SIMON POTTER

GETTING THE MOST FROM YOUR COMMODORE 64

THE INDISPENSABLE GUIDE TO YOUR HOME COMPUTER

THE PENGUIN PERSONAL COMPUTER COLLECTION

Penguin Books

Getting the Most from Your Commodore 64

Simon Potter has worked as a technical writer for computer manuals for many years and he now runs his own computer consultancy. He lives in North London.

The advisory editors for the Penguin *Getting the Most from Your . . .* computer series are Rory Johnston and Martin Banks.

Rory Johnston is a freelance writer on science and technology, and a former Associate Editor of *Computer Weekly*. He worked previously in the computer industry and research, and as a teacher was a pioneer of computer education in schools.

Martin Banks was the first specialist columnist on personal computing in the United Kingdom. He now writes on the subject in a wide range of magazines including *Personal Computer World, Which Micro?, Computer Weekly* and *The Times*.

Simon Potter

Getting the Most from Your Commodore 64

Penguin Books

Penguin Books Ltd, Harmondsworth, Middlesex, England
Penguin Books, 40 West 23rd Street, New York, New York 10010, U.S.A.
Penguin Books Australia Ltd, Ringwood, Victoria, Australia
Penguin Books Canada Ltd, 2801 John Street, Markham, Ontario, Canada L3R 1B4
Penguin Books (N.Z.) Ltd, 182–190 Wairau Road, Auckland 10, New Zealand

First published 1984

Made and printed in Great Britain by
Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk
Set in 9/11pt Monophoto Univers Light

Contents

Acknowledgements

This book is based heavily on work done for and with the Paradox Group, publisher of *Commodore User* magazine (formerly *Vic Computing* but now obviously extended to the 64 and other Commodore computers too). It also draws heavily on Penguin's *Getting the Most from Your Vic-20* by Dennis Jarrett, editorial director of Paradox: obviously the two Commodore machines are very similar. The acknowledgements in that work – notably to Chris Preston, Mike Todd and Jim Butterfield – as well as the entire 64-related content of several books and magazines (especially the U S mag *Compute!*) apply again here.

About This Book

The Commodore 64 is an excellent computer. It has a fine typewriter-style keyboard, a lot of memory, and excellent facilities for sound, graphics and colour. The standard unit is well provided with functions and features; but in addition there's a good range of expansion and add-on options, both from the manufacturer and (importantly) from a whole forest of independent suppliers.

It *is* a good computer . . . with some reservations about its idiosyncrasies, a degree of evident corner-cutting, and Commodore's overriding concern to make the 64 match as far as possible with the idiosyncrasies and corner-cutting in its other computers.

This book should help you understand the 64 a bit better. It should unfold some of the potential inside the thing; it should identify some of the oddities about the 64, too.

It is not a comprehensive technical reference manual and it is not an all-encompassing teaching guide. But it is an introduction for the computer novice, and it should also be of value to people who might know some other computer but haven't yet unpacked their 64. It is as free from heavy technology as I can make it: but it covers those 'technological' concepts and procedures that are necessary to use the 64, explaining them where it's unavoidable, but generally sticking to the pragmatic viewpoint of 'to make *this* happen you'll have to do *that*'.

Section One

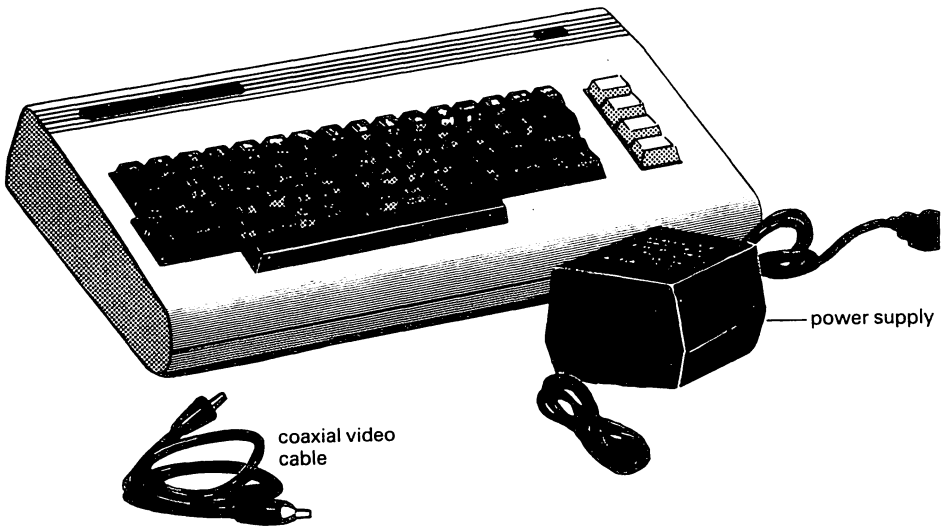
Getting Started

A look at what you've bought, how to organize it, and some first steps in using it

1. Inside the Box

When you unpack your 64, you should find you have the items in the figure below. They are:

- the Commodore 64 computer, which looks like a typewriter keyboard (plus some extra keys) on a mushroom-coloured base that has sockets of various kinds at the back and to the right. It's surprisingly light.
- a mains adaptor – a heavy cubish lump in matching colours with two leads coming from it (this thing 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 barrel-like DIN plug that goes into the 64.
- a TV lead – a length of cable about 75 cm (30 ins) long with what's called a 'phono' plug on one end (that goes into the 64) and an aerial socket connector for the TV on the other.
- instruction booklet – a spiral-bound lie-flat book of around 160 pages called *Commodore 64 MicroComputer User Manual*.

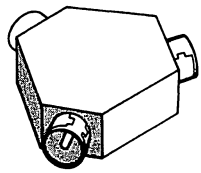


You may have bought more than this – but if you don't have *at least* all of the items listed above, take the 64 back to your dealer.

You'll also need a TV set. It doesn't have to be portable and it doesn't have to be colour. But both are preferable – the 64 will work OK in black and white, but one of the 64's real strengths is its ways with colour. If you can't afford a new colour set, consider buying an ex-rental TV (prices from about £50). The book assumes you do have a colour TV.

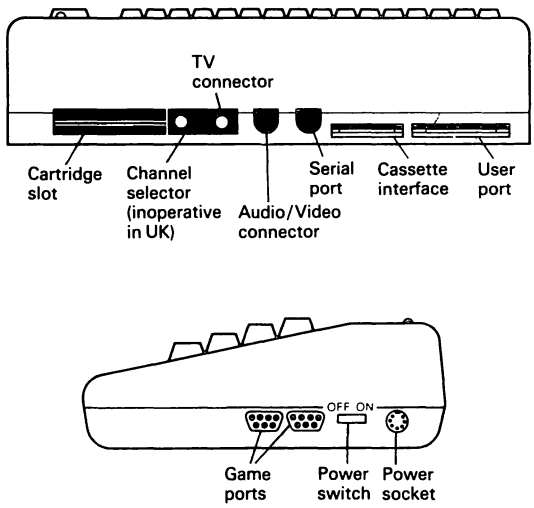
Positioning a full-size television set such that it's comfortable to use with the 64 *and* out of the way of the rest of the family can be a hassle. Portables are recommended if you can get hold of one.

One extra item that's also recommended is a thing called a video splitter – see below. 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 constantly be plugging and unplugging both the 64 and the aerial, which won't do either plug or the aerial socket any good at all.



You can get one from TV or hi-fi shops, and it shouldn't cost you more than three quid at the most.

2. Around the 64



The figure shows all the connections and sockets around the 64.

- on/off switch – a bit small, but obvious enough.
- mains socket – 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. It's still the power plug, though, and it is labelled 'POWER'.
- two joystick sockets – which in fact can connect other things too, like light pens and paddles (more about these later), but one or two joysticks are what are usually plugged in here. If you have only a single joystick (which is usually the case), it's plugged into one of these sockets, leaving the other free.

  They are labelled 'CONTROL PORT 1' and 'CONTROL PORT 2' and are often referred to as the 'game ports'. For no particularly good reason the home computer industry has found it necessary to use the word 'port' to mean 'socket', and since 64 users will probably come across 'port' quite a lot, you'd better get used to the word.
- cartridge slot – not labelled, but it's the big deep slot at the back nearest the power switch. It's used to plug in games or extra program-writing facilities, both of which can come in the form of flattish plastic cartridges which plug in here.

- TV connector – takes the lead from your TV. It passes sound and pictures from the 64 to the television. It is not labelled, but it is very characteristic: it's the small silvery single-pin socket that juts out slightly.
- 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 is nearer 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 a 'composite video signal' for a better picture.
- serial port – the other round socket at the back, next to the video port. It is used to connect the 64 to a printer, a disk drive and some other extras you might be buying. Of the two similar sockets, this is the one with fewer holes: the plug for an audio/video connector won't fit into it.
- cassette port – the smallest of the three slots. The lead from a cassette unit plugs in here (the cassette uses a flat multi-way connector instead of a plug).
- user port – the third slot. It is used for plugging in things like the Commodore modem to allow your 64 to send and receive information over phone lines (e.g. to another 64 modem). It's called the 'user port' because with advanced programming and some clever electronics construction, you can control other devices via this connection.

Also round the back there is a funny little screw. The *User Manual* refers to a 'channel selector' which, you're informed, 'is used to select which TV channel the computer's picture will be displayed on'. Well, this doesn't apply in Britain, so for the UK this switch was omitted. We get a screw instead. Ignore the comment in the *User Manual*.

3. Setting Up

First choose your working space. The different bits of the 64 can sprawl a bit and you will probably want somewhere to put books, notepads and cups of coffee, so give yourself enough room.

The hefty power transformer ought to be well out of the way: if you knock it off, it could do some damage to your foot and/or your 64.

If you have a cassette player, it is wise to site it as far away from the TV as possible, since in some circumstances the unit has been known to pick up unwanted radio emissions from the television set. This can cause faults when storing programs on tape or loading something from cassette.

It makes sense to have the TV set higher than the keyboard, if possible: 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 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 the cassette player (you'll have one soon even if you don't start with one) and a printer (sooner or later you'll want one). You might want to screw a couple of switched extension sockets on it too – the 64 takes one, the TV needs another, and a mains connection is also required for a disk drive and the printer. This prevents too many power leads wandering around the floor.

Making Connections

Now start connecting things together. 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.

If you have a cassette unit, plug that into the smallest of the three slots at the rear of the 64. There's only one way it will fit, so if it isn't sliding in easily turn it over and try again.

Now MAKE SURE THE ON/OFF SWITCH IS OFF – and if the wall socket has a switch, put that off too. The power lead from the heavy mains adaptor goes into an ordinary 13 amp socket at one end and the 64 itself at the other – the 64's end has an unmistakable round multi-pin plug, and there's only one way it will go into the socket. Don't force it! The groove in the barrel should be on the upper side.

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; it's still busily

converting 240 volts into 9 volts whether or not the 64 is running. Like everything else in this sad little world, overuse will wear it out. So switch off at the mains whenever you can.

The Big Turn-on

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.

Turn on the TV and select an unused channel. If you're very lucky you will immediately see the 64's opening display – but that's most unlikely. Find the tuning knob on the TV for the channel you're on and start twiddling. You may pick up TV stations before you reach the 64's signal, but sooner or later you should see light blue words on a dark blue background with a light blue border and a light blue square flashing at you. (Some 64s are now being shipped with the Vic's infinitely more legible colour plan of dark-blue letters on a white ground with a light-blue border.)

- the border – you can subsequently 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. You can fill it with whatever you want; you can change its colour, too.
- 'COMMODORE 64 BASIC V2' – this message tells you that the 64 is ready to start accepting Basic programs and the standard commands you can use. 'BASIC' is the name of the programming language you have available to you; 'V2' means 'version no. 2' (Commodore has four versions of Basic on its different computers).
- '64K RAM SYSTEM' – 'RAM' is 'random-access memory', which is the kind of electronic storage that you use for your programs. 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 hip advertising agencies) to indicate 'times 1,000' (or more accurately 1,024, since K is really an arithmetical 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. The 64 uses some of it for its own functions, though, which is why you also see . . .
- '38911 BASIC BYTES FREE' – this tells you how much memory you have to play with after the 64 has sliced off what it wants for itself. Why '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.
- 'READY' – the 64's waiting for you to do something.
- the blinking square – this is called the *cursor*. It indicates the position on the screen where the next character you type will appear.

If you can't find this display after turning the tuning knob through it's whole range, switch off and wait about ten seconds, then try again. If it's still not there, check all the connections and try again. 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.

4. First Acquaintance

Take a look at the keyboard. It is laid out much like a typewriter's, with a few extra keys and some strange symbols on the front of most keys – they are there for producing graphics: more about this later.

Everything you type appears on the screen. Try typing

```
PRINT 3 + 5
```

The cursor moved every time you hit a key and should be sitting just after the 5 now; if you wanted to type anything else (including spaces), it would appear at the point where the cursor is.

One difference between the 64 and a typewriter is that when you touch the keys you get *capital* letters. A normal typewriter gives you *lower-case* letters until you press the SHIFT key. You can do lower-case on the 64, as you will see later.

PRINT is an instruction, a command 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 – 'PRINT3+5' will work just as well. It's actually an instruction to display the result of adding 3 and 5.

Why 'PRINT'? It's a hangover from the days when computers didn't have screens and all used typewriter-like printers to show you what you'd done.

So how do you actually tell the 64 to get on with it? At the moment the computer has no way of knowing whether you want to type some more. A person might be able to tell that you have finished the instruction from your tone of voice or even by asking you, but the 64 can't hear a tone of voice in your typing.

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 line – the cursor returns to its start position on a line.

Pressing RETURN should immediately give you the answer to 3+5 on the screen, with the 64 telling you it's 'READY' for your next command.

Why can't you just type '3 + 5 =' as on a pocket calculator? Because the 64 needs a *command* to tell it which of the functions available you actually want.

Handling Mistakes

If you made a mistake in your typing, fret not – you can correct it very easily. There are two keys at the bottom right labelled 'CRSR'; they 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. In both cases, *hold down either of the two keys marked* 'SHIFT'.

To correct a mistake you use the CRSR left key to step back along the line. 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 don't take your finger off the cursor key it will carry on moving. You might go too far to the left, in which case you can just step back to the right by hitting the same key *without* holding down the SHIFT.

Let's try this out. Type

```
PRIMT 3 + 5
```

making a deliberate mistake on the N. Before you hit RETURN, take the cursor back to the M and correct it by overtyping. And now try RETURN – no, don't worry about getting the cursor back to the end of the line again. One of the really nice things about the 64 is the way it doesn't really care where the cursor is on a line when you hit RETURN – it will try to obey the instruction anyhow, and so long as the line contains something it understands it will be able to do it. Clever, eh?

5. More Ways with Numbers and Words

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

```
PRINT 12+6-7+33.4
```

(Don't forget the RETURN at the end.) Quick, isn't it? You can also do multiplication and division; use the asterisk for multiply (it's * to avoid any confusion with the letter x) and the slash (/) for divide.

You can also include brackets to force the 64 to do its arithmetic in the right order. So

```
PRINT 3/6+2
```

will produce a different answer from

```
PRINT 3/(6+2)
```

(By now you should be filling up the screen. When that happens you'll notice that everything shifts up one line so that the cursor can sit blinking on the bottom line. You're losing the stuff at the top of the screen by now, but don't worry about it.)

You can use PRINT 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. Instant demonstration: type

```
PRINT "3/(6+2)"
```

and the 64 will display everything in the quotes without doing the calculation.

Obviously, you can put letters in the quotes . . .

```
PRINT "HI THERE FOLKS"
```

and you can also include space by hitting the space bar:

```
PRINT  "HI  THERE  FOLKS"
```

More Mistakes

PRINT is one of a few dozen words that the 64 understands. This limited vocabulary is something you'll have to get used to (it's not difficult); and you'll also have to get acquainted with the way the 64 likes to use its words.

Try typing 'HI THERE FOLKS' without the PRINT command and without the quote marks. You'll get this:

```
HI THERE FOLKS
?SYNTAX ERROR
READY
```

'SYNTAX ERROR' means the 64 hasn't understood what you want to do. That's because it does not recognize 'HI' or 'HI THERE' or 'HI THERE FOLKS' as a word in its vocabulary. You would get the same result if you'd hit RETURN after typing 'PRIMT 3 + 5', because 'PRIMT' isn't in its vocabulary either.

Actually it's not really a *syntax* error at all – syntax is all about the correct arrangement of words in a sentence, and that wasn't your problem: you just used the wrong words altogether. Is this pedantry? Maybe, but why *shouldn't* computers use the right terms?

On the other hand, one of the good things about computers is that you can't hurt them by mistypings like these. And on the 64 they are easy to correct. So make as many as you want!

You've met the DELete key. Now welcome to INSert. It's the same key, but you hold down SHIFT and hit INST/DEL to get the insert function.

Let's correct that SYNTAX ERROR. Take the cursor back up the screen to the 'HI THERE FOLKS' that caused the error – you do that with the other CRSR key, using SHIFT at the same time to give the upwards movement (otherwise the cursor would move *down*). Position the cursor on the 'H' in 'HI', keep one finger on SHIFT, and hit the INST/DEL key.

Voila! A space opens up to the left of 'HI'. We want enough space to insert a PRINT command and the opening quote marks, so hit INST/DEL a few times. Zip the cursor left over the gap to the first position on the line and just type in 'PRINT " ' – use INST again to give yourself more room if you need it. Then move the cursor over to the right and put the closing quotes after the word 'FOLKS' and hit RETURN. Worked, didn't it?

6. Labels

Now here's another word that is in the 64's vocabulary – LET. A computer's memory can hold information for you until you want to use it, but to do that you have to give the information a label for the computer (and you) to remember it by.

With the 64 the label is a letter, any letter you want. And you use LET to assign a particular label to a particular piece of information. If you want to store the number 1,578 somewhere in the 64's memory, the command might look like this:

```
LET A=1578
```

It doesn't have to be the label A: you can use any letter you want. And you don't have to know exactly where in its memory the 64 has put that number; you can always use A to get at it.

So if you then say

```
PRINT A
```

the 64 will come up with the number you stored away under that label. What's more, it will continue to remember the number labelled as A until you store away something else as A – or until you switch off the computer.

Try this:

```
PRINT A+22
```

Yes, you can use labels like this in arithmetic. The 64 will add whatever you've labelled to the number you give and produce the right answer.

Another useful trick:

```
LET A=A+22
```

Now try 'PRINT A' and you should get '1600'. See what happened? The 64 added 22 to the old value of A (which was 1,578) and made that label apply to the new number. You've lost the number originally labelled A.

You can have a huge number of different labels in use at any time – in other words, a vast collection of different numbers for re-use. Each of them must have a different label, but you're not restricted to the twenty-six single letters of the alphabet. You can use *pairs* of letters, like AD or ZB; and you can use a letter followed by a number, like Q6 or M2.

Once you have set up a label or two, you can use them thereafter just as you would use numbers. For instance, type this lot:

```
LET B=1ØØ
LET C=55
LET ZZ=3.3
```

and now:

```
PRINT A+B+C+ZZ
```

If you got the answer '1758.3', it's because the 64 added the values you'd labelled 'A' (still set to 1,600, remember?) to the new labels you've just assigned.

Labels for Text

It's not just numbers you can label like this. You can set up text, too, and store it under a label. Two things to watch: you need a special type of label and the words have to be in quote marks.

Again, you can, in theory, have that huge combination of different labels, because you can use single letters or letter pairs or a letter and a number. But the label has to have a dollar sign after it to tell the 64 that it's text:

```
LET X$="HI THERE FOLKS"
PRINT X$
```

Assuming you've done it correctly the message should appear on the screen.

Let's get a bit more fancy. Type in

```
LET ZZ$=" TODAY IS "
LET A$="."
LET D$=" AND IT'S "
```

(That's a full stop for A$, and get that space in after 'TODAY IS'.) Now try this:

```
PRINT X$+A$+ZZ$+"WEDNESDAY"+D$+"RAINING"+A$
```

Don't worry about running on to the second line: the 64 will look after that for you, and it will believe everything you type in until you hit RETURN. (In fact there's a limit of eighty-eight characters to a line before the 64 won't accept it, but it will be a while before you come up against that restriction.)

In computer jargon these labels are called *variables*, because you can alter their contents so easily.

You don't have to get calluses on your index finger from typing LET all the time. There's a shorthand form that omits the word altogether. 64 assumes that a letter or a pair of letters followed by an equals sign *always* means you want a LET command in there. So you could just type:

```
ZZ$=" TODAY IS "
A$="."
D$=" AND IT'S "
```

and it would have the same effect.

7. Your First Program

So far everything you asked the 64 to do was done as soon as you hit RETURN. But the 64 can hold sequences of instructions in its memory for future use, so that they aren't acted on at the time: this ability is what really gives computers their power.

A sequence of instructions is called a *program*. Here is a very simple program:

```
1 PRINT "HI THERE FOLKS"
```

Now, you might think that's much the same as something you've typed before. But when you hit RETURN at the end of the line, nothing much happens. That's because there's a number at the start of the line; and when the 64 finds a line starting with a number, it assumes that it is part of a program.

What you have just typed is a complete program in itself, sitting in the 64's memory waiting for you to tell it to get working. You do that by typing RUN – try it now (don't forget the RETURN).

When you typed in the instructions before, the computer did what you told it and then forgot the command. This time it has remembered the instruction: and every time you type RUN it will repeat it again.

Try converting some of the 'instant' instructions into a program with numbered lines. For example:

```
1 X$="HI THERE FOLKS"
2 ZZ$=" TODAY IS "
3 A$="."
4 D$=" AND IT'S "
5 PRINT X$+A$+ZZ$+"WEDNESDAY"+D$+"RAINING"+A$
```

A tip: you can add the line numbers just by taking the cursor to the start of each line, using INST/DEL to insert a couple of spaces, and adding the number. Hit RETURN and nothing will happen; the cursor will just move to the start of the next line for you to insert some spaces there.

When you've finished, try typing RUN. It should work; if it doesn't, it's not the 64's fault – you have done something wrong.

Let's tidy up the screen a bit. If you hit the key marked CLR/HOME, the cursor will go to the top left-hand corner of the display – that's it's home position. But if you hold down a SHIFT key at the same time as you hit CLR/HOME, you'll find that the screen clears magically.

This hasn't wiped the 64's memory, though. Your program is still in there, and so are your labelled variables. Type RUN again and the program should do its stuff for you.

Since it's an idea to learn good habits early, you should never use line

numbers in your program that follow this close together. The numbers are there to identify what follows as part of a program: but they also tell the 64 the order you want it to follow when it starts obeying the program's instructions.

Now, it doesn't care whether you leave gaps in your line numbering – it will just obey the line with the lowest number first, then go on to the next lowest and so on. And it doesn't require you to enter the lines in strict number sequence: you can put in line 99 first, then line 5, then 56, and it will run through them with 5 first and 99 last.

So what? Well, this means you can add more lines to a program later, provided you've left enough spare line numbers. Otherwise you might have to change all the line numbers you originally used. On a short program that might not be too much of a problem; on a long one it's a real drag. For starters, then, you should get used to numbering in 10s or even 100s. That way you have lots of spare numbers for new lines you might want to add.

To renumber the program we've been RUNning, we'll have to get it back on to the screen. You do this by typing 'LIST' and hitting the RETURN key. (The command LIST puts on to the screen whatever program is currently in the 64's memory.)

Once it's there in front of you, move the cursor up to line 1 and start renumbering. You can do that simply in this case by adding zeroes: hit RETURN after you make each change.

```
10 X$="HI THERE FOLKS"
20 ZZ$=" TODAY IS "
30 A$="."
40 D$=" AND IT'S "
50 PRINT X$+A$+ZZ$+"WEDNESDAY"+D$+"RAINING"+A$
```

Type LIST again. Now you have the newly numbered lines 10 to 50, but the original versions (lines 1 to 5) will still be there. Why? Because what you LISTed wasn't the actual program – it was a *copy* of it. The original program complete with the original line numbers is still in there. The 64 assumed you were *adding* new lines; and it will accept as a new line anything that's on the screen when you hit RETURN.

How do you get rid of the unwanted five lines? Simple: you type something with a line number that happens to be the same as one you already have in the program. The 64 will junk the original line and use the new one. So you just create five new blank lines with those line numbers: if they don't contain anything, the 64 can't act on them, can it? Type '1' followed by RETURN, then '2' and so on till you get to '5'. That creates five new lines with those numbers – but since they're empty the 64 will ignore them. When you type LIST again you'll see the program starts at line 10.

8. A Guided Tour of the Keyboard

We have seen how some of the 64's keys are used. It's time to summarize what they all do.

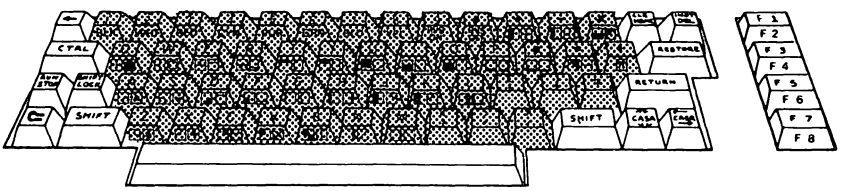
The keyboard might look like a typewriter at first, but most of the typewriter-style keys have at least two symbols on them and many have three.

There are three kinds of functions provided by the 64's keys:

- printing or displaying something
- moving the cursor
- doing something else

Printing Keys

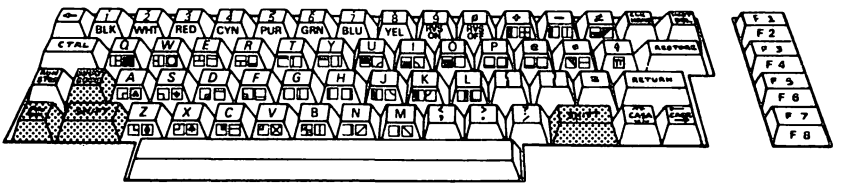
The keys that print something are the alphabetic keys and the numbers along the top row, the punctuation marks and other symbols, and the graphic symbols engraved on the front of most of the letters.



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.

The letters don't work like that. On a typewriter you'd get lower-case letters if you didn't use the shift (usually called 'unshifted' letters) and capitals if you did. On the 64 you get the same when you hit the key without the SHIFT.

If you do hold down the SHIFT and type a letter, you'll find that you get one of the graphic symbols marked on the front of the key – the one on the right, in fact. Don't worry that you don't get it appearing in a kind of ruled box as on the key: the 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.



Either of the two oversize 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 to release it. So you'll get graphic symbols when you type the letter keys.

To get the other shape marked on the front of a key, the one on the left, you type it while holding down the key marked with that curious symbol at the bottom left of the keyboard. In fact it's Commodore's own corporate logo, and it's a kind of extra SHIFT key. You'll see it referred to as the 'COMMODORE' key or the 'CBM' key.

There are a total of sixty-two 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'.

You can get lower-case letters too, by holding down the SHIFT and COMMODORE keys with one hand while you type letters with the other.

Special effects with punctuation

The punctuation marks have a number of wrinkles associated with them. To run through the principal features:

Full stop

This doubles as a decimal point; in fact outside double quotation marks that's its only use.

Comma

Don't use it as a comma except within quote marks; and it won't be recognized if you try to split up long numbers conventionally with commas (so type 1772836 rather than 1,772,836).

There are two special uses: in commands it separates 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. Second, in the lines of a program, the comma between separate PRINT commands means that separate items to be printed will appear one every ten columns – it's an automatic 'tab', in fact.

Semicolon

Again, the semicolon functions as a semicolon only inside quote marks – outside the quotes it has a specific meaning in Basic. When it separates differ-

ent PRINT statements on the same line of a program, it means the different bits of printing will appear on the same line.

Colon

Again, this acts as a colon inside quote marks and has a totally different meaning elsewhere. One is to separate different program statements appearing on a single line:

```
2ØØ FOR T=1 TO 1Ø: PRINT "HI
    THERE": NEXT
```

This is actually three separate statements (FOR, PRINT and NEXT) all on the one line. Go on, try it.

The other use of a colon is to make programs more readable. Ordinarily, the 64 will take out any unnecessary spaces you've left before a statement while typing in a program; so this

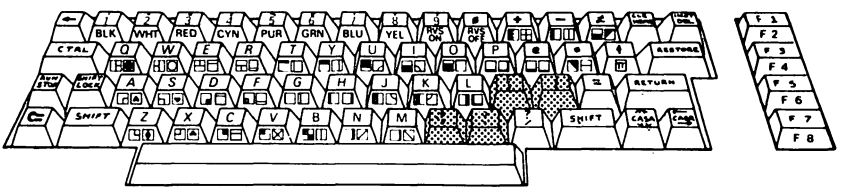
```
55Ø              PRINT "HELLO"
```

would appear as this if you were to LIST it after typing it in:

```
55Ø PRINT "HELLO"
```

But sometimes you might want that gap, for instance to indicate where a new section of program starts. Putting a colon in there will give you the spaces:

```
55Ø :            PRINT "HELLO"
```

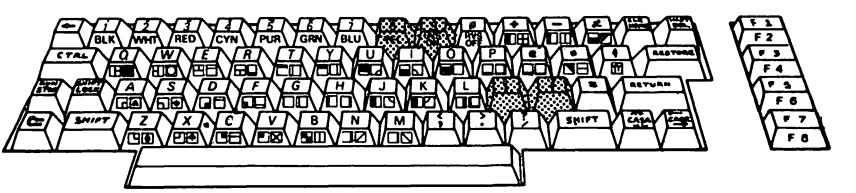


Brackets and parentheses

The square brackets are only square brackets: you can use them only inside quotes.

The parentheses (the curved brackets) can also appear inside quotes. Additionally, they are used as in maths to indicate an order of calculation: '5/(6+0.2)' produces a different answer to '(5/6)+0.2' because you're telling the 64 to do the arithmetic in the parentheses 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, actually. And 'PRINT TAB(8) "HELLO"' puts the word HELLO on the screen eight places from the left.



Arithmetic

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

Addition

Use the plus sign – but *don't* hold down the SHIFT key at the same time. This produces a character which *looks* like a plus, but is just a cross.

Subtraction

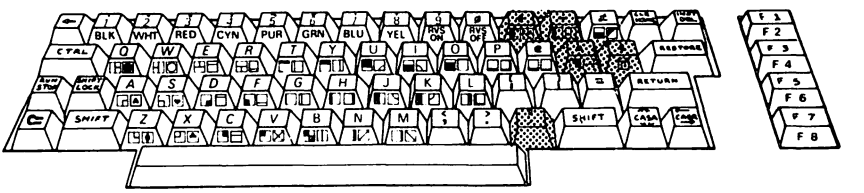
The minus key is normal. If you try holding the SHIFT down you'll get a symbol that looks nothing like a minus.

Division

Use the slash key – it has the same effect as the normal division symbol.

Multiplication

The asterisk has the same effect as the familiar multiplication symbol.



Exponentiation and pi

There are two other helpful maths symbols on the keyboard – the up arrow key, which provides exponentiation, and the pi symbol (π), which you get by pressing the same key while holding down a SHIFT key.

If you're not familiar with exponentiation, it's normally written something like 2^2 and means 'multiply the first number by itself as many times as the second number specifies'. The technical term is 'raise to the power of . . .'. Exponentiation takes the form '34↑4', for example, to raise 34 to the power of 4.

Pi comes out to eight decimal places; you could see that if you were to type 'PRINT π', and you can also use the symbol in arithmetic ('PRINT $\pi*r\uparrow 2$' finds the area of a circle when you replace *r* by an actual radius).

Percent and dollar signs

As well as their normal use in text – within quotation marks, that is – these two symbols are 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 per cent mark, character strings by the dollar. This illustrates all that:

```
1Ø A$="123.4"
2Ø A=123.4
3Ø A%=123.4
4Ø PRINT A$, A, A%
```

A character string has to be enclosed in quotes; neither of the other two can have quotes anywhere. (Try it and see what happens.)

If you now 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 – see how it's lost the decimal point and everything following it?

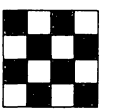
The hidden symbols

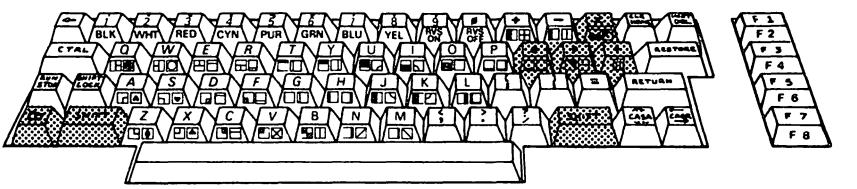
Using the SHIFT and COMMODORE keys, all of the 'printing' keys on the 64 keyboard produce at least two characters (one alphanumeric, one a graphic) and most can print three. The characters are all marked on the key in question. What the handbook doesn't tell you is that there are four other graphics 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 @ key to get 

press COMMODORE and * key to get 

press SHIFT and £ key to get 

press SHIFT and ↑ key to get 



Cursor Controls

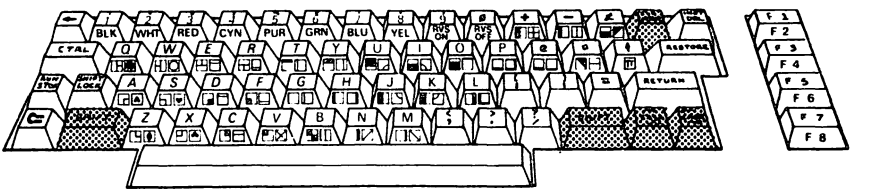
We've already met these – the two CRSR keys at the bottom right and the HOME key on the top row.

Use the CRSR keys unshifted to move down and right, use the SHIFT as well to move up and left. Holding the CRSR down key will move the cursor down the display until it hits the bottom line, at which point it will carry on looking at the 'next' line and so force anything already on the screen off the top edge. Sadly, you can't get that back by using the CRSR up (SHIFTed) key. That will go up till it reaches the top edge of the screen – and there it will stop, flashing impotently at you.

CRSR right 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 to the 'next' line and force stuff off the top of the display.

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

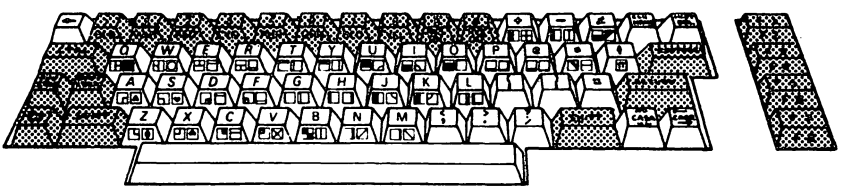
HOME takes you straight to the top left corner of the display without deleting anything: SHIFTed HOME gives you CLR, which does the same thing, but also clears the display for you.



Keys Which Do Something

The rest of the keys on the 64's keyboard don't affect cursor movement or print anything. We've come across a few:

SHIFT
SHIFT LOCK
RETURN
COMMODORE (CBM)



Here are the others:

CTRL
RVS ON
RVS OFF
colour controls
RUN/STOP
RESTORE
function keys

CTRL and the colours

The CTRL key is the biggish one at the top. It's used as yet another SHIFT key to get at the functions marked along the front of the keys in the top row.

Under the number keys 1 to 8 are marked the eight colours in which the 64 can display text on the screen (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 kind of yukky light blue: the border starts off in cyan), purple, green, blue (the cursor and the letters displayed when you switch on are in this blue) and yellow.

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.

CTRL and reverse characters

You also use CTRL to switch in and out of 'reverse' display for anything you type – CTRL and RVS ON (the key with 9 on top) switch it on, CTRL and RVS OFF (the key with Ø 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. So you get a white character on a solid coloured background. This works whatever colour you've selected for the characters and the cursor.

RUN/STOP

The RUN/STOP key is another that has two functions. When the SHIFT key isn't being used, pressing RUN/STOP won't have any effect at all unless you're RUNning a program at the time. Then it will STOP the program midway through it and tell you which line number of the program was being executed when you halted things.

Here's a simple demonstration that introduces another word in the 64's lexicon. When the 64 sees the command GOTO in a program, it goes to the line number specified and does whatever you're asking for there. If it means going back in the program, it does it again . . . and it will do it again and again until you tell it to stop.

So line numbers have a third function. As well as telling the 64 that what

follows is a line in a program and indicating the order you want things done in, line numbers can also act as signposts – with some of the commands you have available, you can direct the 64 to a *particular* line in the program. That way the numerical sequence of lines doesn't have to be the only factor that decides what happens when.

Look at this little program:

```
1ØØ A=Ø
2ØØ PRINT A
3ØØ A=A+2
4ØØ GOTO 2ØØ
```

What's happening here is that you're setting up a label called 'A' and starting off by making sure it's storing the number 0. In line 200 you're printing whatever A refers to; so the screen will display 0. Then you're adding 2 to A – and that GOTO in line 400 takes you back to the PRINT command. When the 64 gets there it will display what's in A again; only this time it will be 0+2, i.e. 2. It goes on to line 300 again, adding 2 to A once more, then goes back to 200 . . . and now A is 2+2, i.e. 4. That will be printed.

And so on. This program will carry on going round and round like this, adding 2 to whatever was last in A and making the result the new value of A before it gets printed. As it stands there's no way of stopping the program until it starts reaching numbers too big for the 64 to handle – unless you use the RUN/STOP key. Try it: type in the program, RUN it, and when the numbers start appearing press the RUN/STOP key. You should have something like this:

```
1ØØ A=Ø
2ØØ PRINT A
3ØØ A=A+2
4ØØ GOTO 2ØØ
RUN
 Ø
 2
 4
 6
 8
 1Ø
 12
 14
 16

BREAK IN 2ØØ
READY.
```

When you hit STOP the program was on one of its cycles back to line 200. That's what 'BREAK IN 200' tells you: you stopped the program at line 200.

RUN/STOP and SHIFT

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. Try it and see what happens:

```
READY.
LOAD

PRESS PLAY ON TAPE
```

That's exactly what will happen if you type 'LOAD'; it tells the 64 to expect something to be loaded in from the tape player. And obviously it's a bit quicker to hold down the shift key and hit RUN/STOP than to type the four letters in LOAD.

RUN/STOP and RESTORE

The biggish key on the right of the second row down labelled 'RESTORE' only works if you press it *while holding down* RUN/STOP *at the same time*.

Go on, try it. The screen has cleared and you've got the cursor blinking at you underneath 'READY', right? It's a rather more dramatic way of stopping a program and/or using CLR to wipe the screen clear.

Pressing RUN/STOP and RESTORE actually 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. But using RUN/STOP and RESTORE doesn't erase the memory. Type LIST and you should see that little program with the GOTO line in it: it's still there waiting for you.

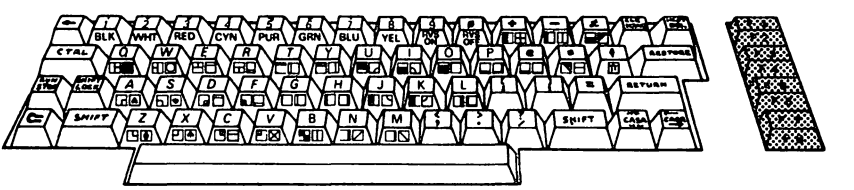
The function keys

The four brownish keys in a row on the right are called 'function' keys. They are labelled f1 to f4 on top, f5 to f8 on their fronts: 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. A second program, though, may well give totally different functions to those keys.

These four keys actually give you 8 possible functions. Four of them, numbers f1 to f4, you get just by pressing the key. For f5 to f8, you hold down SHIFT and hit the key.

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.



9. Using the Cassette Player

Every time you switch off the 64, everything that was in its memory is wiped out. That means if you have a program you're working on, it will be completely lost to you – unless you've taken a copy of it. Now, you *could* do that by hand – you could just write down what you've been typing, so that the next time you want to use the 64 you retype it all.

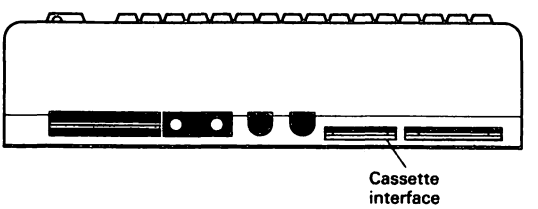
What's more, you can't have more than one program at a time sitting in the 64's memory. If you want to work on a second program, you'll have to get rid of the first one somehow: but what if you want to come back to it sometime?

If you have a Commodore cassette unit, though, you can copy your program on to tape before switching off. And next time you want it, you can load it back in from tape. You get a literal copy of the program, free from any mistakes in copying and retyping. That way you can build up a whole 'library' of programs on cassettes, to be loaded as and when you want. And you can load other people's programs from cassette, too – games you buy, programs you borrow from or swop with other people, and so on. Are you sold on the idea of a cassette unit yet? You should be.

It is possible to use your own portable tape player . . . but it's not easy. This is because you'll need a special connector (the socket for the cassette unit looks a bit odd, remember?). You can buy these connectors, but you'll have to search through the small ads in the microcomputer magazine to find people who sell them.

It's simpler to buy Commodore's own tape unit. This isn't cheap, and it won't let you play music tapes either: what's more, it is also pretty slow in operation – an ordinary tape player moves much more quickly. 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, and that's one reason for the funny plug. It is pretty robust and generally quite 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 tape players from Commodore that you might come across – Commodore calls a tape player a 'Datasette', incidentally, and sometimes a



'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 features, and they both plug in exactly the same way.

Plugging In

To attach the thing, make sure the 64 is turned off (yes, you'll lose your program, but don't worry about it). Slide the plug gently on to the connectors and make sure it's secure there. Don't try forcing the plug in: it only goes one way round. Take a look at the plug – it's divided into two, one gap being narrower than the other. The 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. Then you'll always know which way round it should go.

Now turn the power back on and check that the tape unit is working by putting a cassette in and pressing the PLAY button: if the tape counter is going round, things are probably OK. Hit the STOP button and REWind the tape.

Using the Cassette Unit to Save Programs

You store programs on tape by typing 'SAVE'. If you do that and hit 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*; if you did it right, the tape will 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.

You and the 64 can be a bit cleverer than that, fortunately. Because you can save several programs on one cassette, you'll find it useful to identify each of them separately – and you can do that by giving each program a name. This you do simply by adding a name in quotes after the SAVE command. The name you use can be up to 128 characters long, including spaces and symbols, so you ought to have no problems in finding unique names for each of your programs.

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 again.

Still problems? You may be trying to use a 'write-protected' tape. Take a look at the back of the cassette. All cassettes have a couple of slots there that ordinarily contain a couple of plastic tabs. If you remove those tabs by easing them out with a small electrical screwdriver, you will prevent any recording at all on that tape – it's called 'write protection', because you are protecting the tape from being 'written' on to. When you try pushing down RECord and PLAY with a write-protected tape in there, the RECord key simply won't engage. If you subsequently decide that you *do* want to record on to a write-protected tape, you can just put a piece of sticky tape over the holes.

The 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.

The counter doesn't measure anything particularly (it doesn't correspond to millimetres or tenths of an inch or whatever), but it is fairly regular and reliable. So if you stored something starting at 000 and it finished storing at 010, that count will let you know where the stored program begins and ends.

This is useful because you can store more than one program on a tape. If your first one finishes at 010, you can take a note of that on the cassette's liner card; then hit the FF (fast forward) key while the counter winds on a bit, say to 015 or 020, and press STOP. The next program you put on to the tape can start there . . . and so on. If you take a note of the start point for each program as indicated by the counter, you have a nice easy way of cataloguing what's on the cassette.

Verifying What You've Saved

Once you've saved something, it is a good idea to check that whatever is 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 REW 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 let you know (it'll come back with 'FOUND', and the name of the program, or it will just say 'FOUND' if it's found a program that you didn't name) and it'll start trying to verify it. In fact it goes through the whole program on the tape comparing it character by character with whatever is in the memory.

If it can't find anything on the tape, you'll just see the 'SEARCHING' message until the tape runs out. Either you're using a blank tape with nothing on it (not very likely, surely); or the 64 isn't trying hard enough. Press RUN/STOP and RESTORE together, REWind the tape again, and try 'VERIFY' once more.

If you still can't find your program, it probably isn't there. Use RUN/STOP and RESTORE, rewind the tape, and try again from scratch with another 'SAVE' followed by 'VERIFY'.

If the 64 finds a program and the two do match, it will say 'OK' and 'READY'. If they don't match, you'll get the message '?VERIFY ERROR'. This might be because the program it found isn't the one you want to check, in which case type 'VERIFY' again – if the PLAY button happens to be still down, the 64 will just restart the tape automatically 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. Try rewinding the tape and saving the program again.

You can have the 64 look for a specific program to verify. As with the SAVE command, you just type in 'VERIFY' followed by the program name – as in 'VERIFY "THE GOTO EXAMPLE"'. When you hit RETURN and press PLAY on the cassette deck, the screen will tell you that it's 'SEARCHING FOR THE GOTO EXAMPLE'. It will list every program it comes across until it reaches 'THE GOTO EXAMPLE' – and if it finds unnamed programs you'll get 'FOUND' followed by nothing.

If you don't ask the 64 to VERIFY a particular program name, it will just look for and verify the first program it comes across.

Loading from Tape

To read something into memory from the cassette, you use LOAD. You can type this out in full, or you can hold down a SHIFT key and hit RUN/STOP. This will produce the 'PRESS PLAY ON TAPE' message, and pressing PLAY starts the 64 'SEARCHING'. The screen will go blank – in other words, it goes blue all over – until the 64 finds a program on the tape. It will then flash the message 'FOUND' followed by the program name (if you gave it one when you saved it!). This is the first program it finds on the tape, and the screen will go blank again while the 64 loads it.

Alternatively you can tell it to load a particular program, by giving a program name (in quotes). That way you *can't* use the RUN/STOP trick – you have to type an identifiable name. The safest option is to type 'LOAD "THE GOTO EXAMPLE"' (or whatever) in full, but in fact you don't need to give the full name: you can just give the first couple of characters, enough to identify the program you want. In this case you could type 'LOAD "THE" ' and the 64 would go off searching the tape to load the first program it comes across whose name starts with 'THE'. Or you could even ask it to 'LOAD "T"' and get the same program, if that's going to be the first one it picks up with a T as the first letter of its name.

Looking After Tapes and the Cassette Unit

You can buy tapes that are supposed to be special 'data' cassettes for computers, but in fact any decent-quality audio cassettes will do. Buy short tapes, though – C15s or C30s are preferable. Longer tapes are thinner and they are more prone to stretch slightly with use: besides, with the shorter cassettes it doesn't take so long to fast-forward to the program you want to load.

When you start using a new tape, run it 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.

Store your cassettes sensibly. That means 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. Don't take the risk of putting them on or near anything that generates a magnetic field – it can screw up anything that's recorded on the

tape. This includes your TV, by the way; television 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 in it from time to time by using one of the proprietary tape cleaners that you can get from your local record store: these tend to be cassettes that have a mildly abrasive tape in them which scrubs the accumulated detritus off the bits that actually touch the tape.

It's worth demagnetizing the heads occasionally, too. Rubbing against a magnetic surface such as a cassette tape does cause a little magnetism to build up on the heads, and that can lead to errors. To demagnetize them you buy a little cassette-like thing that runs off a couple of watch batteries: this you just insert, press PLAY, and leave for a few seconds. Alternatively, you can use a hand-held gismo that you plug into a wall socket and gently bring towards the opened tape deck; keep all prerecorded tapes well away unless you want to wipe them too. Again, a record shop should be able to sell you either kind of demagnetizer.

Section Two

Writing Programs

10. First Steps in Programming

Back in Chapter 7 you wrote a program. It used PRINT and LET (though we don't bother typing the whole command 'L–E–T'); and it introduced the idea of variables, labels by which you can refer to different things – words, numbers, phrases, arithmetical expressions.

As it happens, that little program shows two of the key attributes of programming. First, something happens – you can persuade the computer to do something (displaying a message on the screen in this case). And second, in a program you can tell the computer what to do *without necessarily telling it what in fact it is to use to do it.*

In Chapter 7 the variables were given specific meanings. But if you altered those meanings, if you told the 64 the labels referred to something else altogether, the computer would still go through the same actions with the new information.

Try changing any of those variables, for instance to set up the program like this:

```
1Ø X$="DEAR MILKMAN"
2Ø ZZ$="NO MILK PLEASE ON "
3Ø A$=" (SORRY I DON'T KNOW YOUR NAME) "
4Ø D$=" UNLESS IT'S "
5Ø PRINT X$+A$+ZZ$+"WEDNESDAY"+D$+"RAINING"+A$
```

Now try RUN. See? Line 50, which puts the message on the screen, works in exactly the same way as before and prints whatever the variables refer to in the same order . . . but now those variables mean different things, so the message is different.

That's one of the main points about programming. You can write a program to perform some action, like a PRINT or some arithmetic, *independently* of the material it has to work on.

Input

Let's make that a bit more obvious. There is a Basic command called INPUT which will let you put in any acceptable information, and it works like this:

```
1Ø INPUT X$
2Ø INPUT ZZ$
3Ø INPUT A$
4Ø INPUT D$
5Ø PRINT X$+A$+ZZ$+"WEDNESDAY"+D$+"RAINING"+A$
```

If you now RUN the program, you'll see a question mark appear. That means the computer has reached the first INPUT and it's 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.

You'll get the question mark for the next INPUT, so type in anything you want and hit RETURN. Do that until all four INPUTs have been dealt with; when you hit RETURN the last time the screen will display the message you've created. What you typed in response to the first INPUT was labelled as 'X$', so when the program assembled its message it used your INPUT instead of 'X$' in line 50. And so on, for the other four INPUTs.

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 YOUR NAME' or 'WHAT IS THE WEATHER LIKE'. You can do this by having the screen PRINT out a question and following it by an INPUT:

```
1Ø PRINT "HELLO. WHO ARE YOU"
2Ø INPUT A$
```

Using PRINT is OK for requesting an INPUT like this. There's a neater way, though. You can put the question actually in the INPUT command:

```
1Ø INPUT "HELLO. WHO ARE YOU"; A$
```

You can't miss out that semi-colon after the quotes which close the message. Note that 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. 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?

There are two other things to watch about INPUT. First, 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 rude 'REDO FROM START' message. On the other hand, if you are 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 other thing is a restriction on the length of the message you can have with an INPUT statement, 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 twenty-two characters long. And what happens if it is? Try this:

```
1Ø INPUT "PLEASE TYPE IN ANY RESPONSE TO
   CHECK OUT WHAT HAPPENS WHEN THE PROMPT
   LINE IS TOO LONG"; A$
2Ø PRINT A$
```

Another Program

A bloke at the University of Pennsylvania has devised a formula that guarantees to show you how long it will take to find True Love (no, I'm not making this up).

$$M = \frac{0.7}{O \times S \times A \times D \times I}$$

This should tell you the number of months (M) that you'll have to wait before Mr or Ms Right turns up.

O is 'opportunity', the number of eligible candidates to whom you're (as it were) exposed each month; S is 'selectivity', the percentage of them that you find desirable, and A is 'approaches' – the percentage of them that you actually ask for a date. The gloomy news is D for your 'desirability': it's the percentage who say yes and agree to risk going out with you. I is 'intimacy', which isn't necessarily what you might think: it's the percentage of dates that lead to a close relationship of six months' duration or more.

The 0.7 stands for the 70 per cent probability that the equation will work. Maybe it's not so guaranteed after all.

So let's write a program to find out how long you (or anyone else) will have to hang around before being smitten. The calculation itself is easy enough to express:

```
2ØØ M = Ø.7/(O*S*A*D*I)
```

The brackets are in there to force the 64 to do the multiplication before the division – mathematical operations on the computer have an 'order of precedence' just as they do in any arithmetic. To be on the safe side I tend to use brackets all the time to indicate which bit of a calculation I want done first. Besides, it makes things so much clearer when you're checking through a program.

O K, you know how to calculate M. We might as well print the answer:

```
23Ø PRINT "TRUE LOVE SHOULD ARRIVE IN " M
    " MONTHS"
```

Before you can get there, though, the program needs some information to work with. So let's have a bunch of INPUT statements:

```
1ØØ INPUT "OPPORTUNITY"; O
11Ø INPUT "SELECTIVITY"; S
12Ø INPUT "APPROACHES"; A
13Ø INPUT "DESIRABILITY"; D
14Ø INPUT "INTIMACY"; I
```

It makes sense to use widely spaced line numbers. Here I've got line 200 onwards doing the calculation and printing the results; line 100 onwards asks for the INPUT. Since Basic will work through the program in line-number sequence anyway, it doesn't matter what order you type in the lines themselves – so long as the line *numbers* you use are in the right order.

At this point you could now RUN the program, since you have everything

you need – some input, the calculation, and a way of getting the result. But first let's tidy things up a bit.

The following lines clear the display and move the cursor down a couple of lines from the top of the screen (that's what happens if you use print with nothing but a colon following it):

```
1Ø PRINT "[CLR]"
2Ø PRINT: PRINT
```

(I'm putting this preliminary stuff in as line 10 onwards, see?) Let's get a bit fancier and put a title on the screen too, underlining it with a neat row of hearts.

```
3Ø PRINT "TRUE LOVE TESTER"
4Ø PRINT "♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥"
```

And why don't we ask for the name of the hapless individual who's trying to find out when the lightning will strike:

```
5Ø INPUT "WHAT'S YOUR NAME"; A$
6Ø PRINT
```

Line 60 pulls the cursor down again to leave a space before the INPUT prompts appear.

Now the program knows a name, it could use that in the final 'number of months' message. Type 'LIST 230' and you should get the PRINT line back to alter.

```
23Ø PRINT "TRUE LOVE SHOULD ARRIVE FOR " A$
    " IN " M " MONTHS"
```

While we're making the screen look neat and tidy, we might as well clear away everything you've typed in response to the INPUT prompts (that way your sneaky little sister won't be able to report your performance – or lack of it – to the rest of the junior school playground):

```
22Ø PRINT "[CLR]": PRINT: PRINT
```

And we might as well do the formal thing and put in an END statement. That isn't always essential, but it's good practice to indicate the very last statement in a program. I generally use a high line number to leave some room before it for amendments and new lines:

```
999 END
```

You'd better type 'LIST' now and check that everything's there.

```
 1Ø PRINT "[CLR]"
 2Ø PRINT: PRINT
 3Ø PRINT "TRUE LOVE TESTER"
 4Ø PRINT "♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥♥"
 5Ø INPUT "WHAT'S YOUR NAME"; A$
 6Ø PRINT
1ØØ INPUT "OPPORTUNITY"; O
11Ø INPUT "SELECTIVITY"; S
```

```
12Ø INPUT "APPROACHES"; A
13Ø INPUT "DESIRABILITY"; D
14Ø INPUT "INTIMACY"; I
2ØØ M=Ø.7/(O*S*A*D*I)
22Ø PRINT "[CLR]": PRINT: PRINT
23Ø PRINT "TRUE LOVE SHOULD ARRIVE FOR " A$
    " IN " M " MONTHS"
999 END
```

If it is, you can try RUNning the program and see how you do.

Making Decisions . . .

A really important aspect of writing programs is the way a program can make decisions for you and take different courses of action depending on circumstances. For this Basic provides the helpful IF . . . THEN statement. IF such-and-such is the case, THEN do something.

As an example, let's put in a couple of new lines:

```
21 PRINT "I HOPE YOU ARE READY TO RISK
   THIS"
22 INPUT "ARE YOU"; B$
23 IF B$="N" THEN 999
```

These lines let you chicken out before getting to the awful truth. It asks for a confirmation that you're ready to proceed. Line 23 looks to see whether you've typed 'N' (for 'no'); and IF you have, THEN the program goes to line 999. Since the statement at 999 is END, the program will finish there.

Unless it finds a statement that switches the program to a different line number, Basic will try to deal with the lines in numerical sequence. So if the INPUT wasn't 'N' and you don't zip off to line 999, the next statement Basic finds will be executed. That's on line 30.

We could also use IF . . . THEN to do some checks on the other INPUTs. Since they are supposed to be percentages, the number input shouldn't be greater than 100. So we could have a line testing that. LIST line 100

```
1ØØ INPUT "OPPORTUNITY"; O
```

and add a new line:

```
1Ø5 IF O>1ØØ THEN 1ØØ
```

What's happening in line 105 is a check on the response to 'OPPORTUNITY?' IF the number typed in (which is now labelled as variable 'O') is greater than 100 (that > symbol is the 'greater-than' sign) THEN the program should go back to line 100 and request the INPUT again. If it isn't greater than 100 the input will be OK and the program can go on to the next line number in sequence.

You can also build a message into your program to tell the user why it went back to the OPPORTUNITY prompt. You decide that it's simpler to check whether the INPUT is correct and skip the error message if it is:

```
105 IF 0<101 THEN 110
106 PRINT "IT'S SUPPOSED TO BE A PERCENTAGE,
    DUMMY"
107 PRINT "SO IT CAN'T BE MORE THAN 100"
```

Now you're looking at the '0' to see if it is 100 or less (the < means 'less than') and skipping to the next INPUT command – line 110 – if this INPUT is OK. But if it isn't all right, the program will execute the next line . . . which is the error message.

Alternatively (and preferably), you can put the error message on the same line. The IF . . . THEN statement doesn't have to be followed by a line number – practically any Basic statement can come in there. So this will work:

```
105 IF 0<101 THEN PRINT "IT'S SUPPOSED TO
    BE A PERCENTAGE, DUMMY"
```

If you want to try this, don't forget to delete lines 106 and 107 by typing '106' and '107'. Otherwise it would be the next line that Basic encountered, and the program will print the 'duff input' message whether or not you got it right.

. . . and Acting on Them

There are other things you can do after an IF . . . THEN. For instance, you can put that error message somewhere else in the program. Retype line 105 to read:

```
105 IF 0>100 THEN 1000
```

and have a new line numbered 1000:

```
1000 PRINT "IT'S SUPPOSED TO BE A
     PERCENTAGE, DUMMY"
1005 PRINT "SO IT CAN'T BE MORE  THAN 100"
```

and add another line:

```
1010 GOTO 100
```

Now what's happening in line 105 is a check that sends the program off to line 1000 to print the message if the number typed is too big. If it isn't greater than 100 the input will be OK and the program can go on to the next statement which happens to be on the next line number in sequence.

That GOTO in line 1010 is another Basic statement, and it's another way of forcing the program to GO TO a particular line rather than carry on to the line number next in sequence (it can actually be written as 'GO TO' rather than 'GOTO', if you're happier with that). If it wasn't there, the next line to be executed would be the END statement at 999 and the program would stop; GOTO gives you a way to get back into the swim of things.

Subroutines

If your program has a particular clump of instructions it uses more than once to do the same kind of thing at different points within the program, like the last method of printing an error message suggested, there's a neat and simple way of saving time (yours) and space (the 64's in memory). Those instructions *could* be written out every time they're needed: or they could be given only once with the program jumping away to them when it needs to – the repeated instructions could be put in what's called a 'subroutine'.

In other words, going to a subroutine is rather like using GOTO. The difference is that to get back you don't have to specify the line number you want to go back to. So instead of a GOTO you say GOSUB followed by a line number (meaning GO to a SUBroutine that starts at the line number specified).

Let's have a subroutine in our program instead of the instruction to go to line 1000 after the IF statement. Put in this lot:

```
 1Ø5 IF O>1ØØ THEN GOSUB 1ØØØ: GOTO 1ØØ
 115 IF S>1ØØ THEN GOSUB 1ØØØ: GOTO 11Ø
 125 IF A>1ØØ THEN GOSUB 1ØØØ: GOTO 12Ø
 135 IF D>1ØØ THEN GOSUB 1ØØØ: GOTO 13Ø
 145 IF I>1ØØ THEN GOSUB 1ØØØ: GOTO 14Ø
1ØØØ REM subroutine to print message
1ØØØ PRINT "IT'S SUPPOSED TO BE A
     PERCENTAGE, DUMMY"
1ØØ5 PRINT "SO IT CAN'T BE MORE  THAN 1ØØ"
1Ø1Ø RETURN
```

Now the message is being printed by the subroutine. The new lines 105 to 145 are checking whatever you typed in response to each of the INPUT prompts, and if it's not OK, the program GOes to the SUBroutine starting at line 1000 and prints the 'duff input' message. It's a good idea to start subroutines with a REM line to remind you what's happening where in your program. REM denotes a REMark, a message you can put into a program which won't be treated as an executable program statement – in other words, it's there to make things clearer for anyone reading the program. Obviously the line itself won't affect the program, since Basic doesn't try to execute anything after the REM statement – but Basic does recognize the REM, so you can go to a line containing a REM from an IF . . . THEN instruction, from a GOTO, or from a GOSUB.

Note that you've got to have the word GOSUB in there before Basic will realize that you want to go to a subroutine: and within the subroutine there's the command RETURN, which is the magical part – it returns you to the statement following the one containing the GOSUB. Thus, if you mistyped the 'opportunity' entry and put in a number greater than 100, the program would go to the subroutine at 1000 and print the error message and then return you to the statement following the GOSUB, which is a GOTO which switches back and repeats the INPUT line.

One thing about subroutines: it makes sense to put them *after* an END statement, as I have done here – remember the END at line 999? That's because

there's nothing inherently different about the lines in a subroutine; so Basic will just try to execute them in sequence when it reaches them . . . unless it's stopped by an END.

Go on, try deleting line 999. Now type RUN. Everything works, right? Except that the error message is printed at the end and you get a 'RETURN WITHOUT GOSUB' error message . . . If line 999 isn't there, the program will go through the INPUTs until you've got it right; then it will work out the formula; then it will print the answer. And then it will go to the next statement lines it finds, at 1000 onwards, and try to execute them too. But this time it comes across that RETURN statement without there being a GOSUB anywhere which switched the program to the subroutine. So it's baffled, and it lets you know it.

Loops

If you want to keep your True Love Tester, SAVE it on cassette. Otherwise hit RUN/STOP and RESTORE and type 'NEW': we're about to introduce another of the very helpful commands in Basic.

Before you've written many programs you'll find that you frequently need some kind of *loop* with a 'counter' in it: each time it goes round the loop, your program adds one to a variable you've set up as a counter and tests it with an IF . . . THEN to see whether it's reached a required total or not. If it hasn't, the program loops back to go through whatever it was doing once again; if the total *has* been reached, it stops looping around and goes on to something else.

This is such a common requirement in Basic that a special command is provided for it, effectively reducing the number of instructions you have to write. It's the command FOR, and you use it like this:

```
FOR variable=start of count TO end of count
```

For instance, you might want the program to loop around ten times:

```
FOR N=1 TO 1Ø
```

Somewhere following that, you'd have an instruction that adds one to the value currently in N; and at the other end of the sequence of instructions to be repeated you need a NEXT command that sends the program looping around again if N hasn't reached 10 yet.

Suppose you want to print 'HELLO' ten times. Your program might look like this:

```
1ØØ N=Ø
11Ø PRINT "HELLO"
12Ø N=N+1
13Ø IF N=1Ø THEN 15Ø
14Ø GOTO 11Ø
15Ø END
```

This uses N as a counter, with line 100 making sure that it's set to zero before you start. The program then prints one HELLO and adds one to the value of N – first time round, that will be 0; so the first time line 120 is executed N will

contain 1 when the IF . . . THEN instruction is reached. Since N does not equal 10, the program goes back to print another 'HELLO'. When it has gone round ten times the IF . . .THEN takes it to an END statement.

FOR and NEXT do it all so much more simply:

```
100 FOR N=1 TO 10
110 PRINT "HELLO"
120 NEXT
```

You can see it working if you replace line 110 by this:

```
110 PRINT "HELLO"; N
```

When you RUN it now, you'll see how each pass through the loop prints a 'HELLO' . . . because you're also printing the value of N at the time.

As an example of how the variable in the FOR statement can actually be involved in the computation, here is a way of printing out a multiplication table, in this case the seven times table:

```
10 FOR N=1 TO 12
20 PRINT 7*N
30 NEXT
```

On RUN this gives:

```
7
14
21
28
35
42
49
56
63
70
77
84
```

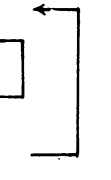
To make the output more meaningful, line 20 could be:

```
20 PRINT N " TIMES 7 MAKES " 7*N
```

Try it. You could also try other tables than seven.

You can get even more clever and make the computer display all the multiplication tables from two to twelve with one program. This involves putting one FOR loop inside another, called *nesting* loops. The easiest way of seeing how this works is by looking at the program:

```
10 FOR M=2 TO 12
20 FOR N=1 TO 12
30 PRINT N " TIMES " M " MAKES " N*M
40 NEXT N
50 PRINT
60 NEXT M
```



The arrows show you how the inner N loop goes round on itself, and the outer M loop does likewise. Each pass round the M loop involves twelve passes round the N loop.

The 'N' and 'M' at the end of the NEXT commands are not strictly necessary for the computer, but they help you keep track of what is going on. The PRINT command with nothing after it at line 50 inserts a blank line in between each table to make them easier to read.

Of course, the tables run off the top of the screen faster than you can read them properly; but you get the idea. You can interrupt with the STOP button, and you can slow things down by pressing CTRL.

Sometimes you have a loop in which the variable needs to be increased by something other than 1 each time it goes round. In this case you use the word STEP at the end of the FOR instruction, followed by a number giving the required increment. This can be a whole number or a fraction, or it can even be negative. Suppose you want a table showing British money converted to American, running from 1p to £5.00, in steps of 1p (£0.01). Assuming the exchange rate is $1.45 to the pound, the program would be:

```
10 FOR L=0.01 TO 5 STEP 0.01
20 PRINT "£" L "= $" L*1.45
30 NEXT
```

FOR loops are sometimes used to do nothing at all – just to waste time. In doing animation on the screen for instance, it is often necessary to draw something, wait a fraction of a second, then draw it again in a slightly different position, and so on. The instructions

```
FOR N=1 TO 500
NEXT
```

will put a short but noticeable delay into a program while the computer goes round the FOR loop five hundred times. Changing the number obviously changes the length of the delay.

Here's another way you can use a timing loop:

```
100 PRINT "THIS PROGRAM PRINTS"
110 PRINT
120 FOR T=1 TO 999: NEXT
130 PRINT "V ";
140 FOR T=1 TO 999: NEXT
150 PRINT "E ";
160 FOR T=1 TO 999: NEXT
170 PRINT "R ";
180 FOR T=1 TO 999: NEXT
190 PRINT "Y ";
200 FOR T=1 TO 999: NEXT
210 PRINT
220 FOR T=1 TO 999: NEXT
230 PRINT "S ";
240 FOR T=1 TO 999: NEXT
250 PRINT "L ";
260 FOR T=1 TO 999: NEXT
```

```
27Ø PRINT "O ";
28Ø FOR T=1 TO 999: NEXT
29Ø PRINT "W ";
3ØØ FOR T=1 TO 999: NEXT
31Ø PRINT "L ";
32Ø FOR T=1 TO 999: NEXT
33Ø PRINT "Y ";
34Ø FOR T=1 TO 999: NEXT
35Ø PRINT
36Ø END
```

Try RUNning that. What slows down the printing is the way the program adds one to the timer variable 'T' 999 times before it goes on to the next line. (Don't forget the semicolons – they're there to put the next PRINTed character on the same line as the last. Without them, each character would appear on the next line down the screen.)

This happens to be a classic case for a subroutine, actually. Rewrite the program like this:

```
1ØØ PRINT "THIS PROGRAM PRINTS"
11Ø PRINT
12Ø GOSUB 1ØØØ
13Ø PRINT "V ";
14Ø GOSUB 1ØØØ
15Ø PRINT "E ";
16Ø GOSUB 1ØØØ
17Ø PRINT "R ";
18Ø GOSUB 1ØØØ
19Ø PRINT "Y ";
2ØØ GOSUB 1ØØØ
21Ø PRINT
22Ø GOSUB 1ØØØ
23Ø PRINT "S ";
24Ø GOSUB 1ØØØ
25Ø PRINT "L ";
26Ø GOSUB 1ØØØ
27Ø PRINT "O ";
28Ø GOSUB 1ØØØ
29Ø PRINT "W ";
3ØØ GOSUB 1ØØØ
31Ø PRINT "L ";
32Ø GOSUB 1ØØØ
33Ø PRINT "Y ";
34Ø GOSUB 1ØØØ
35Ø PRINT
36Ø END
1ØØØ REM delay loop
1Ø1Ø FOR T=1 TO 999: NEXT: RETURN
```

It's neater and easier like that.

11. More Programming

You will find a summary of everything Basic can do in Chapter 24. Meanwhile, this chapter introduces a collection of the more important elements of Basic programming for you to use in real programs.

Functions

Some of the operations the 64 can carry out take a different form from the commands you've been using up to now. These are called *functions*; they are always used as part of an ordinary instruction, rather than as instructions on their own. Some of the functions perform mathematical or trigonometric operations; others work on strings; a few (a very few!) provide access to extra facilities of the computer. And you can set up your own functions, too.

Each function consists of a function name followed by brackets containing whatever it is the function is to operate on (this is called the function's *argument*). Take the Integer function, for instance:

```
INT(x)
```

where *x* is some number. The function quite simply looks at the number inside the brackets and discards anything following a decimal point, retaining only the whole number – the 'integer' part. So a command like this

```
PRINT INT(87.8)
```

gets the reply:

```
87
```

And if you were to try

```
PRINT INT(Ø.15673)
```

you'd get:

```
Ø
```

INT is useful in programming, for instance, to round off numbers or to find out whether one number is divisible by another. In the latter case you'd do the division and test to see whether the integer part of the answer is the same as the answer itself – if it isn't, there's a fraction left over and the numbers you're testing are not divisible.

The SQR function finds the square root of its argument:

```
PRINT SQR(36)
6
```

Several of the functions carry out mathematical tasks like this – finding sines, cosines, logarithms and the like. But unless you aim to do a lot of maths on your 64 it's unlikely that you will need to use them, though you'll find a complete list of them in Chapter 24.

There is one function that you might like to pay some attention to, however. Actually, it's not really a function as such: it's a command that lets you define your own functions. This is handy when a lengthy mathematical formula is used several times in a program. Each time you want to do the calculation you could work it out, or you could have a GOSUB that takes you to a subroutine which does the job. But it's a lot easier to set the calculation up as a function and do it that way.

The command looks like this:

```
DEF FNa(b)=formula
```

It DEFines a FuNction labelled with the variable name *a*, so you can subsequently refer to it; *b* is a second variable that you might want to use in the formula (there again, you might not – but *something* has to be there in those brackets, even if it's not used); and the formula itself follows the equals sign. For instance:

```
2ØØ DEF FNA(R) = 2*π*R
```

That's the formula to find the circumference of a circle, remember? Now you use the new function in exactly the same way as any of the 64's predefined mathematical functions. Somewhere else in the program there might be a need to find out a circle's circumference and PRINT the result, in which case you'd have a line reading

```
57Ø PRINT FNA(24.33)
```

That tells the 64 to use the function you've defined as A and apply the value inside the brackets to the formula.

That value can itself be a variable. For instance, here's a way of calculating the value of the dollar against the pound and vice versa for any particular amount:

```
1ØØ DEF FNA(P) = R*P
11Ø DEF FNB(D) = D/R
2ØØ INPUT "EXCHANGE RATE";R
21Ø INPUT "$ OR £";A$
22Ø INPUT "AMOUNT";N
23Ø IF A$="$" THEN 4ØØ
3ØØ PRINT "CONVERTING POUNDS TO DOLLARS"
31Ø PRINT FNA(N)
32Ø GOTO 5ØØ
4ØØ PRINT "CONVERTING DOLLARS TO POUNDS"
41Ø PRINT FNB(N)
5ØØ INPUT "ANOTHER";Z$
51Ø IF Z$="Y" THEN 21Ø
999 END
```

See how it works? You define two functions for the two calculations, dollars to pounds and pounds to dollars; then the program asks what the exchange rate is today, which calculation you want, and how much you're talking about. If it's dollars to pounds, it'll go to line 400 and use the function labelled B; if it's pounds to dollars, it carries on with line 300 and uses FNA. Either way the argument for the function is the variable N, the amount of money you want converted. At the end of the calculation the result is printed and the GOTO takes the program to line 500 where you're asked if you want another run. Answering 'N' or 'NO' means the next statement in sequence is executed, and since that's END, the program will stop there.

String Functions

The maths functions do have their value, but you'll probably find that you make a lot more use of the character string functions. These enable you to take sequences of words apart and process them in elaborate ways.

One of them is LEN, for 'length'. This gives you a number (not a string) telling you how many characters there are in the string you specify:

```
A$="WINSTON CHURCHILL"
N=LEN(A$)
PRINT N
```

The computer replies:

```
17
```

Another string function is LEFT$. This has *two* arguments inside the brackets, separated by a comma. It takes the left-hand end of the string named in the first argument and gives you another string consisting of however many characters are specified by the second:

```
PRINT LEFT$(A$,7)
```

The machine replies:

```
WINSTON
```

Likewise, MID$(A$,*m*,*n*) gives you a string *n* characters long starting from the *m*th character of string A$:

```
PRINT MID$(A$,5,6)
TON CH
```

There's a RIGHT$ function, too, and that obeys the same rules. The command

```
PRINT RIGHT$(A$,4)
```

will pick out the rightmost four letters of whatever's in A$, so you should get:

```
HILL
```

A good way to see how these operate is to write a short program that takes someone's name and prints it out backwards. You'll need the MID$ to take just one character at a time because that's the string function which can select any

number of characters from anywhere in the string; you could have LEN to find out how many characters there are in total and use that number in a FOR . . . NEXT loop to pick out and PRINT characters one by one starting from the right.

How Much Memory?

There's a collection of functions inside the 64 that provide ready-to-use information. FRE, for example, will tell you how much memory you've got left. Use it like this:

```
PRINT FRE(Ø)
```

and you'll get back a number which indicates how many bytes of memory are still available to you. (All this only works if there are less than 32767 bytes free, otherwise a negative number is returned, to which you have to add 65536.)

This is an interesting one, since while it does have an argument – and that argument *has* to be there – the argument isn't actually used at all. It's a 'dummy' argument. A zero is generally put in there, but it could be any number.

Telling the Time

The TI$ function doesn't have an argument at all. It's an automatic clock: if you set it to zero at the start of the program like this:

```
1Ø TI$="ØØØØØØ"
```

and have PRINT TI$ as the last statement before an END, you'll be able to find out how long the program took to run. The six digits reading from left to right are hours, minutes and seconds respectively, two for each.

You can use string functions to present that a bit more impressively:

```
6ØØ PRINT "HOURS:  " LEFT$(TI$,2)
61Ø PRINT "MINUTES:  " MID$(TI$,3,2)
62Ø PRINT "SECONDS:  " RIGHT$(TI$,2)
```

But if you're using TI$ to find out how long a program takes to RUN, don't forget that executing these instructions will themselves take some time!

Random Numbers

TI$ can be used with another function, TI, which counts in sixtieths of a second (it counts how long the computer has been switched on, or the time since you set TI$ – setting TI$ sets the TI clock too). A frequent use for TI is with another function, RND. This produces a *random number*, a more or less arbitrary value; you might need a number like that for games of chance like dice-throwing or deciding more or less at random when the space invaders peel off to come down on your laser base.

The number you get from RND will be between 0 and 1. RND is used with an argument like this:

```
RND(X)
```

where X can itself be a number, or a variable, or any standard Basic expression combining either or both of those. You could put in a number for yourself; if you put in zero, the 64 will find a number of its own to use as a starting point (the *seed*, it's called) for the random number generation. If X is TI, though, the 64 will figure out a random number by using the TI value as a start. To see what kind of random numbers you can get, try this:

```
1ØØ PRINT "FIRST LOT"
11Ø FOR R=1 TO 1Ø
12Ø PRINT INT(RND(Ø)*1ØØØ)
13Ø NEXT
2ØØ PRINT "SECOND LOT"
21Ø FOR R=1 TO 1Ø
22Ø PRINT INT(RND(TI)*1ØØØ)
23Ø NEXT
3ØØ PRINT "THIRD LOT"
31Ø INPUT "START NO";N
32Ø FOR R=1 TO 1Ø
33Ø PRINT INT(RND(N)*1ØØØ)
34Ø NEXT
999 END
```

Arrays

It often happens in programming that we have to handle a large number of items of data that are separate but closely related in some systematic way. We may want to store in the computer a list of things, or a table of information, and we want to be able to treat this as one thing, rather than as separate variables each with a different letter label (A,B,C,D . . .). A shoe salesman might for instance have a list of prices:

Style 1 costs £9.50
Style 2 costs £26.45
Style 3 costs £17.95
Style 4 costs £11.75
etc.

To give every item in this list a separate variable name would be a nuisance. What the salesman would like is to be able to refer to each entry by its position in the list: $Price_1$, $Price_2$, $Price_3$, and so on. The small number is known as a *subscript*. Basic allows you to do this by putting the subscript in parentheses: P(1), P(2), P(3). So the list can be entered into memory like this:

```
P(1)=9.5Ø
P(2)=26.45
P(3)=17.95
etc.
```

Then the command

```
PRINT P(2)
```

gets the answer

```
26.45
```

The clever thing about this is that a *subscript can itself be a variable.* To see how this works, we can write a program to retrieve any required price from memory. The program asks for the style number wanted, looks it up in the list, and prints the answer:

```
1Ø PRINT "WHICH STYLE"
2Ø INPUT S
3Ø PRINT P(S)
```

Look at that carefully to see how it works. If the user is interested in, say, style 3, he types 3 in reply to the INPUT request. The program sets S equal to 3, and in line 30 puts 3 inside the brackets, so it prints P(3).

Here is a program to print out all the prices for styles 1 to 10:

```
1Ø FOR S=1 TO 1Ø
2Ø PRINT "STYLE " S " COSTS £" P(S)
3Ø NEXT
```

In Basic a list can easily become a *table.* Suppose each shoe style comes in a range of sizes, and each size can be a different price. On paper this would be shown as a table, like this:

	Size				
	1	2	3	4	5
Style 1	9.50	9.85	10.25	10.45	10.80
2	26.45	26.95	27.50	27.95	28.70
3	17.95	18.30	18.95	...	...
4	11.75	12.15	...	...	...
...	...	...	...	...	...

In the computer this is easily done by having *two* subscripts, separated by a comma: P(2,5). Either or both of these can be variables.

Then suppose that the price also depends upon the *colour* of the shoe. This means adding a third dimension to the table, something that is difficult to do on paper but with the computer all you have to do is add a third subscript: P(2,5,4). You can even have a fourth or fifth dimension. These become a bit mind-blowing because you cannot visualize them – they cannot exist in the physical world but the computer accepts them quite happily. You can have as many dimensions as you like, with the limitation that you very soon run out of memory in the computer.

Lists and tables such as these are known as *arrays.* Strings as well as numbers can be held in arrays. Suppose the shoe salesman wants to include the name of each style in his computer program. He simply adds another list, in this case a *string array*:

```
N$(1)="BROGUE"
N$(2)="JACK BOOT"
N$(3)="BROTHEL CREEPER"
N$(4)="TRAINING SHOE"
etc.
```

You might think that each string in a string array would have to be the same length, but this is not the case. You must remember, though, that you cannot mix numerical arrays and string arrays.

DIM

One problem with arrays is that they can take up a great deal of space in memory, depending on how large a range of subscripts you use. The computer has to leave room for whatever subscripts may come along. Normally it assumes that no subscript is going to be larger than 10. If you are going to use subscripts larger than 10, you must tell the machine beforehand, by specifying the dimensions of your array with the instruction DIM. For a two-dimensional array called T with the first subscript going up to 40 and the second up to 15, you say:

```
DIM T(4Ø,15)
```

READ and DATA

An array cannot be written as such as part of your program; any program which uses an array has to *construct* it with normal Basic commands before it does anything else. In the examples so far each element of an array has been specified separately; the alternative is to put the items of data in DATA statements, each separated by a comma. The command READ then takes one item of data at a time from the DATA statements and stores it. This part program replaces the list on p. 60.

```
1Ø FOR S=1 TO 2Ø
2Ø READ P(S)
3Ø NEXT
4Ø DATA 9.5Ø,26.45,17.95,11.75
5Ø DATA . . .
```

You will find this is the easiest way of filling in a list or table of any length.

Practice Makes Perfect

Now would be a good time for you to try writing some more programs on your own. Several suggestions for these are made here; possible solutions for all these are given at the end of the chapter. You can either look at these solutions right away, or try on your own first and then compare your programs with those given. These is no one right answer for any of these suggested programs – all that matters is that they work!

It is a good idea to write your programs out on paper first before you try

typing them into the computer. You will find that pencil and paper are that much less inhibiting to your thoughts than the keyboard.

Teacher

This is a program to test someone's arithmetic. In its simplest form, it can take two random whole numbers between 1 and 100 and print them out. The user is asked to type in the sum, and the computer checks to see if the answer is correct. (The computer of course never gets it wrong!) If the answer is right, another sum is provided; if it is wrong, the 'pupil' is asked to try again.

The random numbers are of course generated by the RND function, like this

```
X=INT(RND(1)*1ØØ)+1
```

to give numbers ranging from 1 to 100 (or any other range you choose). The program could also of course test subtraction, multiplication and division. It could even choose randomly between these four, by getting another random number in the range 1 to 4.

Perpetual calendar

This is an example of using lists, together with READ and DATA statements. The program will tell you the day of the week for any date between the years 1753 to 2199 (Gregorian calendar). The procedure is as follows:

1. Take the last two digits of the year.
2. Take the number from step 1 and divide by 4, disregarding any remainder.
3. Take the 'key number' of the month from this table, *but* if the year is divisible by 4, let the key numbers for January and February be 0 and 3 respectively, instead of 1 and 4.

Jan = 1	May = 2	Sept = 6
Feb = 4	June = 5	Oct = 1
Mar = 4	July = 0	Nov = 4
Apr = 0	Aug = 3	Dec = 6

4. Take the key number of the century from this table:
   1700s = 4
   1800s = 2
   1900s = 0
   2000s = 6
   2100s = 4
5. Take the day.
6. Add together the numbers from steps 1,2,3,4 and 5 and divide by 7.
7. Take the *remainder* from the division in step 6. This gives the day of the week thus:
   0 = Saturday, 1 = Sunday, 2 = Monday, 3 = Tuesday,
   4 = Wednesday, 5 = Thursday, 6 = Friday.

There is plenty of use of the integer part function in this program. To divide while disregarding any remainder, you simply take the integer part of the

answer. To find the century, divide by 100, take the integer part, and multiply by 100 again. Thus 1865 becomes 18.65, then 18, then 1800. Subtract that from the complete year to get the last two digits: 1865 – 1800 = 65.

To get a remainder, take the integer part of the answer, multiply by the divisor, and subtract from the dividend. Thus 17 divided by 5 is 3.4, the 3.4 becomes 3, multiplying by 5 gives 15, 15 subtracted from 17 is 2. As we know already, 17 divided by 5 equals 3, and the remainder is 2.

Dice game

With this program you can play a two-dice game against the computer. The game is known in the United States as 'craps', and is familiar to anyone who has seen *Guys and Dolls*. You roll the dice, add the numbers together, and if you get 7 or 11 you win. With 2 ('snake eyes'), 3 ('Holy Joe') or 12 ('boxcars') you lose. With any other number you roll again, and keep on rolling until you get either 7, in which case you lose, or the number you rolled in the first place, in which case you win.

You simulate the roll of one die by the command

```
X=INT(RND(1)*6)+1
```

Do this again for the second die and then add the two scores together.

Prime factors

A prime number is one which is not divisible by anything other than itself and 1. It is useful to be able to divide up a non-prime number into its prime factors, such as 24 = 2 × 2 × 2 × 3. With large numbers this is extremely tedious, but a computer can do it easily, and in fact computers have been used in some important explorations of the theory of numbers.

The procedure is to try dividing the number which you wish to factorize by each prime number in turn: 2,3,5,7,11,13,17 and so on. You keep dividing out the prime factors until they will go in no longer, and then proceed to the next higher prime number. You can stop as soon as the prime number you are trying is larger than the square root of whatever is left, because there cannot be more than one factor larger than the square root.

There is a trick used in the suggested program for this that is a good example of how computers do things differently from humans. Strictly the machine ought to have a list of all the prime numbers up to however large a number you might want to factorize, but this would be a nuisance to construct and would occupy a lot of space. The program gets around this by dividing by *every odd number* from 3 upwards (there is no point in dividing by even numbers because they are obviously not prime, apart from 2). Of course plenty of odd numbers are not prime, but since the dividing goes upwards from the smallest numbers, all the lower primes will have been divided out by the time an attempt is made to divide in a non-prime number, so that division will always fail. Thus the computer does plenty of futile divisions, but a program which did not do that would have to be much more complicated. The dumb computer does work to save the human programmer effort, and that must be a good thing.

Sorting

Arranging things in order is one of the most useful tasks a computer can carry out. Plenty of sorting programs exist already, but you might like to try writing a very simple one for fun. It should accept a series of numbers from the keyboard and print them out again in ascending order. This will mean using a list in which the numbers are stored as they come in.

The simplest way of arranging the list in memory in the correct order is to make multiple passes through the list, looking at adjacent entries in turn, and asking, 'Are these two in the correct order?' If not, the program interchanges them. You have to go through the list as many times as there are items in it. This is called a *bubble sort*, because with each pass through, the larger items move towards the top. There are many more elaborate ways of sorting, the theory of which is outside the scope of this book.

Debugging

Unless you are superhuman, the programs you write will have mistakes in them to start with. There are two kinds of mistakes. First, there are those in which you ask the machine to do something which is incomprehensible or impossible, such as when you misspell a command word. With these the machine will give you an error message, and it should not be too difficult to find out what is wrong.

The most common error message is 'SYNTAX ERROR', and this is usually caused by a misspelling of a command word or the omission of a vital punctuation mark. Usually the computer tells you the line number where the error was encountered, and by looking at that line you should be able to spot the error and put it right. Other error messages are caused by more specific mistakes. A complete list of the messages and pointers on what to do about them are given in Chapter 23.

In the other sort of mistake, you give the machine instructions that are perfectly valid – they are simply not the right instructions for achieving your desired end. The machine cannot help you here, because it has no understanding of what it is you are trying to do. These errors in programs are often known as 'bugs', and tracking them down can involve detective work that Sherlock Holmes would be proud of.

Some bugs prevent a program from working at all, some make it give the wrong answers while appearing to work properly, and others only cause a problem in particular circumstances. Some big programs have had bugs in them which have lain unnoticed for years, until some unusual situation caused them to make the program go haywire. We can only hope that programs that control nuclear power stations or air traffic control systems contain none of these.

Assuming that you know there is a bug in your program, you first need to satisfy yourself that your original scheme for the program was sound. When you have checked that, you can try a 'hand simulation', going through the program on the screen step by step, pretending to be the computer and doing everything that the computer would. You note down the values of the variables

on a piece of paper as you go along. If you have lots of loops, though, this can take a long time.

The approach to follow after this is to break the program down into sections, and try and confirm that each section is working properly. You can do this by putting in extra PRINT statements that will tell you the state of the variables at each stage in the process. You can also interrupt the program before it is finished, either with the STOP button or by extra END statements inserted. Then you can examine the variables by giving direct PRINT commands. By this process it should be possible to narrow down where the fault might be. It quite often turns out that the problem is not an obvious mistake on your part but a quirk of the Basic language that you were not aware of. Many of these pitfalls are listed in Chapter 23.

When your program is actually working you need to test it to make sure that the answers it gives, or the actions it takes, are the right ones. You need to give it problems for which you already know the answers, to see that it gets them correct. For instance, you can try out the perpetual calendar with today's date, or the date of the battle of Waterloo. Try unusual data: very large or very small numbers, and see what happens. Even after all this, you can never be *absolutely* sure your program has no bugs, but you can feel pretty confident!

Sample programs

Spelling a name backwards

```
10 INPUT N$
20 FOR X=LEN(N$) TO 1 STEP -1
30 PRINT MID$(N$,X,1);
40 NEXT
```

Teacher

```
10 REM Fancy Teacher
20 PRINT "HELLO. PLEASE DO THESE SUMS:"
30 PRINT
40 A=INT(RND(1)*100)+1
50 B=INT(RND(1)*100)+1
60 Q=INT(RND(1)*4)+1
70 ON Q GOTO 80,130,180,230
80 PRINT " "A
90 PRINT "+"B
100 PRINT "-----"
110 SO=A+B
120 GOTO 300
130 PRINT " "A+B
140 PRINT "-"A
150 PRINT "-----"
160 SO=B
170 GOTO 300
180 PRINT " "A
```

```
19Ø PRINT "*"B
2ØØ PRINT "-----"
21Ø SO=A*B
22Ø GOTO 3ØØ
23Ø PRINT " "A*B"/"A
24Ø SO=B
3ØØ INPUT AN
31Ø IF AN=SO GOTO 34Ø
32Ø PRINT "SORRY, THAT IS WRONG. TRY AGAIN."
33Ø GOTO 7Ø
34Ø PRINT "THAT IS CORRECT. TRY ANOTHER."
35Ø GOTO 4Ø
```

Perpetual calendar

```
 1Ø PRINT "PERPETUAL CALENDAR"
 2Ø PRINT "FINDING DAY OF THE WEEK FOR 1753 TO
    2199"
 25 PRINT
 3Ø DIM K(12)
 4Ø FOR M=1 TO 12
 5Ø READ K(M): NEXT
 6Ø FOR C=1 TO 5
 7Ø READ L(C): NEXT
 8Ø FOR X=Ø TO 6
 9Ø READ D$(X): NEXT
1ØØ INPUT "YEAR, PLEASE"; Y
11Ø INPUT "MONTH (NUMBER)"; M
12Ø INPUT "DAY, PLEASE"; D
13Ø C=INT(Y/1ØØ)*1ØØ
14Ø YL=Y-C
15Ø YM=INT(YL/4)
16Ø IF Y/4<>INT(Y/4) GOTO 18Ø
17Ø K(1)=Ø: K(2)=3
18Ø A=YL+YM+K(M)+L(C/1ØØ-16)+D
19Ø X=A-7*INT(A/7)
2ØØ PRINT D$(X)
3ØØ DATA 1,4,4,Ø,2,5,Ø,3,6,1,4,6
31Ø DATA 4,2,Ø,6,4
32Ø DATA SATURDAY, SUNDAY, MONDAY, TUESDAY
33Ø DATA WEDNESDAY, THURSDAY, FRIDAY
```

This program includes no traps for invalid data, such as someone asking for the ninety-third of the month, or for years prior to 1753 (the beginning of the Gregorian calendar). These traps could easily be added.

Dice game

```
 1Ø PRINT "DICE GAME!"
 2Ø PRINT
 3Ø PRINT "PRESS RETURN TO ROLL THE DICE."
 4Ø PRINT
 5Ø W=Ø
 6Ø INPUT "YOUR BET IS"; B
 7Ø GOSUB 1ØØØ
 8Ø IF D=7 OR D=11 GOTO 2ØØ
 9Ø IF D=2 OR D=3 OR D=12 GOTO 25Ø
1ØØ E=D
11Ø INPUT M
12Ø GOSUB 1ØØØ
13Ø IF D=E GOTO 2ØØ
14Ø IF D=7 GOTO 25Ø
15Ø GOTO 11Ø
2ØØ W=W+B
21Ø PRINT "YOU WIN. YOUR WINNINGS ARE" W
22Ø GOTO 6Ø
25Ø W=W-B
26Ø PRINT "SORRY, YOU LOSE. YOUR WINNINGS
    ARE" W
27Ø GOTO 6Ø
1ØØØ GOSUB 11ØØ
1Ø1Ø D=R
1Ø2Ø GOSUB 11ØØ
1Ø3Ø D=D+R
1Ø4Ø RETURN
11ØØ R=INT(RND(1)*6)+1
111Ø PRINT R;
112Ø RETURN
```

The variable M in line 110 of this program is never used. The INPUT statement is simply a way of making the program wait until the player has had a look at the roll of the dice and pressed RETURN before the dice are rolled again. Otherwise the game would always be over in an instant.

Prime factors

```
 1Ø PRINT "PRIME FACTORS"
 15 PRINT
 2Ø D=3
 3Ø PRINT "NUMBER TO BE FACTORIZED"
 35 INPUT M
 4Ø PRINT "HAS THE FACTORS:"
 45 N=M
 5Ø IF N/2<>INT(N/2) GOTO 7Ø
 6Ø PRINT 2: LET N=N/2: GOTO 5Ø
 7Ø IF N/D<>INT(N/D) GOTO 9Ø
 8Ø PRINT D: LET N=N/D: GOTO 7Ø
```

```
 9Ø D=D+2: IF D<=SQR(N) GOTO 7Ø
1ØØ IF M=N GOTO 12Ø
11Ø IF N<>1 THEN PRINT N
115 GOTO 15
12Ø PRINT "NONE. IT IS PRIME.": GOTO 15
```

Again, traps could be added to this program to look for negative or fractional input. It will not work for 2 or 3, or for numbers that are too large to be held exactly in the computer. It often happens in computing that a scheme that is basically sound runs into problems with data at the extremes of the possible range.

Sorting

```
 1Ø PRINT "SORTING"
 2Ø INPUT "HOW MANY NUMBERS"; L
 25 DIM N(L)
 3Ø FOR X=1 TO L
 4Ø INPUT N(X)
 5Ø NEXT
 6Ø FOR I=1 TO L
 7Ø FOR X=1 TO L-I
 8Ø IF N(X+1)=>N(X) GOTO 12Ø
 9Ø Q=N(X+1)
1ØØ N(X+1)=N(X)
11Ø N(X)=Q
12Ø NEXT X
13Ø NEXT 1
14Ø FOR X=1 TO L
15Ø PRINT N(X)
16Ø NEXT
```

As can be seen from line 70. each pass through the list can be shorter than the previous one because at the end of each pass one more item has always been swept to its correct final position at the end of the list. The program will 'crash' if someone tries to order a list of one item.

12. PEEK, POKE and the 64's Memory

Beavering away inside every micro there's a processor which actually does all the work, and the 64 is no exception – the microprocessor in the 64 is called the 6510, as it happens. Every microprocessor works with and understands only its own type of instructions, a unique set of codes that is usually called *machine language*.

If machine language is the only one the processor understands, how come you're able to write programs in Basic? Despite its occasional unfamiliarity, Basic is by no means a machine language. How does the microprocessor understand your Basic programs?

Well, when you turn on the 64 it actually begins to run a huge machine-language 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 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.

When you then type in Basic commands, they are in fact being translated by this machine-language program – it's *interpreting* them, one at a time. Machine language is famously awkward for the average user to grasp – and at the least it's very difficult to use. The 'interpretation' activity 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 the way memory is organized, or how the electrical signals produced when you press a key are decoded, or indeed anything really machine-oriented. But the 64 Basic does have a handful of commands built into it that allow you to get at the machine code. And since they enable you to step from the sunny outside world of near-English statements and understandable commands into the murky technological forest of the 64's internals, these commands are potentially lethal. Once you're able to observe and interfere with the way the computer works, you also have the power to make it *not* work.

Scared yet? Well, don't be. All you can do is reference specific memory locations: you won't be interfering 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. These engaging names are also quite accurate: PEEK allows you to see what's in a particular memory location, POKE puts something specific there.

PEEK isn't actually a command in its own right; it, has to be used with another command. 'PRINT PEEK(24576)' prints the contents of location 24576 – in decimal rather than binary, of course, 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,

```
A=PEEK(24576)
```

or with it being used in loops:

```
IF PEEK(24576)=Ø THEN 4ØØ
```

or participating in compound expressions:

```
PRINT (PEEK(24576)/4)+1/.67
```

POKE is a genuine command. It looks like this:

```
POKE 24576,1Ø
```

which means 'Put the value 10 into memory location 24576.'

Setting up the Display with Pokes

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.

Let's take a look. Turn the 64 on and press CLR. Now type 'PRINT PEEK(1024)' and RETURN: the 64 should respond with '16'.

On the 64, the screen memory starts at location 1024. The 'screen memory' is the area of memory which holds a map of the display containing the codes for any characters currently on the screen. The top left-hand corner is location 1024, the next character position along is 1025, and so on: the leftmost position on the second line is 1064 . . . and so it goes. The bottom right corner is location number 2023.

What you've just done is PEEKed into the location corresponding to the top left corner where there is character code 16 – which happens to be the code for that 'P' from 'PRINT', which is on the screen at present.

Hit CLR again and type 'POKE 1024,3' and RETURN. The 'P' of 'POKE' has changed to a 'C', right? That's because the code for a C is 3, and that's the code you've just put into that position.

Now to the other chunk of memory relating to the display, the colour memory. This works in much the same way except that the value in each location decides what colour anything displayed there will be. The colour memory starts at location 55296, which again corresponds to the top left; and again the numbering goes left to right along the rows down to location 56295 at the bottom right.

Type 'POKE 55296,2' and return. The 'C' should have changed to a murky red – because 2 is the colour code for red. You can have any of the eight main

colours in these locations, numbering them 0 to 7; the numbers are one less than the equivalent keys on the keyboard, so though RED is on key 3 you use the code 2. Code 0 is for black.

Now that you can display characters on the screen and set their colour, you should be ready to delve further. For a start, though, here's a way of specifying a particular point on the screen without having to know all the addresses of all the colour and character position locations on the display.

There are forty columns on the screen. If the top left location is identified as column 0 in row 0, you can compute the actual memory address of any row and column position by the formula

$$position = 1024+y*40+x$$

where x is the horizontal position along a line and y the vertical position in terms of number of lines from the top. You could do the same sort of calculation for the colour memory:

$$position = 55296+y*40+x$$

Here's a program that will show characters on the screen in different colours:

```
 1Ø COL=Ø: CHR=Ø: REM start with zero for
    colour and character codes
 2Ø FOR X=Ø TO 39: FOR Y=Ø TO 24
 3Ø P=1Ø24+Y*4Ø+X: REM work out screen
    position
 4Ø POKE P, CHR: REM POKE character location
    with whatever value is in CHR
 5Ø CHR=CHR+1: REM add one to the value
    in CHR
 6Ø C=55296+Y*4Ø+X: REM work out colour code
 7Ø POKE C, COL: REM POKE colour location
    with whatever value is in COL
 8Ø COL=COL+1: REM add one to the value
    in COL
 9Ø IF COL>7 THEN COL=Ø: IF CHR>255 THEN
    CHR=Ø: REM keep within bounds
1ØØ NEXT Y: NEXT X: REM go on to next
    position
```

When you RUN this, you'll notice you seem to get horizontal blue stripes across the display. Why? Because every seventh character is blue, and since the background is blue too you can't see it.

13. The Function Keys

The 64 has four brownish keys on the right that can be used by programs for a variety of different tasks or *functions* – hence the name 'function keys'. Setting up a program to read the function keys – to see whether any have been pressed – is a doddle. As far as the computer is concerned, pressing one of these keys produces an internal numeric code in the range 133 to 140 as follows:

Function key 1	133	(unshifted)
Function key 2	137	(shifted f1)
Function key 3	134	(unshifted)
Function key 4	138	(shifted f2)
Function key 5	135	(unshifted)
Function key 6	139	(shifted f3)
Function key 7	136	(unshifted)
Function key 8	140	(shifted f4)

You can see this for yourself with this short program:

```
1ØØ GET FK$: IF FK$="" THEN 1ØØ
11Ø PRINT ASC(FK$)
12Ø GOTO 1ØØ
```

RUN this and try pressing any of the function keys.

GET is a command that looks for some input from anywhere – you need that IF in there to keep the GET waiting for you to press something, otherwise the program will just zip along and end before you have time to press something. When you hit a key, that's what gets put into the variable FK$ (this program works for any keys on the 64's keyboard, of course). Line 110 then uses the ASC function which figures out the value of that code, and displays it for you. The final GOTO just takes the program back to the start and it waits for your next key depression.

So you include in your program something like 'GET FK$', and if ASC(FK$)=133 it means function key 1 has been pressed. Or you could check another way, using the CHR$ function; this indicates the character that the 64 has assigned to a particular ASCII code. You can have your 'GET FK$', and if 'FK$=CHR$(139)' it means function key 6 has been pressed with the CBM or the SHIFT key held down.

This is useful for menus of choices and the like. But it's as far as a Basic program can go; if you want a particular string of characters to appear on the screen automatically when you press one of the function keys, you'll have to get into machine-code programming.

14. Run Faster, Use Less Memory

There's quite a lot of memory on the 64, enough for most programs. But there are ways of squeezing even more out of the computer.

For a start, there's one potential drawback with the 64: it uses Basic, and in particular it uses what's called *interpreted* Basic. All programming languages work by giving you the chance to write programs in something that more or less resembles your mother tongue – rather than in a string of 0s and 1s, which would more closely correspond to the way the *computer* actually works. When you come to RUN your program, it's translated into the 0s and 1s that the computer needs.

This is A Good Thing, since it means you don't have to fiddle around too much with the innards. But the problem with the 64's Basic is that after every line the computer has to decode everything in the line into binary before it can execute it and move on to the next one. Most microcomputer versions of Basic work that way, in fact.

(The alternative is a *compiler* language, where you write the program and run it through a 'compiler' before you put it to work. The compilation process turns it into binary for the computer, so when you RUN it in earnest the computer doesn't have to waste any time translating the lines of code – the final program ends up a good deal shorter that way, too.)

So 64 Basic is interpreted, and when you RUN a program the computer will spend more time figuring out what you want to do than it does doing it.

There are ways of cutting back on the space needed, so you can cram your programs into memory: and there are ways of speeding up the run-time for those programs. Sadly, most of them conflict with good programming policies—in particular, 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.

Remarks

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, one byte being the amount of memory a single character or a space takes up (REM takes three characters and one following space); it will take up more bytes 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.

In general, REMs are to be commended because they tell you what the

program is doing. Losing them can be a bit of a drag, but you can at least hand-annotate a printed or handwritten copy: and different chunks of program can be separated by starting each on an identifiable number. Initialization might start at line 10, GOSUBs begin as multipliers of 100, the main chunks of program are separated into 1000s. Don't forget you've got up to 63,999 line numbers available to you!

Line Numbering

On the other hand, longer line numbers do take up more space – one byte per figure. So there's an incentive to keep the numbers low.

The trade-off between readability and space is something you'll have to make for yourself.

Blank Spaces

Spaces in program lines also take up room in memory – one byte per space. Again, they aid the legibility of a program; where possible, the examples in this book do have blanks between Basic statements for precisely that reason. So you'll have to make your own decisions about how much readability you need.

Don't, of course, try to economize too much – don't take any blanks out within quote marks! You need them for spacing on the display or printed output.

Multiple Instructions Per Line

You can get up to eighty-eight characters on a line of program code, and you can use all that length by putting several successive statements on it separated by colons. This takes less memory, because you don't need to store so many line numbers: each line number takes several bytes, a colon occupies only one. And the program runs faster, because the computer has only the one line to decode rather than several.

Watch out for illegal combinations, of course – for instance, one of the lines you're combining may be the target of a GOTO or GOSUB somewhere else in the program.

Abbreviations

To make the most of the eight-eight character line, you can squash Basic commands into two characters by using abbreviations. Normally these are the first letter of the command plus the second letter when you're holding the SHIFT key down, but there is a full list in Chapter 26.

Next time you LIST the program, you'll see that the 64 has helpfully translated the abbreviations back into the original terms. On the display the line, then, may well be over eighty-eight characters – but in memory it's still within the limits.

GOTO and GOSUB

When Basic comes across a GOTO or GOSUB statement it has to go off and look for the line number specified. If that happens to be less than the current line number, the search will start at the beginning of the program: otherwise it starts from the current line number. And if the destination is a long way away it'll take time for the 64 to get there; but then it has to get back again on a RETURN, and for that it'll have to start searching for the point it left by going back to the start of the program.

This might not make so much difference on short programs, but it can be dramatic on longish ones. The best policy is to put all your commonly used subroutines at the beginning of your program.

Variables

Much the same applies to variables. As it executes your 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. So it makes sense to ensure that all variables you're going to be using frequently are right at the beginning of that list set up by Basic – like maybe 'T' for a timer or anything else you use a lot in FOR loops. You can do this by making sure Basic comes across them right at the start of the program, if necessary by 'dummy' assignments – or more usually by setting them to zero (T=0, for instance) as part of your getting-started procedures. That way Basic won't have to look so far down the list before it finds the variable you're referencing.

Incidentally, you can save a little memory space by using short variable names like 'M' rather than 'MINUTES'. But there's very little difference to the 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.

Integer Variables

Simple integer variables are a real time-waster, because Basic always converts integers (i.e. numbers without decimal points) to floating-point numbers (with decimal points). If you had the expression

```
S%=T%+U%+V@+W%
```

Basic would go and find T% and convert that to floating point; then do the same for each of the other three; then add them together; then convert the result to integer again; and then assign the value to S%. Clumsy, isn't it?

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 several per element in a floating-point array. So if you're setting up an array that doesn't need the decimal point, try using 'DIM A%(20)' rather than the apparently simpler 'DIM A(20)'. You'll be saving sixty bytes in this case!

Constants

In some cases you can save a lot of time and probably some memory in a program by using a name for a constant rather than the value itself. This is particularly true with longish decimals (say four or more digits): not only does 33.174 take up six bytes every time it appears, it also has to be converted by Basic into floating-point arithmetic before it's used. Much better to start by giving it a name (B=33.174) and using that when you need the number (W=B*4/2, or even A=B).

But for numbers with one or two digits and without decimal points it's normally better to avoid names. The program will probably run 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.

Arithmetic and Trigonometry

If you know how to use them, it's attractive to find functions like exponentiate and SIN provided in Basic. They do simplify the writing of calculating programs.

On the other hand, they will also slow it down when it's running. The exponentiation operator is particularly sluggish – as well as being prone to give answers .000001 out. It's generally much better to do your 'raising to the power of' by repeated multiplications.

That also applies to SIN, COS, TAN and the other trigonometric functions. On the other hand, if you're using a *variable* – raise T to the power of V, say, or finding COS(XT) – the ready-to-go functions are quicker, because otherwise the program would have to keep checking the list of variable names.

READ, DATA, and DIM

If you have a lot of data to be pushed into a program and used over and over again, the most efficient way to do it memory-wise is with a DATA statement and/or a matrix. This obviously takes up less space than repeating everything.

Loops

There's not a lot you can do to speed up FOR . . . NEXT loops – except to avoid specifying a variable. 'NEXT' on its own means that the loop will complete about 25 per cent quicker than one with 'NEXT T'. This obviously requires some care if you have loops within loops.

And if you are 'nesting' loops, try to make sure that the *innermost* loop has the greatest number of iterations to go through. That's logical enough when you think about it:

```
1ØØ FOR M=1 TO 1ØØØ
2ØØ FOR N=1 TO 1Ø
3ØØ NEXT: NEXT
```

Here the loop going around ten times is being executed a thousand times. In this one the inner loop is only being done ten times:

```
1ØØ FOR M=1 TO 1Ø
2ØØ FOR N=1 TO 1ØØØ
3ØØ NEXT: NEXT
```

So the program doesn't have to skip back to line 100 so often.

Try proving the difference to yourself. Add a couple of lines that time the two operations:

```
 9Ø TI$="ØØØØØØ"
1ØØ FOR M=1 TO 1ØØØ
2ØØ FOR N=1 TO 1Ø
3ØØ NEXT: NEXT
4ØØ PRINT "FIRST RUN ";TI$
5ØØ TI$="ØØØØØØ"
6ØØ FOR M=1 TO 1Ø
7ØØ FOR N=1 TO 1ØØØ
8ØØ NEXT: NEXT
9ØØ PRINT "SECOND RUN ";TI$
```

Garbage

Every time your program uses a string variable (when you assign something inside quote marks to a variable label), Basic takes a copy of it. And if you redefine the variable it copies that too – but it doesn't overwrite the first string associated with that name. In a longish program with a lot of string variables being redefined, this means a lot of garbage can collect in memory with all those unwanted strings. Eventually there may be so many of them that they take up all the free space.

No problem: Basic includes an automatic garbage collector routine which is called in as soon as the strings start to get close to the program in memory. The garbage routine chucks out all the unnecessary strings and then lets you get on.

But it 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 minimize the amount of garbage the program is generating – to cut down as much as possible on the use of strings, in fact.

The major culprit is the overlarge string array. Try not to use too many, and you should avoid some run-time delays.

Displays

You can simplify your screen layouts. This is not my favourite go-faster minimize-memory technique, since I feel that what's on the screen should be as 'friendly' as possible; but obviously all those fancy formatting features, colour changes and wordy messages do take up space in memory and time to get on to the display.

The two obvious suggestions are to abbreviate messages ('TOT' instead of 'TOTAL SCORE', perhaps), and to avoid character string inputs by using single keys – instead of asking the user to type something, he or she hits one key.

Closing Quotes

If you have a line that ends with a pair of closing quotes, you can omit them. That saves one byte – well, it all helps. You can't, of course, omit them if they aren't at the very end of a line.

Overlays

You may be able to get a bit more out of the 64 by separating your program into 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 *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 you can include a LOAD as the last line of the preceding section of a program.

Data from Tape

Don't ignore the INPUT♯ and GET♯ statements. These allow you to read from cassette or disk, and their most effective use in this context is with DATA.

You'd write your program as normal but without READs and without DATA. The Data statements are all in a separate mini-program on tape. READ statements are replaced by something like 'INPUT♯1, X' – you've got a line somewhere like 'OPEN 1, 2, "DATA"' to open a channel to the cassette unit and get at a file called 'DATA' on it. Then your program can read in the DATA a line at a time. Result – a significant reduction in the memory your program needs.

Maximize Basic

That's one illustration of how you can make the most of the commands provided in Basic to simplify and speed up your programs. Getting on top of Basic is one pretty efficient way of producing fast and economical code; you should try thinking in Basic terms.

For example, a FOR . . . NEXT loop can often replace several IF . . . THEN statements – especially in setting up screen layouts. 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. The ON/GOTO statement switches program execution to a specified line number. For instance,

```
ON T GOTO 5ØØ, 6ØØ, 7ØØ
```

will switch to line 500 if T is 1, to line 600 if T is 2, and line 700 if T is 3.

And 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 get you to the top of your memory allowance. SPC and TAB are much more economical when you're PRINTing simply because they take up fewer bytes: SPC inserts a specified number of spaces, TAB zips the cursor to the stated position. And you can save a couple of bytes per command by abbreviating them – 'S' and shifted 'P' gives you 'SPC(', 'T' and shifted 'A' gives 'TAB('.

Section Three

Colour, Graphics and Sound

15. Colour on the 64

One of the best things about the 64 is the way it lets you specify colours on the screen, for whole areas and for individual symbols and characters.

The 64 can use a total of sixteen colours. Characters you put on to the screen can be in any of these, and each character can be a different colour. You can also specify a colour for the background box, and a colour can be specified for the border around the edge too. Each of the three areas of colour can be controlled independently, which isn't the case on some of the so-called 'colour' computers that sell in competition with the 64; with them you're allowed only certain combinations.

OK, so how do you use colour?

Colour from the Keyboard

To change the colour of the next character the 64 comes across you hold down the CTRL key while pressing one of the numeric keys (1 to 0 along the top of the keyboard). On the front of keys 1 to 8 is an abbreviation for the colour you'll get.

CTRL + *Key*	*Abbreviation*	*Colour*
1	BLK	black
2	WHT	white
3	RED	red
4	CYN	cyan (light blue)
5	PUR	purple
6	GRN	green
7	BLU	blue
8	YEL	yellow

Using the CBM key with these numerics gives another eight, though there's nothing marked on the keys:

CBM + *Key*	*Colour*
1	orange
2	brown
3	light red
4	dark grey
5	medium grey
6	light green
7	light blue
8	light grey

You can try this straight away: you should see the cursor change colour to whatever you've selected – unless you press CTRL and 4, in which case you've chosen cyan (and the cursor is light blue already); or unless you went for CTRL and 7, which selects blue (in which case the cursor is still there, but it's the same colour as the background so you can't see it).

You can use these CTRL/number and CBM/number combinations in PRINT statements. Type this (without the square brackets – they just mean hit CTRL and the number given):

```
PRINT "THIS PRINTS [CTRL 3]HELLO IN RED"
```

and this:

```
PRINT "THIS PRINTS [CBM 3]HELLO IN RED"
```

Note that when you press CTRL or the CBM key, at first nothing happens: as soon as you hit the 3 key, though, you get a funny graphic symbol (a reversed pound sign in the first case, a reversed cross in the second). That's how the 64 denotes colour changes for anything that's to be displayed: outside quote marks, pressing CTRL or CBM with one of the colour keys will instantly change the colour of the cursor and anything you type subsequently, but if you're using quote marks (which means you want the stuff inside them to be PRINTed) there won't be any immediate effect. Instead you'll get the colour change when you hit RETURN.

Try it and see. Note that the colour of the cursor has been changed now, and anything else you type will be the same colour as the cursor. And if the blue-on-blue display gets on your nerves, get used to changing the cursor and character colours to white by typing CTRL and 2 at the start of a session.

In the same way you can use these colour controls in programs. Let's turn the examples into a short program:

```
10Ø PRINT "THIS PRINTS [CTRL 3]HELLO IN RED"
11Ø PRINT "[CBM 2] . . . AND IN BROWN"
```

Nothing will happen now until you type RUN.

Colour on the Border and the Background

You can also specify colours for the border (the bit round the edge that you can't otherwise touch) and the background on which characters and shapes appear.

These colours are determined by what's in memory locations 53280 (border) and 53281. The values you POKE into these will be a number from 0 to 15 from the following table:

0	black
1	white
2	red
3	cyan
4	purple
5	green
6	blue

7 yellow
8 orange
9 brown
10 light red
11 dark grey
12 medium grey
13 light green
14 light blue
15 light grey

If you want to check out the range of alternatives and find what you like, type in this short program:

```
  1Ø FOR B=Ø TO 15
  2Ø POKE 5328Ø,B
  3Ø FOR S=Ø TO 15
  4Ø POKE 53281,S
  5Ø GOSUB 1ØØØ
  6Ø NEXT S
  7Ø GOSUB 1ØØØ
  8Ø NEXT B
 1ØØ PRINT "STOPPED WITH THIS COMBINATION:"
 11Ø PRINT "BORDER=",B
 12Ø PRINT "SCREEN=",S
1ØØØ PRINT "NEXT?"
1Ø1Ø GET A$: IF A$="" THEN 1Ø1Ø
1Ø2Ø IF A$="S" THEN 1ØØ
1Ø3Ø RETURN
```

When you RUN the program it will start with a black border (because it first POKEs 53280 with 0) and a black screen (it POKEs 53281 with 0 too). It zips off to a subroutine at line 1000 which prints the 'NEXT?' and then waits for you to type something – that's what the GET command is for: it takes any single-character input without waiting for a RETURN, and puts it into a variable. In this case I'm using A$ as the variable. The 'IF A$="" THEN 1010' will keep the 64 hanging around at line 1010 until *something* has been typed.

If the key you've hit is S, it means you want to know the colour combination you have at the moment. The program then goes off to line 100 to tell you.

Otherwise it RETURNs from the subroutine to POKE 53280 with the next possible screen colour, asks you again if you want the next combination, and (assuming you do) the NEXT command loops back adding one to the number in S to display the next screen colour. You carry on like that for all possible screen backgrounds, then it goes through it all again for the next border colour . . . and so on until it's exhausted the possibilities.

So when you find a combination you like, press S and the 64 will tell you what numbers have to be POKEd into 53280 and 53281 in future to get that combination.

In practice, when you come to write programs that change the border and background colours, it makes sense to simplify things by sidestepping the need constantly to repeat the numbers:

```
 1Ø BRDER=5328Ø: SCREEN=53281
 2Ø BLACK=Ø
 3Ø WHITE=1
 4Ø RED=2
 5Ø CYAN=3
 6Ø PURPLE=4
 7Ø GREEN=5
 8Ø BLUE=6
 9Ø YELLOW=7
1ØØ RANGE=8
11Ø BROWN=9
12Ø LRED=1Ø
13Ø G1=11
14Ø G2=12
15Ø LGREEN=13
16Ø LBLUE=14
17Ø G3=15
```

Thereafter you can POKE a meaningful variable name (SCREEN or BRDER) with a more or less meaningful colour value, as in

```
1ØØØ POKE SCREEN, RED
```

Remember that only the first two letters of the variable names will be recognized by the 64, so BLACK has to be distinguished from BLUE and all those GREYs and GREENs have to be identified uniquely. Nor can you use one of the specially reserved words that Basic needs for its own commands, which is why I'm suggesting BRDER and RANGE (because OR is one of Basic's words).

Colour and Characters from Memory

Many programs – games and others – have displays which involve putting a character on one part of the screen in one colour followed by another somewhere else in a different colour. You *can* do this using PRINT statements and lots of CTRL or CBM codes within quotes; but you'll end up using a lot of them, and the program will be quite slow.

You can get around that by using POKEs to put characters and colours directly on to the screen. But POKE is a command that lets you change the contents of memory: and what's that got to do with the display?

Quite a lot, as it happens. First you'd better understand how the TV screen works. Inside the set there's a device called an electron gun which can shoot a fine stream of electrons at the backside of the screen. That side is coated with a type of phosphorus which glows when it's hit by electrons. So if the gun squirts a pinpoint beam of electrons at a particular spot on the screen, it will start glowing.

The shapes the 64 can display are made up of several such dots, arranged in a combination that looks recognizable to you. Put the @ symbol on the screen and you should see what I mean: there are no curves in there, just a pattern of proximate dots that more or less resemble a curve.

But one dot only glows as long as the electrons are being fired at it. As soon as they aren't, the dot of light on the screen starts to fade. And once it has started one point glowing, the gun has to move on and do the next in sequence.

So to keep a screenful of dots illuminated the gun has to come around pretty quickly and sock in another stream of electrons. That's called 'refreshing', and the 64 does it sixty times a second – any less frequently and you'd notice the dots flickering as they fade and then get reactivated.

In order to be able to refresh the display, the 64 has to remember which dots were illuminated on the last pass of the electron gun. It does that by setting up a 'map' of the screen in memory – a directory of what's on the screen that it can refer to automatically on each sweep of the electron gun. (Someone has said that the TV screen is really a window on to that area of the 64's memory – it lets you see what's inside there.)

Now, we've seen that you can get into the memory and change things there. You can get at the memory-map of the screen, for instance; and by changing some of the stuff in there you can cause the 'window' (the TV screen) to reflect a changed set of contents.

With most computers this involves POKEing a value corresponding to the character you want to display into a memory location corresponding to the position on the screen where you want it to appear. And that happens on the 64 too: you look at the screen character codes map to pick out the place where you want the character, then you check the screen POKE codes table to find the code for the character in question, then you POKE the location with the code. For example, 'POKE 1524,83' puts a heart in the middle of the screen.

Try it. See the heart? No? Ah, well, that's the other thing about the 64 – the heart *is* there, but it happens to be the same colour as the background. And it will stay the same colour as the background until you change its colour. This ability to specify individual colours for individual characters is one of the nice things about the 64, but it does add an extra step if you're putting things on to the screen this way.

To specify the colour you have to do another POKE. This time it's locations 55296 to 56295 that we're interested in, since those make up a colour codes map corresponding to the character codes map.

You find the right location and POKE in a code from 0 to 15: that sets the location specified such that anything displayed in there will take one of the sixteen colours.

The colour map location that you need to see the heart is 55796. So 'POKE 55796,2' (that's the code for red, and hearts *are* red). Now you should see the heart.

How do you know which colour map location corresponds to which character map location? You could do it by laboriously counting squares, but I prefer to let the 64 do the work wherever possible. You know the start locations, the top left-hand corner, in both maps: and you know they both have the same dimensions. So you can add the same number of squares to each of them. If you 'POKE 1024+500,83' and then 'POKE 55796+500,2' you'll get the red in the square where the heart is.

Alternatively you can use this formula:

location = *column* + 40 × *row*

where columns are the vertical wedges in either map and rows are the horizontal slices. You can put this early into your program and then 'POKE 55796 + *location, colour*: POKE 1024 + *location, character*'.

16. Sprites

Sprites are one of the features that makes the 64 rather special among home computers. Commodore literature sometimes calls sprites 'Movable Object Blocks' or 'MOBs'; a sprite is a mini-picture that can be defined separately from other graphics characters and moved around the screen with Basic commands.

Sprites can bump into 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.

Typically, then, you use sprites in an animated 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.

Sprites are created from 'pixels' (on/off dot positions) in a grid of twenty-one rows with twenty-four pixels per row. Since each pixel is a single bit, that means one sprite takes up sixty-three bytes of memory (21 × 24 = 504 and 504/8 = 63). (A 'bit' is a single digit, either 0 or 1; a 'byte' is encoded as eight bits, since the pattern of eight 0s and/or 1s gives enough possible combinations to provide codes for all the different characters in the 64.)

Designing the sprite itself involves these steps:

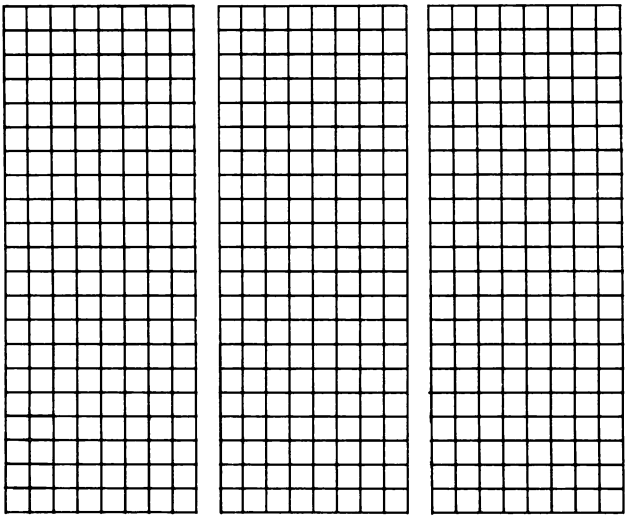
1. Draw the sprite.
2. Find the decimal equivalents.
3. Store those numbers.
4. Tell the 64 where they're stored.
5. Define the sprite's colour.

Then there are two more steps to using the thing:

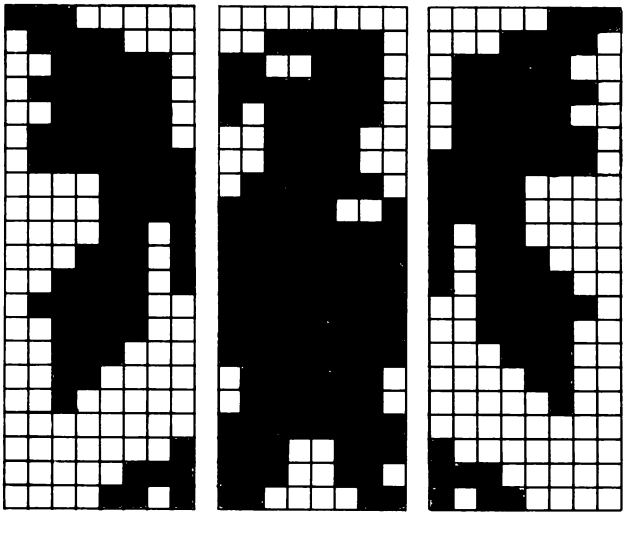
6. Indicate where the sprite should be placed on the display.
7. Switch on the sprite so that it is displayed.

How are Sprites Defined?

OK, let's get started. First you need some paper on which you can draw a 21 × 24 grid (use squared 'graph' paper and it will be much easier!) with the twenty-four columns split into three groups of eight. This should give you a pattern of possible pixel positions like this:



You can now start sketching, filling in the dots that you want to appear as solid parts of the picture. Here's a kind of mean-looking eagle-style animal:



Now to convert the picture into binary equivalents, with a 1 for every pixel that will be used and a 0 for the rest:

```
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
```

Finally, we need the decimal version of each group of eight bits:

```
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
```

To put that a bit more simply:

```
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

You now have to decide where to store the sprite. You need sixty-three bytes per sprite definition – though in fact the 64 finds it more convenient to think in multiples of eight, so you've got to reserve sixty-four (the last location in this sprite 'block' is never used).

Obviously you have to store the sprite somewhere well away from other things like the character set, any background graphics you've defined, and your program itself. For a variety of reasons the best place is the cassette buffer, that area of memory normally used to store stuff coming from or going to the cassette unit. Obviously, you can't then use it for cassette I/O (input/output) 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 sixty-four bytes starting at 828, another at 892, the third at 956.

So here's a program to store the eagle sprite:

```
2Ø FOR A=Ø TO 62: READ S
3Ø POKE 828+A, S
4Ø NEXT A
```

with some DATA for line 10 to READ:

```
1ØØ REM the sprite data
1Ø1 DATA 224, Ø, 7
1Ø2 DATA 12Ø, 62, 3Ø
1Ø3 DATA 62, 2Ø6, 124
1Ø4 DATA 126, 254, 126
1Ø5 DATA 62, 19Ø, 124
```

```
1Ø6 DATA 126, 6Ø, 126
1Ø7 DATA 127, 6Ø, 254
1Ø8 DATA 15, 126, 24Ø
1Ø9 DATA 15, 249, 24Ø
11Ø DATA 13, 255, 176
111 DATA 29, 255, 184
112 DATA 61, 255, 188
113 DATA 124, 255, 62
114 DATA 6Ø, 255, 6Ø
115 DATA 56, 255, 28
116 DATA 48, 126, 12
117 DATA 64, 126, 4
118 DATA Ø, 255, Ø
119 DATA 1, 231, 128
12Ø DATA 7, 23Ø, 224
121 DATA 13, 195, 176
```

To add a couple more sprites, you could append a loop and change line 30 a bit:

```
1Ø FOR SL=Ø TO 2
2Ø FOR S=Ø TO 62: READ SD
3Ø POKE 828+(64*SL)+S, SD
4Ø NEXT S
5Ø NEXT SL
```

with the extra DATA following line 121. Here SL is the sprite number variable, SD the sprite data, and S the individual sprite bytes.

Using a Sprite

To tell the 64 to 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 (for up to eight sprites, remember?) from 2040 onwards with one of the sprite block numbers – which for sprites in the cassette buffer are 13, 14 and 15 for the sprites at 832, 896 and 960 respectively.

So this would put a single eagle at your program's disposal:

```
1ØØØ POKE 2Ø4Ø, 13
```

and this would give it eight:

```
1ØØØ FOR D=Ø TO 7
1Ø1Ø POKE 2Ø4Ø+D,13
1Ø2Ø NEXT D
```

At the moment, the sprite shape doesn't have any colour associated with it. You can give the same shape different colours, though. Sprite colours are controlled by the eight locations running from 53287 on, so to colour one sprite you POKE 53287 with one of the colour numbers – for instance:

```
2ØØØ POKE 53287, 9
```

That gives you a brown eagle. Unless you want them all the same colour, for multiple sprites you have to assign the colours individually:

```
2ØØØ POKE 53287, 9:  REM sprite no. 1
2Ø1Ø POKE 53288, Ø:  REM sprite no. 2
2Ø2Ø POKE 53289, 3:  REM sprite no. 3
2Ø3Ø POKE 5329Ø, 12: REM sprite no. 4
2Ø4Ø POKE 53291, 5:  REM sprite no. 5
2Ø5Ø POKE 53292, 13: REM sprite no. 6
2Ø6Ø POKE 53293, 7:  REM sprite no. 7
2Ø7Ø POKE 53294, 1:  REM sprite no. 8
```

You haven't finished yet, either; the sprites have to be turned on and off before they can be seen. This is done by individual bits of location 53269 – if bit 0 is 1, sprite no. 1 is on: if it's 0, sprite no. 1 is off. So to turn sprite no. 1 on, you have to POKE 1 (the decimal equivalent of binary 00000001) into 53269.

Bit 1 relates to sprite no. 2, bit 2 handles sprite no. 3, and so on. Things can get a bit complicated if you have eight sprites turning on and off at different times; you might have to end up with a table something like this . . .

sprite no. 1	ON	0	0	0	0	0	0	0	1
sprite no. 2	ON	0	0	0	0	0	0	1	0
sprite no. 3	ON	0	0	0	0	0	1	0	0
sprite no. 4	ON	0	0	0	0	1	0	0	0
sprite no. 5	ON	0	0	0	1	0	0	0	0
sprite no. 6	ON	0	0	1	0	0	0	0	0
sprite no. 7	ON	0	1	0	0	0	0	0	0
sprite no. 8	ON	1	0	0	0	0	0	0	0
SUM		1	1	1	1	1	1	1	1

This would mean that all sprites are on. But you can work out the binary number for the sprites you want on at a particular time, convert that to decimal, and POKE 53269 with the value.

Displaying Sprites

And now the really tricky part – deciding where you want the sprites to appear on the screen. For a start, you can place and move sprites on a grid that's larger than the screen, so sprites can appear to move smoothly on to the visible area. The sprite area is 255 vertical co-ordinates by 344 horizontally; the screen is a 'window' placed on top of it, on 50 to 299 vertically and 24 to 320 horizontally.

The position of a sprite is determined by where you put its top left-hand 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.

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.

The horizontal and vertical locations for each sprite go in pairs from 53248, as follows:

Sprite no.	*Horizontal*	*Vertical*
1	53248 plus bit 0, 53264	53249
2	53250 plus bit 1, 53264	53251
3	53252 plus bit 2, 53264	53253
4	53254 plus bit 3, 53264	53255
5	53256 plus bit 4, 53264	53257
6	53258 plus bit 5, 53264	53259
7	53260 plus bit 6, 53264	53261
8	53262 plus bit 7, 53264	53263

```
3ØØØ POKE 53269,255
```

That turns all eight sprites on. Now let's have a pleasantly coloured screen:

```
3Ø1Ø PRINT "[CLR]": POKE 53281,15
```

and set up some variables to simplify the positioning of sprites:

```
3Ø15 SH=53248: SV=53249
```

Now we can distribute the sprites around the screen:

```
3Ø2Ø POKE SH,17: POKE SV,55
3Ø3Ø POKE SH+2,19: POKE SV+2,53
3Ø4Ø POKE SH+4,11Ø: POKE SV+4,1ØØ
3Ø5Ø POKE SH+6,119: POKE SV+6,153
3Ø6Ø POKE SH+8,16Ø: POKE SV+8,18Ø
3Ø7Ø POKE SH+1Ø,19Ø: POKE SV+1Ø,2Ø
3Ø8Ø POKE SH+12,21Ø: POKE SV+12,4Ø
3Ø9Ø POKE SH+14,245: POKE SV+14,253
```

Priorities

When you RUN this, you'll note that some of the sprites may overlap and obscure each other and any text on the screen. When that happens, the sprite with the highest priority does the overlapping – and that's the one with the lower or lowest number. So sprite no. 1 gets the absolute priority.

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.

You can change that by location 53275, where once again each bit relates consecutively to each sprite – bit 0 to sprite no. 1, bit 1 to sprite no. 2, and so on. If a bit is set to 1, the corresponding sprite appears *behind* whatever is on the screen.

Once again you might need a helpful table:

sprite no. 1	behind display	0 0 0 0 0 0 0 1
sprite no. 2	behind display	0 0 0 0 0 0 1 0
sprite no. 3	behind display	0 0 0 0 0 1 0 0
sprite no. 4	behind display	0 0 0 0 1 0 0 0
sprite no. 5	behind display	0 0 0 1 0 0 0 0
sprite no. 6	behind display	0 0 1 0 0 0 0 0
sprite no. 7	behind display	0 1 0 0 0 0 0 0
sprite no. 8	behind display	1 0 0 0 0 0 0 0

Changing Shapes

However you've defined the basic sprite shape, your program can alter it by stretching sideways or up and down – 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.

The table again:

sprite no. 1	horizontal expand	0 0 0 0 0 0 0 1
sprite no. 2	horizontal expand	0 0 0 0 0 0 1 0
sprite no. 3	horizontal expand	0 0 0 0 0 1 0 0
sprite no. 4	horizontal expand	0 0 0 0 1 0 0 0
sprite no. 5	horizontal expand	0 0 0 1 0 0 0 0
sprite no. 6	horizontal expand	0 0 1 0 0 0 0 0
sprite no. 7	horizontal expand	0 1 0 0 0 0 0 0
sprite no. 8	horizontal expand	1 0 0 0 0 0 0 0

And for the up-and-down stretch:

sprite no. 1	vertical expand	0 0 0 0 0 0 0 1
sprite no. 2	vertical expand	0 0 0 0 0 0 1 0
sprite no. 3	vertical expand	0 0 0 0 0 1 0 0
sprite no. 4	vertical expand	0 0 0 0 1 0 0 0
sprite no. 5	vertical expand	0 0 0 1 0 0 0 0
sprite no. 6	vertical expand	0 0 1 0 0 0 0 0
sprite no. 7	vertical expand	0 1 0 0 0 0 0 0
sprite no. 8	vertical expand	1 0 0 0 0 0 0 0

POKE the appropriate location with the decimal equivalent of the binary sum you end up with. For binary-to-decimal conversion, see p. 100.

17. How to Create Your Own Characters

The 64 has a goodly range of characters and graphics symbols – two built-in alternatives, in fact: the upper-case-plus-graphics characters you see when the 64 is first switched on, and the lower-case-plus-some-graphics that you get when you press the SHIFT and CBM keys together.

You can check (and alter) which character set is being used. Both sets are stored in memory, but the pointer to the one currently being used is held in location 53272.

Try this:

```
PRINT PEEK(53272)
```

It should return the number 21 – which means the upper-case character set. Now press SHIFT and CBM and try the PEEK again: it should now be 23, which points to the lower-case character set. You can POKE 53272 with 21 or 23 to force a switch between the two.

But the 64 also has the ability for you to create your own shapes and to use them as easily as any of the predefined characters – you can put them on to the screen by typing at the keyboard, you can use them in programs. 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 fool around with the 64's memory locations, displacing some of what's in there already and replacing a few of the 64's characters with your own.

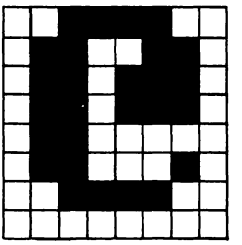
How the 64 Represents Characters

First we need some information about how the 64 stores its characters. On the *screen* each individual character is built up in a kind of map or grid containing eight squares across and eight down, each square being one element of the picture; you get the shape of the character because some of the squares are empty and others are filled.

The 64 *stores* the character shape in a more or less identical manner. Each character takes a block of eight consecutive memory locations, one of each row of the character's shape: and since there are eight bits per location, one bit can correspond to each column in the shape.

The @ sign happens to be the first character in the 64's standard upper-case character set. It occupies the first eight locations, and in those locations you get a pattern of ones and zeros (a binary number, in fact) that corresponds to the presence of filled squares and empty ones respectively:

00111100 = 60
01100110 = 102
01101110 = 110
01101110 = 110
01100000 = 96
01100010 = 98
00111100 = 60
00000000 = 0



Creating Your Own Characters – First Steps

You'd think the easiest way to put your own custom-designed characters on to the screen is simply to put different values into the locations where the 64's standard characters normally reside: if you were to do that for the first eight locations there, you'd be able to include an @ in your programs or to hit the @ key and get the new character every time – right?

Wrong. Sorry, but those locations are inside the 64's read-only memory. And the thing about read-only memory is that you can only read it – you can't *write* anything new into it.

On the other hand, lots of the 64's memory (the bulk of it, in fact) is not read-only. It's RAM, which stands for random-access memory – a bit of a daft name, since there's nothing particularly random about the way you access it. But the main feature of RAM is the fact that you can read from *and* write into it: that's what you're doing when you load a new program or game, or type one in at the keyboard.

As it happens, the 64 doesn't go directly to the memory containing characters when it has something to print. Instead it calls on location 53272. The contents of that can tell you (and the 64) a number of things: the one we're interested in is where the computer is going to look in its memory for the characters it'll put on to the screen.

So if you can set up some characters elsewhere, you could alter 53272 to indicate a different area of memory from which to pull in the characters to be displayed.

Where are you going to put your new characters? Well, you've got a good deal of choice. But these are the safest start locations:

8192
10240
12288
14336

And of these, the best bet is usually 14336. The 64 can access and use a total of 256 characters (irrespective of what they actually are); and since one character needs eight bits to define it, that means a full character set takes up 2,048 locations. That's a lot of characters, but let's say for the sake of argument

that you're going to define (or rather *re*define) a full 256 characters. If you start at 14336 that means you've got all the memory up to 16383 to play with.

Or have you? Maybe not. Normally your *program* is automatically loaded into spare memory – and it will carry on taking up memory until it hits the upper limit you've decided for it. At switch-on, the 64 automatically makes location 40959 the 'highest' point in memory that your program and its associated work needs; so if you just dumped in your special characters at 14336 onwards, you've got a problem – either the program will overwrite the new characters, or vice versa.

So you just tell the 64 not to put programs and associated workspace any higher than 14335. Locations 52 and 56 indicate where the 'top' of Basic memory is; this POKE ensures that it won't be too high:

```
1Ø POKE 52,56: POKE 56,56
```

It makes sense to take a copy of one of the existing character sets and put it into 14336 on – that's to ensure there will be at least some kind of character for each possible position. These lines will do that for you:

```
2Ø POKE 56334, PEEK(56334) AND 254
3Ø POKE 1, PEEK(1) AND 251
```

That gives access to the characters stored at 56334 on.

```
4Ø FOR C=Ø TO 24Ø7
5Ø POKE C+14336, PEEK(53248+C)
6Ø NEXT C
```

This handles the transfer of 2,048 bits containing 256 characters.

```
7Ø POKE 1, PEEK(1) OR 4
8Ø POKE 56334, PEEK(56334) OR 1
```

And that returns the 64 to its original state. It's using the standard character set, but you now have a duplicate copy in 14336 onwards.

New Characters at Last

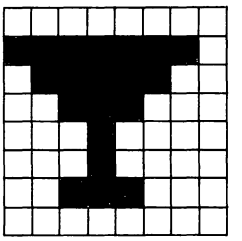
And so we're finally ready to create some characters. It's a doddle. First you need a pencil, a rubber, and a few sheets of squared paper on which you can mark out some 8 × 8 grids. Then you design a new character by filling in the squares that will mark out the shape you want.

Now convert that into eight binary patterns, one for each row, with a 1 for a filled square and a 0 for a blank one.

And finally turn the binary numbers into decimals – if you don't know how to do that, see p. 101. Now you've got a sequence of eight numbers that correspond to the shape of your character.

Let's try it on a Martini glass:

00000000 = 0
11111110 = 254
01111100 = 124
00111000 = 56
00010000 = 16
00010000 = 16
00111000 = 56
00000000 = 0



Right, now you've got your numbers, let's do something with them.

Programming the New Shape

What we're going to do is replace some of the standard characters with your new shapes. For instance, replacing the @ sign with the Martini glass means you'll be able to type the @ key and see the glass appear on the screen instead: and you'll also be able to put the @ character into a program and produce the glass in the same way – so a line reading 'PRINT "@"' would display the booze.

Hit RUN/STOP and RESTORE, then CLR the screen. Now type this:

```
1ØØ DATA Ø, 254, 124, 56, 16, 16, 56, Ø
12Ø FOR A=14336 TO 14336+7
13Ø READ B: POKE A,B: NEXT A
```

What's happening here is that the DATA in line 100 contains the eight rows that make up the shape, and line 120 specifies the eight locations that will hold the data. Line 130 reads the data one item at a time into a variable named B and POKEs it into successive locations.

```
14Ø POKE 53272, (PEEK(53272) AND 24Ø) OR 14
```

When all the data has been copied into the eight locations starting at 14336, that POKE in line 140 tells the 64 it's now got to start looking from 14336 on for the characters to display.

If you type RUN the cursor will disappear for a while and eventually come back with 'READY'. Now try the @ key a few times.

Voila! The glass lives!

Hit RUN/STOP and RESTORE, LIST the program, and add another line:

```
15Ø PRINT "[CLR]@@@@@@@@@@@@@@@@"
```

RUN it again. You should have a row of glasses now, and a demonstration of how easy it is to incorporate the customized characters in a program.

Binary To Decimal

Here is a quick primer on binary-to-decimal conversion. Binary arithmetic works with just two digits (0 and 1) rather than the ten digits in our familiar decimal arithmetic (0 to 9).

In a group of eight binary digits such as the 64 stores in a single location, each successive digit from the right can represent another power of two. So 00000001 is the equivalent of decimal 1 and 00000010 to 2.

Not clear? Ah well, just use this handy chart:

128	64	32	16	8	4	2	1

Wherever there's a 1 in your binary number, add the corresponding decimal figures to make up a total for the whole eight digits. Take the binary form 10111101, for instance:

1	0	1	1	1	1	0	1
128	64	32	16	8	4	2	1

Adding the decimals where there are 1s you get 128+32+16+8+4+1 = 189.

18. Sound

No self-respecting home computer these days can do without some sound-generating functions; and on the 64 you have some of the best currently on the market – good enough in fact for some people (notably Commodore itself) to have produced really quite good speech synthesizers that use the 64's sound generators to imitate the human voice.

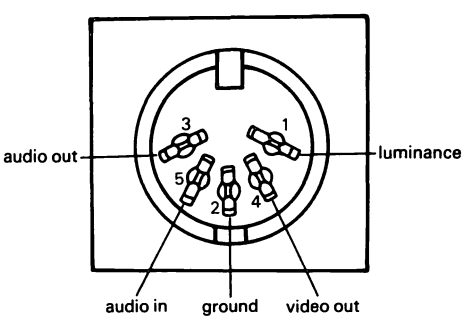
To do this takes a good deal of programming. You might well set your sights a bit lower at first, but even beginners can produce impressive music and sound effects. The 64's sound has three major drawbacks to it, though.

Sound Quality

First (and most easily overcome), there's the quality of reproduction. Normally the 64 plays sound through your TV, and that may be good enough for you. But no matter how sophisticated your sound-generating program, it has to be played through *something*: there is little point in rewriting the '1812 Overture' for the 64 if you're going to play it through the tiny amplifier and often tinny loudspeaker inside an average portable TV. The TV will be adequate for every-day work – but if you want to do something more advanced you'll have to output the sound to a hi-fi system like a home stereo.

Fortunately, the 64 has a socket at the back for this that takes a stereo-type DIN plug. The socket is the one next to the TV connector; despite the *nine* holes in the socket, you need a *five*-pin DIN plug. A hi-fi shop may be able to sell you a ready-wired cable with appropriate plugs at either end, or you can buy the bits and make it up yourself.

Here's the plug:



Note that you don't get two audio outs, because the 64 cannot deliver two-channel stereo sound. As for audio in, no one I know has successfully used it;

but in theory you can take input from something (like an electric guitar) and process that sound signal in the 64.

If you're going to use a hi-fi system for sound reproduction you need pins 2 and 3 to end up on the kind of plug that will fit the AUXiliary socket on your amplifier or tuner. These days that's usually a single-pin phono plug, with the wire from pin 3 of the DIN plug soldered to the central connection on the phono and the ground wire on the outside one. Some systems use a DIN plug at the amp end.

The socket on the 64 does video as well as audio output, clearly, so you'll need to take pins 1 and 4 to a video connector plug – maybe a TV aerial plug, or something that fits a VCR, or (for the best possible picture) a video monitor. A monitor is a kind of TV that can't pick up TV signals; because it doesn't need all the electronics to sort out TV transmissions and compensate for the other garbage cluttering up the air, the picture from a direct computer connection can be much sharper and with more vibrant colours.

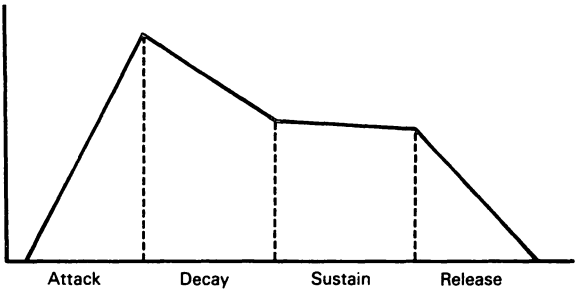
Basics of 64 Sound

The other two problems are more intractable, and you're just going to have to roll up your sleeves and start practising. First, the sheer capability of the 64's sound functions means you get a whole lot of options to play with and different parameters to set: in other words, 64 sound is quite complicated. Second, Commodore somewhat unkindly declined to include some helpful and meaningful commands with which you could use the 64's sound. Instead you have to do it all with POKEs to particular memory locations. As if that wasn't bad enough, you can't use PEEK to check the current setting of those locations – trying that *always* returns the value 0.

Let's get started. This is a summary of what you can do with the 64's sound generator:

- you always have to set the *volume*. This is an overall setting: you can't specify different volumes for different 'voices'.
- you can use any of three 'voices' or tone generators. Up to three of the voices can be in operation at once, so you can play three-note chords.
- each voice can produce four different *waveforms*. These determine the basic type of sound produced – in practice, they approximate to different musical instruments.
- for each voice you can set a *frequency range* or pitch. This is what gives a note its harmonic content or timbre – in other words, it's what makes a musical note sound musical.
- for each voice you have to specify an *Attack/Decay* value. Attack determines how quickly the note 'fades up', so an Attack value of 0 means you start hearing it immediately. Decay is the rate at which it 'fades down'; 0 would be an instant cut-off.
- similarly, for each voice you need a *Sustain/Release* value. Sustain takes over from decay and prolongs the note at the loudness specified by the

Sustain value for the length of time given by the Release value. So putting the four together (and they are usually lumped together by the umbrella term 'ADSR') this is how you might hear a note:



The ADSR combination is sometimes called the sound's 'envelope'.

- also available (though you don't have to use them) are some *modulation* effects, where two tone generators can be combined to produce one complex sound.
- 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.

How to Produce Sound

In short, there are *three* voices: each voice consists of a *tone generator* and an *ADSR* 'envelope' generator. So for any sound-producing program or routine, you need to: set volume; specify waveform for each voice; and set frequency range for each.

These are the memory locations you can POKE to affect sound:

Voice	*Location*	*Effect*
1	54272	low frequency
	54273	high frequency
	54274	lower pulse width
	54275	higher pulse width
	54276	waveform
	54277	Attack/Decay
	54278	Sustain/Release
2	54279	low frequency
	54280	high frequency
	54281	lower pulse width
	54282	higher pulse width
	54283	waveform
	54284	Attack/Decay
	54285	Sustain/Release

Voice	*Location*	*Effect*
3	54286	low frequency
	54287	high frequency
	54288	lower pulse width
	54289	higher pulse width
	54290	waveform
	54291	Attack/Decay
	54292	Sustain/Release
all	54296	master volume

POKEing Values

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 for the voice to be played. See the table for all possible notes in the 64's eight-octave range. In fact the possible range of values for both settings is the usual 0 to 255. And 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.

Musical Notes and Frequencies

Musical Note and Octave	*Frequency High*	*Low*
C – 0	1	12
C# – 0	1	28
D – 0	1	45
D# – 0	1	62
E – 0	1	81
F – 0	1	102
F# – 0	1	123
G – 0	1	145
G# – 0	1	169
A – 0	1	195
A# – 0	1	221
B – 0	1	250
C – 1	2	24
C# – 1	2	56
D – 1	2	90
D# – 1	2	125
E – 1	2	163

Musical Notes and Frequencies

Musical Note and Octave	*Frequency High*	*Low*
F – 1	2	204
F♯ – 1	2	246
G – 1	3	35
G♯ – 1	3	83
A – 1	3	134
A♯ – 1	3	187
B – 1	3	244
C – 2	4	48
C♯ – 2	4	112
D – 2	4	180
D♯ – 2	4	251
E – 2	5	71
F – 2	5	152
F♯ – 2	5	237
G – 2	6	71
G♯ – 2	6	167
A – 2	7	12
A♯ – 2	7	119
B – 2	7	233
C – 3	8	97
C♯ – 3	8	225
D – 3	9	104
D♯ – 3	9	247
E – 3	10	143
F – 3	11	48
F♯ – 3	11	218
G – 3	12	143
G♯ – 3	13	78
A – 3	14	24
A♯ – 3	14	239
B – 3	15	210
C – 4	16	195
C♯ – 4	17	195
D – 4	18	209
D♯ – 4	19	239
E – 4	21	31
F – 4	22	96
F♯ – 4	23	181
G – 4	25	30
G♯ – 4	26	156
A – 4	28	49
A♯ – 4	29	223
B – 4	31	165
C – 5	33	135
C♯ – 5	35	134
D – 5	37	162
D♯ – 5	39	223
E – 5	42	62
F – 5	44	193
F♯ – 5	47	107

Musical Notes and Frequencies

Musical Note and Octave	*Frequency High*	*Low*
G – 5	50	60
G♯ – 5	53	57
A – 5	56	99
A♯ – 5	59	190
B – 5	63	75
C – 6	67	15
C♯ – 6	71	12
D – 6	75	69
D♯ – 6	79	191
E – 6	84	125
F – 6	89	131
F♯ – 6	94	214
G – 6	100	121
G♯ – 6	106	115
A – 6	112	199
A♯ – 6	119	124
B – 6	126	151
C – 7	134	30
C♯ – 7	142	24
D – 7	150	139
D♯ – 7	159	126
E – 7	168	250
F – 7	179	6
F♯ – 7	189	172
G – 7	200	243
G♯ – 7	212	230
A – 7	225	143
A♯ – 7	238	248
B – 7	253	46

Low and high pulse width

Used with the 'pulse' waveform (see below); possible values for the low pulse are 0 to 255 again, for high it's 0 to 15.

Waveform

Again, the possible values are 0 to 255 – but in practice it makes sense to stick with just four:

- 17 – 'triangle' wave; mellow, flute-like sounds
- 33 – 'sawtooth' wave; good for rich sounds like horns or strings
- 65 – 'pulse' wave; define your own wave – can range from hollow clarinet-style sounds to thin oboe-like sound. Varies according to the pulse width
- 129 – 'noise', a fizzy, fuzzy electronic hiss sometimes (erroneously) called 'white noise'. Useful for percussive effects like drumbeats or explosions

Attack/Decay

These are combined in one location. The way to calculate the composite value is

Attack × 16 + Decay

where 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:

Value	*Attack rate*	*Decay rate*
0	2 milliseconds	6 milliseconds
1	8 ms	24 ms
2	16 ms	48 ms
3	24 ms	72 ms
4	38 ms	114 ms
5	56 ms	168 ms
6	68 ms	204 ms
7	80 ms	240 ms
8	100 ms	300 ms
9	250 ms	750 ms
10	500 ms	1.5 seconds
11	800 ms	2.4 s
12	1 second	3 s
13	3 s	9 s
14	5 s	15 s
15	8 s	24 s

Sustain/Release

These too are combined in one location, and again the formula to calculate the composite value is

Sustain × 16 + Release

Sustain interrupts the Decay to prolong the note at its current volume; Release decides how long it's prolonged before allowing it to decay again. Sustain can be any value from 16 (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	6 milliseconds
1	24 ms
2	48 ms
3	72 ms
4	114 ms
5	168 ms
6	204 ms
7	240 ms
8	300 ms
9	750 ms
10	1.5 seconds
11	2.4 s
12	3 s
13	9 s
14	15 s
15	24 s

Poke Locations

Since all the SID chip sound locations run quite neatly from 54272, one easy way to handle them in a program so that you don't have to remember all the location numbers is to kick off with an assignment statement like

```
1ØØ SID=54272
```

Thereafter, SID+4 would reference the waveform location for voice number 1; SID+12 would be voice 2's Attack/Decay setting (54272+12=54284); and so on. (Why SID? Because that's the name Commodore gave to the chip which actually produces the sound – it stands for 'Sound Interface Device'.)

Alternatively, kick off with this (which I actually prefer to use myself):

```
1ØØ V1=54272: V2=54279: V3=54286
```

And you might as well have one for the master volume control too:

```
11Ø VOL=54296
```

Then you can POKE a setting into VOL and use V1, V2 or V3 plus 4 to get at the waveform for any voice: plus 6 for Sustain/Release; and so on. In other words

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

There again, you could make life easier still for yourself by giving these names:

```
12Ø LF=Ø: REM low frequency of note
13Ø HF=1: REM high frequency of note
14Ø LP=2: REM lower pulse width
15Ø HP=3: REM higher pulse width
16Ø WF=4: REM waveform
17Ø AD=5: REM Attack/Decay
18Ø SR=6: REM Sustain/Release
```

So V3+WF now specifies the waveform for voice 3. And you could continue the fill-in-the-blanks approach like this:

```
2ØØ REM set up voices
21Ø FOR N=1 TO 3
22Ø INPUT "WHICH VOICE"; V
23Ø IF V=1 THEN V=V1
24Ø IF V=2 THEN V=V2
25Ø IF V=3 THEN V=V3
3ØØ REM set up note
33Ø INPUT "FREQUENCY RANGE FOR NOTE"; F1,F2
34Ø POKE V+LF,F1: POKE V+HF,F2
35Ø INPUT "DURATION"; T
4ØØ REM set Attack/Decay
41Ø INPUT "ATTACK - Ø TO 15"; A
42Ø INPUT "DECAY - Ø TO 15"; D
43Ø POKE V+AD,A*16+D
5ØØ REM set Sustain/Release
51Ø INPUT "SUSTAIN - Ø TO 15"; S
52Ø INPUT "RELEASE - Ø TO 15"; E
53Ø POKE V+SR,S*16+E
6ØØ REM set up waveform
61Ø INPUT "WHICH WAVEFORM - T, S, N, P"; W$
62Ø IF W$="P" THEN 7ØØ
63Ø IF W$="T" THEN W=17: REM triangle
64Ø IF W$="S" THEN W=33: REM sawtooth
65Ø IF W$="N" THEN W=129: REM noise
66Ø GOTO 8ØØ
7ØØ REM set up Pulse waveform
71Ø W=65
72Ø INPUT "LOW PULSE WIDTH";P1
73Ø INPUT "HIGH PULSE WIDTH";P2
74Ø POKE V+WF,W: POKE  V+LP,P1: POKE V+HP,P2
8ØØ REM another voice?
81Ø INPUT "ANOTHER VOICE - Y OR N"; A$
```

```
 83Ø IF A$="N" THEN 9ØØ
 84Ø IF N=3 THEN PRINT "THREE VOICES
     SPECIFIED": GOTO 9ØØ
 85Ø NEXT N
 9ØØ REM set volume
 91Ø INPUT "VOLUME"; Z
 92Ø POKE VOL, Z
1ØØØ REM timing loop
1Ø1Ø FOR M=1 TO T: NEXT M
1Ø2Ø POKE VOL, Ø: POKE V,Ø
9999 END
```

If you RUN this it will make the noises specified for the length of time given in line 1010. Line 1020 shuts everything off. You can use it to experiment with different effects.

Music

Composing music is slightly more complicated, since you need different durations and 'rests' as well. The easiest way to handle this is to READ a sequence of DATA statements organized in groups of three (high and low frequencies to define the note plus a value for a timing loop to decide its duration) and POKE the data read into the appropriate locations.

For instance:

```
5ØØØ READ H
5Ø1Ø READ L
5Ø2Ø READ DUR
6ØØØ POKE V+HF,H
6Ø1Ø POKE V+LF,L
6Ø2Ø FOR T=1 TO DUR: NEXT
7ØØØ DATA 43, 52, 1ØØØ
```

Assuming you have a voice, volume and ADSR specified somewhere, this should play an E (the 43, 52 in the DATA) for a thousand cycles. Depending on your preference, you could decide that 1000 cycles corresponds to a semibreve – in which case 500 would be a minimum, 250 a crotchet, 375 a dotted crotchet, and 125 a quaver.

Using DATA statements like this, possibly with a DIM in which they can be read, is much the best way to encode music.

Section Four

Buying more of a 64

Adding to the basic system

19. The 64's Printers

A printer is usually one of the first extras that the 64 owner will buy (bank balance permitting). The main reason is the appeal of a printed copy of programs – useful in checking out short programs, probably vital for anything longer.

Commodore sells two printers, the 1515 and 1525. They're both variants of the same unit; Commodore buys them from a Japanese manufacturer called Seikosha which is well known for cheap, simple, and reliable printers.

The 1515 Printer

The 1515 is basically the Seikosha GP-80 with a 64 connection as standard: a number of suppliers can sell you a Seikosha-branded equivalent of the 1515, and a company called Axiom also sells the GP-80 with an own-brand label for connection to any computer.

The 1515 is very compact – surprisingly so if you've seen some of the printers usually sold for computers. It measures 33 by 17 cm (13 by 6.75 ins.), but it does seem a bit high at just over 13 cm (5 ins.) and that makes it look a mite old-fashioned. If that matters, I suspect it's more a comment on you than the printer; inside the thing is a very clever and very up-to-date piece of design that uses very few components and organizes them most impressively.

It prints up to eighty characters a line at a nominal speed of thirty characters a second. I say 'nominal' because with all computer printers you have to take such claims with a truckload of salt: it might print *one* line that fast, particularly if it consists of a simple character like a full stop – but if it has anything more complicated to print, it will take longer. And at the end of each line it has to stop printing to drag the print head back to the left-hand edge before it can get started on the next line. Some printers work 'bi-directionally', which means they print one line left to right and then the next one right to left – so they don't have the return delay.

The 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. It can print 'reversed', too, so the characters come out as white shapes on a dense black background. And it can print anything in double width, useful for headings and emphasis (and pretty pictures).

The 1515 prints on continuous stationery, which 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. It's sometimes called 'fanfold' stationery.

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 in the 1515: 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 roller is moved. If you want to do a single sheet, you'll have to use continuous stationery and neatly trim the edges of one sheet.

The 1515 can cope with paper between 11 cm (4.25 ins.) and 21.5 cm (8.5 ins.) in width. The narrower measure is useful for the kind of stationery you can buy that has peel-off sticky labels on it. In fact, the 64 printers aren't terrifically efficient at sticky labels: some of the labels are pretty heavy-duty and may get stuck inside (where they're a real pig to remove) or they may be too thick to take a decent, dark print.

Anyhow, you'll probably be more interested in buying the usual boxes of paper each sheet of which is 21.5 cm (8.5 ins.) wide and 28 cm (11 ins.) deep. That's not a particularly 'standard' size of paper (most computer stationery is wider) but it is quite readily available from Commodore dealers and elsewhere.

The 1515 leaves a margin on either side when it's printing, so the actual print line is about 16.5 cm (6.5 ins.) in length. The printer does twelve characters per inch, so each printed line can have up to eighty characters in it. On paper 28 cm (11 ins.) long you'll get sixty-six lines on a page.

It's a dot matrix printer, which means it works by pushing tiny needles against an inked ribbon and shoving the ribbon on to the paper to print a small dot. A specific pattern of dots makes up a recognizable character shape, just as the 64 puts characters on to the TV screen – but on the printer the matrix of dots is five wide by seven high. The impact is hard enough to produce carbon copies, if you want to run to the expense of buying the kind of continuous stationery which has carbons interleaved between white sheets (it's called 'multi-part' stationery). The 1515 will manage a top copy and two carbons. There's a simple lever to adjust the force with which the print thumps the ribbon on to the paper, so you could set it at maximum to get the best-quality image on the bottom copy.

In operation the 1515 produces a mellow kind of sound best compared to a number of cats dragging their claws down sheet metal. There's not a lot of sound proofing in the unit, though the see-through plastic cover that slips on over the print mechanism is surprisingly effective at killing some of the noise.

The 1525 Printer

The 1525 is a newer, neater, sleeker and generally more modern-looking unit. It has apparently replaced the 1515, though at the time of writing it's the 1515 that most dealers seem to be stocking still. Internally, it looks much the same, but it prints rather larger and more legible characters (ten per inch rather than twelve) on wider paper – one sheet measures 21 × 28 cm (8.5 × 11 ins.) *after* the perforated strip with the sprocket holes is torn off. This *is* a standard size, and it's much more widely available.

Setting Up

The printer comes with an unpluggable power lead for the mains and a second lead to connect it to the printer. 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 will be plugged into this socket, and the printer then connects to the spare socket at the back of the disk.

The ribbon can be a bit of a pig to fit the first time you do it, but thereafter it's not too bad. It comes 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 slip into plastic lugs inside the printer, with the front edge of the ribbon passing the other side of the print head: you shouldn't get ink on your fingers, but actually getting the cassettes to seat themselves properly on their lugs can take some time.

The ribbon is only about 56 cm (22 ins.) long in total, and it starts 'repeating' itself about every 260 characters. There's quite a lot of ink on it, so you should get a reasonable amount of work from one ribbon before the print quality starts to fade because all the ink's been used: personally, I change ribbons after about 20,000 characters (but then I'm stingy). Ribbons aren't particularly cheap, and you're likely to find that your Commodore dealer is the only local supplier.

Once the ribbon is in you can insert some paper. This too is reasonably straightforward once you've got the knack, though again it can be a bit fiddly. You have to feed one edge through a slot in the back of the printer, grab it when the paper appears at the front, and gently tug it through until you've got enough to reach the sprockets. Be careful: there are a couple of places where the paper can snag as it goes through, and some cheaper brands of paper have an annoying tendency to tear along the perforation before you're ready for it. The trick is to angle the paper and get a corner through first.

Adjust the sprocket holders to the width of the paper – they just slide along – and position the rubber rollers evenly along the paper (they slide too). Then you can engage the sprockets and flip back the covers which clamp the paper on to them. Now you can roll the paper forward with the thumbwheel on the right until you have the paper in position to print. It's safest to roll it on until the next perforation appears above the ribbon – you waste a sheet that way, but at least you won't have a fold or a tear right through the first page of your printing.

Irritatingly, you can't roll the paper *back* with the thumbwheel. So if you move the paper too far forward you either have to disengage the sprockets and pull it back by hand, or you have to carry on rolling and waste a second sheet.

Another slightly annoying aspect of the printer: 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 tubes, you could try making some legs that replace the rubber feet on the underside of the printer: 15 cm (6 ins.) or so should be about right. Alternatively, build or buy a platform – there's at least one manu-

facturer that does a moulded see-through perspex stand, but a wooden stand shouldn't be too difficult to make.

Now you're ready to start printing. First, you might want to check things out, though. Around the back of the printer there's a three-position switch that should be set to 4. 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. It will print all the characters it knows (which means all the characters the 64 can produce, though not in their reversed form) until you move the switch back to 4 or turn off the power.

If you don't get something satisfactory, 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 4 and 5 positions on this switch are there to select the device number. All Commodore peripherals have a device number, and the printer's is usually 4: if you plug in a second printer (unlikely, but just about possible), you have to give them different device numbers unless you want the printing to be done on both printers at the same time. That's what the 5 position is for: you can set the second printer to be device number 5.

Printing

You use the printer by telling the 64 to open a link to it. That's done by an OPEN command followed by a logical file number and a device number.

The device number will usually be 4. And what's a logical file? Difficult to explain simply, that's what. Basically, a logical file is a way of telling the 64 how you want to identify a particular stream of input or output without necessarily telling the 64 where it's coming from or going to. It's a unique number in the range 1 to 255.

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. Actually to persuade the printer to print, you have a version of the PRINT command that has the format PRINT# followed again by the logical file number specified in the OPEN and with a comma coming before whatever it is you want printed.

This will print a little message:

```
OPEN 1,4
PRINT#1,"WOW! IT WORKS!"
```

You can use the printer in direct mode like this. Or you can set it up as a program which will OPEN the link and print your message when you type RUN:

```
10 OPEN 1,4
20 PRINT#1,"WOW! IT WORKS!"
30 CLOSE 1
```

Note that CLOSE command – after OPENing the link you 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.

There's a wrinkle to the OPEN command. As well as the file and device numbers, you can have a third parameter, like this:

```
OPEN 1,4,7
```

That 7 tells the printer to print in the upper-and-lower-case character set. To reset to the normal upper-case-and-graphics, you'd have a 0 there:

```
OPEN 1,4,Ø
```

Clever Ways with PRINT#

Tagging on that 7 is usually done when you want to LIST a program on paper in upper and lower case. In fact there are other ways of slipping between upper and lower case in normal printing by using the PRINT# command, and they're generally easier and more useful to handle. We'll be coming to that.

But first, a word of warning: you can't abbreviate PRINT# in the same way as you can use a question mark instead of an ordinary PRINT command. If you try typing something like –

```
3Ø ?#1, "WOW! IT WORKS!"
```

you'll find that it looks OK when you LIST it on the screen, and the question mark will have been replaced by the word PRINT. But if you try RUNning the program, you'll get a 'SYNTAX ERROR'. You have to type out the 'PRINT#' in full for it to work.

You can PRINT# anything on the printer that you can PRINT on the screen – variables, stuff in quotes, calculations. You can also force the printer to do things that aren't so easy on the screen along with some other formatting and other facilities for which you'd probably use POKEs to achieve on the display. These extras you get by using a CHR$ expression with a code number after the PRINT#. Here's a summary:

CHR$(1Ø)
Forces a line feed after printing: in other words, the paper automatically rolls up one line. But it leaves the print head in the same position within the line that it finished on, so it can be a bit dodgy: the next line you want printed will start from that position rather than the left-hand edge. Advice? Steer clear of CHR$(10).

CHR$(13)
Forces a carriage return after printing. The paper automatically rolls up one line and the print head goes back to the left-hand edge. So in generally you should use this rather than CHR$(10).

CHR$(14)
Everything after it that's to be printed will be in double width.

CHR$(15)
Subsequent printing will be in normal width.

CHR$(16)
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:

```
3Ø PRINT#1, CHR$(16) "Ø8WOW! IT WORKS!"
```

That will print your astonishment starting at the eighth character position along – eight spaces from the left-hand edge, in other words.

CHR$(17)
Everything following will be printed in the upper-and-lower-case character set with some graphics, the same characters that you'd get on the keyboard by hitting the CBM key while holding down SHIFT.

CHR$(18)
Everything following will be printed reversed. The printing will stay that way until you use CHR$(146). Alternatively you can use

```
3Ø PRINT#1,"[CTRL][RVS ON]WOW! IT WORKS!"
```

CHR$(143)
Everything following will be printed in the normal character set of upper-case letters and graphics. That's how you start off, so this is used primarily to get back to normal from a CHR$(17) command.

CHR$(146)
Switches off the reversed print you got from a CHR$(18) to get back to normal.

These CHR$ commands can be mixed in a PRINT# statement. For instance:

```
3Ø PRINT#1, CHR$(1Ø) CHR$(14) CHR$(18) CHR$(143)
   CHR$(16) "Ø6WOW! IT WORKS!"
```

will give a carriage return and then print the message in reversed expanded upper and lower case.

Printing Your Own Graphics

There are these other CHR$ codes, and they're used for printing your own graphics.

CHR$(8)
Sets a graphic mode for the printer. Basically you can define a shape within an 8 × 8 grid and have the printer print the individual dots you specify. Typically, this is done with a DATA statement containing eight values per graphic; those values are the decimal number you get by converting the binary representation of the eight rows that make up a character, with a 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.

CHR$(27)
Works rather like CHR$(16) in that it sets a start position for the printing of user-defined graphics. You need two parameters to specify the start in terms of number of dots away from the left-hand margin.

CHR$(26)
Provides a handy method of repeating any graphic data: they don't have to be your user-defined graphics but you have to have a CHR$(8) preceding them. It's particularly useful for repeating things like bar charts.

Other People's Printers

Commodore's own printers are OK and they aren't an unreasonable price. But there are many other (and probably better) alternatives in the well-filled market-place for printers.

The problem is that you won't be able to plug them in directly; DRG is about the only supplier with an instant plug-in-and-go alternative in the UK, and that's because the printers DRG sells are basically uprated versions of Commodore's own – they come from the same Japanese manufacturers, Seikosha.

Otherwise you'll need some kind of adaptor to persuade a non-Commodore printer to run on the 64: the 64 uses a slightly idiosyncratic method of attaching things.

So if you want to plug in a printer, you've got three choices. You can buy a ready-to-go unit from Commodore or elsewhere that goes straight on. Or 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. This provides what is necessary to plug a Sinclair Microprinter 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 works by scorching the silver coating off the paper in dot-shaped areas to reveal the blackness beneath) and it prints only forty-three characters per line. On the other hand, it does handle graphics rather better than the standard Commodore 1515 or 1525 – the characters are printed larger, so they're more clear – and it's a lot cheaper.

More usually, your non-standard printer will be a conventional dot-matrix plain-paper-and-inked-ribbon printer that offers rather more than the Commodore units but involves a commensurate increase in price. You should expect a better quality print from it, with rather less wobble on the printed lines and true descenders; watch out for the graphics, though – some matrix printers can't manage all the shapes the 64 can produce.

Such printers tend to use one of two standard interfaces (an interface being a standard specification for which wire goes where in a plug-and-socket

connection and exactly what kind of data is being passed along them). Because they are all so different, they actually use plugs and sockets with characteristic shapes.

Many printers have a Centronics interface, which is probably the most widely adopted standard for plugging in printers, and got its name because a manufacturer called Centronics thought of it first. The easiest (perhaps the only way) to use it is to buy one of the several ready-to-go adaptors that plug into the user port, which should 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 many printers therefore assume that you'll want to use RS232. The 64 actually has a kind of RS232 interface built in, but it's a slightly odd version of the standard RS232. More about this in Chapter 21.

20. Disks

A floppy disk drive can be plugged into the 64. It isn't a cheap add-on; but it will allow much greater speed and flexibility in storing and loading programs.

Floppy disks go hand in hand with microcomputers – especially for the more 'serious' uses of micros. That's because you do need the speed of access for complex programming and handling the kind of data files you'd need for business, word-processing, 'what-if' financial planning and the like. The arrival of the microcomputer back in the mid 1970s effectively created the need for something like the floppy disk; and the floppy disk has proved a reasonable and long-lived response to the demand.

For the uninitiated, a floppy disk is a thin disk of plastic that's coated with a brown magnetizable medium. It really is floppy, or at least it would be if the disk wasn't encased in a fairly rigid square cardboard envelope. The envelope has a hole in the middle and a long slot that exposes part of the brown surface of the disk: it's shoved gently into a *disk drive*, where a tapered hub noses into the central hole to spin the disk around. A read/write head touches the disk through the cut-out slot; this is not unlike the record and playback heads in a tape player, since it is able to put information on to the surface of the disk (magnetically) and to read it back again.

Up to a point it's OK to think of a floppy disk as a kind of deformed cassette. Much the same methods are used to store and retrieve information, except that the cassette has a long strip of recording medium while the disk uses concentric rings ('tracks') on a flat surface. But there are two really important differences. First, the disk drive costs a lot more than the cassette deck. And second, disks are a lot faster.

They're faster because they move a lot quicker than tapes, because the read/write head can transfer information to and from the disk more swiftly.

Above all the cassette is slow because it's a serial device. That means that one bit of information is stored after the next along the length of the tape. If you want something that's at the other end of the tape while you happen to be at the start, you'll have to read all through the cassette looking for it.

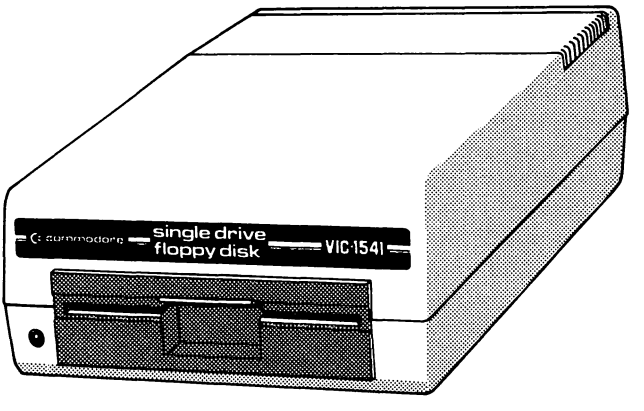
Disks are sometimes described as 'random-access' devices, but that's not strictly true. There's nothing particularly random about them. But they are certainly not serial: when it's looking for something, the read head doesn't have to start at the outer edge of the disk, read all round one track, then move in one and start all over again if it hasn't found it. This is because the read/write head can move. If the disk has some way of telling it which track the information is on, the head can move over to the correct track before it starts reading: so in fact it only has to read a maximum of one trackful of information before it finds what it's looking for. And it is possible to tell the read/write exactly where the information is. This done automatically; the computer or the

disk drive itself maintains a directory of what's where, a 'map' of the disk's contents, and the user need have nothing to do with the process.

Let's see how all this is put into practice on the 64.

The 1541

The original disk drive for the Vic-20 and 64 was called the 1540. The present model is the 1541, a mildly reworked version that looks and works identically but happens to incorporate some extras that allow it to be used with the Commodore 64 as well as the Vic.



Getting started

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 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.

Make no mistake about it, the disk drive is a complicated piece of equipment – and if you're used to venting your neo-Luddite frustrations with the C2N cassette deck by banging it against the wall to make it work, you'd better get used to a more delicate approach.

For a kick-off the 1541 disk drive has in it about the same amount of electronics as the 64 itself. Inside the 1541 is the unit's own microprocessor,

some memory, and integrated circuitry for handling the communications between it and the 64 (that's one of the really good things about the 64, incidentally – all its peripherals have their own 'intelligent' electronic control, so the 64 itself doesn't have to waste too much time looking after them. Nor do they take up the 64's memory: they've all got their own.) Then there's the disk drive itself, a piece of precision engineering. and the clever mechanics for positioning the read/write head. And that lot all leads to a price tag of somewhat more than you paid for the basic 64.

But don't be too disheartened. If you have the uses for it, a disk is probably the most significant system peripheral you'll ever add to the 64. You get a lot of power for your money – and this particular drive compares well for price with the equivalents.

The 1541 is completely software-controlled, coming with its own disk operating system; and because disk operating systems are inevitably quite complicated, efficient use of it necessitates a fair amount of knowledge.

First of all, the manual is terrible. It's full of misprints, for one thing. More important, it doesn't lead you gently and sensibly through the use of the drive in simple fashion – use of commands is explained in an ugly, terse style; and there's a lack of clear examples.

As to operation and performance. though, you can have no real complaints. You plug the disk drive into the serial port at the back of the 64; if you have a printer in that already, you can take it out and plug the printer into an extender port at the back of the disk. All nice and simple.

There's an automatic self-test when you switch on, with a red LED light that flashes if all isn't well. The manual actually says you've got troubles when that light keeps flashing, but it's wrong: the way to stop it is simply to access a disk file – by reading the directory of the loaded disk, for example. That seems to cause no problems.

Using the disks isn't hard, either, though some retries will probably be necessary when you are formatting. You have to 'format' each disk you're going to use; that you do by OPENing a channel to the disk (it's device number 8) and using a PRINT command.

This will include a NEW statement – nothing at all to do with the NEW command in Basic. So you might type:

```
OPEN 1,8,15
PRINT#1,"NEWØ:DISKONE,Ø1"
```

That opens a channel (number 1) to the disk (which is device number 8) and assigns the secondary address 15 to it – as a rule you've always got to have that 15 there. The PRINT also specifies the channel (1) and the command defines a new disk on drive number 0, naming it 'DISKONE' while further identifying it with a unique two-character suffix '01'.

You can shorten the PRINT command:

```
PRINT#1,"N:DISKONE"
```

The NEW needs only its initial letter, in fact, and the 64 will assume you're talking about drive no. 0 if you don't specify a number (since you've probably got only the one disk drive that's acceptable). The identifying suffix is also optional.

Formatting is a once-only job, but the 1541 manual advises you to 'initialize' a disk every time you load it for any reason. This is a precaution rather than a necessity. The format for the initialize command is

```
PRINT#1,"INITIALIZE"
```

or

```
PRINT#1,"I"
```

You can also incorporate it into the OPEN statement:

```
OPEN 1,8,15,"I"
```

The other new commands include two that are self-explanatory. SCRATCH erases something

```
PRINT#1,"S:FILE1,FILE2"
```

You wipe out several files at once by separating them with commas. That's theory, but in practice you might find SCRATCH got bored after two or three and gave up.

RENAME has the new name first, which may not be the natural way to do it (you'd probably prefer to rename OLDFILE as NEWFILE) but is the way computer people have conventionally done things:

```
PRINT#1,"R:NEWFILE=OLDFILE"
```

You can have this in an OPEN statement too:

```
OPEN 1,8,15,"R:NEWFILE=OLDFILE"
```

The VALIDATE command is a housekeeper that will check the contents of a disk, deleting any files that haven't been properly CLOSEd and generally rearranging things to make the most efficient use of the space available. The command can be part of a PRINT or an OPEN:

```
PRINT#1,"V"
```

or

```
OPEN 1,8,15,"V"
```

You can take a look at a directory of files on the disk at any time you want – though this does overwrite anything you happen to have in memory at the time. The format is simple enough:

```
LOAD "$",8
```

Loading is all that this does. To actually *see* the directory you have to type a LIST as well.

Thereafter, you can LOAD, SAVE and VERIFY as normal, except that the file name has to be followed by a comma and the disk's device number:

```
SAVE "INVADERS",8
VERIFY "INVADERS",8
LOAD "INVADERS",8
```

Of course, the quick way to do a VERIFY is to use the abbreviation for SAVE (S and SHIFTed A); then when you get the 'READY' back, move the cursor back up and overtype the abbreviation for VERIFY (V and SHIFTed E).

Hitting RETURN starts the verification with no danger of mistyping the name.

You can use the 64's clever screen editing in the same way to LOAD a program. When you've got the directory on the display with a 'LOAD "$",8' and 'LIST', take the cursor back to the line containing the name of the program you want. The first number there is the file's size, and you overtype that with LOAD. Then move the cursor along to the position following the quotes at the other end of the name, type ',8', press the space bar until the program type identifier has been wiped out, and hit RETURN.

The only problem with this is the fact that some of the file sizes on the screen might be treated by the 64 as line numbers; and if the program you're loading doesn't have the equivalent line numbers, they might appear in the program when you LIST it.

The COPY command is also clever, and it's really redundant for the current state of 64's affairs. It enables you to replicate specific files, giving the copy a new name if you want, and as such it's there primarily to take back-ups from one drive to another. But you can also use it to concatenate (string together) up to four files into one new file with its own name. So you can set up standard routines and call them together with a command like this:

```
PRINT#1,"COPYØ:MYPROG=Ø:SUBONE,Ø:SUBTWO,Ø:
SUBFIVE"
```

Incidentally, disk operating system commands can be abbreviated to a single letter – which is useful in this case, since command lines are restricted to a maximum of forty characters and file names can be quite long.

An OPEN command specifies a logical file number, the device number, and possibly a secondary address; and it associates that lot with a file on disk, which it defines in terms of drive number (always 0), file name, file type, and mode. The mode can be READ or WRITE: file type can be USR, SEQ, or PRG. USR seems to mean a user-specified format. A PRG file is a program, a SEQ is a sequential data file (a very simple cassette-type arrangement in which records can only be read in the order they were written and individual records cannot be altered and amended – all you can do is add new records on to the end of the file). So a full-blown OPEN command might look like this:

```
OPEN 2,8,2,"Ø:FILEX, SEQ, READ"
```

Some of that can be expressed as a variable, which helps:

```
FL$="Ø: FILEX, SEQ, READ"
OPEN 2,8,2,FL$
```

The other Basic commands available to disk users are PRINT#, which transfers a command string to the drive; INPUT#, which picks up data items from the disk and assigns them to a variable; and GET#, which does a similar job for individual bytes of data.

There's a lot more to the 1541, and there's a section on advanced disk programming in the manual that shows you how to get directly to specific parts of the disk for true random access. Using such facilities there's another type of data file you can set up, and it's much more sophisticated; the random access file allows you to read and update individual records (just as on 'real' computers).

21. Other Extras

Joysticks

A joystick gets its name by somewhat stretched analogy with the control stick in aircraft; it plugs into the game port and gives you a stick to push that controls the movement of the cursor or some other object on the screen. So much better than relying on four keys to move your spaceship out of the way of the aliens!

Two joysticks can be plugged into the 64 (they go into the 'game ports' near the on/off switch). That way two people can play a joystick-using game, but obviously you don't *have* to have two joysticks plugged in.

Programs that use two joysticks do have to handle the two ports separately, of course – they have to include commands that sense what's happening at both ports, which way the joystick is being pushed. And a single-joystick program (which, let's face it, is much more likely) won't normally include the programming to cope with *either* port, so you'll be told which one to plug into.

In fact there are two kinds of joysticks that can be plugged into the 64. One is quite clever and very sensitive: the joystick is attached to two potentiometers whose resistances are decided by the position of the joystick. The voltage levels can be read by the 64: one corresponds to the horizontal x-axis, the other to the vertical y-axis. That means the joystick can produce a continuous range of values as it moves round the clock.

These are called *analogue* or *potentiometer* joysticks, and they are not common. About the only one I've seen for the Commodore machines is Stack's. Stack also has a couple of programs that use it; otherwise you'll probably have to write your own.

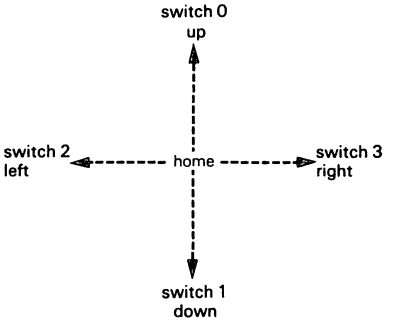
Nearly all games that allow joystick control assume the simpler switch-type joystick popularized by Atari, and in fact most of the joysticks on sale are of this type. (Games and other programs written for the switch type won't work with analogue joysticks.) If you're buying one, Commodore's own joystick is the obvious option: but if you want a more robust unit with a better grip you do have some choice.

These joysticks are really simple: inside the body of the joystick there are four contacts, switches that are closed when the stick is pushed towards them. The switches correspond to the four main compass points. So with the switch type you normally get only four possible readings. Fancier versions stretch to eight – if you push the stick towards the north-east, as it were, the contacts at north and east will both be engaged at once. But that's about the limit. This means the joystick's power of discrimination is considerably less; but it also simplifies the manufacture so they tend to be marginally cheaper and there's less to go wrong.

In practice, however, most joysticks are built down to a price, and fevered use tends to break the contacts quickly on the cheapest joysticks. You're also quite likely to overstrain the spring that returns the stick to the central position. Moral? Buy the best quality you can afford.

Programming for Switch Joysticks

Normally the joystick is upright and none of the switches will be closed. Push it forward and switch 0 closes: pull it to the right and switch 3 closes.



There's also a 'fire' button on the joystick which can be read separately. To include a joystick in your program, you need one of two locations in memory: 56320 for the first one (plugged into port 2), 56321 for the second (port 1), if you're using it.

Single bits in these memory words are set to 1 if nothing is happening; they change to 0 when particular switches in the joystick are 'closed', as they will be when the stick is being moved around. For both of the joystick control locations the bits in question are these:

bit 0 up
bit 1 down
bit 2 left
bit 3 right
bit 4 fire

To see which way the stick is pointing, then, you have to PEEK at 56320 and/or 56321 to find out what the 64 thinks the joystick's doing. And the best way to do that involves some nifty binary-to-decimal conversion and an AND (see Chapter 24):

```
DIRECT=15-(PEEK(5632Ø) AND 15)
```

This statement PEEKs at 56320 (to read the second port it would be 56321); the AND 15 compares the first four bits to 1 and where there's a correspondence it sets the equivalent bit in a working area to 1; and the result is subtracted from 15.

So if the joystick is pointing 'north-east', as it were – up and right – the first four bits of location 56320 will look like this:

bit 0 [up] 0
bit 1 [down] 1
bit 2 [left] 1
bit 3 [right] 0

The AND compares that with the bit pattern for the decimal value 15, which is 1111 in binary; so the result in the workspace word is 0110, decimal 6. Subtracting that from 15 leaves 9. This table of correspondences shows the possible results of applying that DIRECT = statement to the joystick location:

DIRECTion is	*Joystick is pointing*
0	home
1	up
2	down
4	left
5	up and left
6	down and left
8	right
9	up and right
10	down and right

To read the 'fire' button, you have to look at bit 5; and the best way to do *that* is with another AND:

```
FIRE=PEEK (5632Ø) AND 16
```

If bit 5 of the joystick location is 0, it means the fire button is being pressed. This statement compares the contents of 56320 with the binary equivalent of the decimal number 16, and that's 00010000 – so it's applying to bit 5. If the fire button isn't being pressed, the result of the AND will be 1. If it *is* being pressed, the comparison will show that bit 5 of 56320 is different to bit 5 in the comparison word and the result will be 0. Your program can then take some action, if FIRE = 0.

Here's a little routine that will handle the joystick:

```
1ØØØØ JOY1=PEEK(5632Ø): JOY2=PEEK(56321): REM
      create variables for the two joystick
      controllers
1ØØ1Ø S=JOY1 OR JOY2: REM neat, huh? The
      variable S now holds either JOY1 or
      JOY2
1ØØ2Ø FIRE=PEEK(S) AND 16: REM lets you
      check the fire button
1ØØ3Ø DIRECT=15-(PEEK(S) AND 15):
      REM lets you check the joystick
      direction
1ØØ4Ø RETURN: REM go back to the program
```

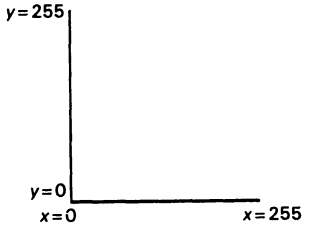
Then you could have something like this inside the main body of the program:

```
1ØØØ GOSUB 1ØØØØ: REM go and look at the
     joystick
1Ø1Ø IF DIRECT=1 THEN 2ØØØ: REM joystick
     pointing north? Go to something that
     happens at line 2ØØØ
1Ø2Ø IF DIRECT=2 THEN 3ØØØ: REM joystick
     pointing south? Go to something that
     happens at line 3ØØØ
1Ø3Ø IF DIRECT=4 THEN 4ØØØ: REM joystick
     pointing west? Go to something that
     happens at line 4ØØØ
1Ø4Ø IF DIRECT=5 THEN 5ØØØ: REM joystick
     pointing northwest? Go to something that
     happens at line 5ØØØ
1Ø5Ø IF DIRECT=6 THEN 6ØØØ: REM joystick
     pointing southwest? Go to something that
     happens at line 6ØØØ
1Ø4Ø IF DIRECT=8 THEN 7ØØØ: REM joystick
     pointing east? Go to something that
     happens at line 7ØØØ
1Ø3Ø IF DIRECT=9 THEN 8ØØØ: REM joystick
     pointing southeast? Go to something
     that happens at line 8ØØØ
1Ø4Ø IF DIRECT=1Ø THEN 9ØØØ: REM joystick
     pointing northeast? Go to something that
     happens at line 9ØØØ
1Ø6Ø GOTO 1ØØØ: REM keep on cycling round
     to check the joystick all the time
```

Paddles and analogue (or potentiometer) joysticks

Potentiometer joysticks are read by the 64 via two totally different locations, which in fact are also used for game paddles. (One effect of this is that programs written for one type of joystick can't be used with the other.)

Analogue joysticks work by having two potentiometers mounted at right angles to each other, in effect to give you an *x*, *y* map looking like this:



The joystick starts in the middle ($x = 127$, $y = 127$). Pushing it in any direction changes the x and y values – and note that you can get a reading as soon as it starts to move: it's not limited to a simple off/on effect at the edges of its range like the switch type.

Game paddles seem more popular in the States than Europe. They're called paddles because they were first developed for and used in 'bouncing ball' games, where you moved a bar up and down your side of the screen to deflect the ball. (Put like that, it doesn't sound much of an explanation for the word 'paddles'. But in some games, bats or racquets are called 'paddles'. See?)

Paddles come in pairs, connected at a plug which fits the game port: there's a wheel on each paddle which is turned to move the bar on the screen, and on some makes they both have a 'fire' button too.

You might think they're a lot simpler than joysticks. After all, you only get two directions of movement, up or down. In fact they're exactly like the less common 'potentiometer' or 'analogue' type of joysticks, and that means they are technically more sophisticated – the paddle actually gives a range of 256 possible positions: turn the wheel to the far right and you're at 0, fully left and you're on 255. Switch-type joysticks are limited to eight positions at most.

And actually most paddles are more sensitive than this implies: you can waggle around at either end of the arc without changing the 0 or 255, for the paddles generally operate only in the middle of the range – often using no more than a quarter of the knob's full range of movement.

Potentiometers give you considerable precision and thus much better control. The drawback is that the range of possible values in the potentiometer joystick/paddle control locations is so great as to be difficult to read from a Basic program – Basic just can't operate quickly enough to keep up with the changes of value. You'll need a machine-code routine to handle it, and that, sadly, is beyond the scope of this book.

Light Pens

A light pen normally plugs into the user port. It generally looks like a biro, and in fact some of them are actually ballpoint-pen bodies with a light sensor at the sharp end and a coiled cable at the other. You point it at the screen, and the computer can read the position at which you're pointing.

They're great for games (though not many published games do allow the use of a light pen) and they are useful for things like menu selection in more 'serious' programs; but they aren't necessarily too accurate – too sensitive at times, not sensitive enough at others. Still, the current crop is a vast improvement.

How do they work? Well, the picture on the TV set is made up of traces of light that are caused by a spot of light flying around the screen – horizontally along parallel lines, in fact, moving from top left on the first line then back the other way along the second, and so on, until it reaches the bottom right when it zips back to the top and starts again. That all happens so quickly the eye can't follow it.

But the sensor in the tip of the light pen can detect the spot passing, and it can relay that to the computer. Inside the 64 there are memory locations inside

which the reading is recording, one for the *x* co-ordinate and one for the *y*. A program can then PEEK at those locations and use them.

The contents of those locations would be constantly changing if there wasn't some way of stopping them when you touch the light pen to the screen – the pen would be constantly detecting the flying spot. So there's a 'trigger', a way of freezing the light pen locations so that the program can check it. The trigger is a pressure sensor that's also in the light pen.

The locations in question are 53267 for the *x* direction (the horizontal axis) and 53268 for *y* (vertical).

The RS232 Interface

RS232 is a true serial interface, probably the most common one in the computer business. Connecting a printer that also uses RS232 is one usual reason for RS232; the other is data communications – sending and receiving data over phone lines, for instance, in which case at both ends of the link you'll need a device called a *modem* that converts the data signals generated by the computer into a form acceptable to the telephone system.

The 64 happens to have an RS232 interface built in. But there are three big problems: first, there isn't a socket that takes an RS232 plug. Second, it's not a standard RS232 interface, so as well as a special cable you'll need some way of getting the 64's RS232 to look like everyone else's. And third, you have to do a bit of extra programming to send stuff to and receive from the RS232 connection – and it's not particularly easy programming.

The first two points are best solved by going out and buying one of the off-the-shelf converter boxes that plug into the user port. These have on them a standard RS232-type socket, and cables are easy enough to come by (there's probably one on your printer already).

How is it used? First you have to OPEN the connection (and you CLOSE it when you're finished). That done, you can accept input from it using GET# and INPUT# and send data to it with PRINT#. So you could OPEN the line – you have to specify a device number (the RS232 chip is device no. 2) and a logical file number – and then PRINT# to it and INPUT# from it:

```
1ØØ OPEN 2,2
11Ø PRINT#2,"ANYONE OUT THERE?"
12Ø INPUT#2,A$
```

The outgoing message would go to a modem and thence to some other computer at the end of a phone line; the response would be sucked into A$ at your end and put on to the screen.

It's a little more complicated than that, of course. And first, some warnings:

- you can't use the disk or a Commodore printer or a cassette deck at the same time as running an RS232 program;
- you'd better have at least 512 bytes of memory free, because the RS232 link takes that much for its send and receive buffers;
- those 512 bytes will overwrite any variables or DIMs you've declared, so make sure you do any variable assignments *after* the OPEN;

● when you OPEN the link you have to specify data transfer speed, something called 'parity', and what type of RS232 interfacing you'll be using.

The parameters for that last one are all determined by the device on the other end of the cable. You can check them out in the handbook for your printer or modem, and you set them up by tagging a couple of entries on to the OPEN command.

The format for OPEN is:

```
OPEN lfn,2,Ø,p1,p2
```

where *lfn* is a logical file number (anything in the range 0 to 255 will do so long as you're not using it elsewhere – 127 is a bit special, though, since it will force an automatic line feed every time a return is sent or received on the communications line). The RS232 chip is treated by the 64 as device no. 2, so that's why that number is there.

And so to the tricky bit: *p1* is a character that indicates the speed of transmission (the baud rate – 'baud' can usually be translated 'bits per second'), the length of each word transmitted (the 'word length'), and the 'stop bits' that indicate the start and end of a transmission. Similarly, *p2* indicates the other key attributes of data transmission – the 'handshaking' convention, full- or half-duplex, and parity checking.

It's easiest to set these up as additions in two CHR$ statements, using the following tables:

p1: Line speed (baud rate)

1	50 baud
2	75 baud
3	110 baud
4	134.5 baud
5	150 baud
6	300 baud
7	600 baud
8	1200 baud
9	1800 baud
10	2400 baud
11	3600 baud
12	4800 baud
13	7200 baud
14	9600 baud
15	19200 baud

p1: Word length

0	8 bits
32	7 bits
64	6 bits
96	5 bits

p1: Stop bits

0	one stop bit
128	two stop bits

p2: Handshaking

0	three lines
1	*x*-On/*x*-Off

p2: Full-/half-duplex

0	half duplex
16	full duplex

p2: Parity

0	none
32	odd
96	even
160	none (mark transmitted)
224	none (space transmitted)

So *p1* might be 'CHR$(1 + 5 + 96)' to set up communications at 50 bps with a five-bit word length and two stop bits: *p2* could be 'CHR$(0 + 0 + 160)' for normal handshaking, full duplex, and mark rather than parity – you could leave out the noughts and use 'CHR$(160)' for the same result. In normal use, most of those options wouldn't be required, though. The 0 settings are the most common ones.

A real example: the normal speed for a modem is 300 baud, with each word transmitted consisting of eight bits and no stop bits. It's an ordinary three-line half-duplex transmission and there's no parity.

Giving the thing a logical file number 2, the OPEN for that would then read:

```
OPEN 2,2,Ø,CHR$(6)+CHR$(Ø)
```

For a printer, a realistic command would set 1200 bps with everything else left at zero:

```
OPEN 2,2,Ø,CHR$(8)+CHR$(Ø)
```

From here on it's easier. There are no oddities about PRINT#, and the only funny thing about the receive commands is a matter of pragmatism – GET# is preferable to INPUT#, because all INPUT statements wait for a return before they let the program get on. If a return never arrives, your program could be waiting for ever. GET# picks up single characters.

The only other consideration is ST, one of the 64's inviolable system variables. It's used at other times for checking the status of I/O operations like reading to or writing from disk or tape: when you OPEN an RS232 channel, it takes on a whole new set of meanings. Basically it can tell you what went wrong when things foul up, but it's a bit tricky to deduce that. The simplest use of ST is to check it – and if it's not zero, kill the transmission and retry.

The IEEE Interface

Commodore tends to connect all external peripherals (disks, printers, etc.) to all of its computers by using what's called the IEEE interface. IEEE is in theory a *parallel* interface, which means all eight bits that make up a character arrive at the same time along eight wires.

That actually happens on the bigger Commodore computers, but for the 64 Commodore produced a kind of cut-down version in which all the bits arrive one after the other. (That's more or less the definition of a 'serial' interface.)

The fact that the 64 doesn't use a 'true' version of the IEEE standard is one reason why standard Commodore peripherals from the Pet/8000 line and the 700s can't be used outright on the 64. But you can buy conversion units – generally cartridges for the expansion port – that do give the 64 instant use of any of the peripherals you might have for a bigger Commodore machine. You shouldn't have to do any extra programming to run printers and the like through one of these plug-on IEEE converters.

Section Five

Troubleshooting

How to get out of trouble

22. What Can Go Wrong Now?

This section lists some of the more common problems and what you can do about them.

Nothing Happens When You Switch On

Presumably you checked that everything *is* switched on – the TV, the 64, and the mains?

1. Is the 64's red POWER light on?
   If yes, go to step 5. If no, try some other small electrical appliance (like a lamp or a radio) in the mains socket that you're using; go to step 2 if it works, go to step 3 if it doesn't.
2. Still nothing?
   Check your home's mains fusebox. There may be a fuse in the socket itself, too. Replace any fuses and try again.
3. Power at the socket?
   If there is power getting to the socket, try replacing the fuses between it and the 64. There's one in the 13 amp plug; and if you're using multi-way adaptors or extension cable adaptors, they have fuses too. Go to step 4.
4. Still no red light?
   Could be the power supply converter; they do burn out from time to time. Not much you can do there, but at least you're probably covered by your guarantee (you *did* return the guarantee card, didn't you?) so take it back to the dealer you got the 64 from.

   Or it could be another fuse. The 64 itself has one inside it, but it's not a standard fuse and you'll need to go to a specialist electrical or electronics shop for a replacement. In any case it's not particularly easy to get at and you may invalidate your warranty by opening up the 64.

   If you want to check it, first make sure the power is disconnected and then open up the 64 (turn it gently and you'll see three Philips-type screws to loosen). Turn it the right way up and ease the top from the bottom. The fuse carrier is on the right, just behind the mains switch. If it should be replaced, you'll need a 3 amp slow-blow fuse.

   Recommendation? Don't open it – take it back to your dealer.
5. Is there anything at all happening on the TV?
   A dirty snowstorm? If yes, go to step 7. If no, the fault is probably in the electrical connection to the TV set. Check its fuses. Go to step 6.
6. Still nothing?
   Sorry, it sounds as though you have a dead TV. Have it repaired. The 64

should be OK, though, so if you have access to a second TV, why not try that one?

7. No 64 display at all?
   If the 64 and the TV both seem to be working but not talking to each other, the fault is either in the tuning or in the video lead connection. Check the lead first: is it snugly in the socket at the back of the 64? Go to 8.
8. Still no display?
   Check the connection to the TV aerial. You have plugged it firmly into the TV's aerial socket, haven't you?
   Still no picture? Go to step 9.
9. Is it tuned?
   If you know the TV has been tuned at some time to pick up BBC and ITV, try replacing the aerial and flipping between the TV channel selections. If you don't get some kind of a picture, there's a fault in the TV. Get it repaired.
   If you do get a picture, replace the aerial with the 64 lead and retune it – select a channel that's not used, make sure the 64 is switched on, and turn the tuning knob or slider or whatever until you get the 'COMMODORE 64 BASIC V2' message.
   If you go through the whole tuning range and still there's no 64 display, the fault is somewhere along the video lead or in the modulator – probably a loose connection. If you're handy with a soldering iron, you might open up the modulator box and the plugs to try sorting it out. But you're better off going back to your dealer.

23. Error Messages and What They Mean

When the 64 itself hits a problem, it puts a message on to the screen preceded by a question mark and followed by 'READY' with the cursor blinking on the line below.

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, that's given here as NNN: it indicates at which line in your program the error was detected, but note that this doesn't necessarily mean the error itself is actually in that line – it may have been caused by something else in the program.

Several of these messages are really obscure and definitely unlikely, some refer only to disk usage, and several involve concepts and commands that are outside the scope of an introductory book like this. But they're all here for the sake of completeness.

BAD DATA ERROR IN LINE NNN
The program was expecting numeric data and it got a character string. Correct the duff command(s).

BAD SUBSCRIPT ERROR IN LINE NNN
You're trying to reference an element in an array that's outside the dimensions you set up with a DIM statement. You'll have to correct the DIM so that the array is larger, or change the array element number so that it's within the range.

BREAK IN LINE NNN
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 leant on the RUN/STOP key.

CAN'T CONTINUE ERROR IN LINE NNN
You've used a CONTinue command, but the program has been deleted, or perhaps amended so that program execution cannot proceed. The most likely cause, however, is that the 64 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.

DEVICE NOT PRESENT
Probably means the device you're trying to get at (usually the printer or the disk, occasionally the cassette) isn't connected to the 64 or the mains. That's easily remedied, of course.

Sometimes, though, you'll get this message for no apparent reason – on my own set-up it sometimes 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 try again.

DIVISION BY ZERO IN LINE NNN
The 64 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
Someone typed too much in response to an INPUT prompt.

FILE ALREADY EXISTS
The 64 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 64 can't see on the tape or disk. It's probably not there; but you may have mistyped the file name or misremembered what you called it.

FILE NOT OPEN
You haven't used an OPEN command when the 64 wants one from you. So OPEN a file and try again.

FILE OPEN
You used an OPEN command on a file that's already open. Either you don't need to OPEN it again, 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 NNN
You've asked the 64 to do too much – you used an expression that has too many brackets or too many functions. Split up the expression somehow; that's good policy anyway, since it will make life easier for anyone (e.g. you) who has to read through and understand the program in the future.

ILLEGAL DIRECT
Most of the 64's commands can be used in immediate mode (i.e. they are executed as soon as you hit RETURN) or in programs (they are executed

only when the program is RUN). But the following are not valid in immediate mode:

```
DATA
DEF FN
GET
GET#
INPUT
INPUT#
```

If you really want to use these, you'll have to write a short program that incorporates what you want to do and RUN it.

ILLEGAL QUANTITY ERROR IN LINE NNN
You have a variable that is outside the 64's range. It usually happens when you're trying to POKE a value less than 0 or above 255. So don't.

LOAD ERROR IN LINE NNN
Something is wrong with an attempted LOAD from cassette – typically the file you're trying to LOAD has been scrambled somehow. You didn't leave the tape on top of the TV, did you? There's not much you can do about this, except put it down to experience – and take more back-up copies in future.

NEXT WITHOUT FOR ERROR IN LINE NNN
The 64 has found a NEXT statement in your program that is not associated with a preceding FOR. You might have missed out the FOR 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 and correct.

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 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 into 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 NNN
The 64 has run out of DATA items to READ – there must be enough entries in your DATA lines to fill all the variables in READ statements. 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 and/or it's creating too many new values for variables. Buy more memory, simplify the

program, or stop it producing so many new values for the 64 to store.

You might also get this error even when 'PRINT FRE(0)' shows that 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 64 that indicates where in the program it has to jump to and when). The solution? Simplify the program.

OVERFLOW ERROR IN LINE NNN

You have a calculation that's produced a number too big for the 64. 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?

REDIM'D ARRAY ERROR IN LINE NNN

The same array name has been used in more than one DIM statement, or you are trying to DIM an array name to which you've already allocated a default value. This is likely to be the result of carelessness: for instance, have you got a
DIM statement within a FOR . . . NEXT loop?

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 64 gets the right 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. 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 subroutine.

STRING TOO LONG ERROR IN LINE NNN

You have too many characters in a string; the maximum allowed is 255. This may have happened because you tried to add two acceptable strings together; 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

You have used an illegal term or construction. The probable cause is mistyping
– common culprits are bad spelling, accidentally hitting one character twice while typing, and too many or too few brackets. Check your program lines.

TYPE MISMATCH ERROR IN LINE NNN

Your program has tried to put the wrong type of value into a variable – string characters into a numeric variable or vice versa. Change the erring command.

UNDEF'D FUNCTION ERROR IN LINE NNN

The program is trying to use a function that you haven't defined by a DEF FN statement. Define your function (and it helps to put all your DEF FNs at the start of your program).

Sometimes this error happens when you weren't actually attempting to reference a user-defined function, but you've just mistyped something that the 64 has detected as a function.

UNDEF'D STATEMENT ERROR IN LINE NNN

You tried to RUN or GOTO or GOSUB a line number that doesn't exist. The target line has been omitted or you got the number wrong.

VERIFY ERROR

The program you're verifying 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.

Section Six

Summaries

24. A Quick Résumé of 64 Basic

This book is not a programmer's textbook, so what you have here is a reference summary and no more. In the examples, lower-case letters have been used to indicate the argument for expressions as follows:

a	any single character
s	any character string
n	any number
i and *m*	an integer (whole number)
r	a real (floating-point) number

Operators

Operators are used to link items in a calculation. There are three possible types, and any of these can be used in combination:

1. Arithmetic	+	addition
	–	subtraction
	*	multiplication
	/	division
	↑	exponentiation (raising to the power of)
2. 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
3. Logical	AND	both expressions must be true to proceed
	OR	either expression must be true to proceed
	NOT	expression must to false to proceed

Commands

Commands can be incorporated into programs in the same way as statements, but they do not operate on information or data within the program itself – instead they work on the program itself or on some other program.

CONT
Continue with the program (after it has been stopped by hitting the RUN/STOP key or by an END or STOP statement in the program itself).

LIST
Prints on the screen the program currently in memory. As it stands it will print the entire program; LIST *n* just lists the single line numbered *n*, LIST *n* – lists from line number *n* to the end, LIST – *n* lists from the start to line number *n*, and LIST *n–m* lists all lines between *n* and *m*.

LOAD
Load program. Loading from cassette, the command can be given as it stands (in which case the first program to be found on the tape will be loaded) or with a program name in full or in part enclosed in quotes (in which event the first program found that matches the name will be loaded).

Loading from disk, the command must always have a program name in quotes followed by a comma and the disk device number which is 8. The name can be abbreviated with an asterisk, in which case the first program found that matches the abbreviated name will be loaded. 'LOAD "$", 8' loads in the disk directory.

NEW
Erases a program from memory and clears out all variables it set up. Without using NEW, typing in a new program may result in a mix of some line numbers and variables from the old one.

RUN
Starts the program running from the beginning. RUN *n* starts it from line number *n*.

SAVE
Saves the program currently in memory. Used as it stands, it will SAVE it on cassette without a name: a name can be assigned by giving one in quotes after the command. For disk, SAVE must be followed by a name and the device number, 8.

VERIFY
Checks a SAVEd program to see if it matches the one in memory. Used alone, VERIFY will check the next program it finds on a cassette: with a name specified in quotes after the command it will look for that one to check. For disk, VERIFY must be followed by a name and the device number, 8.

Variables

Any number of characters can be used as names for variables, though only the first two will be recognized by 64. You cannot use labels that include Basic keywords and the system variables ST, TI and TI$.

Three types of variable can be identified in 64 Basic; the 64 will check that

the value given to the variable is the right kind and will limit the operations performed to those permitted for that type of variable. Types are:

- *integer* – whole numbers (no decimal point) in the range −32768 to +32767. Identified by following the variable name with a per cent sign, e.g. A%=1665.
- *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, e.g. A=3.14159E−4.
- *string variables* – characters (may include numerals) to be used as text, up to 255 characters per string. Identified by a dollar sign after the variable name with the string itself in double quotes, e.g. A$="PHONE 01–607 9489".

Statements

Statements perform operations within a program, and that means they usually appear somewhere after a line number. Most can also be used in 'direct' mode without a line number, in which case they are executed directly you hit RETURN.

CLOSE *n*
Closes an output channel – must be used sooner or later after an OPEN.

CLR
Clears out any variables (not the same as the CLR/HOME key, which just clears the screen). RUN, LOAD and NEW have the same effect but do other things too.

CMD *n*
Sends output to the device number specified instead of the screen. An OPEN has to be used before CMD; CLOSE kills it.

DATA *n*...
Followed by a bunch of constants separated by commas; they are stored in memory in the same order and entered into the program as required by a READ statement.

DEF FN*c*(*i*)
Defines a function (usually a long mathematical formula that you don't want to have to repeat all the time). *c* is an identifying variable of one or two alphabetic characters (so you can DEFine over nine hundred functions with different letter combinations!) and *i* is a numeric variable name that may be used in the formula.

DIM *c*(*i*, . . .)
Defines the DIMension of an array or matrix – several can be set up with one DIM statement at the same time. *c* is the name for the array, and like any variable it can specify string, integer or floating-point contents. The integers in

brackets specify its maximum size in terms of dimensions: if there's only one number it's a one-dimensional array, two makes it 2-D, three makes it 3-D.

For instance, 'DIM S$(22)' sets up a string array of twenty-three elements (the total number of elements is one greater than that specified). 'DIM S%(12,12)' specifies a 12 x 12 array containing up to 169 integer numbers.

END

Stops a program. Not really essential, since when the program runs out of lines to execute it will end anyhow.

FOR *a*=*n* TO *m*

Sets up a loop, a section of program that repeats however many times this statement specifies. *a* is a variable name, *n* the start value, and *m* the end value. A NEXT statement causes the loop to repeat.

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.

GET *a*

Inputs one character at a time from the keyboard (without having to wait for a return) and stores it as variable *a*.

GET# *n,a*

Inputs one character at a time from the device number specified as *n* (it must have been OPENed previously) and stores it as variable *a*.

GOSUB *i*

The program switches to a subroutine starting at line number *i*. To get back to the starting position the subroutine must include a RETURN.

GOTO *i*

The program switches to line number *i*, so that the program no longer has to execute lines in strict numerical order.

IF *expression* THEN *i*

The most important command in Basic; the program switches to line number *i* if the expression is evaluated as 'true', otherwise the next line in sequence is executed. This allows your program to take different courses of action depending on circumstances.

INPUT "*s*"; *a*

Accepts text typed in from the keyboard and stores it as variable *a*. The string "*s*" is optional, but it can be used as a prompt: for instance, 'INPUT "WHAT'S YOUR NAME"; A$' puts the question on the screen (followed automatically by a question mark), and when you've typed something stores that as A$. Omit the prompt (as in 'INPUT A$') and the question mark will still appear on the display.

INPUT# *n, a*
Inputs one block of text at a time from the device number specified as *n* (it must have been OPENed previously) and stores it as variable *a*.

LET *a*=
Followed by an expression and used to set a variable to the value of the expression – which can be a constant (a number, in which case *a* might be unqualified for real numbers or given a per cent symbol for integers) or a string (in quotes, with *a* given the dollar suffix) or a formula that itself may involve other variables (like A=2*π*r).

In fact the word LET is usually omitted altogether. Basic assumes when it sees one or two characters with an equals sign that it's a LET statement.

NEXT *a*
Used to repeat a FOR loop. If the variable is omitted, it's the last loop started that repeats; otherwise the 64 will loop back to the FOR statement named. But if that is not the last FOR, your program will get into a tangle, so the variable name is really only put there for comprehensibility.

ON *a* GOSUB *n*, . . .
A cleverer variant of IF . . . THEN. *a* can be a variable or an expression; either way, if *a* has an integer value of 1 when it's worked out, the program will go to the subroutine starting at the first line number *n* given here. If it's 2, the subroutine at the second line number specified will be jumped to – and so on.

ON *a* GOTO *n*, . . .
An alternative form that takes the program simply to the alternative line numbers *n*, . . . in the same way.

OPEN *n1, n2, n3, s*
OPENs an I/O channel to an externally connected device (disk or printer usually). The first number *n1* is a logical file number; *n2* is the device number; *n3* is a secondary address that may be specified for some types of communication. *s* is a command string that may in some cases be included in an OPEN statement rather than a PRINT#.

POKE *i, a*
Place *a* (which may be a value or the contents of a variable) into memory location *i*.

PRINT . . .
There are numerous options for specifying what is to be printed (or rather displayed on the screen). It can be a value, a string in quotes (including cursor and colour controls), the contents of a specified variable, the result of a formula, or a combination of all of these.

PRINT# *n*, . . .
Directs output to a device number *n* which has previously been OPENed. This can actually be a printer, but it could equally be a non-printing device like a disk. The options include all those for PRINT alone, plus command strings (e.g. 'PRINT#8, "N:NEWDISK" ' is a formatting command for a new disk).

READ *a*, . . .
Fills up a list of variables starting with *a* by reading from a list of data items set up by the next DATA statement following it in the program.

REM *comment*
Basic ignores 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 the sequence. This command 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
Halts a program – just like END, except that you're told the line number at which your STOP was executed.

WAIT *i1*, *i2*, *i3*
Causes the program to pause while the contents of memory location *i1* are ORed (in the exclusive sense of 'or') with *i3* (if it is present) and then logically ANDed with *i2* (which is mandatory). When the result is *not* zero, the program continues.

String Variable Functions

String variables have their own set of special functions:

ASC("*s*")
Gives the ASCII value of the first character in the string *s*; e.g. 'ASC("D")' gives 68.

CHR$(*i*)
Gives the character whose ASCII code is *i*; e.g. 'CHR$(92)' gives the pound sign.

LEFT$("*s*", *i*)
Gives the *i* characters on the left of the string *s*; e.g. 'LEFT$("01–607 9489",2)' gives 01.

LEN("*s*")
Gives the length of the string *s*. Spaces and other non-printing characters are counted; e.g. 'LEN("01–607 9489")' gives the value 11.

MID$("*s*", *m*, *i*)
Gives the *i* characters starting at character number *m* in the string *s*; e.g. 'MID$("01–607 9489",4,3)' gives 607.

RIGHT$("*s*", *i*)
Gives the *i* characters on the right of the string *s*; e.g. 'RIGHT$("01–607 9489", 4)' gives 9489.

STR$(*n*)
Creates a string of characters containing the number *n*; e.g. 'STR$(9489)' gives the string 9489.

VAL("*s*")
Turns the string *s* into a number; e.g. 'VAL ("9489")' gives the real number 9489. If the first character of the string is not a sign or a digit, VAL will return the value 0.

Print and Display Formatting Functions

POS(Ø)
Gives the current position of the cursor in terms of character position across the screen. The (0) means nothing – it is a dummy argument.

SPC(*i*)
Used with a PRINT command to put spaces into a line: 'PRINT "HERE" SPC(13) "THERE"' puts the first word on the left of the screen and the other at the right because it inserts thirteen spaces between them.

TAB(*i*)
Used with a PRINT command to tab to the character position indicated as *i* – so 'PRINT "HERE" TAB(13) "THERE"' puts the first word on the left of the screen (position no. 1) and starts the other at position no. 13.

Arithmetical Functions

ABS(*n*)
Gives the absolute value of the number *n* – which means the value without its sign.

EXP(*n*)
Exponent – gives *e* to the power of *n* (*n* must be less than 88.02969192).

`FNaa(n)`
Specifies a user-defined function set up with a DEF FN statement.

`INT(n)`
Gives the integer portion of *n*. Add 0.5 to round up – INT(*n*+.5).

`LOG(n)`
Gives the logarithm of *n* to the base *e*. For log base 10, use LOG(*n*)/LOG(10).

`RND(n)`
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(n)`
Gives the sign of *n*: +1 if positive, 0 if zero, −1 if negative.

`SQR(n)`
Gives the square root of *n* (*n* cannot be negative).

Trigonometric Functions

`ATN(n)`
Arctangent – gives the angle (in radians) whose tangent is *n*.

`COS(n)`
Gives the cosine of the angle *n* in radians.

`SIN(n)`
Gives the sine of the angle *n* in radians.

`TAN(n)`
Gives the tangent of the angle *n* in radians.

System Variables

There are three key system variables that may be looked at but which should be changed only with considerable care.

`TI`
TI is the system clock. It starts counting time in 'jiffies', a jiffy being 1/60th of a second.

`TI$`
TI$ expresses the TI clock in hours, minutes, seconds.

ST
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(Ø)
Indicates how much memory is left to you for Basic programs; normally used as 'PRINT FRE(0)'. The (0) means nothing – it's a dummy argument.

PEEK(*i*)
Returns the contents of the memory location *i* (must be in the range 0 to 65535).

SYS *i*
Switches to the memory location *i* 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(*i*)
Calls a machine-language subroutine whose starting address is currently stored in locations 1 and 2. *i* is a parameter used by the subroutine.

Reference Summary

This chapter isn't laid out alphabetically, so here's a quick-reference index for specific commands:

25. Memory Maps

This chapter is reprinted with permission from *Commodore User* magazine, based on maps prepared by Jim Butterfield and adapted by Dennis Jarrett.

0	Chip directional register
1	Chip I/O; memory & tape control
3–4	Float-Fixed vector
5–6	Fixed-Float vector
7	Search character
8	Scan-quotes flag
9	TAB column save
10	0 = LOAD, 1 = VERIFY
11	Input buffer pointer/no. of subscript
12	Default DIM flag
13	Type: 255 = string, 00 = numeric
14	Type: 128 = integer, 00 = floating point
15	DATA scan/LIST quote/memory flag
16	Subscript/FNx flag
17	0 = INPUT; 64 = GET; 152 = READ
18	ATN sign/Comparison eval flag
19	Current I/O prompt flag
20–21	Integer value
22	Pointer: temporary string stack
23–24	Last temporary string vector
25–33	Stack for temporary strings
34–37	Utility pointer area
38–42	Product area for multiplication

Basic

43–44	'Start of Basic' pointer
45–46	'Start of variables' pointer
47–48	Pointer: Start-of-Arrays
49–50	Pointer: End-of-Arrays
51–52	Pointer: String-storage (moving down)
53–54	Utility string pointer
55–56	Pointer: Limit-of-memory
57–58	Current BASIC line number
59–60	Previous BASIC line number
61–62	Pointer: BASIC statement for CONT.

63–64 Current DATA line number
65–66 Current DATA address
67–68 Input vector
69–70 Current variable name
71–72 Current variable address
73–74 Variable pointer for FOR/NEXT
75–76 Y-save; op-save; BASIC pointer save
77 Comparison symbol accumulator
78–83 Misc work area, pointers, etc
84–86 Jump vector for functions
87–96 Misc numeric work area
97 Accum no. 1: Exponent
98–101 Accum no. 1: Mantissa
102 Accum no. 1: Sign
103 Series evaluation constant pointer
104 Accum no. 1 hi-order (overflow)
105–110 Accum no. 2: Exponent, etc.
111 Sign comparison, Acc no. 1 vs no. 2
112 Accum no. 1 lo-order (rounding)
113–114 Cassette bull len/Series pointer
115–138 CHRGET subroutine; get BASIC char
122–123 BASIC pointer (within subroutine)
139–143 RND seed value
144 Status word ST
145 Keyswitch PIA: STOP and RVS flags
146 Timing constant for tape
147 Load = 0, Verify = 1
148 Serial output: deferred character flag
149 Serial deferred character
150 Tape EOT received
151 Register save
152 How many open files
153 Input device, normally 0
154 Output CMD device, normally 3
155 Tape character parity
156 Byte-received flag
157 Direct = 128/RUN = 0 output control
158 Tape Pass 1 error log/char buffer
159 Tape Pass 2 error log corrected
160–162 Jiffy clock HML
163 Serial bit count/EOI flag
164 Cycle count
165 Countdown, tape write/bit count
166 Tape buffer pointer
167 Tape Wrt ldr count/Rd pass/inbit
168 Tape Wrt new byte/Rd error/inbit cnt
169 Wrt start bit/Rd bit err/stbit
170 Tape Scan; Cnt; Ld; End/byte assy
171 Wr lead length/Rd checksum/parity

172–173 Pointer: tape buffer, scrolling
174–175 Tape end adds/End of program
176–177 Tape timing constants
178–179 Pointer: start of tape buffer
180 1 = Tape timer enabled; bit count
181 Tape EOT/RS232 next bit to send
182 Read character error/outbyte buffer
183 No. of characters in file name
184 Current logical file
185 Current secondary address
186 Current device
187–188 Pointer to file name
189 Wr shift word/Rd input char
190 No. of blocks remaining to Wr/Rd
191 Serial word buffer
192 Tape motor interlock
193–194 I/O start address
195–196 Kernal setup pointer
197 Last key pressed
198 No. of characters in keyboard buffer
199 Screen reverse flag
200 End-of-line for input pointer
201–202 Input cursor log (row, column)
203 Which key: 64 if no key
204 0 = flash cursor
205 Cursor timing countdown
206 Character under cursor
207 Cursor in blink phase
208 Input from screen/from keyboard
209–210 Pointer to screen line
211 Position of cursor on above line
212 0=direct cursor, else programmed
213 Current screen line length
214 Row where cursor lives
215 Last inkey/checksum/buffer
216 No. of INSERTs outstanding
217–242 Screen line link table
243–244 Screen colour pointer
245–246 Keyboard pointer
247–248 RS232 receive pointer
249–250 RS232 transmit pointer
256–266 Floating to ASCII work area
256–318 Tape error log
256–511 Processor stack area
512–600 BASIC input buffer
601–610 Logical file table
611–620 Device number table
621–630 Sec adds table
631–640 Keyboard buffer

641–642 Start of BASIC Memory
643–644 Top of BASIC Memory
645 Serial bus timeout flag
646 Current colour code
647 Colour under cursor
648 Screen memory page
649 Maximum size of keyboard buffer
650 Repeat all keys
651 Repeat speed counter
652 Repeat delay counter
653 Keyboard shift/control flag
654 Last shift pattern
655–656 Keyboard table setup pointer
657 Keyboard shift mode
658 0 = scroll enable
659 RS232 control reg
660 RS232 command reg
661–662 Bit timing
663 RS232 status
664 No. of bits to send
665 RS232 speed/code
667 RS232 receive pointer
668 RS232 input pointer
669 RS232 transmit pointer
670 RS232 output pointer
671–672 IRQ save during tap I/O
673 CIA 2 (NMI) interrupt control
674 CIA 1 Timer A control log
675 CIA 1 Interrupt log
676 CIA 1 Timer A enabled flag
677 Screen row marker
704–766 (Sprite 11)
768–769 Error message link
770–771 BASIC warm start link
772–773 Crunch BASIC tokens link
774–775 Print tokens link
776–777 Start new Basic code link
778–779 Get arithmetic element link
780 SYS A-reg save
781 SYS X-reg save
782 SYS Y-reg save
783 SYS status reg save
784–785 USR function jump
788–789 Hardwards interrupt vector
790–791 Break interrupt vector
792–793 NMI interrupt vector
794–795 OPEN vector
796–797 CLOSE vector
798–799 Set-input vector

800–801 Set-output vector
802–803 Restore I/O vector
804–805 INPUT vector
806–807 Output vector
808–809 Test-STOP vector
810–811 GET vector
812–813 Abort I/O vector
814–815 Warm start vector
816–817 LOAD link
818–819 SAVE link
828–1019 Cassette buffer
832–894 (Sprite 13)
896–958 (Sprite 14)
960–1023 (Sprite 15)
1024–2047 Screen memory
2048–40959 BASIC RAM memory
32768–40959 Alternate: ROM plug-in area
40960–49151 ROM: BASIC
40960–49151 Alternate: RAM
49152–53247 RAM memory

Video chip [6566 VIC II]

53248 Sprite 0 horizontal location (low-order byte)
53249 Sprite 0 vertical location
53250 Sprite 1 horizontal location (low-order byte)
53251 Sprite 1 vertical location
53252 Sprite 2 horizontal location (low-order byte)
53253 Sprite 2 vertical location
53254 Sprite 3 horizontal location (low-order byte)
53255 Sprite 3 vertical location
53256 Sprite 4 horizontal location (low-order byte)
53257 Sprite 4 vertical location
53258 Sprite 5 horizontal location (low-order byte)
53259 Sprite 5 vertical location
53260 Sprite 6 horizontal location (low-order byte)
53261 Sprite 6 vertical location
53262 Sprite 7 horizontal location (low-order byte)
53263 Sprite 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
[bit 3] number of rows

[bit 4] blank screen
[bit 5] bit-map mode
[bit 6] extended background colour mode
[bit 7] raster position
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
[bit 3] number of columns
[bit 4] multicolour mode
53271 [bit 0] Sprite 0 double-height mode
[bit 1] Sprite 1 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
[bit 3] bit-map pointer
[bits 4–7] character screen or colour map pointer
53273 interrupt flag register
53274 IRQ mask register
53275 [bit 0] Sprite 0 priority register
[bit 1] Sprite 1 priority register
[bit 2] Sprite 2 priority register
[bit 3] Sprite 3 priority register
[bit 4] Sprite 4 priority register
[bit 5] Sprite 5 priority register
[bit 6] Sprite 6 priority register
[bit 7] Sprite 7 priority register
53276 [bit 0] Sprite 0 multicolour mode
[bit 1] Sprite 1 multicolour mode
[bit 2] Sprite 2 multicolour mode
[bit 3] Sprite 3 multicolour mode
[bit 4] Sprite 4 multicolour mode
[bit 5] Sprite 5 multicolour mode
[bit 6] Sprite 6 multicolour mode
[bit 7] Sprite 7 multicolour 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 [bits 0 to 3] border colour
53281 [bits 0 to 3] screen colour
53282 [bits 0 to 3] character multicolour 1
53283 [bits 0 to 3] character multicolour 2
53284 [bits 0 to 3] Sprite multicolour 1
53285 [bits 0 to 3] Sprite multicolour 2
53286 [bits 0 to 3] Sprite 0 colour
53287 [bits 0 to 3] Sprite 1 colour
53288 [bits 0 to 3] Sprite 2 colour
53289 [bits 0 to 3] Sprite 3 colour
53290 [bits 0 to 3] Sprite 4 colour
53291 [bits 0 to 3] Sprite 5 colour
53292 [bits 0 to 3] Sprite 6 colour
53293 [bits 0 to 3] Sprite 7 colour

Sound chip [6581 SID]

54272 Voice 1 low frequency
54273 Voice 1 high frequency
54274 Voice 1 low pulse width
54275 [bit 0 to 3] Voice 1 high frequency
54276 [bits 4 to 7] Voice 1 waveform type
54277 [bits 0 to 3] Voice 1 decay
[bits 4 to 7] Voice 1 attack
54278 [bits 0 to 3] Voice 1 release
[bits 4 to 7] Voice 1 sustain
54279 Voice 2 low frequency

54280 Voice 2 high frequency
54281 Voice 2 low pulse width
54282 [bits 0 to 3] Voice 2 high frequency
54283 [bits 4 to 7] Voice 2 waveform type
54284 [bits 0 to 3] Voice 2 decay
[bits 4 to 7] Voice 2 attack
54285 [bits 0 to 3] Voice 2 release
[bit 4 to 7] Voice 2 sustain
54286 Voice 3 low frequency
54287 Voice 3 high frequency
54288 Voice 3 low pulse width
54289 [bits 0 to 3] Voice 3 high frequency
54290 [bits 4 to 7] Voice 3 waveform type
54291 [bits 0 to 3] Voice 3 decay
[bits 4 to 7] Voice 3 attack
54292 [bits 0 to 3] Voice 3 release
[bits 4 to 7] Voice 3 sustain
54293 [bits 0 to 2] 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 to 7] resonance
54296 [bits 0 to 3] master volume
[bits 4 to 7] passband

CIA 1 [6526 IRQ]

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 to 6] clock – hours
[bit 7] AM/PM indicator
56332 serial port I/O butter
56333 CIA interrupt control
56334 [bit 0] start/stop timer A
[bits 1–6] timer A control
[bits 7] clock control
56335 [bit 0] start/stop timer B
[bits 1–6] timer B control
[bit 7] set alarm or clock

CIA 2 [6526 NMI]

56576 [bits 0–1] video bank select
[bits 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 to 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
[bits 7] clock control
56391 [bit 0] start/stop timer B
[bits 1–6] timer B control
[bit 7] set alarm or clock

Reserved for future I/O expansion

56832–57087
57088–57343

Operating system

57344–65535 ROM
65409–65525 Jump table

26. Abbreviations for Commands and Functions

ABS	A SHIFT B	A	
AND	A SHIFT N	A	
ASC	A SHIFT S	A	
ATN	A SHIFT T	A	
CHR$	C SHIFT H	C	
CLOSE	CL SHIFT O	CL	
CLR	C SHIFT L	C	
CMD	C SHIFT M	C	
CONT	C SHIFT O	C	
DATA	D SHIFT A	D	
DEF	D SHIFT E	D	
DIM	D SHIFT I	D	
END	E SHIFT N	E	
EXP	E SHIFT X	E	
FOR	F SHIFT O	F	
FRE	F SHIFT R	F	
GET	G SHIFT E	G	
GOSUB	GO SHIFT S	GO	
GOTO	G SHIFT O	G	
INPUT#	I SHIFT N	I	
LET	L SHIFT E	L	
LEFT$	LE SHIFT F	LE	
LIST	L SHIFT I	L	
LOAD	L SHIFT O	L	
MID$	M SHIFT I	M	
NEXT	N SHIFT E	N	
NOT	N SHIFT O	N	

OPEN	O SHIFT P	O	
PEEK	P SHIFT E	P	
POKE	P SHIFT O	P	
PRINT	?	?	
PRINT#	P SHIFT R	P	
READ	R SHIFT E	R	
RESTORE	RE SHIFT S	RE	
RETURN	RE SHIFT T	RE	
RIGHT$	R SHIFT I	R	
RND	R SHIFT N	R	
RUN	R SHIFT U	R	
SAVE	S SHIFT A	S	
SGN	S SHIFT G	S	
SIN	S SHIFT I	S	
SPC(	S SHIFT P	S	
SQR	S SHIFT Q	S	
STEP	ST SHIFT E	ST	
STOP	S SHIFT T	S	
STR$	ST SHIFT R	ST	
SYS	S SHIFT Y	S	
TAB(	T SHIFT A	T	
THEN	T SHIFT H	T	
USR	U SHIFT S	U	
VAL	V SHIFT A	V	
VERIFY	V SHIFT E	V	
WAIT	W SHIFT A	W	

Screen Character Codes

SET 1	SET 2	POKE	SET 1	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	SPACE	32			64	SPACE		96
A	a	1	!	33		A	65			97
B	b	2	"	34		B	66			98
C	c	3	#	35		C	67			99
D	d	4	$	36		D	68			100
E	e	5	%	37		E	69			101
F	f	6	&	38		F	70			102
G	g	7	'	39		G	71			103
H	h	8	(	40		H	72			104
I	i	9	)	41		I	73			105
J	j	10	*	42		J	74			106
K	k	11	+	43		K	75			107
L	l	12	,	44		L	76			108
M	m	13	-	45		M	77			109
N	n	14	.	46		N	78			110
O	o	15	/	47		O	79			111
P	p	16	0	48		P	80			112
Q	q	17	1	49		Q	81			113
R	r	18	2	50		R	82			114
S	s	19	3	51		S	83			115
T	t	20	4	52		T	84			116
U	u	21	5	53		U	85			117
V	v	22	6	54		V	86			118
W	w	23	7	55		W	87			119
X	x	24	8	56		X	88			120
Y	y	25	9	57		Y	89			121
Z	z	26	:	58		Z	90			122
[		27	;	59			91			123
£		28	<	60			92			124
]		29	=	61			93			125
↑		30	>	62			94			126
←		31	?	63			95			127

Codes 128 to 255 are reversed images of codes 0 to 127.

CBM ASCII Codes in Full for PRINT CHR$

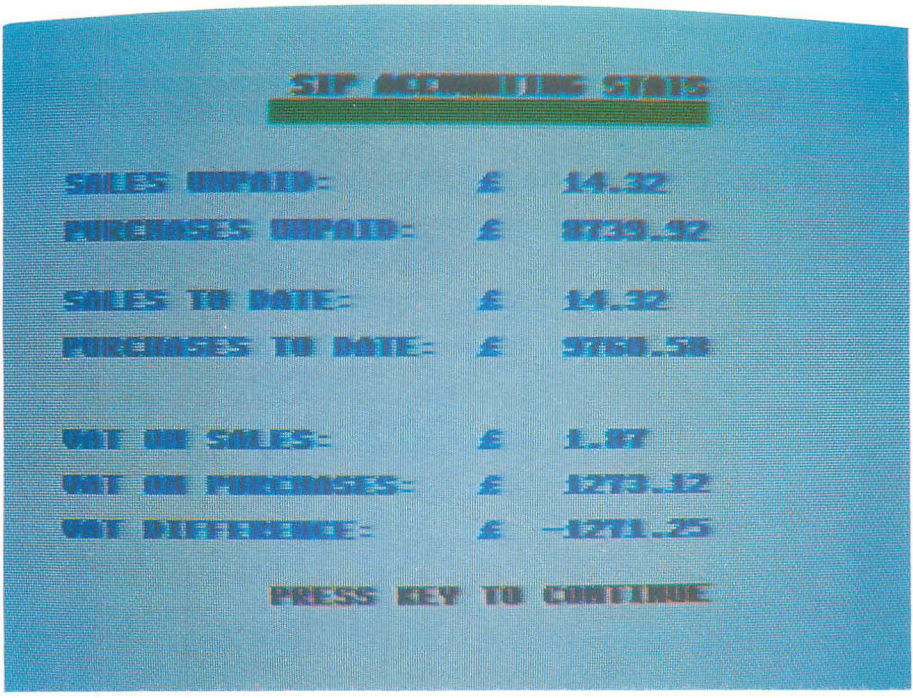
Prints	CHR$	Prints	CHR$	Prints	CHR$	Prints	CHR$	Prints	CHR$
		%	37	K	75		114		153
	0	&	38	L	76		115		154
	1	'	39	M	77		116		155
	2	(	40	N	78		117	PUR	156
	3	)	41	O	79		118	CRSR	157
	4	*	42	P	80		119	YEL	158
WHT	5	+	43	Q	81		120	CYN	159
	6	,	44	R	82		121	SPACE	160
	7	–	45	S	83		122		161
DISABLES SHIFT C=	8	.	46	T	84		123		162
ENABLES SHIFT C=	9	/	47	U	85		124		163
	10	0	48	V	86		125		164
	11	1	49	W	87		126		165
	12	2	50	X	88		127		166
RETURN	13	3	51	Y	89		128		167
SWITCH TO LOWER CASE	14	4	52	Z	90		129		168
	15	5	53	[	91		130		169
	16	6	54	£	92		131		170
CRSR	17	7	55	]	93		132		171
RVS ON	18	8	56	↑	94	f1	133		172
CLR HOME	19	9	57	←	95	f3	134		173
INST DEL	20	:	58		96	f5	135		174
	21	;	59	♠	97	f7	136		175
	22	<	60		98	f2	137		176
	23	=	61		99	f4	138		177
	24	>	62		100	f6	139		178
	25	?	63		101	f8	140		179
	26	@	64		102	SHIFT RETURN	141		180
	27	A	65		103	SWITCH TO UPPER CASE	142		181
RED	28	B	66		104		143		182
CRSR	29	C	67		105	BLK	144		183
GRN	30	D	68		106	CRSR	145		184
BLU	31	E	69		107	RVS OFF	146		185
SPACE	32	F	70		108	CLR HOME	147		186
!	33	G	71		109	INST DEL	148		187
"	34	H	72		110		149		188
#	35	I	73		111		150		189
$	36	J	74		112		151		190
					113		152		191

Codes 192–223 are the same as 96–127
Codes 224–254 are the same as 160–190
Code 255 is the same as 126

27. Code Tables

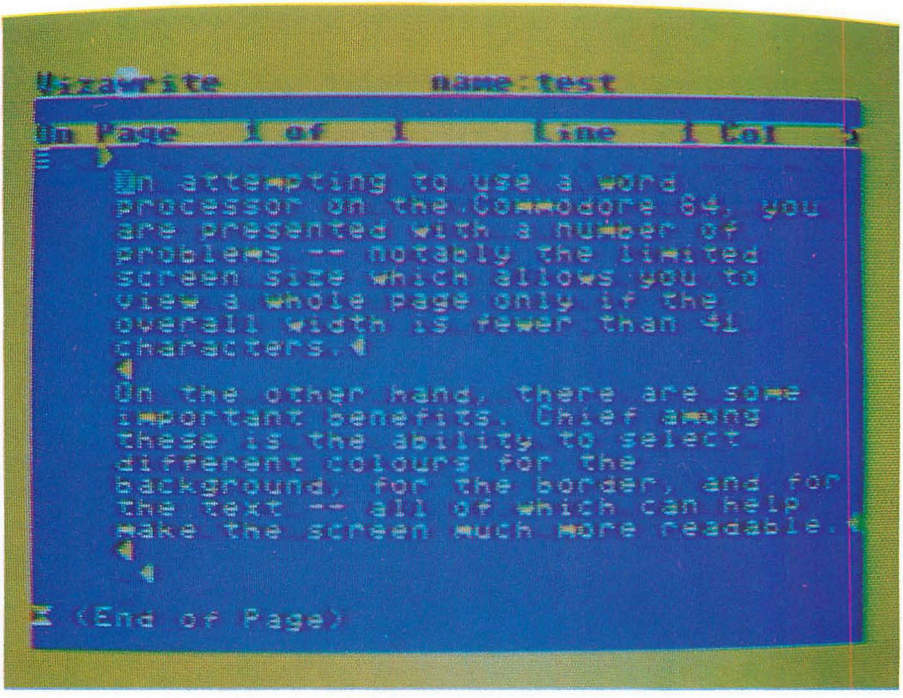
Screen Control Codes

PRINT CHR$	*Effect*	*Appears as*
5	PRINT WHITE	
13	RETURN	
14	SWITCH TO LOWER CASE	
17	CURSOR DOWN	
18	REVERSE FIELD ON	
19	HOME	
20	DELETE	
28	PRINT RED	
29	CURSOR RIGHT	
30	PRINT GREEN	
31	PRINT BLUE	
133	FUNCTION KEY FI	
134	FUNCTION KEY F3	
135	FUNCTION KEY F5	
136	FUNCTION KEY F7	
137	FUNCTION KEY F2	
138	FUNCTION KEY F4	
139	FUNCTION KEY F6	
140	FUNCTION KEY F8	
141	SHIFT AND RETURN	
142	SWITCH TO UPPER CASE	
144	PRINT BLACK	
145	CURSOR UP	
146	REVERSE FIELD OFF	
147	CLEAR	
148	INSERT	
156	PRINT PURPLE	
157	CURSOR LEFT	
158	PRINT YELLOW	
159	PRINT CYAN	

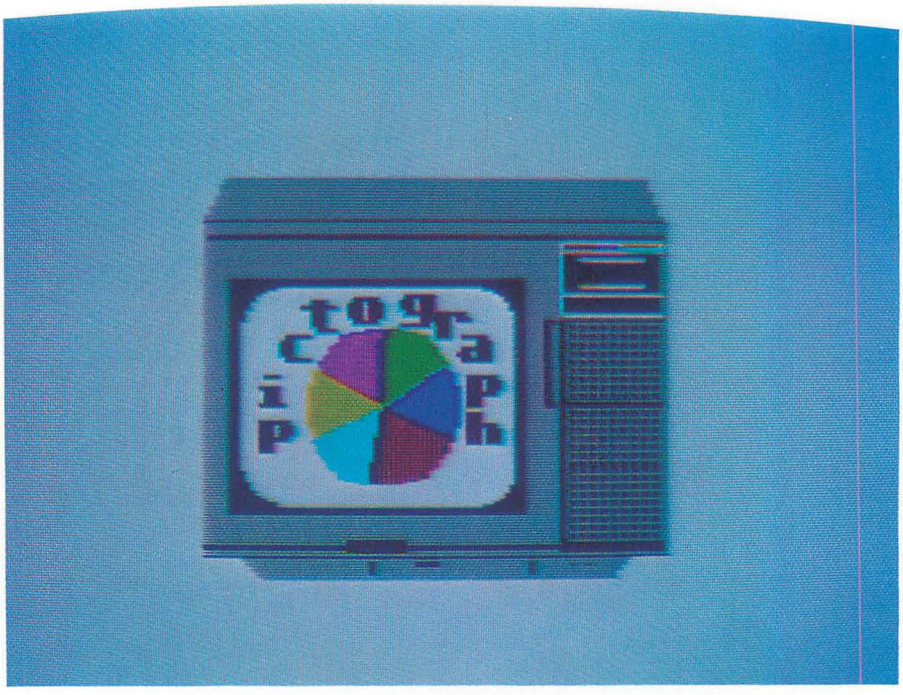


Run your business on an under-£250 computer? Possible with packages like SIP's Sales, Purchase and Nominal ledger accounts

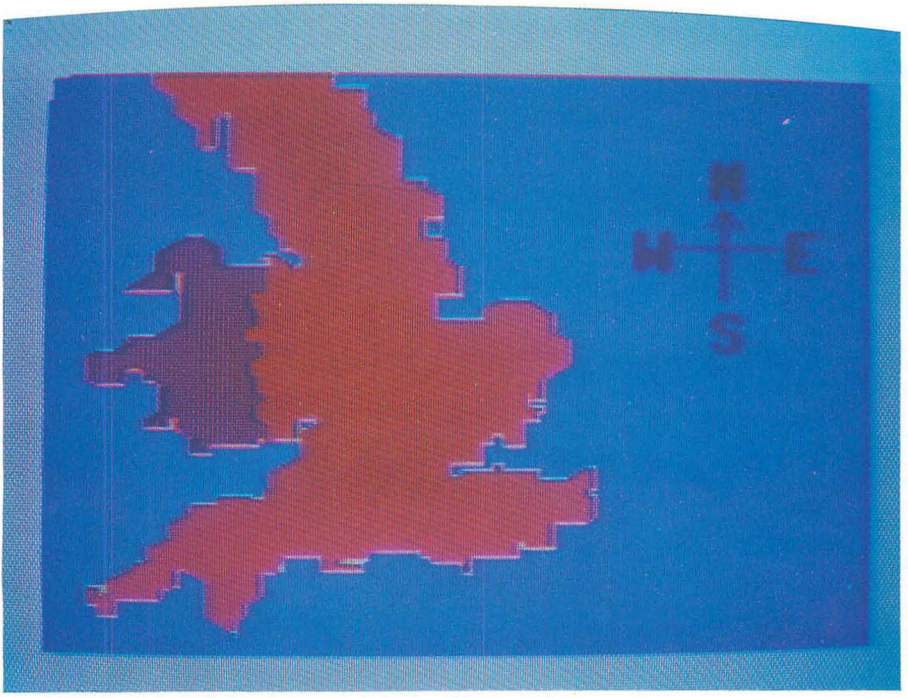
The classic use for a computer with a lot of memory, excellent graphics, good colour and very good sound capabilities is in complex and appealing games



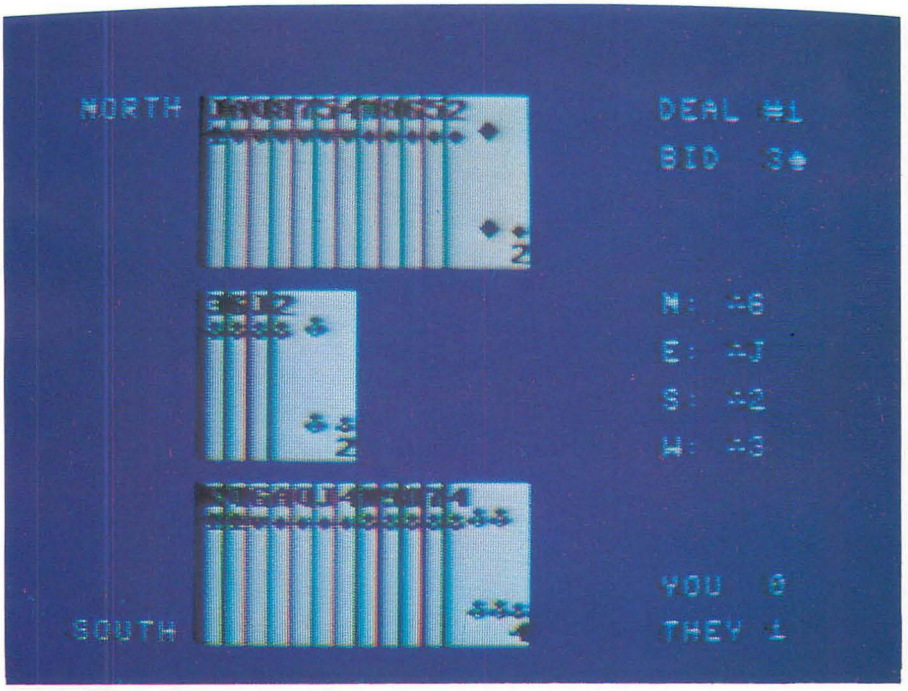
Anything an office computer can do . . . the 64 makes a reasonable word processor for home and business use



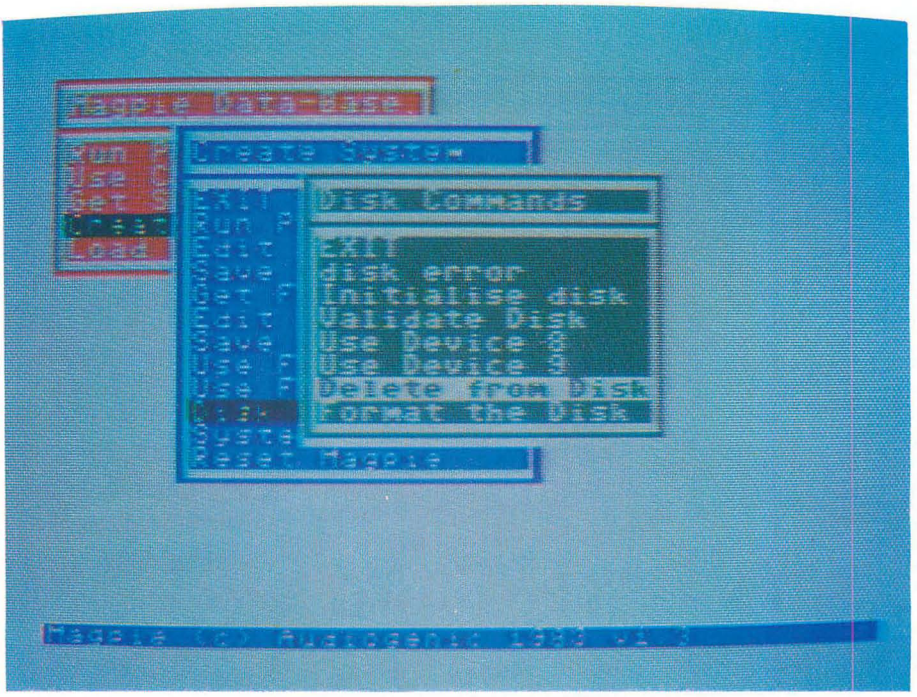
Strong colours and strong images on the 64's screen from the Pictograph joystick 'drawing' program



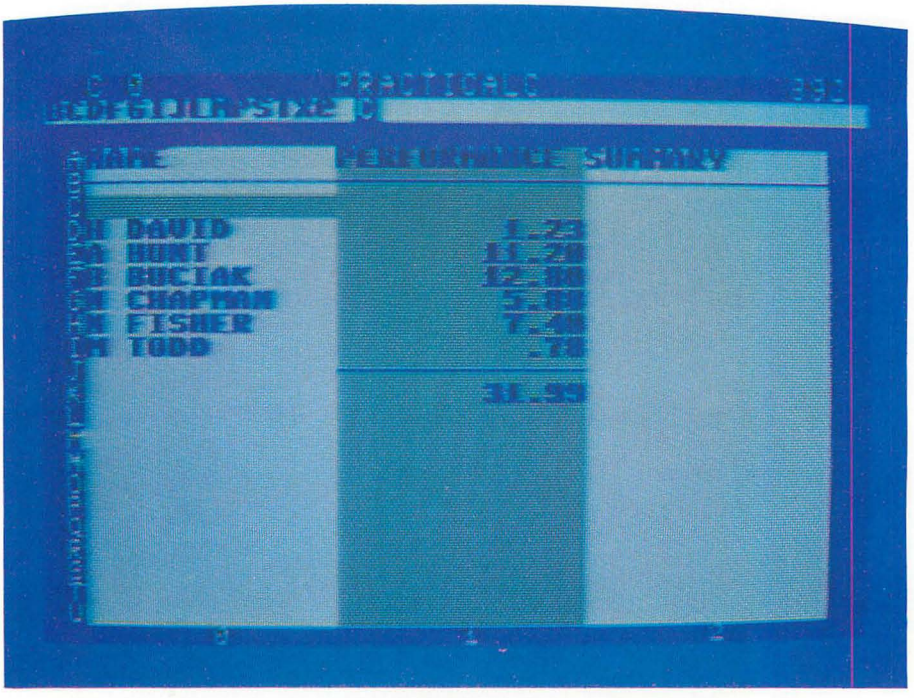
Superb colour in this not unreasonable map of Britain



A good version of bridge for the 64 – makes full use of the memory available and good use of the colour and graphics capabilities



The Commodore 64 is capable of running some quite sophisticated software. This is Audiogenic's Magpie database package, featuring 'pop-up' menus of choices at various stages of use



'Spreadsheet calculators' let you do 'what-if' forecasting for business finances. This is PractiCalc, one of the best available

SO YOU'VE JUST BOUGHT A COMMODORE 64

But how does it work? What can it do? How can you use it for games? How do you write programs? How do you produce graphics? What do you do if something goes wrong?

Getting the Most from Your Commodore 64 is a comprehensive and carefully designed introduction to your machine. The book is written and edited by professional journalists and writers and, through the use of diagrams, colour photographs, programs and entertaining examples, it takes you, in a clear and painless way, from the elements of computing through to mastery of your machine.

The Commodore 64 is a function-filled home computer with enough power for business use as well. And that is reflected in the wide range of proprietary software now available for it – from some of the best games ever seen on a home computer to genuine and useful word processing and accounting packages.

If you want to understand what your Commodore 64 is **really** capable of doing, then this is the book for you.

Cover design by Keith Pointing

U.K. £5.95
AUST. $9.95
(recommended)
N.Z. $13.95
CAN. $14.95

Computing
ISBN 0 14
00.7813 4