in one chunk, rather than one character or one line at a time. Try this:

```
10 B=53265
20 PRINT "<clr>NOW YOU SEE ME ..."
30 GOSUB 100
40 PRINT "<clr>... NOW YOU DON'T"
50 GOSUB 100
60 GOTO 20
100 POKE B, PEEK (B) AND 239
110 FOR T=1 TO 999: NEXT
120 POKE B, PEEK (B) OR 16
130 FOR T=1 TO 999: NEXT
140 RETURN
```

What's happening is that the text is being put on the screen *before* you get to see it.

- **Location 53267: Light pen X location**

This holds the horizontal location of a lightpen in the form of a value from 0 to 255. That doesn't correspond exactly to any particular position on the screen, though, and to read it successfully you may have to do some experimentation and calibration.

- **Location 53268: Light pen Y location**

This holds the vertical position in the same way.

- **Location 53270: screen shake**

This location serves a number of functions, notably selecting multi-colour mode. But it can also provide a convincing visual explosion effect. Try this:

```
FOR A=0 TO 255: POKE 53270,A: NEXT
```

- **Location 53272: Pointer to screen and character memory**

This one can be used to switch between upper and lower case – POKE 53272, 23 gets you into lower case, POKE 53272,21 returns you to the normal upper case.

53272 also determines the location of the screen memory. Normally it starts at 1024, but you can shift it (or flip between different screens using location 648). The safest formula is ...

POKE 53272, (PEEK (53272) AND 15) OR A

value of A	*screen location*
0	0
16	1024
32	2048
48	3072
64	4096
80	5120
96	6144
112	7168
128	8192
144	9216
160	10240
176	11264
192	12288
208	13312
224	14336
240	15360

To use a character set of your own you need this one ...

POKE 53272, (PEEK(53272) AND 240) OR B

value of B	*characters start at*
0	0
2	2048
4	4096
6	6144
8	8192
10	10240
12	12288
14	14336

- **Location 53280: Border colour**

The value in here determines the border colour, as follows:

0 black	8 orange
1 white	9 brown
2 red	10 light red
3 cyan	11 light grey
4 purple	12 medium grey
5 green	13 light green
6 blue	14 light blue
7 yellow	15 dark grey

- **Location 53281: Screen colour**

POKE this location with colour values as above to change the background colour. This little program runs through all the possible combinations and lets you see the POKEs to 53280 and 53281 for any combination you like the look of:

```
10 B2=0: BRDER=53280: SCREEN=53281:
POKE BRDER, 0: POKE SCREEN, 0
20 PRINT "<clr><cud x 4>" TAB(8)
```

```
"<rvs>COLOUR COMBO SELECTOR"
30 PRINT TAB(8) "PRESS ANY KEY TO START"
40 GET A$: IF A$="" THEN 40
100 FOR B1=0TO15
110 POKE BRDER,B1
120 PRINT "<cud>NEXT BORDER? NEXT
SCREEN?": PRINT "OR SEE COMBINATION?"
130 PRINT "<cud>TYPE B OR S OR C"
140 GET B$: IF B$="" THEN 140
150 IF A$="C" THEN GOSUB 900
160 IF A$="B" THEN 200
170 B2=B2+1: POKE SCREEN,B2
180 PRINT "<cup x 5>": GOTO 120
190 IF B2>15 THEN 300
200 PRINT "<cup x 5>": NEXT B1

300 PRINT "AGAIN?"
310 GET C$: IF C$="" THEN 310
320 IF C$="Y" THEN 20
899 END

900 PRINT TAB (4) "<cud>BORDER = ",B1
910 PRINT TAB (4) "BACKGROUND = ",B2
920 PRINT "<cud X 4>CONTINUE?"
930 GET D$: IF D$="" THEN 930
940 IF D$="N" THEN 899
```

- **Location 55296: Start of screen colour memory**

The screen colour map runs from 55296 (top left position) to 56319 (bottom right); if you POKE a character into a position at 1024 onwards, you have to POKE a value into the equivalent colour location here before you can see it. The relevant POKE values are as for 53280 and 53281.

- **Location 56320: Control port 2**

PEEK here to find out what's happening with a joystick or whatever connected to control port 2. Bit 4 looks to see whether the FIRE button is being pressed. Bits 0 to 3 handle joystick direction, with decimal values in the range 0 to 15 – though you'd ordinarily check for only nine values, as follows:

Since this location also scans the keyboard, you can have a kind of fake joystick by holding down one or more of the back-arrow, 1, 2 and CTRL keys. Key 1 equates to North, back-arrow to South, CTRL to West and 2 to East; CTRL and 1 together give you North-West, and so on.

You can see that at work for joystick or keys by RUNning this one-liner. Move the stick or hold down the keys to see the values change:

```
10 PRINT 15-(PEEK(56321) AND 15)
```

This location also provides for paddles, but you'll need machine code to read them because of their complexity and sensitivity.

- **Location 56321: Control port 1**

The same applies for the second control port, an additional or an alternative place for a joystick. In addition it handles a lightpen – bit 4 then detects a lightpen trigger.

- **Location 56325: Control port 1**

When you SAVE or LOAD programs on the 64, the screen goes blank. This can be a bit disconcerting, but it's worth noting that any program will run around five per cent faster if the screen is blank at the time. You can blank the screen by:

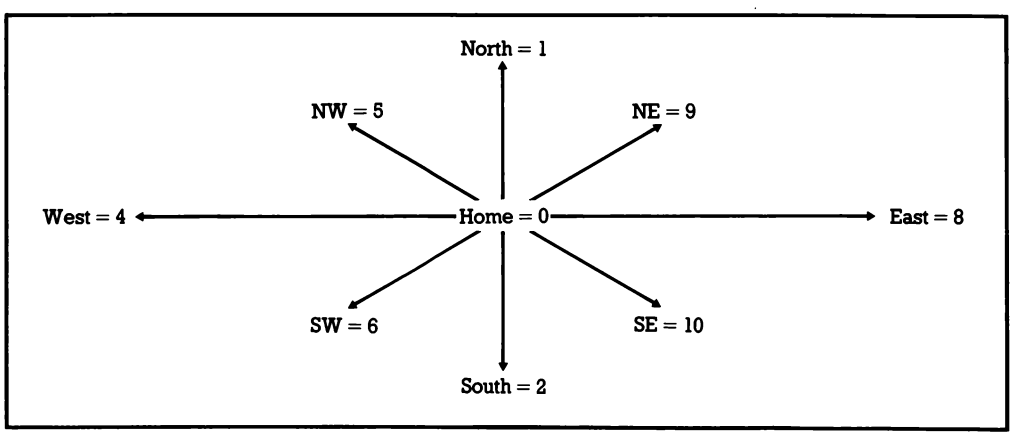
```
POKE 56325,PEEK(56325) AND 239
```

Return to normal:

```
POKE 56325,PEEK(56325) OR 16
```

- **Location 64738: Cold restart**

SYS 64738 will give you a 'cold restart' without having to switch off and on again. It will clear memory, reset all variables, and generally give you the same effect as powering off and on without the wear and tear on the switch.



CHAPTER 30
A Practical Glossary

The Commodore world is full of trade names, product numbers and "buzzwords' (a term the computer industry felt it had to invent to describe computer-industry jargon). This A-to-Z of the 64 includes most of them.

accuracy Being free from error, obviously, but in computing it means quite specifically the size of the error – in terms of the difference between the actual number and its value as stored in the computer.

adapter A means of attaching extra parts, typically some external device.

address The reference of a particular location in memory.

ADSR The four phases of a sound – Attack (how quickly it reaches maximum pitch), Decay (how quickly it fades), Sustain (how long it lasts), Release (how long it takes to become completely silent again.

algorithm A set of well-defined rules for the solution of a problem in a specified number of steps.

alphanumeric Comprising letters of the alphabet and numeric digits. Also called 'alphameric' by some (not by me).

application A particular job that the computer can do for the user (as opposed to tasks that it does by way of internal 'housekeeping'). So an 'application program' or '*package*' is software designed for such a job, like a word processor or a sales ledger program.

array A collection of values which the 64 can reference by a single *variable* with a *subscript*.

ASCII American National Standard Code for Information Interchange – the commonly-used code for passing information around between computers and between different parts of a computer system. Like all such codes, it decides what particular *bit* patterns relate to which particular characters.

assembler A form of programming language that comprises a near-English approximation of *machine code* – it permits you to use mnemonics rather than hexadecimal numbers, for instance.

bank-switching The ability of a computer to use alternative 'banks' of memory depending on circumstances. The range of possible memory addresses remains the same; the actual *location* of those addresses can be changed by bank-switching.

baud A measure of transmission speed denoting the number of signals per second – which for practical use can be treated as identical to 'bits per second', though in fact that's not true (and at higher transmission speeds can be positively misleading).

binary Pertaining to two possible states. So a binary number system such as computers use expresses everything using only two digits. This suits the internal operation of computers very well, since it can be related to the presence or absence of specific voltage levels – and that is quite easy to design in electronics terms.

bit A binary digit, usually expressed as a '0' or a '1'. Groups of eight bits make one *byte*.

Boolean A form of logical mathematics used in programming to permit different actions if two conditions are the same, if they're not the same, or if one or the other matches a specific third condition.

branch A switch in the flow of a program, usually caused by an IF/THEN, GOTO or GOSUB statement, which means that the next line number in sequence is *not* the one that is executed next.

buffer An area of memory used as temporary storage.

bug Fault or error in a program (errors in hardware may also be called 'bugs', but aficionados tend to prefer the term 'glitch').

byte A group or pattern of eight *bits* that can be used to encode a single *character*.

call A branch to a *subroutine*.

carriage return Literally, the action of returning a printing mechanism to the start of the next line. From that genesis the term has also come to refer to returning the *cursor* to the start of the next line, though it is more usually abbreviated to *Return*.

cassette buffer An area of the 64's memory that is used for temporary storage while transferring information between memory and a cassette unit.

chaining Linking two or more programs together so that one operates on or with another, or RUNs it. Typically this is done without losing all the variables set up by the first program (as normally happens

when your program includes a LOAD command). In other words, program one can define some variables to be used by program two: that way you can have in effect a single chained program that's larger than the maximum memory size available for it.

channel An electronic 'pathway' along which signals (and thus data) can be sent. Usually means a 'conceptual' or 'logical' path rather than an actual set of wires.

character A letter, numeric digit, punctuation mark, arithmetic operator sign, and other symbols collectively used to represent or delimit data.

character set A group of characters – or more specifically, the collection of *all* the characters that the 64 can recognise at one time.

CIA Complex Interface Adapter – the 64 has two of these multi-function chips.

clear Referring to the screen, it means 'blank the display' and is done by PRINTing the <clr> character or keying the CLR/HOME key (the contents of memory are not erased, just the contents of the screen). As a Basic command it effectively resets all variables to zero or to *empty strings*.

cold start A complete reset, such as would happen if you were to switch off and then on again. Anything in memory will be lost.

communication The transmission and reception of data.

compiler A type of programming language that goes through an intermediate stage before the code the programmer writes can be used on a computer. The programmer writes a program and then 'compiles' it into a form the computer can handle; when the compiled version (the 'object' code) is loaded, the bulk of the translation between the programmer's neo-English commands and the computer's own more or less illegible representation of it has already been done.

By contrast with an *interpreter* language for programming, the compiler approach can produce more complex and more elegant code (because the program development part doesn't have to leave any room for the resulting program actually to run, it can take all the computer's memory to provide the programmer with extra facilities). The program itself will also run faster (because the computer doesn't have to translate each programming command as it goes – that's largely been done in the compilation process). And the program can also be larger, providing more facilities, for a given size of computer (because at run-time it doesn't need to have the whole of the language in memory with it, just a cut-down version which can exclude all the program development tools).

concatenate Join two *strings* together.

constant A fixed value, like 'pi' or '*K*'. More generally, a particular numeric value or a set of characters to be stored in a memory location and/or used in a Basic statement – an *unchanging* element of the statement, in other words, as opposed to a *variable*.

co-ordinates A method of identifying a particular point on a two-dimensional grid (which could be the 64's screen, or a business graph) expressed in terms of its horizontal and its vertical positions.

CP/M By Digital Research. The Control Program for Microcomputers first appeared in 1974, subsequently becoming the *de facto* standard operating system for personal computers based on eight-bit microprocessor chips like Intel's 8080, 8085 and Zilog's Z80. Now a dinosaur that won't lie down and die, it makes extensive use of obscure acronyms for simple functions. Less user-friendly than the average video recorder, but just as popular: four hundred thousand users can't be wrong.

crash A sudden and often inexplicable halt midway through a program – could be anything from a *bug* in the program to a power failure.

cursor Moving marker on the screen to indicate where the next character will appear.

data Loosely, 'information'. More literally, data is any information that is meaningful to the computer – could be parts of a program, could be stored *files*, could be anything sent or received, could be keyboard input ...

data management system Same as *filing system*, but may in fact be more of a *database management system*.

database Loosely, a collection of *files*.

database management system Generally the same as a *filing system*, though the term implies more of an ability to maintain files of records such that they are *not* specific to one *application* and can in fact be used sensibly by several different applications.

debug To correct mistakes in a program – to remove *bugs*.

decimal The number system we use conventionally, where ten digits are used in combination to express a value (as opposed to *binary*, which uses two: and *hexadecimal*, which uses 16).

decrement To reduce the value of.

default A value or a setting for a variable condition that will be used if no other is specified.

Direct mode Typing commands and statements without a line number, such that they will be executed as soon as Return is pressed.

disable To render something temporarily inoperative without actually switching it off completely.

disassemble To produce a *listing* of a *machine-code* program, typically part of the *Kernal*.

edit To enter, modify or delete data.

empty string A *string* containing absolutely nothing, usually written as ''.

enable To permit operation.

exponentiation The mathematical operation of raising one number to a power.

expression A compound of variables, values, operators and/or functions that can be used in a Basic statement.

file A collection of *records*, usually grouped so as to apply sensibly to one or more specific *applications*.

filing system A *package* designed to create, maintain and utilise *files* of *records*.

floating point A number that may be expressed with a fractional portion following a decimal point.

FP Abbreviation for *floating point*.

function The name for a particular formula that is applied to an accompanying value or *variable*.

garbage collection An automatic action within Commodore Basic – the computer stops momentarily to reorganise its memory and discard variables that are no longer current.

graphics Using pictorial representation rather than conventional *characters*.

hard copy Output on paper.

hexadecimal A number system that expresses values in terms of 16 characters (numerals 0 to 9 and letters A to F).

high-byte That byte in a pair that holds the 'high' value of a hexadecimal number – hex values are stored in two consecutive bytes.

Immediate mode Alternative name for *Direct mode*.

increment To increase a value.

initialise To reset everything to zero or to start-up values.

integer A whole number, one without a fraction or a decimal point.

interpreter A form of programming language where the actual program coding resides in memory with the programmer's development tools and the part of the programming system that handles the conversion of near-English programming commands into the 'machine language' the computer uses. The main plus point is that programs can be written and run immediately without the need for an intermediate translation stage; so they can be altered and re-run *ad infinitum* to suit the programmer, until the desired end is attained. This makes life a lot easier for the programmer, but has a number of drawbacks – the need for the *whole* programming system to be available in memory alongside the end-program cuts down the space available, the step-by-step conversion the computer has to do every time it runs the program slows things down, and the ease of amendment encourages sloppy programming with little forethought that in turn can produce inefficient code.

The generally preferred alternative is a *compiler* language; the 64 comes with interpreted Basic as standard, but you can buy compiler versions of Basic from a couple of independent suppliers.

interrupt To halt an operation in such a way that it can be resumed.

iteration Repetition, usually of instructions in a program *loop*.

jiffy One-sixtieth of a second, as used by the 64's TI clock.

joystick A moveable lever that can pivot in all two-dimensional directions. A joystick cane be used as cursor-moving pointer or a 'drawing' device on a graphics display; such units generally come with one or more 'fire' buttons that effectively duplicate the Return/Enter keys to select a particular position that has been reached by moving the stick.

justify To align characters to fit the positioning constraints of a particular format. In printing terms, 'left-justification' means to arrange the text so that it starts at the left-most position: 'right-justification' formats it against the right-hand margin.

K A constant conventionally used in measuring computer and disk memory capacity to mean '1,024' (or more loosely '1,000'). So '128K bytes' is 128x1024 = 131,072 bytes.

KB Kilobytes – shorthand for '*K* bytes'.

Kernal A collection of *machine-language* instructions and routines permanently resident in the 64 and controlling basic operations of the computer.

light pen A photosensitive device connected to the computer by cable and used to select particular points on the screen, or to move a cursor or line-drawing point.

line feed A movement up or down one line of the paper in a printer – and by extension of the display on a screen.

listing The results of typing "LIST" – a display or printed copy of the lines in a program.

location The physical equivalent of an *address* – a place in memory at which data can be stored.

loop A set of program lines that are repeated, usually because a FOR/NEXT combination or a GOTO statement forces them to be.

low-byte The 'low' portion of a two-byte pair holding a hexadecimal number.

M One million. So '1.2M bytes' is 1,200,000 bytes.

M/C Abbreviation for *machine code.*

MB Megabytes – shorthand for '*M* bytes'.

machine code The most elemental form of instructions the computer can understand – meaning that those instructions correspond directly to individual operations and activities within the computer.

machine language Alternative name for *machine code.*

matrix An *array* of two or more dimensions.

memory Used by some to mean *any* computer storage medium, but by me to refer only to the *internal* memory of the computer – the memory chips, in fact.

memory map A key, index or guide to the organisation of the computer's memory – shows what goes where in terms of the addresses at the beginning and end of discrete areas of memory.

menu A list of options.

microcomputer Low-cost and compact (typically desktop) computer that contains all the elements – processor, storage, input and output devices – of the room-sized mainframes of the 1960s. And sometimes just as powerful.

microprocessor Not a complete microcomputer, but the name of the central processor chip used as the heart of a microcomputer system. It's a computer-on-a-chip (maybe more than one chip) but not a chip that's a complete computer because other electronics are required to control storage, keyboard, and display devices. Used to be a popular subject in the latter 1970s for television documentary makers.

ML Abbreviation for *machine language.*

modem Abbreviation for 'modulator-demodulator', a device that converts between the digital signals a computer uses and the analogue signals employed by telephone lines. A modem then allows information generated by the computer to be transmitted over phone lines; another modem at the other end could convert it back into digital form for use by another computer.

monitor A display unit, like a TV screen. Or a program that allows you to access and alter memory locations using *hexadecimal* notation and *machine code.*

monochrome Displaying a single colour on a background.

moveable object block Alternative (and little-used) term for *sprite.*

nesting Putting one *loop* inside another.

null string Same as *empty string.*

nybble Half a *byte* – four bits.

operator Symbol for an operation to be performed.

overlay A programming technique that uses the same area of memory for different parts of the program at different times.

overwrite To record over some existing material on disk or in memory.

package A collection of one or more ready-to-use programs for a particular *application.*

parameter A variable element in an operation that the user can specify.

peripheral Any externally-connected device plugged into a computer.

pixel Acronym for 'picture element', meaning a single addressable point on a display – in other words, the smallest element of a computer-generated graphic display: one point that can be lit or unlit.

port An access point – or to put it another way, a socket.

precedence Usually meaning the order of priority in which arithmetic and other *operators* are executed.

precision A measure of the ability to distinguish between nearly identical values.

program A sequence of instructions for a computer to follow.

program development Writing and *debugging* a program.

Program mode Entering instructions with a preceding line number, such that they are not executed when Return is keyed; instead they will be executed along with other instructions as part of a program when RUN is typed.

prompt An indication that the computer's waiting for you to supply a command or some information.

radian A unit of measure for a circle or part of a circle.

RAM Random-Access Memory – though access isn't really 'random'. 'Read/write memory' would be a better term: it's a type of internal component that can hold data (including programs), but the contents of which can be replaced by fresh data as required.

random file A type of *file* on disk only in which all items are referenced in an index such that access to an individual *record* within the file and amendment or replacement of it are both possible and quick (since the file does not have to be read through from the start).

range A set of permitted values.

record A collection of related information that can be treated as a single unit. Individual items of information that make up the record are *fields*; a group of records is a *file*.

read-only No alteration or addition permitted.

real number Alternative term for *integer*.

relative file A type of *file* on disk only in which all items are referenced in an index such that access to an individual *record* within the file and amendment or replacement of it are possible.

reset To return everything to a start-up condition.

Restore The Commodore Basic facility that returns *arrays* to null values so that they can be reused in a program.

Return Derived from *carriage return*, meaning the end-of-line or end-of-entry terminator – usually generated, of course, by hitting the Return key on the 64 keyboard.

ROM Read-Only Memory. Unlike *RAM*, the contents of these internal memory components cannot be changed; so ROM is used for factory-supplied programs providing specific functions that all users will get – like the *diagnostic* checks done by the 64 at start-up. Because ROM doesn't have to provide for input and alterations, it can be much faster to 'read' than RAM.

routine Part of a program that can be reused frequently to provide a commonly-required function: a group of instructions with a specific purpose and so usually treated as a single entity – generally a *subroutine*, in fact.

RS232C By the US Engineering Industries Association. It is a standard electrical connection for computers, communications equipment, and printers based on lozenge-shaped 25-pin plugs and sockets. Is used for most *serial* transmissions – the simplest sort, where binary codes for the information are sent down the wire character-by-character. Drawback is that that is slow. RS232 only lays down the electrical characteristics of the link, not the way the information is coded into binary digits. Result: there are almost as many different types of RS232 interfaces as there are varieties of *Basic*.

scroll To move the display horizontally or vertically such that new information appears at one edge while existing material disappears from another.

serial A transmission method that contrast with *parallel* communications, in that each character is sent one bit after another down a single wire rather than as eight bits travelling in parallel down eight wires.

sequential file A type of *file* on disk or cassette in which all items are stored in sequence and have to be read into the computer in bulk. There is no way to access an individual *record* within a sequential file, or to amend it.

SID Sound Interface Device, the sound-generating chip in the 64.

sprite A small, independently-defined picture that can be placed on and moved over the display screen.

string A collection of characters.

structured programming An approach to programming that emphasises the logical progression of program code, typically featuring great legibility and eschewing the use of the GOTO statement (the use of which can destroy a logical flow to the program listing).

subroutine A *routine* that referenced by a GOSUB statement. Once the program lines of the subroutine have been executed, the program flow returns to that point in the main program from which a branch was made to the subroutine.

syntax Rules covering the structure, form and use of a language.

tab Tabulate – a function derived from typewriting terminology whereby the cursor can be moved directly to a specific position on the screen.

toggle To switch back and forth between two conditions.

update To modify with current information – usually applies to stored files.

variable A label that during the course of a program can reference different values or *strings* – so its 'meaning' can vary.

version number As errors are corrected in proprietary and/or published programs and amendments made, the result is frequently given a qualifying Version Number to indicate which version it is. Major changes are signified as 'Version 2', 'Version 4' and the like; less significant amendments are indicated as 'Version 2.1', 'Version 2.2' and so on.

VIC The Video Interface Chip – the chip that provides the 64 with sprites and other graphics capabilities.

Vic-20 No relation to *VIC*: Commodore's predecessor to the 64.

warm start A method of resetting the computer such that the contents of memory are left undisturbed.

wedge A set of user-written *machine-code* instructions that add extra commands to the 64 and/or extra statements to Basic. Wedges are placed in *Zero Page* and the CHRGET routine there is amended to detect calls to the new code.

wild card A special character used in string matching, in particular for searches (with extended Basic commands) and for disk and cassette operations. The commonest 'wild card' character is the asterisk,

meaning 'match anything that starts with the specified characters' – so LOAD "PR*" will load the first program found on tape that has a name starting 'PR'.

write To record data on a disk or tape, or in memory.

write-protect To prevent data being recorded – typically to preserve the existing contents of a tape or disk.

zero-page The area of memory containing locations 0 to 255

CHAPTER 31
Now Read On

In the opening words for this book I suggested that it wouldn't be the last one you'll ever need if you are going to get stuck into the 64. This Chapter lists some of my favourites – it's definitely a biased selection, but at least I have read them all ...

Getting started

Usborne Guide to Computers
by Brian Reffin-Smith
Usborne, 1983: £1.85

– excellent low-cost illustrated intro for beginners

The Good Computing Book for Beginners
by Dennis Jarrett
Hutchinson, 1984 (third edition): £2.95

– by me. A cheap and bulky little number providing a general-purpose introduction to computers, a glossary with personal comments, a few jokes

COMPUTE!'s First Book of 64
Ed. COMPUTE! magazine
COMPUTE! Books, 1984: $12.95 (about £11.70 from UK distributors)

– collected articles from the best of the Commodore-oriented US magazines. Highly recommended for all levels of experience

Commodore 64 – Getting the most from it
by Tim Onosko
Prentice-Hall, 1983: £7.95

– very good all-round introduction in depth. Stops short of assembler and multi-colour mode, but then so do most us. Meaty and well-written; recommended despite a few bugs in the program examples

Discover your Commodore 64
by Philip Williams
Century Publishing, 1984: £2.95

– small but clear, concise and cheap intro to programming – nothing too complicated but presented in a nice style

Moving on

COMPUTE!'s First Book of 64 Sound and Graphics
Ed. COMPUTE! magazine
COMPUTE! Books, 1984: $12.95 (about £11.70 from UK distributors)

– an even better source-book. Collected articles again, and again highly recommended

Commodore 64 Programmer's Reference Guide
by A Finkel, N Harris, P Higginbottom and M Tomczyk
Prentice Hall: £9.95

– practical, good-value guide with programming tips, much explanation (albeit a bit patchy in parts) and an intro to assembler

Mastering the Commodore 64
by A J Jones and G J Carpenter
John Wiley, 1983: £6.95

– a pot-pourri for more experienced users, jam-packed with information. Highly recommended

Handbook of Basic for the Commodore 64
by Frederick Mosher and David Schneider
Prentice-Hall, 1984: £8.95

– must be the best book yet on the 64's Basic. Explanation, examples, comments and applications for each Basic statement *plus* good sections on sprites and music *plus* lots of incidental info of relevance

The Master Memory Map
by Paul Pavelko and Tim Kelly
Prentice-Hall, 1983: £14.35

– location-by-location guide to the 64's memory (very interesting, lots of example programs, good learning tool) plus some less good sections on sprites, sound, reconfiguring memory etc. Apart from the price (it's only 185 pages) this is recommended as a friendly intro to PEEKs and POKEs

Sprites and Sound on the Commodore 64
by Peter Gerrard
Duckworth, 1984: £6.95

– sane, sound, comprehensive guide for more advanced users. The best yet from this prolific author?

Basic Subroutines for Commodore Computers
by Eddie Adams
John Wiley, 1983: £10.50

– problem-and-answer approach with many useful programs (a few too many conversion routines for a five-star rating, though) and a good deal of incidental education about subroutine-writing

Commodore 64 Exposed
by Bruce Bayley
Melbourne House, 1983: £6.95

– terse but packed collection of Useful Facts – especially good for machine-code routines

Secrets of the Commodore 64
by P Cornes and A Cross
Bernard Babani, 1984: £1.95

– even smaller but even cheaper. Doesn't live up to the promise of the title, but still an interesting collection of tips, techniques and tools

Assembler and machine-code programming

6502 Assembly Language Programming
by Lance Leventhal
McGraw-Hill, 1979: £12.95

– despite its age, still one of the best books on machine code for the 64's microprocessor

6502 Assembly Language Subroutines
by Lance Leventhal and Winthrop Saville
McGraw-Hill, 1982: £12.95

– companion volume – an excellent collection of over 50 routines.

Advanced Basic and Machine Code for the Commodore 64
by Peter Gerrard
Duckworth, 1984: £6.95

– not at all a bad stab at the subjects in the title

Beyond Basic

Comal Handbook
by Len Lindsay
Prentice-Hall, 1984: £15.25

– best Comal book from the bloke who's done most to propagandise the language. Expensive, but includes a complete Comal on disk

The Complete Forth
by Alan Winfield
Sigma, 1983: £6.95

– best intro I've seen to Forth

Some magazines

Commodore User
monthly, £12pa or 85p per copy
The Metropolitan, Enfield Road, London N1 5AZ

– my own mag. 'Nuff said, except that we try hard to live up to the user bias implied in the title rather than reprinting Commodore's press handouts

COMPUTE!
PO Box 5406, Greensboro, NC 27403, USA
monthly, $20pa

– 6502 computers only. Always lots of 64 coverage in well-filled pages. Recommended, even for the UK in spite of US import prices

Commodore Magazine
487 Devon Park Drive, Wayne, PA 19087, USA
monthly, $15pa

– official Commodore US magazine. Lightweight marketing tool, but with some real nuggets in its pages

Commodore Horizons
167-169 Great Portland Street, London W1
monthly, £15pa or 95p per copy

– amiable enough

Section 6

Summaries

APPENDIX 1
Basic in Summary

Modes

Basic can operate in two modes – **direct** or **immediate** mode, and **program** mode.

- **Direct mode**

All commands and most statements can be typed in without a line number and executed immediately Return is pressed. Multiple statements can be entered on a single line with the colon separator. The following statements cannot be used in direct mode:

CONT
DEF FN
GET
GET#
INPUT
INPUT#
DATA

- **Program mode**

Programs are defined as sequences of commands or statements preceded by a valid line number. They will not be executed until the RUN command is used in direct mode.

The LIST, LOAD, NEW and RUN commands can be incorporated into programs, whereupon they will be executed in direct mode as soon as they are encountered – in other words causing an exit from the program.

Operators

- **Arithmetic:**

\+ addition
– subtraction
* multiplication
/ division
↑ raise to a power

- **Comparison:**

= is equal to
< is less than
<= is less than or equal to
> is greater than
>= is greater than or equal to
<> is not equal to

- **Logical:**

AND both expressions must be true to proceed
OR either expression must be true to proceed
NOT neither expression must be true to proceed

Variables

- **Restrictions**

Any number of characters can be used as names for variables, though only the first two will be recognised by 64. The first character must be a letter, the second can be a letter or a numeral.

You cannot use FN, TO, IF, ON, or OR; and you should avoid using other Basic keywords and the system variables ST, TI and TI$.

- **Types**

Three types of variable can be identified; the 64 will check that the value given is of the right type and will limit the operations performed to those permitted for that type of variable. Types are:

integer: whole numbers (with no decimal point) in the range -32768 to +32767. Identified by following the variable name with a percent sign, eg ZA%=1984

real or **floating point**: numbers including a decimal point and/or an exponent, with exponents in the range -38 to +38. Identified by having no qualifier on the name, eg AB=12.456 or D1=6.12279E-5

string variables: characters (may include numerals, screen controls, colour controls etc) to be used as text – up to 255 characters total per string. Identified by a dollar sign after the variable name with the string itself in double quotes (eg A$="PHONE 01-607 9489") or given as a CHR$ code (eg RE$=CHR$(13))

Commands

Commands can be used in direct mode – and normally use. They may also be incorporated into programs in the same way as statements, but they do not operate on information or data within the program – instead they work on the program itself or on some other program.

- **CONT**

Continue executing the program after it has been stopped by the RUN/STOP key or by an END or STOP statement.

- **LIST**

Prints on the screen the program currently in memory. Options:

LIST will print the entire program

LIST *line number* lists the single line numbered indicated

LIST *line number* – lists from the line number to the end of the — —program

LIST – *line number* lists from the start to the line number

LIST *line number* – *line number* lists lines between the two numbers

- **LOAD**

Load program. Options for loading from tape:

LOAD loads the first program to be found on cassette

LOAD '*program name*' loads from cassette the first program found — —that matches the full or partial name given

LOAD '*character(s)* *' loads from cassette the first program found — —whose name starts with the characters given and includes any other characters ('*' is the 'any-characters' wild-card)

LOAD '*character(s)* ?' loads from cassette the first program found — —whose name starts with the characters given and includes any other character in the position(s) denoted by the question mark(s) ('?' is the 'single-character' wild-card)

Options for loading from disk (the command must always have a program name in quotes followed by a comma and the disk device number, normally '8'):

LOAD '$',8 loads the disk directory

LOAD '*', 8 loads the first program to be found on the disk

LOAD '*program name*',8 loads from disk the first program found that — —matches the full or partial name given

LOAD '*character(s)* *',8 loads from disk the first program found whose — —name starts with the characters given and includes any other characters ('*' is the 'any-characters' wild-card)

LOAD '*character(s)* ?',8 loads from disk the first program found whose — —name starts with the characters given and includes any other character in the position(s) denoted by the question mark(s) ('?' is the 'single-character' wild-card)

- **NEW**

Erases a program from memory and clears out all variables it set up (if NEW is not used, a subsequently-entered program may have mixed in with it some line numbers and variables from the previous one.

- **RUN**

Starts the program running. Options:

RUN starts from the beginning

RUN *line number* starts from the line number given

- **SAVE**

Saves the program currently in memory. Options:

SAVE stores program on cassette without a name

SAVE '*program name*' stores program on cassette with the name given

SAVE '*program name*',8 stores program on disk with the name given

SAVE '@:*program name*',8 stores program on disk with the name given, — —replacing any currently-stored program of the same name

- **VERIFY**

Checks that a previously SAVEd program matches the one currently in memory. Options:

VERIFY will check the next program it finds on a — —cassette

VERIFY '*program name*' will look for and then check the program — —specified on cassette

VERIFY '*program name*',8 will look for and then check the program — —specified on disk

Statements

Statements perform some operation within a program, in which case they will appear somewhere after a line number. Most can also be used in direct mode without a line number.

- **CLR**

Clears current values of any variables, including array variables (but does not have the same effect as the CLR/HOME key, which merely clears the screen). RUN, LOAD and NEW have the same effect but all do other things as well.

- **CLOSE**

Closes an output channel that has been opened by the OPEN command – takes the form ...

CLOSE *channel number*

- **CMD**

Send output to the device number specified instead of to the screen. The OPEN command must have been used previously. Takes the form ...

CMD *device number*

- **DATA**

Indicates that following information comprises DATA items to be read by the READ command. Items are separated by commas and are stored in memory in the same order to be entered into the program in that sequence as required by one or more READ statements.

- **DEF FN**

Enables the programmer to define a numeric function (usually a formula that you don't want to have to repeat all the time). Takes the form ...

DEF FN *variable name* (*value*) = *rule*

... where the identifying variable name can be of one or two alphabetic characters (so you can DEFine up to 676 functions with different letter combinations!) and *value* is a numeric variable name that may be assigned in the *rule*. For instance:

```
10 DEF FN SQUARE (S) = S*S
20 PRINT FN SQUARE (3)
RUN
 9

READY
/text
```

- **DIM**

Defines the DIMension(s) of an array or matrix. Options:

DIM *array name* (*size*) sets up a single-dimensional array of the size — __specified

DIM *array name* (*rows, columns*) sets up a two-dimensional array — __of the size specified

DIM *array name* (*size, size, size ...*) sets up a multi-dimensional — __array of the size specified

The array name can be any valid one- or two-character variable name in the string, integer or numeric formats. Array size is always one greater than the number(s) specified since position 0 is included.

Several arrays can be set up with one DIM statement at the same time by separating the array name and dimensions by commas.

- **END**

Stops a program. Not really essential, since when the program runs out of lines to execute it will terminate anyhow; but it is good practice to use END, and it is in any case essential when subroutines are placed at the end of the main body of a program (without END the program would continue executing, run on into the subroutine code, and eventually crash with a ?RETURN WITHOUT GOSUB error).

- **FOR ... TO**

Sets up a loop, a section of program that repeats for however many times this statement specifies. Takes the form ...

FOR *variable name* = *start value* **TO** *end value*

A STEP can also be specified, as in FOR A=250 TO 0 STEP-5 (note that loops can go "backwards' too). If the command includes no STEP the increment is assumed to be +1.

A NEXT statement causes the loop to repeat.

- **GET**

Scans the keyboard continuously and inputs one character at a time from the keyboard without having to wait for a Return (which itself can be the keyboard input). Takes the form ...

GET *variable name*

... and stores the keyboard input as the variable specified. If no key is being pressed, a "no-key' value is stored as the variable.

- **GET#**

Scans the device number indicated and inputs one character at a time from it. The device number must have been OPENed previously. Takes the form ...

GET# *variable name*

... and stores the input as the variable specified.

- **GOSUB**

Switches program execution to a subroutine starting at the line number given. Takes the form ...

GOSUB *line number*

To get back to the starting position the subroutine must include a RETURN

- **GOTO**

Switches program execution unconditionally to the line number given. Takes the form ...

GOTO *line number*

- **IF ... THEN**

Switches program execution to the line number given if the expression is evaluated as "true'; otherwise the next line in sequence is executed. This allows a program to take different courses of action depending on circumstances. Takes the form ...

IF *expression* **THEN** *line number*

- **INPUT**

Accepts text typed in from the keyboard and stores it as a named variable. Options:

INPUT *variable name* prompts only with question mark

INPUT *"string"*; *variable name* prompts with the string followed by the question mark

Variable names and the acceptable assignment of the input follow the usual rules, except that input for a character variable need not be enclosed in quotes.

- **INPUT#**

Inputs one block of text at a time from the device number specified (it must have been OPENed previously) and stores it as the variable name given. Takes the form ...

INPUT# *device number, variable name*

- **LET**

Sets a named variable to the value of the expression given. In fact the word "LET" is usually omitted altogether; Basic assumes that when it sees one or two characters with an equals sign that it's a LET statement. Options:

LET *variable name = expression*

variable name = expression

The expression can be a constant (a number, in which case the variable would be unqualified for real numbers or given a percent symbol for integers): a string (characters in quotes or CHR$ codes, with the variable name given the dollar suffix): or a compound expression, such as a concatenation or a formula, that itself may involve other variables.

- **NEXT**

Repeats a FOR ... TO loop. Options:

NEXT repeats the last FOR loop to have been started

NEXT *variable name* repeats the FOR loop that uses the variable name specified

- **ON ... GOSUB**

A more flexible alternative to the IF/THEN construction for conditional branches. Takes the form ...

ON *variable or expression* **GOSUB** *line number(s)*

If the variable or expression has an integer value of 1 when it's worked out, the program will go to the subroutine starting at the first line number given here. If it's 2, the jump will be to a subroutine at the second line number specified – and so on.

- **ON ... GOTO**

A more flexible alternative to the IF/THEN construction for conditional branches. Takes the form ...

ON *variable or expression* **GOTO** *line number(s)*

If the variable or expression has an integer value of 1 when it's worked out, the program will jump to the first line number given here. If it's 2, the jump will be to the second line number specified – and so on.

- **OPEN**

Sets up an input/output link to an externally connected device (disk or printer, normally). Subsequently PRINT#, INPUT# and GET# commands will refer to this link.

OPEN always includes a logical file number and a device number: a secondary address may also be specified for some types of communication with disk or cassette, and for those devices a command string can also be included for some operations (rather than separately in a PRINT# statement).

Options:

OPEN *file number*, **4** opens a link to the printer (device number 4)

OPEN *file number*, **6** opens a link to the printer (device number 6)

OPEN *file number*, **1, 1,** *"file name"* opens a link to the cassette — —(device number 1) for writing only

OPEN *file number*, **1, 0,** *"file name"* opens a link to the cassette — —for reading only

OPEN *file number*, **8,** *channel "file name,***S,W***"* opens a link to — —the disk (device number 8) for writing only to the Sequential file specified

OPEN *file number*, **8,** *channel "file name,***S,W***"* opens a link to — —the disk for reading only on the file

OPEN *file number*, **8, 15** opens a link to the disk for using PRINT# — —subsequently with disk commands

OPEN *file number*, **8, 15,** *"disk commands"* opens a link to the for — —disk commands to be used directly (without a separate PRINT#)

- **POKE**

Places a value, a compounded expression, or the contents of a variable into a specified memory location. Takes the form ...

POKE *memory location, expression*

- **PRINT**

Displays information and/or controls display options for screen output. Options:

PRINT *"string"*
PRINT *"value"*
PRINT *"variable name"*

... or a combination of all of these.

- **PRINT#**

Directs output to a specified device number which has previously been OPENed. This can be a printer, but it could equally be a non-printing device like a disk. Options:

PRINT# *file number "string"*
PRINT# *file number "value"*
PRINT# *file number "variable"*
PRINT# *file number "disk command"*

...Or a combination of these. Disk commands take the form ...

"command disk number: name"

The command can be abbreviated to a single letter, and the disk number can usually be omitted (since "0' is assumed on single-disk set-ups: for twin-disk 64s, one disk would be number 0 and the other number 1).

- **READ**

Fills a list of specified variables by taking items in sequence from the next DATA statement line that is encountered in the program. Takes the form ...

READ *variable, variable, ...*

- **REM**

Text following is a comment and Basic will ignore everything following the REM statement, so you can use it to annotate the program as you wish for your own benefit.

- **RESTORE**

When a list of items from a DATA statement is READ, a pointer marks the next one in sequence. This commands resets the pointer back to the start of the list, so the same DATA can be used again by a subsequent READ.

- **RETURN**

Included in a subroutine to return to the line following the GOSUB.

- **STOP**

A debugging aid, generally used as a temporary measure during program development, that stops execution of a program when it is encountered. As such it operates just like the END statement – except that you're told the line number at which your STOP was executed: and you can restart from that point by using CONT.

- **WAIT**

Causes a program to pause until the contents of a specified memory location fit the conditions implied in the statement. Options:

WAIT *memory location number* pauses the program until the contents of the memory location have a 1 in one or more of the same positions that have a in the binary representation of the number – in other words, the program WAITs until *PEEK (location) AND number* is not zero

WAIT *memory location number*, number pauses the program until the contents of the memory location have a 0 in one or more of the same positions that have a 1 in the binary representation of both numbers: or until it has a 1 in positions where the first number has a 1 and the second has a 0

String variable functions

- **ASC('string')**

Gives the ASCII value of the first character in the string.

- **CHR$(*ASCII code*)**

Gives the character for that ASCII code.

- **LEFT$('string', *number of characters*)**

Gives the specified number of characters starting from the leftmost position in the string.

- **RIGHT$('string', *number of characters*)**

Gives the specified number of characters starting from the rightmost position in the string.

- **MID$('string', *start position, number of characters*)**

Gives the required number of characters starting at the specified position within the string

- **STR$(*number*)**

Creates a string of characters containing the number.

- **!VAL('string')**

Turns the string into a number. If the first character of the string is not a sign, a dollar, or a digit, VAL will return the value '0'.

- **LEN('string')**

Gives the length of the string – spaces and other non-printing characters are counted.

Arithmetic functions

- **ABS(*number*)**

Gives the absolute value of the number (the value without its sign)

- **FN** *variable name* (*value*)

Specifies a user-defined function set up with a DEF FN statement

- **EXP(*number*)**

Gives E to the power of the number (which must be less than 88.02969192)

- **INT(*number*)**

Gives the integer portion of the number

- **LOG(*number*)**

Gives the logarithm of the number to the base E. For log base 10, use LOG(n)/LOG(10)

- **RND(*number*)**

Gives a (nearly) random number. RND(0) produces the same random numbers for each sequence; using the current contents of the TI clock – RND(-TI) – is a reasonable way to start a sequence

- **SGN(*number*)**

Gives the sign of the number as '+1' if positive, '0' if zero, '-1' if negative

- **SQR(*number*)**

Gives the square root of the number (which cannot be negative)

Trigonometric functions

- **ATC(*number*)**

Arctangent – gives the angle (in radians) whose tangent is the number specified

- **COS(*angle*)**

Gives the cosine of the angle specified in radians

- **SIN(*angle*)**

Gives the sine of the angle specified in radians

- **TAN(*angle*)**

Gives the tangent of the angle specified in radians

PRINT and display formatting functions

- **POS(0)**

Gives the current position of the cursor in terms of character position across the screen. The (0) is a dummy argument.

- **SPC(*number*)**

Used with a PRINT command to put the specified number of spaces into a line.

- **TAB(*number*)**

Used with a PRINT command to tab to the character position indicated by the number.

System variables

There are three key system variables that may be looked at – but which should be changed only with considerable care.

- **TI** is the system clock. It counts time in 'jiffies', a jiffy being 1/60th of a second

- **TI$** expresses the TI clock in hours, minutes, and seconds as a string variable

- **ST** is the status variable for the last I/O operation. Its meaning is as follows:

Bit no.	Value	Cassette	Serial devices
0	1	none	write time-out
1	2	none	read time-out
2	4	short block	none
3	8	long block	none
4	16	read error	none
5	32	checksum error	none
6	64	end of file	end of input
7	-128	end of tape	device not present

Other functions and commands

- **FRE(0)**

In theory, indicates how much memory is left to you for Basic programs; it is normally used as PRINT FRE(0), the (0) being a dummy argument. In practice this may return a negative number because of a silly bug in the 64 (this happens when the number of available bytes is greater than 32767). This formula will always produce the correct result:

```
FR = FRE(0) - (FRE(0) < 0) * 65535
```

A simpler version works only for memory availability greater than 32767:

```
FR = 65535 + FRE(0)
```

- **PEEK**

Specifies the memory location indicated as ...

PEEK(*memory location*)

The memory location number must be in the range 0 to 65535)

- **SYS**

This command is used in the form ...

SYS *memory location*

It switches to the memory location specified and executes a machine-language subroutine there. That may include the machine-language RTS, equivalent to RETURN in Basic, which gets you back into the Basic program.

- **USR**

This function is used in the form ...

USR (*memory location*)

It switches to a user-written machine-language subroutine located at the memory location specified and executes it.

Ready-reference

This summary isn't laid out alphabetically, so here's a quick-reference index for specific Basic statements and commands:

Basic word	*Abbreviation*	*Type*
ABS	A <Shifted> B	function
ASC	A <Shifted> S	function
ATN	A <Shifted> T	function
CHR$	C <Shifted> H	function
CLOSE	CL <Shifted> O	command
CMD	C <Shifted> M	command
CONT	C <Shifted> O	command
COS	none	function
DATA	D <Shifted> A	statement
DEF	D <Shifted> E	statement
DIM	D <Shifted> I	statement
END	E <Shifted> N	statement
EXP	E <Shifted> X	function
FN	none	function
FOR	F <Shifted> O	statement
FRE	F <Shifted> R	function
GET	G <Shifted> E	statement
GET#	G <Shifted> E#	statement
GOSUB	GO <Shifted> S	statement
GOTO	G <Shifted> O	statement
IF	none	statement
INPUT	I <Shifted> N	
	(delete)	statement
INPUT#	I <Shifted> N	statement
INT	none	function
LEFT$	LE <Shifted> F	statement
LEN	none	function
LET	L <Shifted> E	statement
LIST	L <Shifted> I	command
LOAD	L <Shifted> O	command
LOG	none	function
MID$	M <Shifted> I	function
NEW	none	command
NEXT	N <Shifted> E	statement
ON	none	statement
OPEN	O <Shifted> P	statement
PEEK	P <Shifted> E	function
POKE	P <Shifted> O	statement
POS	none	function
PRINT	?	statement
PRINT#	P <Shifted> R	statement
READ	R <Shifted> E	statement
REM	none	statement
RESTORE	RE <Shifted> S	statement
RETURN	RE <Shifted> T	statement
RIGHT$	R <Shifted> I	function
RND	R <Shifted> N	function
RUN	R <Shifted> U	command
SAVE	S <Shifted> A	command
SGN	S <Shifted> G	function
SIN	S <Shifted> I	function
SPC	S <Shifted> P	function
SQR	S <Shifted> Q	function
ST	none	variable
STOP	A <Shifted> T	command
STR$	ST <Shifted> R	function
SYS	S <Shifted> Y	statement
TAB	T <Shifted> A	function
TAN	none	function
THEN	T <Shifted> H	statement
TI	none	variable
TI$	none	variable
TO	none	statement
USR	U <Shifted> S	function
VAL	V <Shifted> A	function
VERIFY	V <Shifted> E	command
WAIT	W <Shifted> A	statement

APPENDIX 2
The Memory Map

Operating system RAM: locations 0 to 1023

0 'chip directional indicator'

1 memory and tape control. Includes memory configuration,

2 temporary workspace

3 – 4 converts floating-point real numbers to integer

5 – 6 converts integer to floating-point numbers

7 search character (stores ASCII value 34, quote symbol)

8 flag – scan for end-of-string quotes

9 number of screen column from last TAB

10 load/verify flag indicating last tape or disk operation (0=load)

11 number of subscripts

12 flag – use default DIMensions for arrays (10 or 10, 10). Otherwise holds value of first letter of the array most recently DIMensioned

13 type of data being read – 0=numeric, 255=string

14 data type for numerics – 0=floating-point, 128=integer

15 flag – DATA scan for LIST and garbage collection. Normal value 4

16 flag – subscript reference *or* user function (FNx) call

17 flag – Basic input type (0=INPUT, 64=GET, 152=READ)

18 flag – verify TAN sign (tests SIN/COS division results). Normal value 0, goes to 255 when TAN argument is greater than 259.1

19 INPUT query flag (0 for query mark as INPUT prompt, 1 to suppress it)

20 – 21 temporary storage for integer value (used by locations 3-4 and 5-6)

22 pointer to start of temporary string stack (normally points to location 25)

23 – 24 last temporary string stack

25 – 33 stack area for temporary string storage

34 – 37 pointers to utility routines in Basic ROM

38 – 42 workspace for product of multiplication

43 – 44 pointer to start address of Basic program (normally address 2049)

45 – 46 pointer to start of program's variables (normally address 2091)

47 – 48 pointer to start of program's arrays

49 – 50 pointer to end of program's arrays

51 – 52 pointer to end of string storage

53 – 54 utility string pointer

55 – 56 pointer to end address of Basic program space (normally address 40960)

57 – 58 current line number of Basic program

59 – 60 previous line number of Basic program

61 – 62 pointer to line number for CONT

63 – 64 current DATA line number

65 – 66 pointer to current DATA address

67 – 68 pointer to temporary data storage for INPUT within input buffer

69 – 70 name of current variable

71 – 72 address of current variable

73 – 74 pointer for FOR/NEXT variable

75 – 76 temporary workspace for Basic pointers *or* saving Y register *or* op-code

77 comparison symbol accumulator

78 – 83 miscellaneous workspace area and Basic pointers

84 – 86 jump area for functions

87 – 96 miscellaneous work area for numerics

97 value in accumulator 1 – exponent

98 – 101 value in accumulator 1 – mantissa

102 value in accumulator 1 – sign

103 pointer for series evaluation constant

104 value in accumulator 1 – overflow digit

105 value in accumulator 2 – exponent

106 – 109 value in accumulator 2 – mantissa

110 value in accumulator 2 – sign

111 result of sign comparison between contents of accumulators 1 and 2

112 accumulator 1 rounding (low-order)

113 – 114 length of contents of cassette buffer

115 – 138 get Basic character routine (CHRGET). Includes **121** for CHRGOT (get same byte again) and **122 – 123** for pointer to current byte of Basic text

139 – 143 seed value for RND

144 ST status word (normally contains 0 for normal status)

145 flag – STOP and RVS keys. Normal value 255

146 tape timing constant

147 indicator for last disk or tape operation – 0=load, 1=verify ???

148 flag – serial bus *or* output character buffer

149 buffered character for serial bus

150 end-of-tape character (EOT) received *or* cassette synchronisation number

151 temporary data area (register save)

152 number of OPENed files

153 device number of input device (normally 0, for keyboard)

154 device number of output device (normally 3, for screen)

155 parity indicator for character read from or written to tape

156 flag set to indicate byte received from tape

157 flag – Direct or Program mode (0=Program mode)

158 tape pass 1 – error log

159 tape pass 2 – error log

160 – 162 jiffy clock

163 bit count for serial transfers *and* end-of-input (EOI) flag

164 count of cycles

165 tape synchronisation countdown

166 pointer to tape I/O buffer

167 temporary storage for tape leader on writing *or* tape read pass *or* RS232 input bits

168 temporary storage for new byte on tape writing *or* tape read error *or* count for number of RS232 input bits

169 temporary storage for start bit on tape writing *or* first bit of tape read error *or* check for start bit of RS232 input

170 temporary storage for tape scan, count, start and end; *or* byte assembly buffer for RS232 input bits

171 temporary storage for tape leader length on writing *or* tape read checksum *or* parity status for RS232 input

172 – 173 pointer to tape buffer *or* screen scrolling

174 – 175 I/O pointer for tape end address *or* end-of-program address

176 – 177 tape timing constants

178 – 179 pointer to start address of tape buffer

180 set to 1 for tape timer enabled; *or* bit count for RS232 output

181 flag – end of tape (EOT); *or* RS232 next bit to send

182 error in character read *or* RS232 output byte buffer

183 number of characters in file name

185 current secondary address

186 current device number

187 – 188 pointer to file name

189 temporary storage for tape read/write word *or* parity status for RS232 output

190 number of blocks remaining to read or write

191 serial word buffer

192 tape motor interlock

193 – 194 I/O start address

195 – 196 Kernal setup pointer – used for temporary input storage on cassette reads

197 last key pressed (normally 64, 'no key')

198 number of characters in keyboard buffer

199 flag for reversed-characters display

200 end-of-line flag for INPUT (screen column number of end of INPUT record)

201 cursor position for start of INPUT – row

202 cursor position for start of INPUT – column

203 flag – printed Shifted characters. Normally 64

204 cursor blink (normally 0 for on, set to 1 for off)

205 cursor blink timing (normally 2)

206 ASCII value character currently under cursor

207 flag – last cursor blink. Normally 0

208 flag – INPUT or GET from keyboard. Normally 0

209 – 210 pointer to current column on screen line

211 cursor position along current screen line in terms of number of character positions along the line

212 flag – cursor in quotes mode (ie programmed). Normally 0 for Direct-mode cursor

213 length of current screen line. Normal default value is 39

214 current cursor position on screen in terms of number of lines from the top of the screen

215 temporary workspace for checksum

216 number of INSerts yet to be made

217 – 242 screen line link table *or* temporary edit buffer in Direct mode

243 – 244 pointer to current screen colour

245 – 246 pointer to keyboard decode tables

247 – 248 pointer to RS232 receive buffer

249 – 250 pointer to RS232 transmit buffer

251 – 255 [unused]

256 – 266 Basic temporary data storage area, primarily for strings

267 – 318 tape input error log

319 – 511 processor stack working area

512 – 600 input buffer for Basic

601 – 610 table of logical files active

611 – 620 table of device numbers for each logical file active

621 – 630 table of secondary addresses for each logical file active

631 – 640 keyboard buffer queue

641 – 642 pointer to start (bottom) of Basic memory. Normally 2048

643 – 644 pointer to end (top) of Basic memory. Normally 40960

645 serial bus timeout flag

646 current character colour code

647 background colour currently under cursor

648 bottom of screen memory page (equivalent to bottom left-hand corner of the screen display – multiply by 256 to get the decimal value of the address. Normally holds 4, since the screen memory generally begins at location 1024)

649 maximum number of characters that can be held in keyboard buffer (normally 10)

650 keyboard repeat flag (normally 0 for repeat only on INST/DEL, the two cursor movement keys, and space bar)

651 keyboard repeat speed counter. Normal value 4

652 keyboard repeat delay counter

653 flag for Shift, CTRL and CBM keys – set appropriately if any combination is currently held down

654 last keyboard Shift pattern

655 – 656 pointer to keyboard table setup. Normally 60232

657 flag – shows Shift key status (0=disabled, 128=enabled)

658 flag – automatic scrolling (0=on)

659 RS232 control register (set by first CHR$ with RS232 OPEN command)

660 RS232 command register (set by second CHR$ with RS232 OPEN command)

661 – 662 non-standard RS232 baud rate (set by third and fourth CHR$ codes with RS232 OPEN command)

663 status of RS232 transfer

664 number of bits still to send over RS232 link

665 – 666 code for RS232 baud rate

667 index pointer to end of RS232 receive pointer

668 pointer to start of RS232 receive buffer

669 pointer to start of RS232 transmit buffer

670 index pointer to end of RS232 transmit pointer

671 – 672 temporary save area for IRQ vector values during tape I/O

673 CIA 2 interrupt control: RS232 enable

674 CIA 1 timer A control log: TOD sense during tape I/O

675 CIA 1 interrupt log *or* temporary data storage during tape I/O

676 CIA 1 timer A enabled flag *or* temporary tape read IRQ indicator

677 temporary storage for line index (screen row marker)

678 flag – PAL or NTSC video signals (0=NTSC, 1=PAL)

679 – 767 unused, though **704 – 766** normally will hold data for sprite no. 1

768 – 769 vector to Basic – print error message

770 – 771 vector to Basic – 'warm' start

772 – 773 vector to Basic – tokenise Basic text

774 – 775 vector to Basic – LIST Basic text

776 – 777 vector to Basic – despatch character and start new Basic text

778 – 779 vector to Basic – evaluate Basic token

780 A register save prior to SYS command

781 X register save prior to SYS command

782 Y register save prior to SYS command

783 SP register (stack pointer) save prior to SYS command

784 op-code for JMP. Normal value therefore 76

785 – 786 low and high bytes for USR(x) start address

787 [unused]

788 – 789 vector to hardware interrupt routine (IRQ). Normally 59953

790 – 791 vector to BRK interrupt routine

792 – 793 vector to NMI (non-maskable interrupt) routine. Normally 65095

794 – 795 vector to Kernal OPEN routine

796 – 797 vector to Kernal CLOSE routine

798 – 799 vector to Kernal CHKIN set-input routine

800 – 801 vector to Kernal CHKOUT set-output routine

802 – 803 vector to Kernal CLRCHN restore I/O routine

804 – 805 vector to Kernal CHRIN input routine

806 – 807 vector to Kernal CHROUT output routine

808 – 809 vector to Kernal STOP test RUN/STOP routine

810 – 811 vector to Kernal GETIN get input routine

812 – 813 vector to Kernal CLALL abort I/O routine

814 – 815 vector to user-defined destination

816 – 817 vector to Kernal LOAD routine

818 – 819 vector to Kernal SAVE routine

820 – 827 [unused]

828 – 1019 cassette I/O buffer

832 – 894 (alternatively) data for sprite no. 13

896 – 958 (alternatively) data for sprite no. 14

960 – 1022 (alternatively) data for sprite no. 15

1020 – 1023 [otherwise unused]

Screen RAM: locations 1024 to 2039

1024 normal start of screen memory (equivalent to top left corner of display)

2023 normal end of screen memory (equivalent to top left corner of display)

2024 – 2039 [unused]

sprite pointers RAM: locations 2040 to 2047

2040 pointer to block containing data for sprite no. 0

2041 pointer to block containing data for sprite no. 1

2042 pointer to block containing data for sprite no. 2

2043 pointer to block containing data for sprite no. 3

2044 pointer to block containing data for sprite no. 4

2045 pointer to block containing data for sprite no. 5

2046 pointer to block containing data for sprite no. 6

2047 pointer to block containing data for sprite no. 7

Basic program RAM: locations 2048 to 40959

2048 normal start of Basic program space

40959 normal end of Basic program space

32768 – 40959 (alternatively) ROM space for plug-in cartridges

Basic interpreter ROM: locations 40960 to 49151

40960 ROM control vectors

40972 keyword action vectors

41042 function vectors

41088 operator vectors

41118 keywords

41374 error messages

41768 error message vectors

41829 miscellaneous messages

41866 scan stack for FOR or GOSUB

41912 move memory

41979 check depth of stack

41992 check memory space

42037 OUT OF MEMORY error

42039 error routine

42144 BREAK message and routine

42260 READY message and routine

42112 ready for Basic

42140 handle new line

42291 re-chain lines

42336 receive input line

42361 crunch Basic tokens

42515 find Basic line

42562 NEW routine

42590 CLR routine

42638 backup text pointer

42652 LIST routine

42818 FOR routine

42988 execute statement

43037 RESTORE routine

43052 break

43055 STOP routine

43057 END routine

43095 CONT routine

43121 RUN routine

43139 GOSUB routine

43168 GOTO routine

43218 RETURN routine

43256 DATA routine

43270 scan for next statement

43304 IF routine

43323 REM routine

43339 ON routine

43371 get fixed-point number

43685 LET routine

43648 PRINT# routine

43654 CMD routine

43680 PRINT routine

43806 print string

43835 print format character

43853 bad input routine

43899 GET routine

43941 INPUT# routine

44025 prompt and input

44038 READ routine

44284 messages for input errors

44318 NEXT routine

44408 check for type match

44446 evaluate expression

44712 evaluate pi

44785 evaluate within parentheses

44791 closing parentheses

44799 evaluate comma

44808 SYNTAX ERROR

44820 range check

44840 search for variable

44967 create FN reference

45030 OR routine

45033 AND routine

45078 compare

45185 DIM routine

45195 locate variable

45331 check alphabetic

45341 create variable

45460 array pointer routine

45477 value 32768

45490 convert floating- to fixed-point

45521 set up array

45637 BAD SUBSCRIPT

45640 ILLEGAL QUANTITY

45900 compute array size

45949 FRE routine

45969 convert fixed- to floating-point

45982 POS routine

45990 check for Direct mode

46003 DEF routine

46049 check syntax of FN

46068 FN routine

46181 STR$ routine

46197 calculate string vector

46215 set up string

46324 make room for string

46374 garbage collection routine

46525 check salvageability

46598 collect string

46653 concatenate strings

46714 build string in memory

46755 discard unwanted string

46811 clean up description stack

46828 CHR$ routine

46848 LEFT$ routine

46892 RIGHT$ routine

46903 MID$ routine

46945 pull string parameters

46972 LEN routine

46978 exit string-handling

46987 ASC routine

47003 input byte parameter

47021 VAL routine

47083 parameters for POKE and WAIT

47095 floating-fixed

47117 PEEK routine

47140 POKE routine

47149 WAIT routine

47177 add 0.5

47184 value to subtract from

47187 subtraction routine

47210 addition routine

47431 complement FAC no. 1

47486 overflow detect

47491 multiply by zero byte

47594 LOG routine

47659 multiplication routine

47705 multiply one bit

47756 move from memory to FAC no. 2

47799 adjust FAC no.1 and 2

47828 check under/overflow

47842 multiply by 10

47865 add 10 in floating-point

47870 divide by 10

47890 division routine

48034 move from memory to FAC no. 1

48071 move from FAC no. 1 to memory

48124 move from FAC no. 2 to FAC no. 1

48140 move from FAC no. 1 to FAC no. 2

48155 round FAC no. 1

48171 get sign

48185 SGN routine

48216 ABS routine

48219 compare FAC no. 1 to memory

48283 floating-fixed

48332 INT routine

48371 move string to FAC

48510 get ASCII character

48589 print line number

48605 float to ASCII

49117 decimal constants

48954 TI constants

49019 powers routine

49076 negative routine

49133 EXP routine

49151 end of Basic interpreter

4K free RAM: locations 49152 to 43247

Video chip (6566 VIC II)
ROM: locations 53248 to 53295

53248 sprite no. 0 horizontal location (low-order byte)

53249 sprite no. 0 vertical location

53250 sprite no. 1 horizontal location (low-order byte)

53251 sprite no. 1 vertical location

53252 sprite no. 2 horizontal location (low-order byte)

53253 sprite no. 2 vertical location

53254 sprite no. 3 horizontal location (low-order byte)

53255 sprite no. 3 vertical location

53256 sprite no. 4 horizontal location (low-order byte)

53257 sprite no. 4 vertical location

53258 sprite no. 5 horizontal location (low-order byte)

53259 sprite no. 5 vertical location

53260 sprite no. 6 horizontal location (low-order byte)

53261 sprite no. 6 vertical location

53262 sprite no. 7 horizontal location (low-order byte)

53263 sprite no. 7 vertical location

53264 *bit 0*: sprite 0 horizontal location, high-order bit
bit 1: sprite 1 horizontal location, high-order bit
bit 2: sprite 2 horizontal location, high-order bit
bit 3: sprite 3 horizontal location, high-order bit
bit 4: sprite 4 horizontal location, high-order bit
bit 5: sprite 5 horizontal location, high-order bit
bit 6: sprite 6 horizontal location, high-order bit
bit 7: sprite 7 horizontal location, high-order bit

53265 *bits 0-2*: vertical scroll over a display (between 1 and 8 pixels per movement)
bit 3: number of rows per display (0 = 24, 1 = 25)
bit 4: blank/unblank screen (set to 1 if you want the display area the same colour as the border)
bit 5: select bit-map mode for work on a grid 320 pixels wide by 200 high
bit 6: select extended background colour mode (each character takes on a background colour in addition to the normal screen background)
bit 7: register compare for raster position – used with 53266

53266 raster position

53267 light-pen horizontal position

53268 light-pen vertical position

53269 *bit 0*: turn sprite 0 on/off
bit 1: turn sprite 1 on/off
bit 2: turn sprite 2 on/off
bit 3: turn sprite 3 on/off
bit 4: turn sprite 4 on/off
bit 5: turn sprite 5 on/off
bit 6: turn sprite 6 on/off
bit 7: turn sprite 7 on/off

53270 *bits 0-2*: horizontal scroll on/off
bit 3: number of columns (0 = 38, 1 = 40)
bit 4: multi-colour mode
bits 5-7: [unused]

53271 *bit 0*: sprite 0 double-height mode
bit 1: sprite double-height mode
bit 2: sprite 2 double-height mode
bit 3: sprite 3 double-height mode
bit 4: sprite 4 double-height mode
bit 5: sprite 5 double-height mode
bit 6: sprite 6 double-height mode
bit 7: sprite 7 double-height mode

53272 bits 0-2: character set pointer (upper-case or upper-and-lower-case)
bit 3: bit-map pointer
bits 4-7: pointer to character screen map (normally 16 for 1024) or to colour map

53273 interrupt flag register

53274 IRQ mask register

53275 *bit 0*: sprite 0 priority over background
bit 1: sprite 1 priority over background
bit 2: sprite 2 priority over background
bit 3: sprite 3 priority over background
bit 4: sprite 4 priority over background
bit 5: sprite 5 priority over background
bit 6: sprite 6 priority over background
bit 7: sprite 7 priority over background

53276 *bit 0*: sprite 0 multi-colour mode
bit 1: sprite 1 multi-colour mode
bit 2: sprite 2 multi-colour mode
bit 3: sprite 3 multi-colour mode
bit 4: sprite 4 multi-colour mode
bit 5: sprite 5 multi-colour mode
bit 6: sprite 6 multi-colour mode
bit 7: sprite 7 multi-colour mode

53277 *bit 0*: sprite 0 double-width mode
bit 1: sprite 1 double-width mode
bit 2: sprite 2 double-width mode
bit 3: sprite 3 double-width mode
bit 4: sprite 4 double-width mode
bit 5: sprite 5 double-width mode
bit 6: sprite 6 double-width mode
bit 7: sprite 7 double-width mode

53278 *bit 0*: sprite 0 – sprite-vs-sprite collision detect
bit 1: sprite 1 – sprite-vs-sprite collision detect
bit 2: sprite 2 – sprite-vs-sprite collision detect
bit 3: sprite 3 – sprite-vs-sprite collision detect
bit 4: sprite 4 – sprite-vs-sprite collision detect
bit 5: sprite 5 – sprite-vs-sprite collision detect
bit 6: sprite 6 – sprite-vs-sprite collision detect
bit 7: sprite 7 – sprite-vs-sprite collision detect

53279 *bit 0*: sprite 0 – sprite-vs-character collision detect
bit 1: sprite 1 – sprite-vs-character collision detect
bit 2: sprite 2 – sprite-vs-character collision detect
bit 3: sprite 3 – sprite-vs-character collision detect
bit 4: sprite 4 – sprite-vs-character collision detect
bit 5: sprite 5 – sprite-vs-character collision detect
bit 6: sprite 6 – sprite-vs-character collision detect
bit 7: sprite 7 – sprite-vs-character collision detect

53280 border colour

53281 screen background colour 0

53282 screen background colour 1

53283 screen background colour 2

53284 screen background colour 3

53285 sprite multi-colour 0

53286 sprite multi-colour 1

53287 sprite 0 colour

53288 sprite 1 colour

53289 sprite 2 colour

53290 sprite 3 colour

53291 sprite 4 colour

53292 sprite 5 colour

53293 sprite 6 colour

53294 sprite 7 colour

53295 – 54271 VIC images

Sound chip (6581 SID) ROM: locations 54272 to 54300

54272 Voice 1 low frequency

54273 Voice 1 high frequency

54274 Voice 1 low pulse width

54275 Voice 1 high frequency

54276 Voice 1 waveform type

54277 *bits 0-3*: Voice 1 decay
bits 4-7: Voice 1 attack

54278 *bits 0-3*: Voice 1 release
bits 4-7: Voice 1 sustain

54279 Voice 2 low frequency

54280 Voice 2 high frequency

54281 Voice 2 low pulse width

54282 Voice 2 high frequency

54283 Voice 2 waveform type

54284 *bits 0-3*: Voice 2 decay
bits 4-7: Voice 2 attack

54285 *bits 0-3*: Voice 2 release
bits 4-7: Voice 2 sustain

54286 Voice 3 low frequency

54287 Voice 3 high frequency

54288 Voice 3 low pulse width

54289 Voice 3 high frequency

54290 Voice 3 waveform type

54291 *bits 0-3*: Voice 3 decay
bits 4-7: Voice 3 attack

54292 *bits 0-3*: Voice 3 release
bits 4-7: Voice 3 sustain

54293 filter low frequency

54294 filter high frequency

54295 *bit 0*: Voice 1 filter
bit 1: Voice 2 filter
bit 2: Voice 3 filter
bit 3: external filter
bits 4-7: resonance

54296 *bits 0-3*: master volume
bits 4-7: passband

54297 A-D converter 1 for game paddle or analogue joystick

54298 A-D converter 2 for game paddle or analogue joystick

54299 voice 3 random number generator (used when voice 3 is set to the 'noise' waveform)

54300 voice 3 envelope generator output

54301 – 54527 SID images

Unused reserved memory: locations 54528-55295

Commodore describes this as 'reserved for future I/O expansion' whatever that means.

Screen colour RAM: locations 55296-56295

55296 start of screen colour map (equivalent to top left position on screen)

56319 end of screen colour map (equivalent to bottom right of screen)

56296 – 56319 [unused]

CIA 1 (6526 IRQ) ROM: locations 56320-56335

56320 *bit 0*: joystick 0 up
bit 1: joystick 0 down
bit 2: joystick 0 left
bit 3: joystick 0 right
bit 4: joystick 0 fire

56321 *bit 0*: joystick 1 up
bit 1: joystick 1 down
bit 2: joystick 1 left
bit 3: joystick 1 right
bit 4: joystick 1 fire

56322 Data Direction Register, port A

56323 Data Direction Register, port B

56324 timer A, low byte

56325 timer A, high byte

56326 timer B, low byte

56327 timer B, high byte

56328 clock – tenths of seconds

56329 clock – seconds

56330 clock – minutes

56331 *bits 0-6*: clock – hours
bit 7: AM/PM indicator

56332 serial port I/O buffer

56333 CIA interrupt control

56334 *bit 0*: start/stop timer A
bits 1-6: timer A control
bit 7: clock control

56335 *bit 0*: start/stop timer B
bits 1-6: timer B control
bit 7: set alarm or clock

56336 – 56575 CIA 1 images

CIA 2 (6526 NMI) ROM: locations 56576-56591

56576 *bits 0-1*: video bank select
bit 2: RS232 output on user port
bits 3-5: IEEE output on serial port
bits 6-7: IEEE input

56577 user or RS232 port input/output

56378 Data Direction Register, port A

56379 Data Direction Register, port B

56380 timer A, low byte

56381 timer A, high byte

56382 timer B, low byte

56383 timer B, high byte

56384 clock – tenths of seconds

56385 clock – seconds

56386 clock – minutes

56387 *bits 0-6*: clock – hours
bit 7: AM/PM indicator

56388 serial port I/O buffer

56389 CIA interrupt control

56390 *bit 0*: start/stop timer A
bits 1-6: timer A control
bit 7: clock control

56391 *bit 0*: start/stop timer B
bits 1-6: timer B control
bit 7: set alarm or clock

56592 – 56831 CIA 2 images

Unused reserved memory: locations 56832-57343

Two 255-byte blocks again described as 'reserved for future I/O expansion'.

Kernal ROM: locations 57344-65535

The following information isn't a complete summary of the Kernal – but it does indicate the start addresses of the more important ROM routines. It's based on information published by Commodore in 1982, and at least some of the start addresses are reported to have changed for current 64s.

57411 series evaluation 1

57433 series evaluation

57495 RND routine

57593 breakpoints

57642 SYS routine

57686 SAVE routine

57702 VERIFY routine

57704 LOAD routine

57790 OPEN routine

57799 CLOSE routine

57812 parameters for LOAD and SAVE

57862 check default parameters

57870 check for comma

57881 parameters for OPEN and CLOSE

57956 COS routine

57963 SIN routine

58036 TAN routine

58126 ATN routine

58235 warm start routine

58260 initialise

58264 CHRGET for zero page

58303 initialise Basic

58439 vectors for address 768 (print messages)

58451 vectors for initialisation

58463 power-up message

58624 get I/O address

58629 get screen size

58634 get/put screen row/column

58648 initialise I/O

58692 clear screen

58726 cursor to home position

58732 set screen pointers

58784 set I/O defaults

58800 input from keyboard

58930 input from screen

59012 test for quotes

59025 set up screen print

59062 advance cursor

59117 cursor back

59127 cursor back into previous line

59158 output to screen

59516 got to next line

59537 RETURN routine

59553 check line decrement

59571 check line increment

59595 set colour code

59610 colour code table

59626 scroll

59749 open space on screen

59848 move a line on screen

59872 synchronise colour transfer

59888 set start-of-line

59903 clear line of screen

59923 print to screen

59940 synchronise colour pointer

59953 hardware interrupt routine

60039 read keyboard

60281 keyboard select vectors

60289 keyboard 1 – unShifted

60354 keyboard 2 – Shifted

60419 keyboard 3 – CBM key

60484 graphics/text control

60495 set graphics/text mode

60536 keyboard 4

60601 video chip set-up

60647 Shift/RUN equivalent for tape load

60656 screen line address low

60681 send 'talk'

60684 send 'listen'

60736 send to serial bus

60850 serial bus timeout

60857 send 'listen SA'

60862 clear ATN line

60871 send 'talk SA'

60876 wait for clock

60893 send serial deferred character

60911 send 'untalk'

60926 send 'unlisten'

60947 receive from serial bus

61061 serial clock on

61070 serial clock off

61079 serial output '1'

61088 serial output '0'

61097 get serial input *and* clock

61107 1ms delay

61115 RS232 send

61190 send new RS232 byte

61230 DSR-absent error

61233 CTS-absent error

61243 disable timer

61258 compute bit count

61273 RS232 receive

61310 set up to receive

61381 parity error on receive

61386 overflow on receive

61389 break in receive

61392 framing error

61409 submit to RS232

61453 DSR-absent error

61463 send to RS232 buffer

61517 input from RS232

61574 get from RS232

61604 check for serial bus idle

61629 messages

61739 print if direct

61758 get ...

61774 ... from RS232

61783 input

61849 get from tape/serial/RS232

61909 output ...

61917 ... to tape

61966 set input device

62032 set output device

62097 close file

62223 find file

62239 set file values

62255 abort all files

62259 restore default I/O parameters

62282 do 'file open'

62421 send SA

62473 open RS232

62622 load program

62895 SEARCHING message

62913 print filename

62930 LOADING or VERIFYING message

62941 save program

63119 SAVING message

63131 bump clock

63164 log PIA key reading

63197 get time

63204 set time

63213 check stop key

63227 output error messages

63277 find tape header

63338 write tape header

63440 get buffer address

63447 set buffer pointers

63466 find specific header

63501 bump tape pointer

63511 PRESS PLAY ON TAPE message

63534 check tape status

63544 PRESS RECORD AND PLAY message

63553 initiate tape read

63588 initiate tape write

63605 common tape code

63696 check tape stop

63714 set tape read timing

63788 read bits from tape

64096 store tape characters

64398 reset pointers

64407 set up new character

64422 send transition to tape

64456 write data to tape

64461 IRQ entry point

64599 write tape leader

64659 restore normal IRQ

64696 set IRQ vector

64714 stop tape motor

64721 check read/write pointer

64731 bump read/write pointer

64848 initialise system constants

64738 system reset

64770 check 8K ROM

64784 8K ROM mask

64789 Kernal reset

64794 Kernal move

64816 vectors

64923 IRQ vectors

64931 initialise I/O

64989 enable timer

65017 save filename data

65024 save file details

65031 get status

65048 flag status

65052 set status

65067 set timeout

65061 read or set top of memory

65063 read top of memory

65069 set top of memory

65076 read or set bottom of memory

65091 NMI entry point

65126 warm start

65206 reset IRQ and exit

65212 interrupt exit

65218 RS232 timing table

65238 NMI for RS232 input

65287 NMI for RS232 output

65347 fake IRQ

65352 IRQ entry point

65409 jump table

65530 hardware vectors

Index

abbreviations 72, 118, 123, 125
ABS 92
access mode 134
addition 40
ADSR 16, 113
aerial splitter 29
AMP Keyboard Technologies 22
amplifier 35
amplitude 112
AND 101
APL 22
APPEND 118
Apple 15, 19, 20
ARC 122
arithmetic keys 40
arrays – additional commands 121
– detailed 80
– sizes 82
ASC 88
assembly language 123
asterisk 40, 51
Atari 15, 19, 21
ATN 95
attack 112, 113, 114
audio/video port 29, 35
AUTO 117
AVERAGE 118
AVG 118

B Series 22
B128 22
B256 22
background 31
Basic – additional commands 118
– Commodore's 49, 50
– extensions 117
– faster programs 124
– history 49
– in opening message 31
– on 64 16
– programming 49
– reserved words 61, 62
– versus machine language 98
BC BASIC 122
BELL 119
bi-directional 163
bit masking 100
BLANK 119
blocks 146
BMB Compuscience 21
BORDER 119

border 31
BOX 122
Braunschweig 20
BREAK 119
buffer 135
BX256 22
byte 31, 98
Bytec Comterm 25

C language 26
C2N 139
capitals 37
cartridge port 29
Casio 18
cassettes – control by location 1 101
– connection 30
– port 29
– detailed 139
– LOAD 142
– possible problems 145
– program files 140
– SAVE 140
– VERIFY 142
CBM key – and colour changes 43
– in book's listings 45
– use of 41
CENTRE 119
Centronics 17, 169, 171
CHAIN 121
chaining 128, 143
CHANGE 118
channel 133
character sets 37
characters – strings and dollar sign 40
characters – as control codes 73
characters – vs numerics 51
CHINKLE 119
CHR$ – printer controls 170
– as control codes 73
– use of 88
CIRCLE 122
CLOCK 119
clock 62, 63
CLOSE 135
CLR – command 136
– in book's listings 45
key 41
CMD 168
COHERRENT 20. 26
COHOS 26
COL$ 119
COLLECT 123
colon – uses 39
colours – and PRINT 73
– in book's listings 45
comma – uses 39
command channel 152
Commodore – history 18
– microprocessors 15
– PC 25
– undelivered products 23
– 116 25
– 16 25
– 264 24
– 364 25
– 4000 19
– 500 23
– 8000 19
– 9000 23

composite video 35
CONCAT 123
conditional IF 122
CONT 57
continuous stationery 161
control port 29
COPY 122, 152
Corby 20
correspondence-quality 157, 160
COS 94
CP/M-86 20
CRSR (keys) 41
CTRL – and colour changes 42
– and reversed characters 43
– in book's listings 45
– use of 42
CURSOR 119
cursor – additional commands 119
– and PRINT 73
– and Return 42
– at start-up 32
– CRSR keys 41
– in book's listings 45
– position 76

daisywheel 160
Dartmouth College 49, 50
DATA 77, 127
Data 20 Corporation 25
Datasette 139
DATE$ 119
David's Basic 118
decay 112, 113, 114
decimal point 38
decimal tab 94
DEF 94
DEL – in book's listings 45
– use of 41, 135
DELETE 117
device numbers 133, 134
Digital Research 20
DIM 80, 127
DIN plug 27
DIP switches 162
DIR 123
Direct Mode 50, 142
DISABLE 119
diskette 147
disks – additional commands 123
– buying 148
– COPY 152
– deletion 152
– detailed 146
– directory 123, 151
– error channel 151
– error channel command 123
– formatting 149
– handling 153
– INITIALISE 152
– insertion 147
– OPEN 149
– relative files 153
– RENAME 152
– SAVE 151
– SCRATCH 152
– sequential files 153
– VERIFY 151
DISP 119
display – background 31
– border 31
– on monitors 34
– setting up with POKEs 99
division 40, 51
dollar sign 40, 59
dot matrix 157
DRAW 122
DX-64 24

EasyCalc 25
EasyScript 20
edge connector 29
electrostatic printer 157
ELSE 122
END 57
Enter (key) 42
envelope 112
ENVELOPE 122
example programs – calculator 83
– DIARY 81
– CARD SHUFFLE 1 84
– DICE 91
– CARD SHUFFLE 2 91
– COMPOUND INTEREST 94
– ARCTAN ALIEN 94
– USER-DEFINED ALIEN 107
– SOUND SETTING 115
– FANFARE 115
– CLUB 135
– ACCENTED 'E' 106
EXECUTE 119
Executive 64 24
exponentiation 40, 51

files – detailed 133
– file reference number 133
– mode 134
– name 134
– sequential 134
– type 134
FILL 122
filter 113
FIND 117
FN 94
FOR/NEXT 55, 126
formatting 149
frequency 112,113,114
friction feed 161
Frontier Manufacturing 18
full stop 38
function keys – additional commands 118
– with GET 69
– on keyboard 43
– in book's lsitings 45
fuse 29

garbage collection 126
GENIN 120
GET 55, 68
GET# 135, 138
GOSUB 54, 125
GOTO 54, 125
Gould, Irving 21
graphics – detailed 105
– from keyboard 37
guarantee 32

harmonic content 112
header 140

HELP 117
HHC-4 19, 21
hi-fi 35
high-resolution graphics 105, 107
HOME – in book's listings 45
– use of HOME key 41
Homebrew Computer Club 19
hub reinforcement 148
Hyperion 25

IBM 17
IEEE 17, 147, 159
IF/THEN 54
index hole 147
INITIALISE 152
INK 120
INPUT 67
INPUT# 135, 137
INST 41
INT 88
integer variables 59, 60
Intel 26
interpreter 98

jiffy 62
Jones, Myrrdin 21
joystick 29, 121

Kemeny, John 49
Kernal 101, 139
KEY 118
keyboard – detailed 36
– of 64 16
KILL 118
KIM 19
Kurtz, Tom 49

labels 119
leader 140
LEFT$ 87
LEN 86
LET 58
light pen 29, 121
LINE 122
line numbers 51, 117, 124
LINES 118
LIST 53, 56, 168
LOAD 135, 142
LOG 92, 94
logarithms 92, 94
logic-seeking 163
logical file number 133
loops – and improved programs 127, 128
– delay 113
– general 55
– timing 62
lower case 37

M6809 21
machine language 98, 123
mains adapter 27
Mark Williams Company 20, 26
mask 100
matrix printer 157
MAX 120
Max 22
memory – alternative configurations 101
– detailed 98
– in opening message 32
– outline memory map 99
– program storage 143
– saving 103

MERGE 118
Micro Display Systems 18
Micromainframe 21
Microprinter 157, 169
Microsoft 20, 50
MID$ 87
Miller, Bill 21
MIN 120
mini-floppies 147
Mitsumi Electric 20
MMF9000 21
modulation of sound 112, 113
modulator 32
monitor (machine-language) 122, 123
monitor (video) 34
MOS 6502 15, 19, 21
MOS 6510 15
MOS Technology 15, 18
Motorola 21, 26
MOVE 122
MPS-801 165, 169
MS-DOS 20
multiplication 40, 51
MUSIC 123

National Computer Conference 19
near-letter-quality 160
NEW 149
NEXT – and improved programs 127
– use of 55, 126
noise (waveform) 112
NOKEY 120
non-impact printer 157
numeric variables 59
NUMIN 120

ON 71, 127
ON ERROR 120
ON KEY 120
OPEN – files 133
– for disks 149
– for printers 168
– for RS232 172
operators 50, 86
OR 101
overlaying 128

P Series 23
PAD 120
paddle 29, 121
PAINT 122
Panasonic 19
paper 161
parallel interface 17
parentheses – uses 39
Pascal 22
Peddle, Chuck 18, 19
PEEK 99
PEN 121
percentage-sign 40, 59
period 38
Pet – colour version 21
– launched 18
– SuperPet 21
phono plug 27
pi 40, 51
Pilot 26
pin feed 161
pitch 112
platen 161

PLAY 123
PLOT 122
plotters 166
Plus 4 25
POKE 99
POP 120
port 29
POS 76
potentiometer 32
precedence, order of 50
Precision Software 20
PRG 135
PRINT – and improved programs 127
– detailed 72
– Direct vs Program mode 50
– formatting numbers 74, 94
– specifying print position 75
PRINT AT 74, 94
PRINT USING 74, 94
PRINT£ 135, 137
Printerface 169
printers – additional commands 123
– and graphics 158
– by Commodore 165
– CHR$ controls 170
– converted typewriters 160
– controls 162
– consumables 164
– correspondence quality 157, 160
– daisywheel 160
– detailed 157
– electrostatic 157
– listing programs 168
– lower-case 168
– maintenance 164
– matrix 157
– near-letter-quality 160
– non-impact 157
– non-Commodore 159, 169
– OPEN 168
– reliability 164
– speed 163
– thermal 157
Program mode 50
Programmer's Aid 117
pulse (waveform) 112

quotation marks 38, 51
QWERTY 36

RAM 31, 49, 99
Random Access Memory (see *RAM*)
READ 77, 127
Read-Only Memory (see *ROM*)
READY 32
REL 135
relational operators 86
relative files 153
release 113, 114
REM 56, 124
RENAME 152
RENUMBER 117
REPLACE 118
RESCUE 118
reserved words 61
Reset 28
RESTORE command 43, 121
RESTORE key 43, 54, 78
Return 42, 56, 72
RGB 35
ribbons 164
Richard, Don 21
Ricoh 18
RIGHT$ 87
RND 90
ROM 29, 49, 98
ROTATE 122
ROUT 121
RS232 17, 169, 171
RUN/STOP 54, 119

SAVE 135, 140, 151
sawtooth (waveform) 112
SCAN 121
SCR WAIT 122
SCRATCH 152
screen – copy 123
– editing 52
– save 121
secondary address 134
sectors 146
Seibu 19
Seikosha 158
semi-colon – and PRINT 72
– uses 39
SEQ 135
sequential files 134, 153
serial interface 17, 29
SGN 93
Shift 36
SHR$ 121
SHRINK 118
SID – in 64 16
– POKEs 113, 114
Simon's Basic 117
SIN 94
Sinclair 157
slash 40, 51
Smith, Marshall F 21
Softchip 64 118
Softex 157, 169
SORT 121
sound – additional commands 122
– adjustment 34
– detailed 112
– range of POKE values 114
– SID POKEs 113
Sound Interface Device (see *SID*)
space bar – in book's listings 45
SPACES 120
SPC 72, 73
SPC$ 121
SPRITE ON 122
sprites – activating 109
– additional commands 122
– and bit masking 100
– collisions 111
– colour 109
– design 108
– detailed 108
– displaying 110
– placement 111
– priorities 111
– storing sprite data 109
SQR 92
square root 92
ST 62, 137, 144
stack 120
STEP 118
STOP 57

STR$ 87
strings – and DATA 77
– and dollar sign 40
– comparison 86
– string-handling detailed 86
– variables 59, 60
– vs numerics 51
structured programming 122
subroutines 56
subtraction 40
SUM 121
SuperScript 25
sustain 113, 114
SWAP 121
SX-100 24
SX-64 – alternative names 24
SYS 99, 103

TAB 72, 73, 137
TAN 94
Tandy 19
television – aerial splitter 29
– buying 29
– connecting 29
– location
– lead 30
– picture 32
– tuning 31, 32
– versus monitors 34
– work-top 30
thermal printer 157
Thomas, Roy 21
TI 62
timbre 112
time 62
title (taps) 140
Toolkit 117
TRACE 118
tracks 146
tractor feed 161
Tramiel, Jack 18
triangle (waveform) 112
trigonometry 93, 94, 126
Tripler 159
'turbo' loading 139

Ultimax 22
Unix 20, 26
UNNEW 118
up-arrow (key) 40
upper-case 37
USER 102
user port 29
user-defined functions – DEF FN 94
– additional commands 121
user-defined graphics 105, 122
USR (command) 99
USR (files) 135

VAL 88
variables – and improved programs 126
– detailed 58
– DEF FN 94
– 'list ' command 118
– names 59
– reserved 62
– time 62
– types 40, 59
VERIFY 142, 151
VIC (see *Video Interface Chip*)
Vic-10 22
Vic-20 15
Vic-40 22
video splitter 29
voices 112, 113
volume 113

WAIT 102
Warner Brothers 21
Waterloo University 22
WAVE 122
waveform 112, 113, 114
WEDGE 151
Whitby Computers 118
wild card 142, 151
Wozniak, Steve 19
write-protection 141, 147

XPD 121

Z8000 20, 25
Zilog 20, 25, 26
ZX Microcomputers 157

This could be the only 64 book you'll ever need...

The Commodore 64 has become the world's best-selling home computer. It looks good, works well, is jam-packed with features and functions. . . . but it's not always easy to use those facilities.

This book offers a critical tour and explanation of the 64. It's a guide, a reference, and a tipster for all 64 owners — and for anyone considering the 64.

It's not simply a programming handbook, though. It's *the* book of the 64: it will give you the information you need to use the 64 to the full — whether you are writing your own programs or buying someone else's, whether you intend to put a 64 to work in your business or sharpen your reflexes on Space Invader games, whether you'll stick with the minimum 64, plus cassettes or whether you will use a system with disks, printers, modems and colour monitors.

About the author

Dennis Jarrett is Managing Editor of *COMMODORE USER,* Britain's largest-selling monthly magazine devoted to Commodore's home computers. He says he's not a computer enthusiast as such, and he's certainly no expert: but he likes learning the hard way, and he enjoys running a magazine that he regards as a kind of extended Commodore club. He lives alone on the edge of gentrification in trendy Islington with a decaying Gilbern and an aberrant answering machine.

Cover design by The Pinpoint Design Company

ISBN 0-09-154981-7

ISBN: 0 09 154981 7

Price: £7.95 net
IN UK ONLY



This was brought to you

from the archives of

http://retro-commodore.eu