AMSTRAD COMPUTING

IAN SINCLAIR

WITH THE CPC464

Amstrad Computing

Other books for Amstrad users

Sensational Games for the Amstrad CPC464
Jim Gregory
0 00 383121 3

40 Educational Games for the Amstrad CPC464
Vince Apps
0 00 383119 1

Adventure Games for the Amstrad CPC464
A. J. Bradbury
0 00 383078 0

Introducing Amstrad CPC464 Machine Code
Ian Sinclair
0 00 383079 9

Practical Programs for the Amstrad CPC464
Owen Bishop and Audrey Bishop
0 00 383082 9

Filing Systems and Databases for the Amstrad CPC464
A. P. Stephenson and D. J. Stephenson
0 00 383102 7

Amstrad Computing

Ian Sinclair

COLLINS
8 Grafton Street, London W1

Collins Professional and Technical Books
William Collins Sons & Co. Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing Ltd 1984
(previous ISBN 0 246 12665 5)
Reprinted by Collins Professional and Technical Books 1984, 1985 (twice)

Distributed in the United States of America
by Sheridan House, Inc.

British Library Cataloguing in Publication Data
Sinclair, Ian R.
Amstrad computing.
1. Amstrad CPC464 (Computer)—Programming
I. Title
001.64'2 QA76.8.A4

ISBN 0 00 383120 5

Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface

The Amstrad CPC464 computer offers much more for either the home or the business user than any previous machine at a comparable price, more even than many machines at much higher prices. Because of this, many buyers of the machine will never have made use of a computer before, certainly not a computer with the range of facilities that the CPC464 offers. The manual which comes with the machine is one of the best that has ever been produced for a home computer, but it is an impossible task to cater for every need. In particular, it's one thing to learn how to operate a computer, another to learn how to program it for yourself, and yet another to learn how to design your own programs. This book is aimed to offer just that type of progressive help to the novice buyer.

There will invariably be applications for your Amstrad CPC464 which cannot be covered by programs which you buy. When your computing has moved to the stage where you need to design your own programs to solve your own problems, you will need to learn how to program the Amstrad CPC464 for yourself. Programming, after all, is one of the features for which a computer is intended. Owning a computer and not programming it for yourself is like buying a Mercedes and getting someone else to drive it. If you have never programmed a computer for yourself, this book will show you how. If you have some experience of earlier makes of computers, then this book will open a new world of programming to you. The Amstrad CPC464 does not use the same simple version of the BASIC programming language as earlier designs of machines – it needs to be learned almost from scratch if you have previously used one of these machines.

I would like to emphasise that this book was written while I was *using* an Amstrad CPC464 which had been delivered to my home, and that the listings of programs in this book were obtained from an Epson printer that was connected to the Amstrad CPC464. This

might seem to be an unnecessary claim, but many books still appear in which the program listing have been retyped, with errors appearing in many of them. Every program which appears in this book, and every example of programming commands, has been tested on the Amstrad CPC464 which I have here in front of me. Nothing has been copied untested from the manual, and where a command has operated in a way that is not obvious from the manual description, I have pointed out the difference. All of the screen displays which I have described were obtained on the Amstrad colour monitor. This has been done because I feel that many of the readers of this book are likely to be using such a monitor because of the remarkably low price of the computer/monitor package. Those readers who have purchased the computer with its power supply/TV adaptor have not been forgotten, however.

As always, I am greatly indebted to many people who made this book possible. The machine was provided by Amstrad Consumer Electronics plc, and Richard Miles of Collins Professional and Technical Books worked tirelessly to ensure that I had the Amstrad CPC464 on my desk as soon as possible. Among many others at Collins Professional and Technical Books, he and Janet Murphy worked wonders with my manuscript, and the most efficient team of typesetters and printers that I know operated to produce the book in record-breaking time. I am sure that the result of all this work will be a book that will match the capabilities of your Amstrad CPC464. In my opinion, this is one of the best home computers that I have used to date, and I would like to congratulate Amstrad on a remarkable entry into the computer marketplace. It was a new experience for me to come across a new computer on which everything that was claimed for it worked, and worked so well.

Ian Sinclair

Chapter One

Setting up the CPC464

When I opened my CPC464 computer I found that several of the keytops were loose. If you find that this has happened to yours, don't panic, because most of the keytops simply plug back into place. The exception is the long spacebar. The holes in the brackets attached to the spacebar have to be hooked into the two metal prongs in the slot in which it fits. You will need the assistance of a pair of tweezers to do this. Once done, and with the spring located, the spacebar presses into place, and your CPC464 is ready for action.

The CPC464 package contains the CPC464 itself, a demonstration tape, and a large manual. In addition to this package you will need one of three additional packages. One of these is the green-screen monitor, another is the colour monitor, and the third is the power supply/TV adaptor. If you have bought either of the monitors, then you are ready to start work with the CPC464. If you have bought the power supply/TV adaptor package, however, you will need to have a TV receiver handy as well. The manual very clearly shows you how the monitors or the power supply are to be connected together. If you are using the power supply, you will also have to make a connection to the aerial socket of your TV receiver. Advice on how to tune your TV to the CPC464 signals follows later in this chapter.

If you are using the monitors, be careful with the six-pin plug. This must not be forced into its socket, but unless the monitor is at the same level as the computer, and very close, you may find that the plug does not make good contact. I used the monitor on a shelf above the computer, and I had to stretch the cables to prevent the six-pin plug from losing contact. If you find when you switch on the machine that there is no display on the monitor, try wiggling this plug. You won't do any damage if this plug loses contact even when you are programming the machine. If the smaller plug for the power supply comes loose, however, any program that you were using will be lost unless you have a recording of it on tape. For that reason, it's very

important to be able to use the cassette tape recorder which is built into the machine, and there is advice on that point in this chapter also. First of all, however, no matter which version of the machine you have bought, you will need to connect a mains plug.

The plug is connected as indicated in Fig. 1.1. There are only two leads, one blue and the other brown, and the cable should be tightly clamped. The fuse should be a 3 amp type, not the 13 amp variety

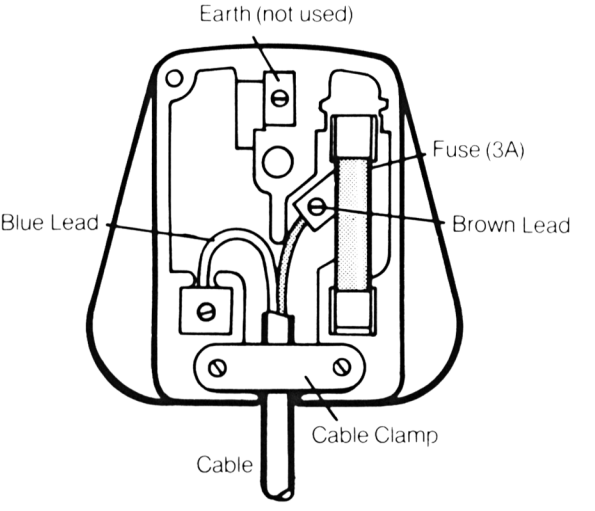

Fig. 1.1. The connections to a three-pin mains plug. Only the live and neutral leads are used. If you haven't done this sort of thing before, play safe and hand it to an electrician.

which usually comes with the three-pin plug. If you are accustomed to fitting plugs for yourself then the diagram should be enough to remind you of what is needed. If you don't want to have anything to do with mains supplies, then get an electrician to fit a plug for you, with a 3 amp fuse. You don't have to give the computer to the electrician – only the power supply box or the monitor, whichever you are using. When you have done this, make sure that the power supply box is located away from the CPC464 and clear of the TV. Put it where the air can circulate round it. It gets only slightly warm while you are using the computer, but you should always try to keep it in a cool place. If you are using a monitor, place it where you can see the whole of the screen and at a distance that doesn't put too much strain on that curled cable.

Once over that hurdle, you are almost ready to put the CPC464 to work for you, but if you are using the power supply, you also need the use of a TV receiver. A computer is a device which is arranged so as to

send out electrical signals that can be used to form images on a TV screen. There are two ways in which this can be done. One is to use a special form of TV display, which is designed to use the signals directly from the computer. Such a device is called a *monitor*, and it can't normally be used for receiving TV pictures from an aerial. The alternative method is to convert the signals from the computer into the same form as the signals sent by TV transmitters, so that they can be connected to the aerial socket of an ordinary TV receiver.

There is an important difference between the two. Although it's convenient to be able to use a TV receiver (and cheap as well), the picture is of a poor standard. This is because a TV receiver was never designed to accept computer signals, and because of the need to convert the computer's signals into the same form as transmitted signals. The result is that the letters and other shapes on the screen look fuzzy, and the colours looks streaky. If you use the Amstrad monitor, which is very reasonably priced compared with others, the shapes on the screen will be clearer, and the colours much better. The only problem that I encountered with the Amstrad colour monitor I used was that the brightness control did not allow enough brightness for use during the day. If you are troubled with this, or possibly excessive brightness, return the monitor for internal adjustment. Do not, under any circumstances, attempt to remove the case of the monitor to make your own adjustments unless you have extensive TV servicing experience.

Seeing is believing

Unless you connect a TV receiver or monitor to the CPC464 you won't be able to see what the computer is doing. It will still compute for you just as well, but you won't see what is going on. To connect the CPC464 to a TV receiver, you will need to plug your aerial lead to the TV from the power pack. Unless you can keep a TV receiver specially for use with the CPC464, you will find that you have to keep plugging and unplugging the aerial cable and the CPC464 cable. This is never a good thing to have to do, because it loosens the contacts of the socket on the TV. A useful alternative is to use the type of adaptor that is illustrated in Fig. 1.2. This allows you to plug a lead into the aerial socket of the TV so that the TV can be used both for CPC464 and for *Dynasty* without having to pull plugs out. The two-way TV adaptor that I used is sold in TV stores under the name of 'Panda'.

You need to connect the CPC464 to the TV or to the adaptor using

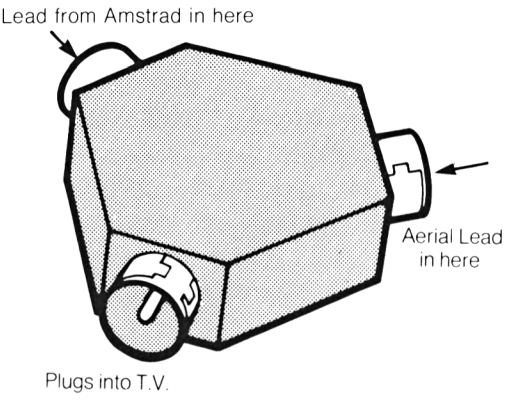

Fig. 1.2. A TV aerial cable adaptor, which allows you to keep both the computer and the aerial connected.

the special cable that is provided with the CPC464 power supply. If this lead is too short for you, you can buy extension pieces of cable, and cable joiners. If you have used the adaptor, then all that you have to do to change between computing and TV watching is to change channels! If you are using a monitor, of course, all you have to do is ensure that the plugs are correctly inserted.

The TV or monitor that you use to display the CPC464's signals need not be a colour type, not to start with at least. The skills of programming a CPC464 do not require you to see the results in colour until you come to the colour instructions of the CPC464 in Chapter 8. When you use a black and white TV or monitor to show the CPC464 signals, the colours appear as shades of grey, and they are quite distinct. If you use a colour receiver, you will see the colours appear in all their glory, though not all makes of TV receivers will give equally good displays.

The big switch-on

Now before you plug in everything in sight and switch on, it's a good idea to see how many mains sockets you have nearby, and where you are going to house everything. When you use the CPC464 with its monitor you will need only one mains socket. If you use the power supply and a TV receiver, you will need two sockets, one for the CPC464 and one for the TV receiver. Most houses have desperately few sockets fitted, so you will find it worthwhile to buy or make up an extension lead that consists of a three- or four-way socket strip with a

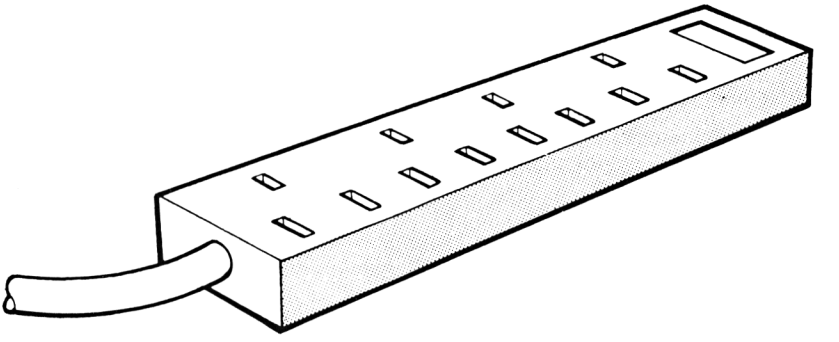

Fig. 1.3. A four-way socket strip which avoids the use of the old-style adaptors.

cable and a plug (Fig. 1.3). Even if you need only one socket at the moment, the odds are that it will not be where you want it. In addition, the CPC464 is such an inviting computer that you will probably want to add a printer, and then disk drives to it later. All of these extras will need power sockets. Using a socket strip avoids a lot of clutter – you don't want to bring your CPC464 crashing to the floor when you trip over a cable. Don't rely on the old fashioned type of three-way adaptor – they never produce really reliable contacts. The CPC464 has an on/off switch, but you should *always* take out the mains plug after you have finished a computing session. The switch on the monitor will also remove power from the computer, because the monitor contains the power supply for the computer.

When you have the essential equipment to start computing, consisting of the CPC464 keyboard, power supply and TV (or monitor), quite a lot of flat surface is needed. Later on, you will probably want to add a printer, possibly disk drives and other extras which make the difference between having a computer *system* and just having a computer. All of this needs space, and the best way that I have found of organising this is one of the computer stands made by Selmor (Fig. 1.4). If you aren't at that stage yet, then a good-sized desk or table will have to suffice for the time being. Computing is like hi-fi – there's always something else that you can buy!

With everything housed and connected up, you now have to get used to some dos and don'ts. The CPC464, like practically all home computers, makes use of a cassette recorder for storing information. This is because the information that the computer stores in its own memory is lost whenever the computer is switched off. The cassette recorder can record signals on tape, and these remain recorded after

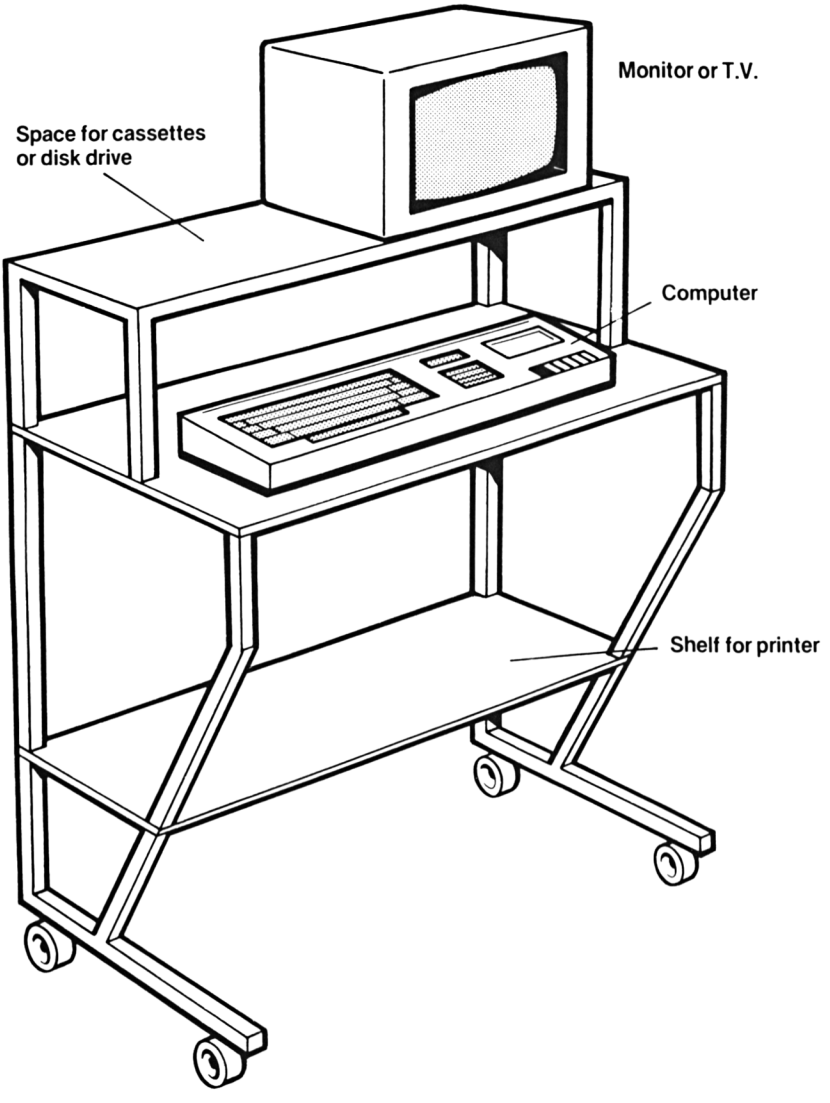

Fig. 1.4. Using a Selmor stand to house all the bits and pieces of a CPC464 computer system.

the machine has been switched off. When these signals are replayed into the computer, the memory is filled with the information again, and the computer can make use of it. Unlike most home computers, however, the CPC464 has its own built-in cassette recorder. This has several advantages. For one thing, you have no fussy adjustments to make, as you often have when using a separate cassette recorder, and there are no extra connections to make either. The other point is that

this built-in cassette recorder is one which is designed for computer signals, unlike ordinary domestic recorders, and because of that it's a much more reliable way of storing information on tape. Even tape is not perfect, though, and it's wise always to make two copies of any computer program that is specially valuable to you – the second copy is called a *back-up*.

The next step is to switch on the TV receiver and the CPC464, or monitor and CPC464. Unless you are exceptionally lucky, or using the monitor, you will probably see nothing appear on the screen. This is because a TV receiver has to be tuned to the signal from the CPC464. Unless you have been using a video cassette recorder, and the TV has a tuning button that is marked 'VCR', it's unlikely that you will be able to get the CPC464 tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the CPC464's signals. If you are using the monitor, you can ignore this section.

Figure 1.5 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system that is illustrated in Fig. 1.5(a). This is the type of tuning system that you find on black and white portables, and to get the CPC464's signal on the screen, you only have to turn the dial. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial isn't marked, which is unusual, then start with the dial turned fully anticlockwise as far as it will go, and slowly turn it clockwise until you see the CPC464 signal appear.

What you are looking for, if the CPC464 hasn't been touched since you switched it on, is the screen display that is illustrated in the manual on page Fl.6. The second line of this display is a copyright notice – on my CPC464 it carried the date 1984. When you can see the words of the copyright notice, turn the dial carefully, turning slightly in each direction until you find a setting in which the words are really clear. On a colour TV receiver the words may never be particularly clear, but get them steady at least, and as clear as possible.

The older types of colour and B/W TV receivers used mechanical push-buttons (Fig. 1.5(b)) which engage with a loud clonk when you push them. There are usually four of these buttons, and you'll need to use a spare one, which for most of us means the fourth one. Push this one fully in. Tuning is now carried out by rotating this button. Try rotating anticlockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the CPC464's signal during this time, you'll see the message

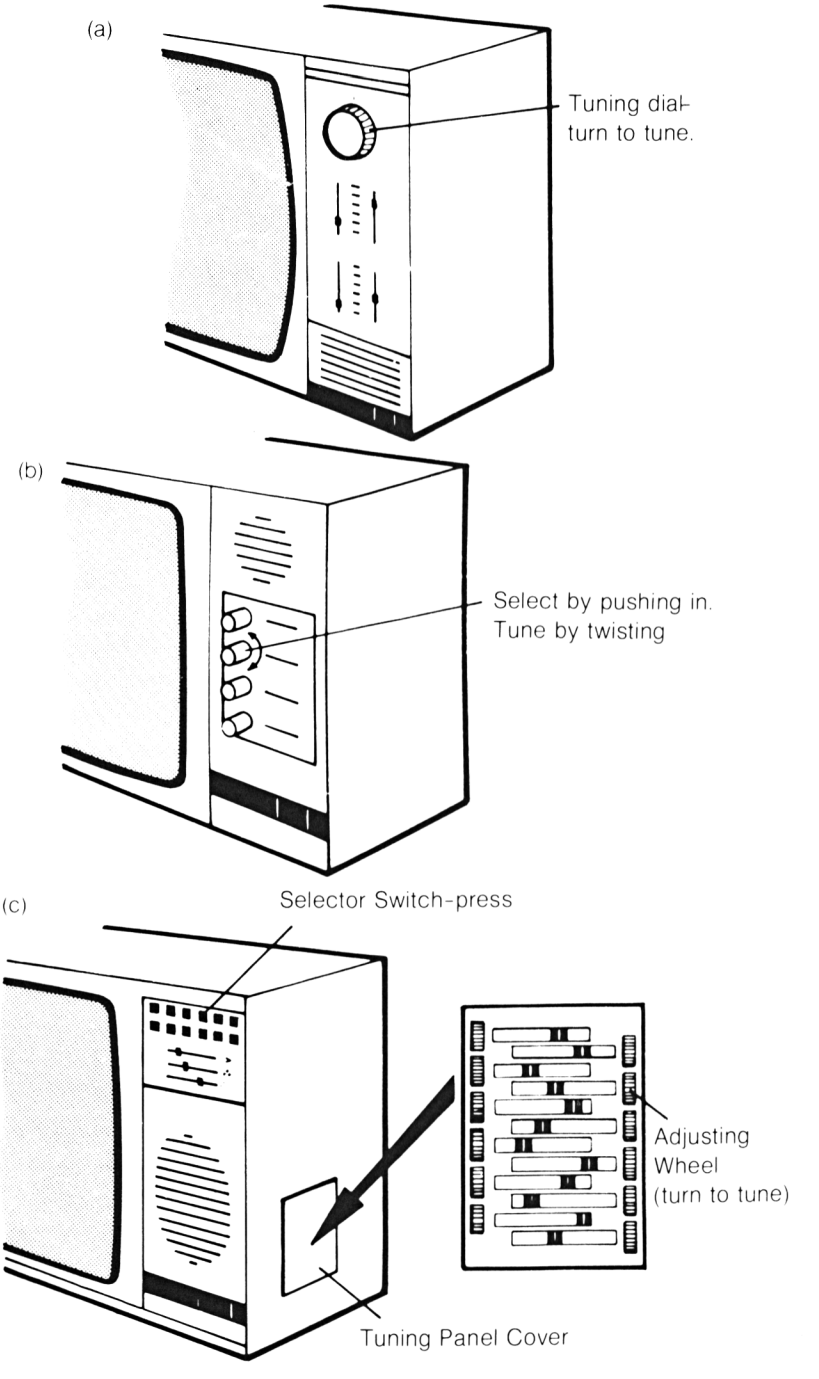

Fig. 1.5. TV tuning controls. (a) Single dial, as used on black and white portables, (b) four-button type, (c) the more modern touch-pad or miniature switch type.

on the screen. If you've turned the button all the way anticlockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the CPC464 signal at any setting, check that you have connected the cables to the TV aerial socket correctly. If you are using the Panda adaptor, you can push one of the other tuning buttons to check that you can receive normal TV signals. If you can, there's nothing wrong with the TV, so switch back and try again to find the CPC464 signal.

Modern TV receivers are equipped with touch-pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the side or at the front of the receiver (Fig. 1.5(c)). The buttons or touch-pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number available (usually 6 or 12), press the pad or button for this number, and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen and silence from the loudspeaker. On this type of receiver, the picture is usually 'fine-tuned' automatically when you put the cover back on the tuning panel, so don't leave it off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so that you have to keep re-tuning. The CPC464 should give a good picture on practically any TV receiver. If your TV exhibits faults like a shaking picture, or very blurred colours, then check the tuning carefully. If the faults persist, and the TV is correctly tuned, you will have to contact the service agents for the TV – or use a different model in future!

If you are using the monitor, then you should get a good picture with practically no need for adjustment. The picture brightness will probably need to be adjusted for the lighting conditions inside the room, but little else needs to be done. The colour monitor has only a vertical hold control accessible (a horizontal hold control can be screwdriver-operated), but it's most unusual to find that these need to be tweaked. Only if the picture on the screen is very unsteady should you attempt these adjustments.

And now to work

Once you have achieved a screen display from your CPC464, the

business of mastering the use of the CPC464 begins. When the copyright notice is being displayed, you'll see a bright (gold) square immediately under the word 'Ready'. This square is called the *cursor*. It is used as a marker, and when you press a key, a letter (or number, or whatever is marked on the key) will appear at the position of the cursor. The cursor will then move across the screen to the next position. It's important to note at this point that nothing that you can do just by pressing keys on the keyboard can possibly damage the CPC464 machine – the worst you can do is to lose a program that was stored in the memory. You can, however, damage the CPC464 by spilling coffee all over it, dropping it, or connecting it up to other circuits while the power is switched on. You can also scramble some of the signals on your tapes if you attempt to take them out while they are still running, or if you switch the machine on or off with a tape in place. *Always* switch off the computer, and everything that is connected to it when you insert or remove any of the plugs at the back. *Never* switch the computer on or off with a cassette in place, or remove a cassette while the cassette drive is being used. Keep your tapes in a cool place, well away from the monitor or power supply.

Key-tapping time

It's time now to look at the keyboard, because the keyboard is the way that you pass instructions to the CPC464. If we ignore the groups of keys at the right-hand side, then most of the CPC464 keys look like typewriter keys. The arrangement of letters and numbers is the same as that of a typewriter, and if you've ever used a typewriter, particularly an electric typewriter, then you should be able to find your way round the keyboard of the CPC464 pretty quickly. When you press any of the letter keys, you will see the letter appear on the screen. What you see is a *lower-case* (small) letter, not an *upper-case* (capital) letter. Just like a typewriter, the keyboard of the CPC464 normally gives you lower-case letters. If you want a capital, you need to press one of the two SHIFT keys as well as a letter key.

Commands that you give to the computer in typed form can be in either lower-case (small letters) or upper-case (capitals). If you want capitals all the time, press the key that is marked CAPS LOCK. To return to lower-case letters you only have to press the CAPS LOCK key again. Unfortunately, there is no indicator light to show you whether CAPS LOCK is on or off – you just have to try it and see! Whatever the setting of the CAPS LOCK key happens to be, the keys

which show two symbols on them, like most of the number keys, will need the use of SHIFT to get the symbol on top. When you press one of these keys alone, you get the symbol that is marked on the lower part of the key. Using SHIFT along with the key gives the symbol on the upper part of the key. For example, if you press the 8 key by itself, you get 8. If you press the 8 key and SHIFT together, you get (.

As well as the ordinary typewriter keys, there are a number of special keys which are not found on any typewriter. At the top left-hand side of the keyboard, for example, you will find the key that is labelled ESC (escape) and on the right-hand side of the long spacebar, you will find the key which is marked CTRL (control). These ESC and CTRL keys are used in ways that are outlined in the manual. The ESC key is particularly useful because it allows you to stop anything that is happening. Pressing any other key will then allow things to continue, and pressing ESC again will make the stop permanent. When a program is stopped in this way, you can always restart it, because the program is not wiped from the memory. Another way to stop a program from running is to press the CTRL, SHIFT *and* ESC keys together, but this always has the effect of wiping out any program from the memory. We'll make use of this later, because it's sometimes the easiest way of returning the computer to its correct state. You must make sure before you take this step, however, that you have recorded any program that you have been working with. The CTRL key can be used to display some shapes that are quite different from the letters that are marked on the keys – but more of that later. One action that you should know about at the moment, however, is the letter delete. Type a word, and then press the DEL key, which is on the right of the top row of keys. You'll see that one letter disappears off the end of the word that you have typed. If you hold the DEL key down, you'll see the other letters being deleted also, one by one, rapidly, as the cursor moves left. When there are no more letters to delete, you'll hear the built-in loudspeaker of the CPC464 sound a warning. Turn up the volume control (next to the on/off switch at the right-hand side) if you can't hear the warning. This DEL action is one of the useful ways that you can rub out mistakes in your typing.

The set of four keys that are marked with arrows are cursor keys. Try pressing them, and you will see that they make the cursor move in the direction of the arrow on the key. These keys are used, together with the COPY key in the centre of the group, for one type of editing. This is explained in detail in Appendix A when you are ready for it. Finally, the set of twelve keys at the lower right-hand side of the

keyboard comprises what are called the number pad keys. For those of you who are right-handed, they are supposed to make it easier to enter numbers. I'm left-handed, and I never use them!

The most important of all of the special keys, however, as far as we are concerned at the moment, is the large key that is marked ENTER. This is in the position of the carriage return key of an electric typewriter, but its action is not the same in all respects. Pressing the ENTER key is a signal to the computer that you have completed typing an instruction and that you now want the computer to obey it. If you are accustomed to using an electric typewriter you will have to change some of your habits as far as this key is concerned. When using a typewriter you would press the carriage return key each time you wanted to select a new line, with typing starting at the left-hand side of the new line. The ENTER key of the computer does rather more than this. If the material that you are typing into the CPC464 takes more than one line on the screen, the machine will *automatically* select the next screen line for you. The ENTER key must *not* be used for this purpose. The ENTER key is used only when you want the machine to carry out a command or store an instruction, not simply when you want to use a new line. When you press ENTER, however, it will always provide a new line for you, and select a position at the left-hand side. The position where a letter or other character will appear when you press a key is indicated by the cursor.

You will find that the action of each key repeats if you hold your finger on it, and this repeat action is quite fast. If you type a set of meaningless letters and then press ENTER, the computer usually responds with the phrase:

Syntax error

which is followed underneath by the word 'Ready' and the cursor. The syntax error message occurs because the computer is a simple machine. It can understand only a few words – the words that we call its *reserved words* or *instruction words*. If what you type does not include these words, or uses these words incorrectly, then this is a syntax error as far as the computer is concerned. It may make sense to you, but it doesn't make sense to the computer! The word 'syntax' means the way that words are used in any language, and what your computer uses for instructions is a sort of language.

Cassette tryout

The CPC464 computer, like all others, stores instructions and other information in its memory – but only while the machine is switched on. Whenever you switch off your CPC464, anything that was stored in its memory is instantly lost, and you can't get it back even if you switch on again very quickly. For that reason, we need to store the programs and the information that we need to use in programs in another form. The most useful forms are magnetic disks and magnetic tape. The CPC464 has at the time of writing no disk system available, though it will probably be on sale by the time this book appears in print. A disk system has the advantage of being very much faster as well as more reliable than cassettes.

The computer has circuits which will convert the instructions of a program into signals, which can then be recorded on the tape of the cassette recorder. When these signals are replayed, another set of circuits will convert the signals back into the form of a program. In this way, the use of the CPC464 cassette system allows you to record your programs on tape and then to replay them. Before you tackle the rest of this book it's important to check now that you can use the cassette recorder to record and replay programs.

The easiest thing to try first is to use the cassette to put a program into memory. This is called *loading*, and since your CPC464 comes with a set of programs on a cassette, these are a convenient way of gaining experience. You will see from the notes that come with the programs that you cannot easily make a copy of the programs because they have been 'copy-protected' to prevent you doing so. Normally, you would not want to buy programs that were copy-protected in this way, because you cannot make back-up copies. It is essential to make a back-up copy of any program which you value. This does not mean that the cassette system is particularly unreliable; it is simply a normal safeguard. All recording systems are subject to loss of signals, and people who use computers always keep several copies of anything that is valuable. It's really rather foolish to pay a lot of money for a program that you can't copy. If a program-seller announces proudly that his program can't be copied, then *don't buy it!* In any case, if you have any experience with using cassettes, you can make a back-up copy of any tape by using a twin-deck recorder, or two cassette recorders linked by a suitable cable.

Load in the 'Welcome' tape that comes with your computer, following the very clear instructions in the manual. When you hear the tape stop, press the STOP key on the recorder. This is important.

Although the computer stops the motor from moving the tape, the cylinder which moves the tape is still held tightly against the tape by a rubber pulley, called a pinch-wheel. If this pinch-wheel is left clamped against the cylinder (or capstan) for a long time, it will develop a flat on one side, and this can cause trouble with recording and replaying. When this happens, repair is not so simple as it would be with a separate cassette recorder. You have to take the whole computer to a repair shop because the cassette recorder is built-in and can't easily be separated. When you press the STOP key of the recorder the program will start running. Only certain tapes that have been recorded by the manufacturers and by professional software suppliers will need to be loaded in this way, however. The important thing you need to learn now is how to record your own programs.

Recording your own

Before you can make a recording to test the system, you need a program to record, and this involves some typing. This is easy if you have just switched on the CPC464, but if you have been pressing keys at random then it's a good idea to switch off again, then on, with any cassette removed.

Type the number 10 (1 and then 0), and then the word rem. It doesn't matter whether you type rem or REM. This is a command word for the computer, and no matter whether you type it in upper-or lower-case, the computer will (later) convert it into upper-case. Check that this looks correct, and then press the ENTER key. The effect of this is to place the instruction line '10 REM' into the memory of the CPC464. As you type the first digit, the character will be seen on the screen at the cursor position. When you press the ENTER key, the cursor moves to the next line down. At the same time, your command stays where it was typed on the screen. If you used lower-case, then you will still see it in lower-case, like:

```
10 rem
```

If you see a mistake as you type the line, just use the back space action, by pressing the DEL key. This shifts the cursor backwards, and deletes the letter that the cursor is now over. You can then type the correct letter, which will replace the incorrect one. If this makes the line correct, pressing ENTER will enter it into the computer. If you have typed something that is incorrect, like REN or RWM then you will not be warned in any way – the computer accepts anything

that you type following a number. To correct a line such as:

```
20 Ren
```

you can, instead of using DEL, just type the correct version:

```
20 rem
```

and press the ENTER key. Now type the rest of the lines, as illustrated in Fig. 1.6, remembering to press the ENTER key after you have completed typing each line. The numbers are called *line*

```
10 REM
20 REM
30 REM
40 REM
```

Fig. 1.6. A program for testing the cassette recording and replaying actions.

numbers, and they are present for two reasons. One is to remind the computer that this is a program; the other is to guide it, because the computer will normally carry out instructions in the same order as the line numbers. You can check that your program looks correct by asking the computer to 'list' it. *Listing* means that the computer prints on the screen whatever you have stored in its memory. Using the line numbers ensures that the instructions are stored, and if you type 'list', and then press ENTER, you will see your program. Don't be surprised to find that all the lower-case letters (like rem) have been converted to upper-case (like REM), because this is part of the action of the computer, along with putting line numbers in order, and leaving a space between the last digit of the number and the first letter of the command. Check from this 'listing' that the program is like the printed version in Fig. 1.6.

Now make sure that you have a cassette ready. Choose a new cassette, preferably a C12 or C15 computer cassette. Suitable cassettes can be bought wherever you bought your computer, or in branches of main high street stores. Place the cassette in the recorder (the manual shows this very clearly), and press the REW key of the recorder to make sure that the cassette is fully rewound. When you are sure it is, press the little button on the tape counter so that it reads 000. Now press the FF (fast forward) key of the recorder, and stop the tape when the counter shows the figure 002. This makes sure that you have a piece of fresh tape to record onto, *and this is very important.* At the beginning of each cassette there is always a piece of plastic tape with no magnetic coating on it. This is called the *leader*, and if you

attempt to record a program on this piece of the cassette, your attempts will quite certainly end in failure! In addition, the first piece of genuine recording tape is often slightly creased or marked. By moving the tape beyond this portion, you are much more likely to obtain good-quality recordings.

Now to make the recording. First of all you have to type:

SAVE "TEST"

and then press the ENTER key. The word SAVE is the instruction to the computer meaning that you want to save (record) a program onto a cassette. The TEST part is a *filename* which the computer will use to recognise the program if it is asked. It is possible to save a program without a filename, as long as you have the 'save' and the first quotemark (") present, but you should not make a habit of this. Normally, you will keep a number of different programs on each cassette, and you need the filenames to instruct the computer to load back the correct one. When you press the ENTER key, you will see the message:

Press REC and PLAY then any key:

and you have to press down the REC and PLAY keys of the recorder firmly until they lock. Now the 'press any key' message is almost correct, but not quite. The SHIFT, CAPS LOCK and CTRL keys will have no effect, and the ESC key will stop the whole process. You can, however, press the spacebar, or ENTER, or any of the letter or number keys. When you do this, the motor of the cassette will run, and you will soon see a message appearing on the screen. In this case, the message will be:

Saving TEST block 1

and if you have the loudspeaker volume control turned up, you may hear the faint noises which are the signals that represent your 'program'. When the program has been recorded, the motor of the recorder stops, and the word 'Ready', followed by the cursor reappears on the screen. The program is now recorded, and the size of the program is indicated by the number that follows the word 'block'. Short programs, such as are used in this book, will mainly need only one block to record. The motor of the cassette recorder will stop automatically at the end of the process, but you still have to press the STOP key of the recorder to release the tape. That's all that's involved in making the recording.

Now comes the crunch. You have to be sure that the recording

worked. Type NEW and press ENTER. This should have wiped your program from the memory. Just to make it look better, type CLS and press ENTER. This clears the screen, so that there will be no confusion. Now type LIST and press ENTER. Nothing should appear – LIST means put a list of the program instructions on the screen, and there shouldn't be any! An alternative to all of this is just to press CTRL,SHIFT and ESC to clear the memory.

You can now load the instructions in from the tape. Type LOAD "test" and then press ENTER. You will once again get a message, this time to press the PLAY key (you *must not* press the REC key) and then 'any key'. When you do this, the cassette motor will run again, and you will soon see a message on the screen. This time, the message is:

Loading TEST block 1

and if the volume control is turned up you will hear noises again. Once again, when the program is loaded, the motor of the cassette recorder will stop, and you should then press the STOP key of the recorder. Type LIST now, then press the ENTER key. You should see your program appear on the screen. Once you can reliably save programs on tape, and re-load them, you can confidently start computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few minutes more work will save your effort on tape so that you won't have to type it again. The CPC464 cassette system is a very reliable one, though programs take rather longer to save or load than with some other systems. The only thing that you have to be careful about is care of your cassettes. The table in Fig. 1.7 should act as a useful reminder to you in this respect.

1. Keep cassettes dry, and in a cool place, free from condensation.
2. Store and use well away from magnets. This includes loudspeakers, TV receivers and electric motors.
3. Never touch the tape with your hands.
4. If the tape jams, take the cassette out, hit it against your hand, and then try to fast-wind it fully in each direction.
5. Never use the first part of the tape in any cassette.
6. Never keep a tape stored for too long without playing it or at least fast-winding it.

Fig. 1.7. A checklist to ensure long life from your cassettes.

Later on, when you are dealing with much longer programs, you will probably find the time taken using a cassette rather irritating. The ultimate answer is to move to a disk system, but you *can* get faster cassette action! If you type:

SPEED WRITE 1 (then press ENTER)

this switches the cassette system to record at double-speed. The tape moves at the same rate, but the recording is carried out more quickly. You don't have to use this command when you are loading – the computer will select the correct speed automatically. It's not quite so reliable as the slower speed, but if you take good care of your cassettes, the difference in reliability is not noticeable. The difference in speed, however, is very noticeable when you are using long programs. It's a good idea to keep a back-up copy of a long program recorded at slow speed and use an 'everyday' copy which has been recorded at fast speed. To get back to normal (slow) speed, just type:

SPEED WRITE 0 (then press ENTER)

or reset the machine with CTRL, SHIFT and ESC if you don't mind losing the program from the memory.

Load and run

Later on, when you start to write your own programs, you will want to know how to make programs load and run automatically like the Welcome tape demonstration. The method is quite simple. You type the SAVE command, followed by the program name in the usual way. You then follow this with a letter P, meaning 'protected'. For example, you might have a command such as:

SAVE "longprog",P

being used to save your program. The SAVE procedure works as normal, but what is placed on the tape is not normal. To load back this program, you must either:

(1) Press CTRL and the *small* ENTER key, then press the PLAY key of the recorder, followed by 'any key', or
(2) Type RUN", then press ENTER (the main ENTER key), then operate the cassette recorder as before.

Either way, the program will load and run automatically. You will *not*, however, be able to list it or RUN it for a second time when it has

finished. Never save a program with protection unless you have another copy, because a program which is protected in this way will be lost if anything goes wrong, and you can't make a copy from it. Protection methods help slightly to reduce pirating of programs, but they punish the honest user who doesn't know how to make back-ups of such programs. You can, however, expect to find hints in the magazines about how to overcome these protection methods, when your programming is more advanced.

When you're ready

The first time you read this book, you may as well skip what follows, because it will become useful to you only when you have some more experience of computing. There are two more ways of loading a program apart from using LOAD and RUN. You can use CHAIN, followed by the filename. This will load the program and run it at once, so that you don't need to use LOAD and then RUN. Another type of loading is offered by MERGE. Normally when you load a program, it wipes any existing program from the memory. Using MERGE will *add* one program to another, so that you can make a lot of short programs into one long one. The programs must use different sets of line numbers. You can use CHAIN and MERGE together (such as CHAIN MERGE "BITS) to add one program to another and run the resulting longer program. Another facility is to read a catalogue of all the programs on a tape, using CAT – this is more useful for short tapes that have been recorded at the higher speed.

Chapter Two

Putting It All On The Screen

Chapter 1 will have broken you in to the idea that the CPC464, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the ENTER key is pressed. You will by now have used the command LIST which prints your program instructions on to the screen. There are two other useful points that you need to know before we go much further. One is that you can clear the screen by typing CLS (or cls), and then pressing the ENTER key. As your familiarity with the computer keyboard increases, you will want to make use of the editing commands, and these are explained in Appendix A.

Now there are two ways in which you can use a computer. One is called *direct mode*. Direct mode means that you type a command, press ENTER, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In program mode the computer is issued with a set of instructions, with a guide to the order in which they are to be carried out. A set of instructions like this is called a *program*. The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated only if you type the whole command again, and then press ENTER. The set of command words that can be used, along with the rules for using them, make up what is called a *programming language*. The CPC464 provides you with a new and very modern version of the most commonly-used of all programming languages for small computers; BASIC. BASIC is short for 'Beginners All-purpose Symbolic Instruction Code', and it was originally devised for teaching purposes. Since then, it has developed into a useful language in its own right. The version of BASIC that your CPC464 uses is, however, enriched by a lot of extra commands.

An important point about all computer commands, whether they are direct commands or program instructions, is that they have to be in a precise form. The spelling of a command word must be perfect, for example, or it won't be obeyed. It must also be *used* in the right way. For example, you can't just type SAVE, then press ENTER, and expect the computer to do anything. The computer expects a quote mark to follow the word SAVE, and it can't act on the command if the command is incomplete. Some commands include spaces between words, and will not work if a space if missed out. In other places, you find that a space is not important. You have to learn these things by experience, because there is no reliable guide as to when you must put in a space and when you can leave one out.

The CPC464, as you know by now, allows you to type command words in either lower- or upper-case, but converts to upper-case when the program is listed on the screen. This can cause confusion when you are trying to follow a printed listing, however, if instruction words are typed in lower-case, so from now on all instruction words will be printed, in listings and in the text, in upper-case. You know that if you type them in lower-case, it doesn't matter, but it makes it clearer which are the instruction words in the book. You now also know that a command has to be followed by pressing the ENTER key, so that I don't have to keep reminding you by printing things like LIST (press ENTER) each time.

Let's now take a look at the difference between a direct command and a program instruction. If you want the computer to carry out the direct command to add two numbers, 1.6 and 3.2, then you have to type:

PRINT 1.6 + 3.2 (and then press ENTER)

You have to start with PRINT (or print), because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of knowing that what you want is to see the answer on the screen. It doesn't recognise instructions like GIVE ME or WHAT IS – only a few words that we call its *reserved words* or *instruction words*. PRINT is one of these words. Remember that there *must* be a space following the T of PRINT. You do not need to have spaces between the 6 and the + or the + and the 3.

When you press the ENTER key after typing PRINT 1.6 + 3.2, the screen shows the answer, 4.8. This answer is not shown in the same place as your typed command, however. It is on the next line, and starting one space in from the left-hand side. If you have just cleared

the screen before typing, the answer will appear near the top left-hand corner. When you are working with a colour monitor or TV, the CPC464 clears its screen to a blue background colour. Still on the subject of the PRINT command, there's an oddity about this one. You can type a question mark (?) in place of PRINT. What's more, if you use a question mark, you don't need to leave a space between it and the first number! You can type:

```
?1.6+3.2
```

and press ENTER to see the result without any fear of getting the dreaded syntax error message. Since using the question mark saves so much typing, it's very useful to know when you have a lot of lines which start with a PRINT command.

Once a direct command has been carried out, however, it's finished. A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press the ENTER key. Instead, the instructions are stored in the memory, ready to be carried out as and when you want. The computer needs some way of recognising the difference between your commands and your program instructions. On computers that use the 'language' called BASIC this is done by starting each program instruction with a number which is called a line number. This must be a positive whole number – the type of number that is called a positive integer. This is why you can't expect the computer to understand an instruction like 5.6 + 3= because it takes the 5 as being a line number, and the rest doesn't make sense.

```
10 PRINT 5.6+6.8
20 PRINT 9.2-4.7
30 PRINT 3.3*3.9
40 PRINT 7.6/1.4
```

Fig. 2.1. A four-line arithmetic program.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. I'm doing this just because these are simple to work with. Computers aren't used all *that* much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.1 shows a four-line program which will print some arithmetic results.

Take a close look at this, because there's a lot to get used to in these four lines. To start with, the line numbers are 10,20,30,40 rather than 1,2,3,4. This is to allow space for second thoughts. If you decide that

you want to have another instruction between line 10 and line 20, then you can type the line number 15, or 11 or 12 or any other whole number between 10 and 20, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 10 and 20. If you number your lines 1,2,3 then there's no room for these second thoughts, though you can change line numbers if you have to by using the editing commands.

The next thing to notice is how the number zero on the screen is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the 0 in place of O, nor the O in place of 0, and the slashing makes this difference more obvious to you, so that you are less likely to make mistakes. The zero that you see on the keyboard is also slashed, it is on a different key, and is differently shaped. Type some zeros and Os on the screen so that you can see the difference. In the listings you will see the zero slashed, but it will not be in the text.

Now to more important points. The star or asterisk symbol in line 30 is the symbol that the CPC464 uses as a multiply sign. Once again, we can't use the × that you might normally use to indicate multiplication because × is a letter. There is no divide sign on the keyboard either, so the CPC464, like all other small computers, uses the backslash (/) sign in its place. This is the diagonal line on the same key as the question mark, *not* the one on the key just next to it.

So far, so good. The program is entered by typing it just as you see it. You don't need to leave any space between the line number and the P of PRINT, because the CPC464 will put one in for you when it displays the program on the screen. You *must* leave a space following PRINT though, and I have to emphasise this because not all computers are quite so fussy. If you use the ? abbreviation for PRINT you don't have to worry about the space. Getting back to the program example, you will have to press the ENTER key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in the illustration. When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. There are two things that you need to know now. One is how to check that the program is actually in the memory; the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You can use the CLS key to wipe the screen first if you like, then type LIST and press the ENTER key. When you press the

ENTER key, and not until, your program will be listed on the screen. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. The ? signs have been converted to the word PRINT, and a space has been inserted between the word and the first digit of a number. To make the program operate, you need another command, RUN. Type RUN, then press the ENTER key, and you will see the instructions carried out. To be more precise, you will see:

```
12.4
4.5
12.87
5.42857143
```

That last line should give you some idea of how precisely the CPC464 can carry out this type of arithmetic. You'll notice, by the way, that if you listed the program before you ran it, the results are printed under the program listing. You can avoid this by using CLS (press ENTER) before you use RUN.

When you follow the instruction word PRINT with a piece of arithmetic like 2.8 * 4.4, then what is printed when the program runs is the *result* of working out that piece of arithmetic. The program *doesn't* print 2.8 * 4.4, it just prints the result of the action 2.8 * 4.4.

Now this is useful, but it's not always handy to get a set of answers on the screen, especially if you have forgotten what the questions were. The CPC464 allows you a way of printing anything that you like on the screen, exactly as you type it, by the use of what is called a *string*.

```
10 PRINT"2+2="2+2
20 PRINT"2.5*3.5="2.5*3.5
30 PRINT"9.4-2.2="9.4-2.2
40 PRINT"27.6/2.2="27.6/2.2
```

Fig. 2.2. Using quote marks. In this and other examples, the abbreviation ? was used in place of the PRINT instruction word, but PRINT appears in the listing.

Figure 2.2 illustrates this principle. In each line, some of the typing is enclosed between quotes (inverted commas) and some is not. Enter the short program in Fig. 2.2, clear the screen, and run it. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed *exactly* as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

```
2+2=4
```

Now there's nothing automatic about this. If you type a new line:

```
15 PRINT "2+2="5*1.5
```

then you'll get the daft reply, when you RUN this, of:

```
2+2= 7.5
```

The computer does as it's told and that's what you told it to do. Only a loony would believe that computers could take over the world!

This is a good point at which to take notice of something else. The line 15 that you added has been fitted into place between lines 10 and 20 - LIST if you don't believe it. No matter in what order you type the lines of your program, the computer will sort them into order of ascending line number for you. In addition, you'll see that the computer has put a space between the = sign and the first digit of the answer. This is to allow for a + or − sign, and we sometimes want to put in an extra space here. This can be done by leaving a space between the = sign and the final quotemark (like = ") in the line.

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, used alone in this way, always means print on to the TV screen. For activating a paper printer (*hard copy*, as it's called), there's a separate variety of PRINT instruction which is followed by a hashmark (#) and the number 8. PRINT#8, for example, will make printing go to a printer, *if you have one connected.* If you don't have a printer connected, you have to avoid this command, because it will make the computer appear to lock up, paying no attention to the keys and doing nothing. If you get into this state, you can either connect a printer, or just press ESC twice to get out of trouble. Appendix B illustrates how to use a parallel printer, such as the Epson RX80, with the CPC464.

```
10 PRINT"This is"
20 PRINT"the excellent"
30 PRINT"Amstrad CPC464 computer"
```

Fig. 2.3. Using the PRINT instruction to place words on the screen.

Now try the program in Fig. 2.3. You can try typing the lines in any order you like, to establish the point that they will be in line number order when you list the program. When you clear the screen and RUN the program, the words appear on three separate lines. This is because the instruction PRINT doesn't just mean print on the screen.

It also means take a new line, and start at the left-hand side! You will also find, incidentally, that when the words on the screen reach the bottom line of the screen, then all the lines appear to move up, and the top line disappears. This is the action that is called *scrolling*, and it's the way that the machine deals with displaying lots of lines on a screen which holds only 25 lines altogether.

Now the action of selecting a new line for each PRINT isn't always convenient, and we can change the action by using punctuation marks that we call print modifiers. Start this time by acquiring a new habit. Type NEW and then press the ENTER key. This clears out the old program, and you might also like to use the CLS action to clear the screen. If you don't use the NEW action, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored. In Fig. 2.2, for example, the line 15 that you added would be left in store even when you typed a new line 10 and a new line 20.

```
10 PRINT"This is ";
20 PRINT"the excellent ";
30 PRINT"Amstrad CPC464"
```

Fig. 2.4. The effect of semicolons.

Now try the program in Fig. 2.4. There's a very important difference between Fig. 2.4 and Fig. 2.3, as you'll see when you RUN it. The effect of a semicolon following the last quote in a line is to prevent the next piece of printing starting on a new line at the left-hand side. When you RUN this program, all of the words appear in one line. It would have been a lot easier just to have one line of program that read

10 PRINT "This is the excellent Amstrad CPC464"

to do this, but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at that sort of thing later in program examples. In the meantime, try a small change. Alter line 30 so that it reads:

30 PRINT "Amstrad CPC464 computer"

and RUN the program again. This time, you'll see quite a different result. The first two lines have been joined, but the last line is on its own. This is because if the last line had been joined on, a word would

have been split from one line to the next. The CPC464 will not allow this, and it's a feature unique to this machine, which makes it easier to design neater printing!

Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT will cause a new line to be selected, so the action of Fig. 2.5 should not come as too much of a surprise. Lines 10 and 20 contain a novelty,

```
10 CLS:PRINT"This is the CPC464"
20 PRINT:PRINT
30 PRINT"Ready to work for you."
```

Fig. 2.5. Clearing the screen with the CLS instruction, and using multistatement lines. The action of the CLS key is not the same as that of CLS in a program line.

though, in the form of two instructions in one line. The instructions are separated by a colon (:), and you can, if you like, have several instructions following one line number in this way, taking several screen lines. The only practical limit to this is that it makes your instructions too hard to read if you put too many instructions together in this way. In a 'multistatement' line of this type, the CPC464 will deal with the different instructions in a left-to-right order. The instruction CLS should not surprise you either – this clears the screen, and makes the printing start at the top left-hand corner. It's the same action as the CLS direct command, but done automatically within the program.

Another point about Fig. 2.5 is that line 20 causes the lines to be spaced apart. The two PRINT instructions, with nothing to be printed, each cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space, it's very handy.

Figure 2.6 deals with columns. Line 10 is a PRINT instruction that acts on the numbers 1 and 2. When these appear on the screen, though, they appear spaced out just as if the screen had been divided into columns. The mark which causes this effect is the comma, and the action is completely automatic. The comma is on the key next to the letter M, and if you use the apostrophe on the 7 key, you will not

```
10 PRINT 1,2
20 PRINT 1,2,3,4
30 PRINT"ONE","TWO"
40 PRINT"ONE","TWO","THREE","FOUR"
50 PRINT"THIS ITEM IS LONGER","TWO"
60 PRINT 1,2,3,4,5,6
```

Fig. 2.6. How the comma causes words to be placed into columns.

get the same effect! The two look rather alike on the keyboard, but completely different on the screen. As line 20 shows, you can get only three columns, each one of which allows room for up to fourteen characters, depending on whether you use letters or numbers. Anything that you try to get into a fourth column will actually appear on the first column of the next line down. The action works for words as well as for numbers, as lines 30 and 40 illustrate. When words are being printed in this way, though, you have to remember that the commas must be placed *outside* the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into a column something that is too large to fit, the long phrase will spill over to the next column, and the next item to be printed will be at the start of the following column. Line 50 illustrates this – the first phrase spills over from column 1 all the way to column 2, and the word TWO is printed starting at column 3 on the same line. Line 60 shows what happens if you keep using commas – the columns just take up the same positions on the next line.

This action of commas is used on practically every home computer, but the CPC464 adds a new twist. Type ZONE 6, press ENTER, then RUN your program again. This time, things fit better! It's as if you suddenly had more columns, and you do. The number that follows ZONE gives the number of spaces between columns, so it allows you to mark out the screen (invisibly) for printing in any way that you like. You can change the value of ZONE during a program, for example, so that different bits of printing are differently arranged. This is particularly useful if you want to do tabulated work for business purposes.

Commas are useful when you want a simple way of creating columns which all have the same spacing. A much more flexible method of placing words on the screen exists, however. This is programmed by using the command word TAB, which has to follow PRINT. TAB is short for 'tabulate', and it means 'split into columns'.

For the purpose of using TAB, we need to remember that the screen, as it exists when we switch on, is divided into 40 columns across by 25 lines down. Figure 2.7 shows a TAB map of these column numbers.

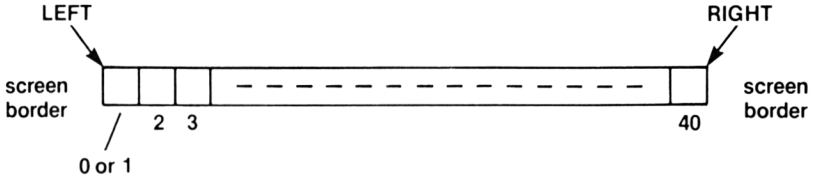

Fig. 2.7. The TAB map, which shows how the TAB numbers correspond to positions in a line.

The positions are numbered 1 to 40, but you can also use TAB(0), which is the same position as TAB(1), the left-hand edge. TAB(1), then, would mean the left-hand side, and TAB(40) would be the right-hand side. The word TAB must follow PRINT, and must itself be followed by the TAB number in brackets. If you omit the brackets, you will see the odd effect of a zero being printed as well as your number or phrase – you'll understand why later.

```
10 PRINT TAB(18)"Centre"
20 PRINT TAB(5)"Start here"
30 PRINT TAB(45)"45 is here"
```

Fig. 2.8. How TAB is used to position the cursor in a program.

Now try a TAB example, as in Fig. 2.8. The first word is printed in the centre of the screen, and the second one near the left-hand side. The third word, however, is printed in the same position in its line as the second. That's because numbers of 1 to 40 operate the TAB command, and if you use 41, then it's equivalent to starting all over again with 1. You *don't* get a line skipped by using a TAB number greater than 40, as you do with some machines. There's another point here too. The word 'centre' in this example was printed in the centre of the screen by using TAB(18). Figure 2.9 shows the formula that is used to find the TAB number for making any word or phrase appear centred on the screen. Remember when you use this, however, that the screen which the CPC464 uses is not always perfectly centred on the screen of the monitor or TV. On my colour monitor, for example, the margin at the left is larger than the margin at the right.

1. Count number of characters in the title, including spaces.
2. Subtract this number from 40 if it is even; from 41 if it is odd.
3. Divide the result by 2.
4. Use the result as the TAB number.

Fig. 2.9. The formula for centring a title.

Figure 2.10 shows a rather different way of spacing out figures and letters on the screen. This uses the SPC command, and though you might think that it's pretty much like the TAB command, it isn't! Figure 2.10 shows the difference. When you TAB the positions, the

```
10 PRINT TAB(2)"CPC464"TAB(16)"COMPUTER"
20 PRINT TAB(2)"CPC464"SPC(16)"COMPUTER"
```

Fig. 2.10. Using SPC to space printing.

first letter of each word is placed in the appropriate TAB position. Note, incidentally, that you can have more than one TAB in a line following a PRINT command. Unlike most computers, the CPC464 doesn't need any semicolons to separate bits of this command. In line 20, SPC is used. Now this does not TAB to position 16 – it prints 16 spaces between the end of CPC464 and the start of COMPUTER. That's quite different from having the C of COMPUTER in TAB position 16. The general rule is that you use TAB if you want neat columns, with the first letter of each word starting in the same position of each line. You use SPC if you want to fix the amount of space between words or numbers, even if these words or numbers are of different lengths.

There's yet another way of positioning your printing, and it's even more free-ranging than TAB. The instruction word is LOCATE, and it allows you to place words on the screen at any position, and in any order. Normally, when you PRINT on to the screen, a PRINT instruction causes the computer to take a new line and you cannot go back to the previous one. By using LOCATE, you can place words and numbers where you like, and even have one line replace another if you want to.

The important difference is that the CPC464 LOCATE command *must* be issued before the PRINT instruction that it refers to. It can be in the line just before the PRINT instruction, or it can be in the same line, separated by a colon.

LOCATE has to be followed by two numbers. Of these, the first

number is a *column number*. You can print 40 characters (letters, numbers, punctuation marks) in a line across the screen of the CPC464. Each of these characters is in one column, because characters in all of the other lines are also spaced out in the same way. We can use a number, then, to represent the position of a character on a line. The number can range from 1 (left-hand side) to 40 (right-hand side), just like the TAB numbers. The second number of the LOCATE command is a *screen-line number*. The lines on the screen are numbered starting with 1 (the top line), and ending with 25 (bottom line of the display part of the screen).

```
10 CLS
20 LOCATE 1,1:PRINT"TOP LEFT"
30 FOR N=1 TO 1000:NEXT
40 LOCATE 29,25:PRINT"BOTTOM RIGHT";
50 FOR N= 1 TO 1000:NEXT
60 LOCATE 1,25:PRINT"BOTTOM LEFT"
70 FOR N=1 TO 1000:NEXT
80 LOCATE 32,1:PRINT"TOP RIGHT"
90 FOR N=1 TO 1000:NEXT
100 LOCATE 17,12:PRINT"MIDDLE"
```

Fig. 2.11. How LOCATE allows you to print anywhere on the screen.

An example would certainly help here. Take a look at Fig. 2.11. Line 10 clears the screen, and line 20 introduces the first LOCATE. This is 1,1 meaning that printing will start on the top left-hand corner of the screen. There *must* be a space between the E of LOCATE and the first number. The T of TOP LEFT will therefore be placed at the top left corner of the screen, as it would anyhow just following a CLS. The next line then causes a pause. We haven't looked at this type of command yet, but we'll come to it later.

The next LOCATE is followed by 29,25, so that the phrase BOTTOM RIGHT is placed with the T of RIGHT in the bottom right-hand corner of the screen. How was this calculated? The answer is by counting the T of right as column 40, then counting back 39,38,37 and so on until reaching the B, which was at 29. This is to be on the bottom line, number 25. To prevent the screen from scrolling at this point, I have had to add the semicolon following this phrase. Normally, the semicolon isn't needed, but the T of BOTTOM RIGHT comes at the last position of the screen, and this would normally cause a scroll action. With this much information now, you can see how the rest of the words have been put into position. Because

of the pauses, you can see that we are not following the normal order of left-to-right, top-to-bottom for printing.

Figure 2.12 is a map that you will find useful for placing words where you want them on the screen with LOCATE. This map shows

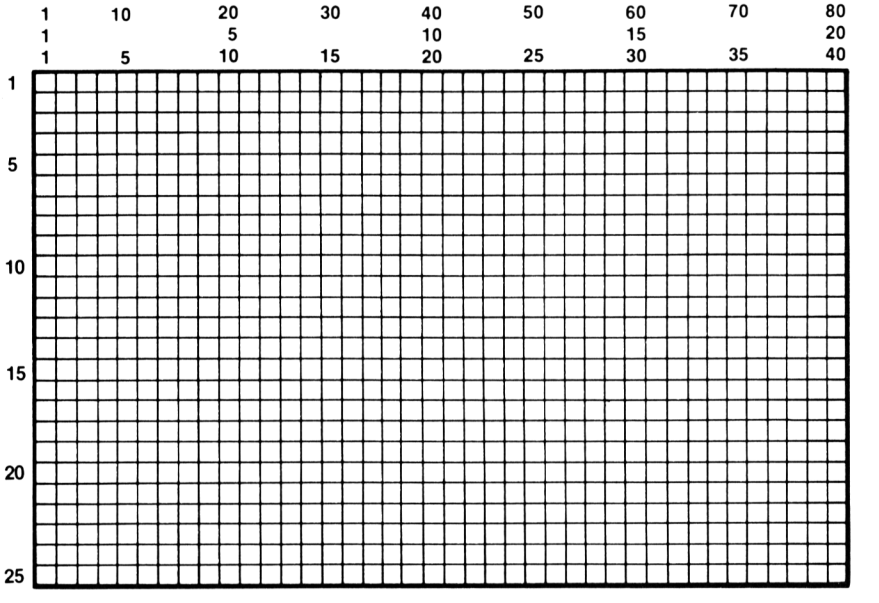

Fig. 2.12. A LOCATE map to help you to make the best use of this command.

several sets of numbers, and you may wonder why. The reason is that you can choose three different character sizes on your screen. Keep the program of Fig. 2.11 in the memory, and type MODE 0 (then ENTER). There *must* be a space between the E of MODE and the zero. If you now LIST your program, you'll find that all the characters are double the size that they were previously. When you run the program, you will see the top left and bottom left appear in the correct places, but not the right-hand words. This is because the range of LOCATE numbers is now 1 to 20 across, though it is still 1 to 25 down. Now try typing MODE 2 (ENTER). This time, you'll find that the characters are much smaller, 80 to a line. They are also much harder to read, especially on a colour monitor or TV receiver. The LOCATE horizontal numbers are now in the range 1 to 80. When you switch on the CPC464, the MODE is always MODE 1, with 40 characters per line. This in an excellent compromise between making the characters easy to read and getting a reasonable number on to the screen at one time. Unless you are using the green-screen monitor, it's best to avoid the MODE 2 letters.

One final point. The CPC464 contains two of the sort of commands that so many machines lack. One is AUTO. If you type AUTO (ENTER), then the machine will number lines for you automatically. All you have to do is type whatever has to appear in each line. Each time you press ENTER, another line number will appear ready for you! To stop the auto-numbering, press ESC twice. You can also start the AUTO action at any number you like (try AUTO 100, for example), and you need not have the line numbers incrementing in tens either – try AUTO 10,5. This is very handy if you are copying a program from a printed listing, and the listing has been renumbered in tens.

RENUM is the other command. Type RENUM, and your program will be renumbered starting at line 10, and with the line numbers incrementing in tens, no matter how many odd numbered lines you used. You can choose to make your renumbering start at some existing line, say line 240, you can choose what number you want this first line to have, and you can choose the increment number. Believe it or not, there are machines that don't have these facilities!

Chapter Three
A Few Variations

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at some of the actions that go on before anything is printed. One of these is called *assignment*. Take a look at the program in Fig. 3.1. Type it in, run it, and contrast what you see on

```
10 CLS
20 X=23
30 PRINT"2 TIMES"X" IS";2*X
40 X=5
50 PRINT"X IS NOW"X
60 PRINT"AND TWICE"X" IS";2*X
```

Fig. 3.1. Assignment in action. The letter X has been used in place of a number.

the screen with what appears in the program. The first line that is printed is line 30. What appears on the screen is:

2 TIMES 23 IS 46

but the numbers 23 and 46 don't appear in line 30! This is because of the way we have used the letter X as a kind of code for the number 23. The official name for this type of code is a *variable name.*

Line 20 assigns the variable name X, giving it the value of 23. 'Assigns' means that wherever we use X, *not* enclosed by quotes, the computer will operate with the number 23 instead. Since X is a single character and 23 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as X=2174.3256, for example. Line 30 then proves that X is taken to be 23, because wherever X appears, not between quotes, 23 is printed, and the 'expression' 2*X is printed as 46. We're not stuck with X as representing 23 for ever, though. Line 40 assigns X as being 5, and lines 50 and 60 prove that the change has been made.

That's why we call X a 'variable' – we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned. Even after you have run the program of Fig. 3.1, providing you haven't added new lines or deleted any part of it, you can type PRINT X, and pressing ENTER will show the value of X on the screen. The listing also shows a curiosity. You can follow a phrase which is in quotes by a variable name, as in PRINT"2 TIMES"X in line 30. You *cannot*, however, do this if the phrase is followed by an expression, something like 2*X or X–7. You will get an error message here if you attempt to use " IS"2*X in the program. Because of this, a lot of programmers automatically use a semicolon wherever a quote sign is followed by a variable name or expression.

Getting back to variables, this very useful way of handling numbers in code form can use a 'name' instead, which must start with a letter, upper-case (capital) or lower-case (small). You can add to that letter other letters, making a complete word if you like, or digits – but not spaces, arithmetic symbols (+,–,*,/) or punctuation marks of any kind. Names like TOTAL, lastname, ALLTHEREIS and R2D2 can all be used for number variables, and each can be assigned to a different number. One thing that you need to be careful about, though, is that the CPC464 *does not distinguish between upper- and lower-case names*. If you assign the variable name of jam to the number 45, and then assign the name of JAM to 88, you will find that both jam and JAM have been assigned to the same value, 88 in this case. This will be whichever number you assigned most recently. Another thing to watch for is the use of *reserved words*. The reserved words of the CPC464 are its instruction words – words like PRINT, NEW, RUN and so on. You *cannot* use these as variable names, and you will get a 'Syntax error' message if you attempt to use them. Some computers won't even allow you to use words which *contain* these reserved words, so that you could not use words like NEWLY, for example. The CPC464, however, is more tolerant, and will allow you to use any word which is not identical to a reserved word. If you ask the CPC464 to print the value of a word which has not been assigned, it will come up with the value of 0 rather than with an error message, as some machines do.

Just to make it even more useful, you can use similar 'names' to represent words and phrases also. The difference is that you have to add a dollar sign ($) to the variable name. If N is a variable name for a number, then N$ (pronounced en-string or en-dollar) is a variable name for a word or phrase. The computer treats these two, N and N$, as being entirely separate and different. They also have to be assigned

in rather different ways. When you assign a number to a number variable, using the = sign, you don't have to type a quotemark (") on each side of the number. When you assign a string variable in this way, however, you have to make use of quotemarks. We'll look at other methods of making assignments later. In the next example, we shall make use of 'long' variable names, by which we mean more than two letters. Using just one letter saves memory space, but a good choice of variable name can help to remind you of what the variable represents. When you have so much memory to make use of as the CPC464 permits, you can afford to be generous with your names!

Serenade for strings

Figure 3.2 illustrates *string variables*, meaning the use of variable names for words and phrases. Lines 10 and 20 carry out the assignment operations, and lines 30 to 50 show how these variable

```
10 CLS:NAME$="CPC464"
20 FIRST$="The excellent":LAST$="Compute
r system"
30 PRINT FIRST$;" ";NAME$;" ";LAST$
40 PRINT"This uses the ";NAME$
50 PRINT FIRST$;" ";NAME$;" in action!"
```

Fig. 3.2. Using string variables. These are distinguished by the dollar sign.

names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text, which *must* be surrounded by quotes. You have to be careful when you mix these two, because it's easy to run words together. Note in lines 30 to 50 how spaces have been left between words. When you are printing one variable after another, the space is created by typing quotes, then pressing the spacebar, then another quotemark, like " ". To leave a space between text in quotes and a variable name, you only need to press the spacebar at the point where you need the space. The semicolons are *not essential* when you are joining up bits of text in this way. If you omit the semicolon at a join, you will get the same effect as if you had included it, provided no number expressions are used.

Figure 3.3 shows another example, this time using the variable names BLURB$ and PUFF$ for longer phrases. There wouldn't be much point in printing messages in this way if you wanted the

```
10 CLS:BLURB$="The new computer"
20 PUFF$="that brings real computing at
less cost"
30 PRINT"The CPC464- "
40 PRINT BLURB$;PUFF$
50 PRINT BLURB$+PUFF$
```

Fig. 3.3. Illustrating the use of longer variable names for phrases of several words.

message once only, but when you continually use a phrase in a program, this is one method of programming it so that you don't have to keep typing it! There's another thing to note here too. In line 40, the two variable names are separated by a semicolon. Even this separation isn't necessary for the program to work, but it helps a lot when you are trying to read the program.

When you use the semicolon as a spacer between variables in this way, or if you simply type the variables one after the other, the printing on the screen will try to avoid splitting words. If the 'PUFF$' phrase were printed directly following BLURB$ on the same line, the word 'computing' would be cut in half, as line 40 shows. If, of course, you make one of the variables equal to a line of text that takes more than one line on the screen, this form of joining phrases won't prevent a word from being split. In line 50, the phrases have been forced together by using a + sign between them. We'll come back to this sort of use of the + sign later.

Strings and things

Because the name of a string variable is marked by the use of the $ sign, a variable like A$ is not confused with a number variable like A. We can, in fact, use both on the same program knowing that the computer at least will not be confused. Figure 3.4 illustrates that the

```
10 A=2:B=3
20 A$="2":B$="3"
30 CLS
40 PRINT A,B
50 PRINT A$,B$
60 PRINT A" times"B" is";A*B
70 PRINT A$" times "B$" is impossible!"
```

Fig. 3.4. String and number variables might look alike when they are printed, but they are different!

difference is more than just skin deep. Lines 10 and 20 assign number variables A and B, and string variables A$ and B$. When these variables are printed in lines 40 and 50, you can't tell the difference between A and A$ or between B and B$. They only noticeable difference when you see them printed in columns is that the number variables always have a space in front of the value. The difference appears, however, when the computer attempts to carry out arithmetic. It can multiply two number variables because numbers can be multiplied, but it can't multiply string variables, whether these represent numbers or not. You can multiply "2" by "3", but you can't multiply "2 LABURNUM WAY" by "3 ACACIA AVENUE". The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other arithmetic operation on strings. Attempting to do a forbidden operation in line 70 causes an error message when the program runs, and this error will always halt a program. The message that appears is 'Type mismatch in 70' and it means that you have tried to perform an operation with strings that can be done only with number variables. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and that attempts to perform these operations on numbers will also cause the same error message. The difference is an important one. The computer stores numbers in a way that is quite different from the way it stores strings. The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's only a machine!

There is one operation that looks like arithmetic that can be carried out on strings, but not on numbers. It uses the + sign, but it isn't addition in the sense of adding numbers. Figure 3.5 illustrates this

```
10 A$="FORBES"
20 B$="JONES"
30 CLS
40 PRINT"Just call me "A$+"-"+B$+" ,he s
aid."
50 PRINT:A$="123":b$="456"
60 PRINT"Joined string is ";A$+B$
70 PRINT"Addition would give";579
```

Fig. 3.5. Concatenating or joining strings. This is not the same action as addition!

action of joining strings, which is often called *concatenation*. This is nothing like the action of arithmetic, as you'll see by lines 40 and 60.

Line 60 uses numbers in place of the names placed between the quotes. Just to point out the difference, line 70 shows what would have happened if these had been number variables. Concatenation is a very useful way of obtaining strings which otherwise would need rather a lot of typing, and you have seen already how it forces one string to be tacked on to the end of the other. Take a look at Fig. 3.6.

```
10 A$="***":B$="###"
20 S$="THE NEW CPC464"
30 CLS
40 PRINT TAB(8)A$+B$+S$+B$+A$
```

Fig. 3.6. Using concatenation to make a frame for a title.

This defines A$ and B$ as characters which can be used as 'frames' around a title. The title is defined in line 20 as 'THE NEW CPC464'. Line 40 then prints a concatenated string.

Along with this business of concatenation, there's a very useful command which will join a number of identical characters into a string for you. The command is STRING$, and it has to be followed by two items, enclosed in brackets. The second item is the character that you want to use, and the first item is the number of these characters you want. For example, if you program G$=STRING$ (20,"$"), this will make the string G$ contain twenty dollar signs. Figure 3.7 illustrates this STRING$ action used to make a frame for a title.

```
10 CLS
20 A$=STRING$(10,"*")
30 B$=STRING$(5,"#")
40 PRINT:PRINT A$+B$+"TITLE"+B$+A$
```

Fig. 3.7. Using STRING$ to provide a set of identical characters. See Fig. 8.10 for a different way of using STRING$.

Getting some input

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, number or name, into a program while it is running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT.

```
10 CLS
20 PRINT"What is your name"
30 INPUT NAME$
40 CLS:PRINT:PRINT
50 PRINT NAME$;" -this is your life!"
```

Fig. 3.8. Using the INPUT instruction. The name that you type is put into the phrase in line 50.

Figure 3.8 illustrates this with a program that prints your name. Now I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

What is your name

are printed on the screen. On the line below this you will see a question mark, followed by the (gold) cursor. The computer is now waiting for you to type something, and then press ENTER. Until the ENTER key is pressed, the program will hang up at line 30, waiting for you. If you're honest, you will type your own name and then press ENTER. You *don't* have to put quotes around your name – simply type it in the form that you want to see printed. When you press ENTER, your name is assigned to the variable NAME$. This is one method of assignment to a string variable that doesn't need the use of quotes. The program can then continue, so that line 40 clears the screen and spaces down by two lines. Line 50 then prints the famous phrase with your name at the start. You could, of course, have answered Mickey Mouse or Donald Duck or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. Even if you type nothing, and just press ENTER, it will carry on, with no name at all. Don't listen to the nutters who tell you that computers know everything!

```
10 CLS:PRINT"Enter a number, please"
20 INPUT n
30 CLS:PRINT
40 PRINT"Twice"n" is";2*n
```

Fig. 3.9. An INPUT to a number variable. The quantity that you type must be a number.

We aren't confined to using string variables along with INPUT. Figure 3.9 illustrates an INPUT step which uses a number variable n. The same procedure is used. When the program hangs up with the question mark and the cursor appearing, you can type a number and

then press the ENTER key. The action of pressing ENTER will assign your number to n, and allow the program to continue. Line 40 then proves that the program is dealing with the number that you entered. When you use a number variable in an INPUT step, then what you have typed when you press ENTER must be a number. If you attempt to enter a string, the computer will refuse to accept it, and you will get an error message – 'redo from start'. Unlike most error messages, this does *not* cause the program to stop; it simply allows you another chance to get it right. If your INPUT step uses a string variable, then *anything* that you type will be accepted when you press ENTER, but you will get an error message if you try to perform arithmetic on a string.

The way in which INPUT can be placed in programs can be used to make it look as if the computer is paying some attention to what you type. Figure 3.10 shows an example – but with INPUT used in a

```
10 CLS
20 INPUT"Type your name, please ";NAME$
30 PRINT
40 PRINT"Very pleased to meet you, ";NAM
E$
```

Fig. 3.10. Using INPUT to print a phrase which requests the input.

different way. This time, there is a phrase following the INPUT instruction. The phrase is placed between quotes, and is followed by a semicolon and then the variable name NAME$. This line 20 has the same effect as the two lines:

```
15 PRINT "Type your name, please";
20 INPUT NAME$
```

and this time the question mark and the cursor appear on the *same line* as the question. Your reply is also on the same line – unless the length of the name causes letters to spill over on to the next line. You can also use the comma in place of the semicolon in a line like this. Try the effect for yourself. The comma causes the question mark to be deleted, but the cursor is still present.

The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can mix variable types in one INPUT line. Figure 3.11, for example, shows two variables being used after one INPUT. One of the variables is a string variable NAME$, the other is the number variable NR. Now when the computer comes to line 20, it will print the message and then wait

```
10 CLS
20 INPUT"Name and number, please ";NAME$
,NR
30 PRINT:PRINT
40 PRINT"The name is ";NAME$
50 PRINT"The number is ";NR
```

Fig. 3.11. Putting in two variables in one INPUT step.

for you to enter *both* of these quantities, a name and then a number. There is just one way of doing this correctly. That is to type the name, then comma, then the number, and the press ENTER. If you press ENTER after typing the name but before you have typed the number, you will get the 'Redo from start' message, and you must then enter both the name and the number, then press ENTER. The name and number will then be printed again in lines 40 and 50. From this, you can see that you can never enter anything that contains a comma when you have an INPUT. For example, if you have an INPUT NAME$ step, you can't enter BLOGGS, FRED. This would be treated as two entries, and only BLOGGS would be accepted. Attempting to enter BLOGGS, FRED would cause the 'Redo from start' message. They are very particular about detail, these computers, and no two makes are exactly alike! There is, however, another command which allows you to enter items which contain commas. It's LINE INPUT, and you can use it in exactly the same way as you use INPUT.

```
10 CLS
20 INPUT"Four numbers, please ";A,B,C,D
30 PRINT
40 PRINT"The sum of these is ";A+B+C+D
```

Fig. 3.12. An INPUT step which calls for four numbers.

We can extend this principle further. Figure 3.12 calls for four numbers to be entered. Once again, these must be entered in one go, separated by commas, pressing ENTER only after all four have been typed. The numbers are assigned to the variable names, and the program will print the sum in line 40.

Reading the data

There's yet another way of getting data into a program while it is running. This one involves reading items from a list, and it uses two

instruction words, READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items of the list must be separated by commas. Each time an item is read from such a list, a 'pointer' is altered so that the next time an item is needed, it will be the next item on the list rather than the one that was read the last time round.

We'll look at this in more detail in Chapter 5, but for the moment we can introduce ourselves to the READ...DATA instructions. Figure 3.13 uses the instructions in a very simple way. Line 20 reads

```
10 CLS
20 READ J
30 PRINT"ITEM ";J;
40 PRINT" is a ";
50 READ NAME$
60 PRINT NAME$
70 DATA 5, disk drive
```

Fig. 3.13. Using the READ and DATA words to place information into a program.

an item number, which is the first item on the list and assigns it to the variable J. This is printed in line 30, with the semicolon keeping printing in the same line so that the phrase in line 40 follows it. The semicolon at the end of line 40 once more keeps the printing in the same line, and line 50 reads the name which is the second item in the list. This is assigned to the variable name NAME$ and printed in line 60. Line 70 contains the DATA, one number and one phrase. The number can be typed as it is, and you can see also that the phrase needs *no* quotes round it. Anything that you assign to a string variable in this way need have no quotes around it in its DATA line. If you do put in quotes, they will be ignored, but no error message will appear.

You must always be careful about how you match your READ and your DATA. If you use a number variable in the READ line, like READ A, then what is in the DATA line being read *must* be a number. If it is not, then the program will stop with an error message. This will show that the DATA line is faulty, and present you with a chance to correct it. Perhaps this is a good time to break off and read the Appendix on editing! If you use a string variable, as in READ A$, then it doesn't matter whether your DATA line contains a number or a string. Remember that if you read a number using READ A$, then

the CPC464 will not allow you to carry out any arithmetic on that number. Note that you *must* have a space following the word DATA, and that your data items must be separated by a comma. You can't use items of data that contain commas.

The READ...DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step. These would be items that you would need every time that the program was used, rather than the items you would type in as replies. We're not quite ready for that yet, so having introduced the idea, we'll leave it for now.

Number antics

The amount of computing that we have done so far should have persuaded you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for word processing or even for accounts. It's time, then, to take a very brief look at the number abilities of the CPC464. It is a brief look because we simply don't have space to explain what all the mathematical operations do. In general, if you understand what a mathematical term like sin or tan or exp means, then you will have no problems about using these mathematical functions in your programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is counting. Counting involves the ideas of *incrementing* if you are counting up and *decrementing* if you are counting down. Incrementing a number means adding 1 to it, decrementing means subtracting 1 from it. These actions are programmed in a rather confusing-looking way in

```
10 CLS
20 X=5:PRINT"X IS"X
30 PRINT
40 X=X+1
50 PRINT"IT'S CHANGED - "
60 PRINT"X IS NOW"X
```

Fig. 3.14. Incrementing, using the equals sign to mean 'becomes'.

BASIC, as Fig. 3.14 shows. Line 20 sets the value of variable X as 5. This is printed in line 20, but then line 40 'increments X'. This is done using the odd-looking instruction X=X+1, meaning that the new value that is assigned to X is 1 more than its previous value. The rest of the program proves that this action of incrementing the value of X has been carried out.

The use of the = sign to mean 'becomes' is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could equally well have a line:

```
X=X-1
```

which would have the effect of making the new value of X one less than the old value. X has been *decremented* this time. We could also use X=2*X to produce a new value of X equal to double the old value, or X=X/3 to produce a new value of X equal to the old value divided by three. Figure 3.15 shows another assignment of this type,

```
10 CLS
20 X=5:PRINT"X is"X
30 PRINT
40 X=2*X+4
50 PRINT"It's changed -"
60 PRINT"X is now"X
```

Fig. 3.15. A more elaborate assignment, using an *expression*.

in which both a multiplication and an addition are used to change the value of X.

Figure 3.16 illustrates the use of some number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 10 picks the value of 2.5 for X. Line 20 then prints the value of X squared, meaning X multiplied

```
10 CLS:X=2.5
20 PRINT"X squared us";X^2
30 PRINT
40 PRINT"It's square root is";SQR(X)
50 PRINT
60 PRINT"It's natural log. is";LOG(X)
70 PRINT"and it's ordinary log. is";LOG1
0(X)
```

Fig. 3.16. Using some number functions.

by X. This is programmed by typing X↑2, and the character which the CPC464 uses for this is on the same key as the pound sign. To obtain the square root of the number that has been assigned to X, we use the instruction word SQR. An alternative is X↑.5, but SQR(X) is easier to type and remember. For other roots, like the cube root, you can use expressions like X↑(1/3) and so on. LOG(X) produces the natural logarithm of X. This is not the type of logarithm that you may want, and to find the ordinary (base 10) log, you have to use LOG10(X).

Figure 3.17 illustrates the various number functions that can be used, with a brief explanation of what each one does. Some of these actions will be of use only if you are interested in programming for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs. Figure 3.17 also shows the order of priority of actions. This makes sure that when you type something like 3+4*5, then what you get is the logical result. In this case it is 23, because the multiplication is *always* done first, and the addition follows.

ABS(X) Converts negative sign to positive.
ATN(X) Gives angle (in radians) whose tangent is X.
BIN$(X,Y) Converts number X into binary, and fills with zeros to a length Y.
CINT(X) Converts X to an integer. This will be rounded if X contains a fraction.
COS(X) Gives the cosine of angle X (radians).
CREAL(X) Converts X to a 'real' number, as distinct from an integer.
DEG Sets angles in degrees.
EXP(X) Gives the value of *e* to the power X.
FIX(X) Strips fraction from X.
FRE(X) Gives amount of memory not in use or reserved.
HEX$ Converts number into hex (base 16).
INT(X) Gives the whole-number part of X, rounded to the nearest smaller whole number.
LOG(X) Gives the natural logarithm of X.
LOG10(X) Gives the logarithm to base 10 (common log) of X.
MAX(X,Y,Z,...) Gives the number which is the greatest in a list.
MIN(X,Y,Z,...) Gives the number which is the minimum in a list.
PI Gives the value of pi, the ratio of the circumference to the diameter of a circle.
RAD Sets angles in radians.
RANDOMIZE(X) Sets new sequence of 'random' numbers.
RND(X) Gives a random fraction between 0 and 1.
ROUND(X,Y) Rounds X to the number of places specified by Y.

SGN(X) Gives the sign of X. The result is +1 if X is positive, −1 if X is negative, 0 if X is zero.
SIN(X) Gives the sine of angle X (radians).
SQR(X) Gives the square root of X.
TAN(X) Gives the value of the tangent of angle X (radians).
UNT(X) X must be between 0 and 65536. UNT converts it into a number between −32768 and +32767.

Order of priority
For ordinary arithmetic, the order of priority is MDAS - multiplication and division, followed by addition and subtraction. The full order of priority is:

1. Raising to a power, using ↑.
2. Multiplication and division.
3. Addition and subtraction.
4. Comparison, using = <>.
5. AND
6. OR
7. NOT

Fig. 3.17. Number functions, with brief notes. (Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!)

How precise?

One of the problems of small computers is precision of numbers. You probably know that the fraction ⅓ cannot be expressed exactly as a decimal. How near we can get to its true value depends on the number of decimal places we are prepared to print, so that 0.33 is closer than 0.3, and 0.333 is closer still. The computer converts most of the numbers it works with into the form of a fraction and a multiplier. The fraction is not a decimal fraction but a special form called a *binary fraction*, and this conversion is seldom exact. The conversion is particularly awkward for numbers like 1, 10, 100 and .1, .01, .001; all the powers of ten, in fact. To avoid embarrassments like printing 3−2=.9999999, the computer will round numbers of this type up or

```
10 CLS
20 PRINT 1/3,2/3
30 PRINT 1/11,10/11
40 PRINT 1/3+2/3,1/11+10/11
```

Fig. 3.18. How the computer copes with 'awkward' numbers. Very small or very large numbers are converted into *standard form*.

down as need be, before displaying them. Not all computers do this well – you can be glad that you bought a CPC464! Figure 3.18 shows how the CPC464 copes with fractions like 1/3, 2/3, 1/11 and 10/11. The numbers that you see on the screen have been rounded to nine places of decimals, but the number that the computer *stores* must be of many more decimal places. The rounding works to make sure that adding the fractions gives the correct result.

You can also see from the way the computer prints the fraction 1/11 that there is an alternative method of printing numbers. If you have worked in any subject that uses very large or very small numbers (physics, chemistry, engineering), you will know about this method already. If you haven't met it before, it's called *standard form*. A number in standard form consists of a quantity which lies somewhere between 0 and 10, never quite equal to either of these, and multiplied by a power of ten. Take, for example, a number like 132000. If we shift the decimal point five places *left* (equivalent to *dividing* by ten to the power 5), then this becomes 1.32. To get the value correct, this would have to be multiplied by 10 to the power 5, or, as we write it E5. The number 132000 could therefore be written as 1.32E5.

Suppose we try a small number, like 0.00036. This is 3.6 times ten to the minus 4, or 3.6E−4, with the minus sign there because we have had to shift the decimal point four places *right* to get this result. The CPC464, like most small computers, will accept and print numbers in this form. The conversion is automatic, and the CPC464 will display numbers in this form only when it has to – which means when the numbers would need more than nine figures to display.

Most computers allow a limited range of numbers to be stored in a much more precise way, called an *integer variable*. An integer, as far as the CPC464 is concerned, is a whole number whose value lies between the limits of −32768 and +32767. An *integer variable name* consists of the variable name followed by the % sign. If you assign a number to an integer variable, then only numbers in the correct range can be used, and any fractions will be discarded. You will see the error message 'Overflow' if you try to do something like:

```
A%=32800
```

for example. Figure 3.19 illustrates the action of rejecting fractions, because when the variable X, whose value is 3.7, is assigned to X% in line 40 then printing X% in line 50 gives 4 only. The fraction .7 has been rounded up to 1. This is unusual, and many other computers, given this line, would simply omit the .7, giving 3.

```
10 CLS:X=3.7
20 PRINT"X is an ordinary number equal t
o"X
30 PRINT" X% is an integer."
40 X%=X
50 PRINT"The value of X% is"X%
60 Y%=7/5
70 PRINT"7/5 is"Y%" in integers!"
```

Fig. 3.19. Using integers. These need less memory to store, and can be more precise – but not for division!

The advantage of using integer variables is two-fold. One advantage is that any arithmetic, apart from division, that we carry out on integers is exact, with no rounding up or down needed. Division is the exception because fractions are ignored, as line 70 of Fig. 3.19 shows. The other advantage of integers is that they need less memory to store. A program that uses integers will also run much faster than one which uses any other type of variables. The CPC464 allows you to carry out integer division, using the functions \ and MOD (the \ key is next to the right-hand SHIFT). The \ sign means the whole-number result of an integer division. If you type:

PRINT 7\3

for example, you will see the result 2 appear. This is because $7/3$ is 2 and a fraction, and integer division ignores fractions. MOD is used to find the remainder after an integer division. If you type:

PRINT 7 MOD 3

then the result you see this time is 1. This is the *remainder* after 7 has been divided by 3. DIV and MOD come into their own when you get involved in more advanced programming methods that we have space for in this book.

Another action which is specially useful for mathematical work is *defined functions*. This is a way of making use of a mathematical action many times, but writing it just once into your program. Figure 3.20 shows a way of using a defined function which works. It is a very

```
5 DEF FN sum(A,B,C)=A+B+C
10 CLS
20 PRINT"Enter three numbers, please"
30 INPUT A,B,C
40 PRINT"Sum is ";FNsum(A,B,C)
```

Fig. 3.20. Using a very simple defined function.

simple example, and you would never use a defined function for anything so easy, but being easy makes it simpler to follow. Lines 20 and 30 ask you to input three numbers. Line 40 then prints a quantity called FNsum(A,B,C). Now this has no meaning unless it has been defined *earlier in the program*, and it has to be defined by a line that starts with DEF FN. When you enter this line, you have to follow this with the name that you have decided on. The rest of the line shows what quantities the defined function has to work with (its parameters), and what it is supposed to do with them. In this case, it is going to add the numbers that have been assigned to A, B, and C. This has to be completed in one line. You can look on the DEF FN as providing a formula which the machine will look for and use whenever it finds a FN in a program. In this example, the DEF FN has been placed in a line numbered 5, to ensure that it is recognised by the machine before it is needed. If you used DEF FN in a line numbered 100 in this example, you would get a 'Syntax error' message when line 40 ran.

```
10 DEF FNhypot(a,b)=SQR(a^2+b^2)
20 CLS
30 INPUT"Two sides of a right-angled tri
angle, please ";x,y
40 PRINT"The third side is ";FNhypot(x,y
)
```

Fig. 3.21. Another defined function, this time for working out the length of the longest side of a right-angled triangle.

Figure 3.21 shows another example of this useful action. Once again, the definition is given early in the program. FNhypot will find the square root of a↑2+b↑2, but lines 30 and 40 use variables x and y as the two sides of the right-angled triangle. The point is that the DEF FN part tells the computer what to do with a pair of numbers, and it *doesn't matter what they are called.* This makes programming very much easier when you are getting past the beginner stage and starting to flex your muscles a bit!

Tailpiece

At some stage, you may find yourself working with a program in which most of the variables are of one type. You can save a lot of

typing in such a case by using another form of DEF command. If you type, for example, DEFINT A-Z, then all variables that start with any of the letters A to Z will be integer variables, and you won't have to use A%, B%, C% and so on in the program – just A, B, C and so on. You can use DEFSTR to define letters as meaning strings, and DEFREAL to mean real numbers. You can also mix these, using, for example, DEFSTR A-I, O-Z:DEFINT J-N to make all variables that start with the letters J to N integers, and all others strings. This can save a lot of typing of $ and % signs!

Chapter Four

Repeating Yourself

One of the activities for which a computer is particularly well suited is repeating a set of instructions, and every computer is equipped with commands that will cause repetition. The CPC464 is no exception to this rule, and is equipped with more of these 'repeat' commands than is usual for computers in this price range – or even in higher price ranges. We'll start with one of the simplest of these 'repeater' actions, GOTO.

GOTO means exactly what you would expect it to mean – go to another line number. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again. The command word GOTO has to be followed by a space and then a line number; you will receive an error message if the space is omitted.

Figure 4.1 shows an example of a very simple repetition or 'loop', as we call it. Line 10 contains a simple PRINT instruction. When line 10 has been carried out, the program moves on to line 20, which

```
10 PRINT"Amstrad fills your screen!"
20 GOTO 10
```

Fig. 4.1. A very simple loop. You can stop this by pressing the ESC key twice.

instructs it to go back to line 10 again. This is a never-ending loop, and it will cause the screen to fill with the words:

Amstrad fills your screen!

until you press the ESC key to 'break the loop'. Any loop that appears to be running forever can be stopped by pressing the ESC key. This does what it says – stops the program running – but not completely. If

you press any other letter or number key (or space or ENTER), the program will take over from where it left off. We'll see later that this is very useful if you are chasing faults in a program. If you want to stop the program completely, so that you can record it or change it, then you have to press the ESC key again.

Now try a loop in which there is slightly more noticeable activity. Figure 4.2 shows a loop in which a different number is printed out

```
10 CLS:N=10
20 PRINT N
30 N=N+1
40 GOTO 20
50 REM USE ESC ESC TO END
```

Fig. 4.2. A loop which carries out a count-up action very rapidly. You will also have to use the ESC key to stop this one.

each time the computer goes through the actions of the loop. We call this *each pass through the loop*. Line 10 sets the value of the variable N at 10. This is printed in line 20, and then line 30 increments the value of N. Line 40 forms the loop, so that the program will cause a very rapid count-up to appear on the screen. Once again, you'll have to use the ESC key to stop it, and this gives you a chance to see how the program will carry on when you press another key. As before, pressing ESC again will break out of the program.

Now an uncontrolled loop like this is not exactly good to have, and GOTO is a method of creating loops that we prefer not to use! We don't always have an alternative, but the CPC464 offers two, and one of them is the FOR...NEXT loop. As the name suggests, this makes use of two new instruction words, FOR and NEXT. The instructions that are repeated are the instructions that are placed between FOR and NEXT. Figure 4.3 illustrates a very simple example of the

```
10 CLS
20 FOR N=1 TO 10
30 PRINT"Amstrad beats the lot!"
40 NEXT
```

Fig. 4.3. Using the FOR...NEXT loop for a counted number of repetitions.

FOR...NEXT loop in action. The line which contains FOR must also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, N is the counter variable, and its limit numbers are 1 and 10.

The NEXT is in line 40, and so anything between lines 20 and 40 will be repeated.

As it happens, what lies between these lines is simply the PRINT instruction, and the effect of the program will be to print 'Amstrad beats the lot!' ten times. At the first pass through the loop, the value of N is set to 1, and the phrase is printed. When the NEXT instruction is encountered, the computer increments the value of N – from 1 to 2 in this case. It then checks to see if this value exceeds the limit of 10 that has been set. If it doesn't, then line 30 is repeated, and this will continue until the value of N exceeds 10 – we'll look at that point later. The effect in this example is to cause ten repetitions.

You don't have to confine this action to single loops either. Figure 4.4 shows an example of what we call *nested loops*, meaning that one loop is contained completely inside another one. When loops are

```
10 CLS
20 FOR N=1 TO 10
30 PRINT"Count is ";N
40 FOR J=1 TO 1000:NEXT
50 CLS:NEXT
```

Fig. 4.4. A program that uses nested loops, with one loop inside another.

nested in this way, we can describe the loops as inner and outer. The outer loop starts in line 20, using variable N which goes from 1 to 10 in value. Line 30 is part of this outer loop, printing the value that the counter variable N has reached. Line 40, however, is another complete loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J, and we have put nothing between the FOR part and the NEXT part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The last action of the main loop is clearing the screen in line 50. The overall effect, then, is to show a count-up on the screen, slowly enough for you to see the changes, and wiping the screen clear each time. In this example we have used NEXT to indicate the end of each loop. We could use NEXT J in line 40 and NEXT N in line 50 if we liked, but this is not essential. It also has the effect of slowing down the computer slightly, though the effect is not important in this program. When you do use NEXT J and NEXT N, you must be absolutely sure that you have put the correct variable names following each NEXT. If you don't, the computer will stop with a 'NEXT missing' error; meaning that the

NEXTs don't match up with the FORs in this case. You would also get this message if you had omitted a NEXT.

Even at this stage it's possible to see how useful this FOR...NEXT loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

```
FOR N=1 TO 9 STEP 2
```

which would cause the values of N to change in the sequence 1,3,5,7,9. When we don't type STEP, the loop will always use increments of 1.

```
10 CLS
20 FOR N=10 TO 1 STEP -1
30 PRINT N;" seconds and counting"
40 FOR J=1 TO 800:NEXT
50 CLS:NEXT
60 PRINT"Blastoff!!"
```

Fig. 4.5. A count-down program, making use of STEP.

Figure 4.5 illustrates an outer loop which has a step of −1, so that the count is downwards. N starts with a value of 10, and is decremented on each pass through the loop. Line 40 once again forms a time delay so that the count-down takes place at a civilised speed. This is a particularly useful way of slowing the count-down. If we want to speed up the rate, the easiest way is to use an integer variable such as J% in place of J. If we do this, however, we can't use steps that contain fractions, like .1.

Every now and again, when we are using loops, we find that we need to use the value of the counter, such as N or J, after the loop has

```
10 CLS
20 FOR N=1 TO 5
30 PRINT N
40 NEXT
50 PRINT"N is now ";N
60 FOR N=5 TO 1 STEP -1
70 PRINT N
80 NEXT
90 PRINT"N is now ";N
```

Fig. 4.6. Finding the value of the loop variable after a loop action is completed.

finished. It's important to know what this will be, however, and Fig. 4.6 brings it home. This contains two loops, one counting up, the other counting down. At the end of each loop, the value of the counter variable is printed. This reveals that the value of N is 6 in line 50, after completing the FOR N=1 TO 5 loop, and is 0 in line 90 after completing the FOR N=5 TO 1 STEP −1 loop. If you want to make use of the value of N, or whatever variable name you have selected to use, you will have to remember that it will have changed *by one more step* at the end of the loop. You can, of course, use negative values of N in loops. If you use integer variables like N% or J% for speed, however, remember that there are limits to these values. You must not attempt to use an integer greater than 32767 or less than −32768. If you attempt to make the number go outside this range you will get an 'Overflow' error message.

One of the most valuable features of the FOR...NEXT loop, however, is the way in which it can be used with number variables instead of just numbers. Figure 4.7 illustrates this in a simple way.

```
10 CLS
20 A=2:B=5:C=10
30 FOR N=A TO B STEP B/C
40 PRINT N
50 NEXT
```

Fig. 4.7. A loop instruction that is formed with number variables.

The letters A, B and C are assigned as numbers in the usual way in line 20, but they are then used in a FOR...NEXT loop in line 30. The limits are set by A and B, and the step is obtained from an expression, B/C. The rule is that if you have anything that represents a number or can be worked out to give a number, then you can use it in a loop like this.

Loops and decisions

It's time to see loops being used rather than just being demonstrated. A simple application is in totalling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. From what we have done so far, it's easy to see how this could be done if we wanted to use numbers in fixed quantities, like ten numbers in a set. The program of Fig. 4.8 does just this.

```
10 Total=0:CLS
20 PRINT TAB(8)"TOTALLING NUMBERS PROGRA
M"
30 PRINT:PRINT"Enter each number as requ
ested."
40 PRINT"The program will give the total
."
50 FOR N =1 TO 10
60 PRINT"Number";N;" please ";
70 INPUT J:Total=Total + J
80 NEXT
90 PRINT:PRINT"Total is ";Total
```

Fig. 4.8. A number-totalling program for ten numbers.

The program starts by setting a number variable 'Total' to zero. This is the number variable that will be used to hold the total, and it has to start at zero. As it happens, the CPC464 arranges this automatically at the start of a program, but it's a good habit to ensure that everything that has to start with a value actually does. We couldn't, incidentally, use TO for this variable, because TO is a reserved word, part of the FOR... NEXT set of words. You will get a 'Syntax error' message when the program runs if you have used a reserved word as a variable name.

Lines 20 to 40 issue instructions, and the action starts in line 50. This is the start of a FOR... NEXT loop which will repeat the actions of lines 60 and 70 ten times. Line 60 reminds you of how many numbers you have entered by printing the value of N each time, and line 70 allows you to INPUT a number which is then assigned to variable name J. This is then added to the total in the second half of line 70, and the loop then repeats. At the end of the program, the variable total contains the value of the total – the sum of all the numbers that have been entered.

It's all good stuff, but how many times would you want to have just ten numbers? It would be a lot more convenient if we could just stop the action by signalling to the computer in some way, perhaps by entering a value like 0 or 999. A value like this is called a *terminator*; something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of 0 is very convenient, because if it gets added to the total it won't make any difference. We need, however, some way of detecting this terminator, and this is provided by a new instruction word, IF.

IF has to be followed by a condition. You might use conditions like IF N=20, or IF NM$="LASTONE" for this purpose. After the condition, you can use the word THEN, and that has to be followed by what you want to be done, all within the same line. You might simply want the loop to stop when the condition is true. This can be programmed by placing a line number following THEN. If this is the number of the last line in the program, the program will end when the condition is true.

Now if all of that sounds rather complicated, take a look at the simple illustration in Fig. 4.9. We can't use a FOR...NEXT loop

```
10 CLS:PRINT TAB(10)"Another total finde
r"
20 PRINT:PRINT"The program will total nu
mbers for you"
30 PRINT"until you enter a zero."
40 Total=0
50 INPUT"Number, please ";N
60 Total=Total+N
70 PRINT"Total so far is";Total
80 IF N<>0 THEN 50
90 PRINT"END OF TOTALLING"
```

Fig. 4.9. A running-total program which can't use FOR...NEXT. Line 80 carries out the test which decides whether or not to loop.

here because we don't know how many times we will want to go through the loop, so we have used IF...THEN to control the loop. The instructions appear first, and we make the variable total equal to zero in line 40. Each time you type a number, then, in response to the request in line 50, the number that you have entered is added to the total in line 60, and line 70 prints the value of the total so far. Line 80 is the loop controller. IF is used to make the test, and in this case, the test is to find if N is *not* zero. If it's not, then we go back to line 50. The odd-looking sign (<>) that is made by combining the 'less than' and 'greater than' signs, is used to mean 'not equal'. You can put a GOTO following THEN, or leave it out as you please. Since it's just more to type, I have left it out.

The effect, then, is that if the number which you typed in line 50 was not a zero, line 80 will send the program back to line 50 again for another number. This will continue until you enter a zero. When this happens, the test in line 80 fails and the program goes to line 90. This announces the end of the program, and since there are no more lines,

the program stops. If you press ENTER without having typed a number, then the program takes this as equivalent to entering a zero, and stops. Not all machines behave so sensibly!

This example uses just one test with its IF, but you aren't so limited. You could, for example, decide to terminate if either a 0 or 999 was entered. In this case, your IF line could read:

```
IF N<>0 AND N<>999 THEN 50
```

- making two conditions. You can also use the word OR when you make a test, such as:

```
IF X=0 OR X=999 THEN...
```

but you would have to be careful in a number totalling program that 999 did not get added to your total! You have to be careful about where you place some of these tests!

Figure 4.10 illustrates the type of tests that you can perform using IF. These use the mathematical signs for convenience, but remember that all of these signs will have a meaning for strings as well. For the moment, it's easy to see what = and <> might mean, but later on we'll be looking at how < and > can be used.

Sign	Meaning
=	Exact equality
>	Left-hand quantity greater than right-hand quantity
<	Left-hand quantity less than right-hand quantity

The signs can be combined as follows:

<>	Quantities not equal
>=	LHS greater than or equal to RHS
>=	LHS less than or equal to RHS

Note: When the < or > sign is combined with =, then the < or > sign *must* be used first. Using a combination like => will always cause an error message.

Fig. 4.10. The mathematical signs that are used for comparing numbers and number variables.

What ELSE?

IF...THEN forms a test which can be very useful in programs.

There's another extension to IF...THEN, however. You can use the word ELSE to carry out another test and cause a different sort of action. An example makes this a lot clearer, so take a look at Fig. 4.11.

```
10 PRINT TAB(14)"HEADS OR TAILS"
20 PRINT"(Enter E to stop)"
30 N=1+INT(2*RND(2))
40 IF N=1 THEN PRINT"HEADS"ELSE PRINT"TA
ILS"
50 PRINT"Type E to stop"
60 INPUT A$:IF A$="E" THEN END ELSE 30
```

Fig. 4.11. A simple heads-or-tails program, with ELSE used to provide the alternative course of action.

This is a simple heads-or-tails gamble, with no scoring. Lines 10 and 20 set things up as usual, while line 30 starts the main loop of repeated actions. This is the important gambling line. RND means 'select at random', and when it is followed by a positive number in brackets, it means that the machine will pick a fraction at random, lying between 0 and 1. The fraction is never quite zero, however, and never reaches 1. By multiplying this fraction by 2, we'll get a number which can be almost zero, or almost 2, like 1.999999. Taking INT, the integer part, will give a whole number between 0 and 1. If we add 1 to this, we get a number which can be either 1 or 2, and that's what we want for a heads-or-tails choice. The test is made in line 40, so that if N is 1, the word 'HEADS' is printed, and if it's not 1, 'TAILS' is printed. In this example, ELSE is being used to choose the alternative action. ELSE is used again in line 60 to decide whether to end the program or not. Now it's up to you to turn this into a more useful game, asking the user to guess what is coming, and keeping a score!

The importance of ELSE is that you can have an option. If a test succeeds, something can be done (a message printed, perhaps), and with the use of ELSE, another action (a different message, perhaps) can be taken if the test fails. Later on we'll look at how we can program for a larger number of tests. For the moment, it's time to look at another type of loop that allows you a lot more choice in how you terminate it.

WHILE you WEND

Very few home computers offer you any more than a simple

FOR...NEXT loop; some don't even have the ELSE command. The CPC464, however, offers you a very different type of loop – the WHILE...WEND. This can often make it much easier to program a loop, and it avoids the use of GOTO, and even the use of IF...THEN. The principle is that you start your loop with a condition, then you have as many lines as you like of what has to be done in the loop, and finally, the word WEND (from While and END) marks the end of the loop.

```
10 CLS:PRINT TAB(15)"MORE TOTALS":Total=
0:N=1
20 PRINT:PRINT"Enter numbers for totalli
ng, enter 0 to":PRINT" end"
30 WHILE N<>0
40 INPUT N
50 Total=Total+N
60 PRINT"Total so far is";Total
70 WEND
80 PRINT"End of program"
```

Fig. 4.12. A number-totalling program which uses the WHILE...WEND loop.

An example would certainly help, so cast an eye on Fig. 4.12. This is another version of an old friend, the number-totalling program. This time, as well as making Total=0, we have N=1 near the start. This is needed because of the way that a WHILE...WEND loop works, as we'll see. The start of the loop is in line 30, WHILE N<>0. What this means is that the loop will be repeated for as long as N is not zero. When the program starts, however, you will not have input any number N by this stage. This is why a 'dummy' value for N has to be included in line 10 before the loop starts. Without this N=1 step, the program would finish as soon as it got to line 30!

The steps in the loop are familiar, and we needn't go over them again. The important one to note is line 70, WEND. This marks the end of the loop, and will automatically send the loop back to the WHILE test. There's no need for IFs and THENs here, and you can even nest WHILE...WEND loops, as the more complicated example in the manual shows. The only snag is that the test is made right at the start of the loop. You must have a value for whatever is being tested at this stage, or the loop simply won't run.

Take a look at another example, Fig. 4.13, this time using READ...DATA inside the loop. This is a program which simply reads a number of data items from a list until it encounters an 'X'. In

```
10 CLS
20 A$=" "
30 WHILE A$<>"X"
40 PRINT A$
50 READ A$
60 WEND
70 DATA Glenfiddich,Glenmorangie, Laphro
aig
80 DATA Islay Mist,Glenduff,X
```

Fig. 4.13. Another example of WHILE...WEND loop, using READ...DATA to read a list of items.

line 20, the variable A$ is set to a space, obtained by typing a quotemark, then the spacebar, then another quotemark. The loop starts in line 30, with the condition that the loop continues until an 'X' is found as the value of A$. The loop consists of printing the value of A$ (a blank first time round), then reading the DATA list for another value of A$. Line 60 is the WEND for the loop, sending the value of A$ back to line 30 to be tested. The overall effect is that the program prints the list of DATA items. Very tasty!

Figure 4.14 shows yet another use for the WHILE...WEND loop. In this case, it acts as a *mugtrap*. A mugtrap (its polite name is *data*

```
10 INPUT"Type a number 1 to 5";N
20 WHILE N<1 OR N>5
30 PRINT"Unacceptable answer - 1 to 5 on
ly"
40 INPUT N
50 WEND
60 PRINT"YOU PICKED"N
```

Fig. 4.14. Using a WHILE...WEND loop in a mugtrap for a number entry.

validator) is a piece of program that tests what you have entered. If what you have entered is unacceptable, like a number in the wrong range, then the mugtrap refuses to accept the entry, shows by a message on the screen why the entry is unacceptable, and gives you another chance. Mugtraps are very important in programs where a piece of incorrect entry might stop a program with an error message. For an expert (you, after you have finished this book!) this is no problem; a crafty GOTO will get back to the program. For the inexperienced user, the error message seems like the end, and it's

likely that the whole lot will be lost, even if it took all day to enter the data!

In this example, then, you are invited to enter numbers in the range 1 to 5. If the number that you enter is in this range, all is well, but if not (try it!) then the WHILE... WEND loop swings into action. This prints an error message of your own, and gives you another chance to get it right. That's the essence of a good mugtrap, and the WHILE... WEND loop is ideal for forming such traps. Note, despite the emphasis on numbers in some examples, that the WHILE... WEND loop is just as much at home with strings. You can have lines like:

```
WHILE Name$ <>"X"
```

to allow you to keep entering names into a list, or:

```
WHILE AN$<>"Y" AND AN$<>"N"
```

to make a mugtrap for a 'Y' or 'N' answer.

Single key reply

So far, we have been putting in Y or N replies with the use of INPUT, which means pressing the key and then pressing ENTER. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press ENTER. For snappier replies, however, there is an alternative in the form of INKEY$. INKEY$ is an instruction that carries out a check of the keyboard to find if a key is pressed. This checking action is very fast, and normally the only way that we can make use of it is by placing the INKEY$ instruction in a loop which will repeat until a key is pressed. Figure 4.15 shows such a loop used to produce a 'wait until ready' effect. The WHILE... WEND loop is a very simple one

```
10 CLS
20 PRINT"PRESS ANY KEY..."
30 WHILE INKEY$="":WEND
40 PRINT"END"
```

Fig. 4.15. Using INKEY$ in a WHILE...WEND loop to find when a key has been pressed. Some keys will cause the program to operate, but will not print anything on the screen.

which will keep repeating for as long as INKEY$ is a blank. The *blank string* is produced by typing two quotes, with nothing between

them. Each time the computer deals with INKEY$, it searches the keyboard to find if any key is pressed. If none is, then INKEY$ is a blank, and the WHILE...WEND loop keeps looping. When you press a key, however, the loop is broken, and the program moves on. This is useful to have at the end of a set of instructions. The user then has as much time as is needed to read the instructions, and can press any key to start things happening. As usual, 'any key' really means any character key, because several keys, such as SHIFT and CTRL have no effect, and ESC will stop the program.

The INKEY$ instruction will produce a string quantity when any key is pressed, so we can assign INKEY$ to a string variable, K$. In this way, when any key is pressed, the quantity that it represents will be assigned to K$, and we can then test this string as we wish. Figure 4.16 shows INKEY$ being used in this way to get a 'Y' or 'N' answer,

```
10 PRINT"Please type Y or N"
20 K$=INKEY$:IF K$=""THEN 20
30 IF K$<>"N" AND K$<>"Y"THEN PRINT"Inco
rrect answer- Y or N only, please"
40 PRINT"Your answer is ";K$
```

Fig. 4.16. Using INKEY$ to get a 'Y' or 'N' answer.

with a mugtrap incorporated. In line 20, INKEY$ is put in a loop, and is continually being tested. Only when a key is pressed will K$ have a value that is not blank, and the loop will then be broken. The value of K$ is then tested again, to see if the answer is acceptable. A piece of program like this is not so easily put into a WHILE... WEND loop form, and is very clumsy when it is so formed. This simpler version is much better.

There's another, rather different, method of testing for a key being pressed. This uses the command word INKEY, and it's based on the idea that each key can produce a number code. These number codes are shown in the manual, in Appendix III, page 16. The point about using INKEY is that it is not a 'press any key' type of instruction – it is used to detect just one specific key. Figure 4.17 shows this in action,

```
10 PRINT"PRESS SPACEBAR TO CONTINUE"
20 WHILE INKEY(47)<>0:WEND
30 PRINT"CONTINUE HERE.."
```

Fig. 4.17. The INKEY instruction, which can detect a specific key being pressed.

with INKEY(47) being used to detect the spacebar. You can use INKEY(47)=0 to test for the spacebar being pressed, or INKEY(47) =−1 to test for the spacebar *not* being pressed. You can also use numbers that will allow you to detect when the SHIFT or CTRL (or both) keys are pressed along with the one you want. For example, INKEY(47)=32 will test for the spacebar and SHIFT being pressed together, INKEY(47)=128 will test for CTRL and the spacebar together, and INKEY(47)=160 will test for all three keys being pressed.

INKEY is a very useful way of testing for a key, particularly in games, because *any* key can be detected, including the arrowed keys, and the COPY DEL and others. Even the ESC key can be tested in this way!

Chapter Five
Strings and Other Things

String functions

In Chapter 3 we took a fairly brief look at number functions. If numbers turn you on, that's fine, but string functions are in many ways more interesting. What makes them so is that the really eye-catching and fascinating actions that the computer can carry out are so often done using *string functions*. What is a string function, then? As far as we are concerned, a string function is any action that can be carried out with strings. That definition doesn't exactly help you, I know, so let's go into more detail.

A string, as far as the CPC464 is concerned, is a collection of characters which is represented by a *string variable*, a name which ends with the dollar sign. You can pack practically as many characters as you are likely to need into a CPC464 string – a maximum of 255 characters per string, which is a longer string than most of us will ever need. Like other computers, the CPC464 stores its strings in a special way, making use of what is called ASCII code. The letters stand for 'American Standard Code for Information Interchange', and the ASCII (pronounced Askey) code is one that is used by most computers. Don't confuse it with the number codes that are used with INKEY, because the ASCII codes are quite different. Figure 5.1 shows a printout of the ASCII code numbers and the characters that they produce on my printer (Epson RX 80).

Each character is represented by a number, and the range of numbers is from 32 (representing the space) to 127. On the printer, 127 produces a black square, but on the CPC464 screen, you'll see a chequered pattern. You can assign characters to a string variable by using the equality sign. When you assign in this way, you need to use quotes around the characters. You can also assign using INPUT, and using READ...DATA, when no quotes are needed.

Now all number variables are represented in a different type of

32		33	!	34	"
35	#	36	$	37	%
38	&	39	'	40	(
41	)	42	*	43	+
44	,	45	-	46	.
47	/	48	0	49	1
50	2	51	3	52	4
53	5	54	6	55	7
56	8	57	9	58	:
59	;	60	<	61	=
62	>	63	?	64	@
65	A	66	B	67	C
68	D	69	E	70	F
71	G	72	H	73	I
74	J	75	K	76	L
77	M	78	N	79	O
80	P	81	Q	82	R
83	S	84	T	85	U
86	V	87	W	88	X
89	Y	90	Z	91	[
92	\	93	]	94	^
95		96	`	97	a
98	b	99	c	100	d
101	e	102	f	103	g
104	h	105	i	106	j
107	k	108	l	109	m
110	n	111	o	112	p
113	q	114	r	115	s
116	t	117	u	118	v
119	w	120	x	121	y
122	z	123	{	124	¦
125	}	126	~	127	

Fig. 5.1. The standard ASCII code numbers.

coding, one that uses the same number of codes no matter whether the value of the variable is large or small. There's one type of number coding for integers, and another for ordinary (called *real*) numbers. Because a string consists of a set of number codes in the memory of the computer, one code for each character, we can do things with strings that we cannot do with numbers. We can, for example, easily find how many characters are in a string. We can select some characters from a string, or we can change them or insert others. Actions such as these are the actions that we call 'string functions'.

LEN strikes again

One of these string function operations that I mentioned was finding out how many characters are contained in a string. Since a string can contain up to 255 characters, an automatic method of counting them is rather useful, and LEN is that method. LEN has to be followed by the name of the string variable, within brackets, and the result of using LEN is always a number so that we can print it or assign it to a number variable. You don't need to put a space between the N of LEN and the opening bracket.

```
10 CLS
20 A$="You can expand the CPC464"
30 PRINT"There are";LEN(A$);" characters
 in that phrase."
40 INPUT"Try a phrase for yourself ";B$
50 PRINT B$;" has";LEN(B$);" characters.
"
```

Fig. 5.2. Introducing LEN, a member of the string function family.

Figure 5.2 shows a simple example of LEN in use. Line 20 assigns a variable and line 30 tells you how many characters are in this variable. Note the word 'characters', it's not the same as 'letters'. Each space, full stop, comma and so on counts as a character for the purposes of a string, because each one is represented by an ASCII code number. Lines 40 and 50 illustrate how you can find the length of a string which you have entered. This is something that is useful if you want to ensure that a name entered at the keyboard is not too long for the computer to use. You might, for example, have a program that places names in a form, with columns of restricted width, such as fifteen characters. If too long a name is entered, it could spill over into another column. You'll probably notice, if you have a long name or address, that when you get letters that have been computer-addressed, some part of the name or address may have been shortened. Now you know why, and how!

All this is hardly earth-shattering, but we can turn it to very good use, as Fig. 5.3 illustrates. This program uses LEN as part of a routine which will print a string called T$ centred on a line. This is an extremely useful routine to use in your own programs, because its use can save you a lot of tedious counting when you write your programs. The principle is to use LEN to find out how many characters are present in the string T$. This number is subtracted from 42, and the

```
10 CLS
20 T$="The remarkable Amstrad"
30 PRINT TAB((42-LEN(T$))/2);T$
```

Fig. 5.3. Using LEN to print titles centred.

result is then divided by two. If the number of characters in the string is an even number, then the TAB number will contain a .5, but this is completely ignored by TAB when the string is printed. You can, incidentally, use 41 or 42. Whichever one you use, you will find that words are reasonably well centred – 42 works better with phrases which have an even number of characters, and 41 works better with phrases which have an odd number of letters. Yes, you could program that with a few IF...THEN...ELSE steps! We can use a routine of this type to centre anything that has the name T$. In the next chapter, we'll be looking at the idea of *subroutines*, which allow you to type the set of instructions (for centring a title, for example) just once, and then use them for any string that you like.

STR$ and VAL

You know by now that there are some operations that you can carry out on numbers but not on strings, and some which you can carry out on strings but not on numbers. This might be inconvenient, but as it happens, we can convert numbers from one form into another quite easily. This allows us to perform arithmetic on a number that has been in string form, and to use string functions on a number that was formerly only in number form. Take a look at Fig. 5.4. To start with,

```
10 N$="22.5":V=2
20 CLS:PRINT
30 PRINT N$;" times";V;" is";V*VAL(N$)
40 PRINT
50 V$=STR$(V)
60 PRINT"There are";LEN(V$);" characters
 in";V" !"
70 PRINT
80 PRINT N$;" added to";V$;" gives ";N$+
V$;" ?"
```

Fig. 5.4. Converting numbers to and from string form, using STR$ and VAL. Note the space that STR$ puts in.

we make N$ a number in string form, and V a number in number form. Line 30 then shows how we can carry out arithmetic with N$. By typing VAL(N$) in place of N$ alone, the number value of N$ is used in the calculation, and the correct result is obtained. In line 50, number V is transformed into a string, V$. There's a warning here, however, as line 60 shows. Number V was equal to '2', a single digit number. When STR$ has been used to convert to string form, however, the number of characters is increased mysteriously to two. This is because of that invisible space which is put in to allow for a + or − sign. The STR$ routine always includes the space, so that the length of a string which has been obtained from a number is always one character too many unless there has been a − sign in the number. Line 80 is there to remind you of what can happen if you forget about VAL and try to add two strings!

By the left, slice!

The next group of string operations that we're going to look at is *slicing operations*. The result of slicing a string is another string; a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string.

All of that might not sound terribly interesting, so take a look at Fig. 5.5. The string A$ is assigned in line 20, and another string is

```
10 CLS
20 A$="Amstrad"
30 B$="ateur computing with a profession
al machine."
40 PRINT LEFT$(A$,2)+B$
```

Fig. 5.5. Using the string slicing action LEFT$.

assigned in line 30. What's printed in line 40 is a phrase which uses the first two letters of 'Amstrad'. Now how did this happen? The instruction LEFT$ means 'copy part of a string starting at the left-hand side'. LEFT$ has to be followed by two quantities, within brackets and separated by a comma. The first of these is the variable name for the string that we want to slice, A$ in this example. The second is the number of characters that you want to slice (copy, in fact) from the left-hand side. The effect of LEFT$(A$,2) is therefore to copy the first two letters from Amstrad, giving Am. The other

string in line 30 is then added to this, so giving us the phrase which is printed on the screen in line 40.

For a more serious use of this instruction, take a look at Fig. 5.6.

```
10 CLS:PRINT:PRINT
20 INPUT"Your surname, please ";SN$
30 PRINT:INPUT"Your first name, please "
;FM$
40 PRINT:PRINT
50 PRINT"You'll be known as ";LEFT$(FM$,
1)+"."+LEFT$(SN$,1)+". round here."
```

Fig. 5.6. Extracting initials with LEFT$ string slicing.

This has the effect of extracting your initials from your name, and it's done by using LEFT$ along with a bit of concatenation. The INPUT steps in lines 20 and 30 find your surname and first name, and assign them to variable names SN$ and FM$. We can't use the more obvious FN$ for forename, because FN is a reserved word in BASIC, and you may not use reserved words as variable names. You will get a 'Syntax error' report if you try to do this. Line 50 then prints your initials by using LEFT$ to extract the first letter of each string. The letters are then assembled along with full-stops, using concatenation in line 50. If you have two players in a game, it's often useful to show the initials and score rather than print the full name, but the full names can be held stored for use at various stages in the game.

All right, Jack?

String slicing isn't confined to copying a selected piece of the left-hand side of a string. We can also take a copy of characters from the right-hand side of a string. This particular facility isn't used quite so much as the LEFT$ one, but it's useful none the less. Fig. 5.7 illustrates

```
10 CLS
20 A$="Amstrad magic"
30 PRINT:PRINT
40 PRINT"It's all ";RIGHT$(A$,5);" to me
"
```

Fig. 5.7. Using RIGHT$ to extract letters from the right-hand side of a string.

the use of the instructions to avoid having to type a word over again. There are more serious uses than this. You can, for example, extract

the last four figures from a string of numbers like 010-242-7016. I said a *string* of numbers deliberately, because something like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable, you'll get a silly answer. Why? Because when you type N = 010−242−7016 then the computer assumes that you want to subtract 242 from 10 and 7016 from that result. The value for N is −7248, which is not exactly what you had in mind! If you use N$="010-242-7016" then all is well.

We can get quite a lot of interesting effects from LEFT$ and RIGHT$. Take a look at Fig. 5.8 for an example, which does odd

```
10 CLS
20 INPUT"Your name, please";A$
30 N=LEN(A$)
40 FOR J=1 TO N
50 PRINT LEFT$(A$,J);TAB(21)RIGHT$(A$,J)
60 NEXT
```

Fig. 5.8. Slicing both left and right sides of a string.

things with the letters of your name. The program prompts you to enter your name in line 20, and the name is assigned to A$. In line 30, we use LEN so that the number variable N contains the total number of characters in your name. This will include spaces and hyphens – nobody's likely to use asterisks and hashmarks! Line 40 starts a loop which uses the total number of characters as its end limit. Line 50 is the action line. When J is 1, line 50 prints the first letter on the left of your name on to the left-hand side of the screen, and the first letter on the right of your name on the right-hand side. On the next pass through the loop, a new line is selected, and two letters are printed. This continues until the entire name is printed. If you use a LEFT$ or RIGHT$ with a number that is more than the number of letters in the string, then you simply get the whole string.

Middle earth?

There's another string slicing instruction which is capable of much more than either LEFT$ or RIGHT$. The instruction word is MID$, and it has to be followed by three items, within brackets, and using commas to separate the items. Item 1 is the name of the string that you want to slice, as you might expect by now. The second item is a number which specifies where you start slicing. This number is the

number of the characters counted from the left-hand side of the string, and counting the first character as 1. The third item is another number, the number of characters that you want to slice, going from left to right and starting at the position that was specified by the first number.

```
10 CLS
20 A$="Amstrad CPC464"
30 L=LEN(A$)
40 FOR N=1 TO L
50 PRINT MID$(A$,N,1);" ";:NEXT
60 PRINT:PRINT
70 FOR N=1 TO L
80 PRINT MID$(A$,N,1)+"+";:NEXT
```

Fig. 5.9. Using MID$. This can extract from any part of a string, and can, like LEFT$ and RIGHT$, be controlled by variables.

It's a lot easier to see in action than to describe, so try the program in Fig. 5.9. Line 20 assigns A$ to Amstrad CPC464, and line 30 finds L, the number of characters in this phrase. The loop that starts in line 40 then prints letters taken from the word phrase. With the value of N equal to 1, the letter that is sliced is A, because its position in the word is 1, and we're copying one letter from this position. If we used MID$(A$,1,2), we would get Am, and if we used MID$(A$,3,2) we would get st. As it is, we select a letter at a time, and print a space. The semicolon in line 50 then ensures that the next sliced letter is printed on the same line. The net effect is that the letters are printed spaced out. The second loop in lines 70 and 80 performs the same kind of effect, but places a + sign between the letters rather than a space.

One of the features of all of these string slicing instructions is that we can use variable names or expressions in place of numbers. Figure 5.10 shows a more elaborate piece of slicing which uses expressions.

```
10 CLS
20 INPUT"Your name, please ";NM$
30 L=LEN(NM$):C=INT(L/2)+1
40 FOR N=1 TO C
50 PRINT TAB(21-N)MID$(NM$,C-N+1,N*2-1)
60 NEXT
```

Fig. 5.10. Making a letter pyramid to show the action of MID$ with a formula.

It all starts innocently enough in line 20 with a request for your name. Whatever you type is assigned to variable NM$, and in line 30 a bit of mathematical juggling is carried out. How does it work? Suppose you type DONALD as your name. This has six letters, so in line 30 L is assigned to 6, and C is the whole number part of L/2 (equal to 3), plus 1, making 4. Line 40 then starts a loop of 4 passes. In the first pass you print at TAB(20) (because N=1 and 21−N is 20) the MID$ of the name using C−N+1, which is 4−1+1=4, and N*2−1, which is also 1. What you print is therefore MID$(NM$,4,1), which is A in this example. On the next run through the loop, N is 2, C−N+1 is 3, and N*2−1 is also 3. What is printed is MID$(NM$,3,3), which is NAL. The loop goes on in this way, and the result is that you see on the screen a pyramid of letters formed from your name. It's quite impressive if you have a long name! If your name is short, try making up a longer one!

More priceless characters

It's time now to look at some other types of string functions. If you look back a few pages, you'll remember that we introduced the idea of ASCII code. This is the number code that is used to represent each of the characters that we can print on the screen. We can find out the code for any letter by using the function ASC, which is followed, within brackets, by a string character in quotes or a string variable (no quotes). The result of ASC is a number, the ASCII code number for that character. If you use ASC("CPC464"), then you'll get the code for the C only, because the action of ASC includes rejecting more than one character. Figure 5.11 shows this in action. String

```
10 A$="CPC464 Computing"
20 CLS:PRINT
30 FOR N=1 TO LEN(A$)
40 PRINT ASC(MID$(A$,N,1));" ";
50 NEXT
```

Fig. 5.11. Using ASC to find the ASCII code for letters.

variable A$ is assigned in line 10 and in line 30 a loop starts which will run through all the letters in A$. The letters are picked out one by one, using MID$(A$,N,1), and the ASCII code for each letter is found with ASC. Watch how the brackets have been used! The space between quotes, along with the semicolons in line 40 makes sure that

the codes are all printed with a space between the numbers, and without taking a new line for each number. Simple, really.

ASC has an opposite function, CHR$. What follows CHR$, within brackets, has to be a code number, and the result is the character whose code number is given. The instruction PRINT CHR$(65), for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for coding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 5.12 illustrates this use. Line 40 is a WHILE...INKEY$...WEND loop to make the program wait for

```
10 CLS:PRINT
20 PRINT"What's the code for computing s
uccess?"
30 PRINT"Press any key to reveal the sec
ret"
40 WHILE INKEY$="":WEND
50 FOR J%=1 TO 6
60 READ D%:PRINT CHR$(D%);
70 NEXT J%
100 DATA 67,80,67,52,54,52
```

Fig. 5.12. Using ASCII codes to carry a coded message, and then using CHR$ to obtain the character that corresponds to a code number.

you. When you press a key, the loop that starts in line 50 prints 6 characters on the screen. Each of these is read as an ASCII code from a list, using a READ...DATA instruction in the loop. The PRINT CHR$(D%) in line 60 then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it! If you wanted to conceal the letters more thoroughly, you culd use quantities like one quarter of each code number, or 5 times each code less 20, or anything else you like. These changed codes could be stored in the list, and the conversation back to ASCII codes made in the program. This will deter all but really persistent decoders! This example, incidentally, illustrates the use of READ and DATA in a loop. We would normally use READ and DATA only for information that we particularly wanted to keep stored in a program like this. Another feature is the use of integer variables such as J% and D%. This takes less memory space, and using J% in the loop makes it operate significantly faster.

There's another instruction which belongs with READ and DATA, while we are on the subject. This one is RESTORE. When you use RESTORE by itself, it means that the DATA pointer is to be reset. For example, if you have just read six items from a DATA line that holds only six, you can't have another READ, because there is nothing more to read. If you use RESTORE just before another READ, however, you will read the first item again. There's another twist to the use of RESTORE, however. RESTORE followed by a line number means that the DATA list will start again from the beginning *of that line*. You must make sure, of course, that the line number which you have chosen *is* a data line! Take a look at Fig. 5.13. This offers a choice of data to be read by making use of

```
10 CLS
20 PRINT"Which list do you want?"
30 INPUT" 1,2 OR 3 PLEASE";N
40 IF N<1 OR N>3 THEN PRINT"Mistake, ple
ase try again":GOTO30
50 IF N=1 THEN RESTORE 140
60 IF N=2 THEN RESTORE 150
70 IF N=3 THEN RESTORE 160
80 A$=""
90 WHILE Q$<>"X"
100 PRINT Q$;" ";
110 READ Q$
120 WEND
130 END
140 DATA  Opel, Mercedes, Porsche,X
150 DATA Renault,Peugeot,Citroen,X
160 DATA  Fiat, Alfa-Romeo,Lancia,X
```

Fig. 5.13. How RESTORE can be used to select different DATA lines.

RESTORE along with a number. When you pick a number, it is used in lines 50 to 70 to carry out a RESTORE command. It's unfortunate that the RESTORE command does not allow an expression to be used. If it did, we could program this more tidily, such as RESTORE 1000*N. Each DATA line contains three items ending with X, and the loop which reads the DATA is arranged so that it will stop when the X is read. We could have used a FOR...NEXT loop for reading, since each line contains the same number of items. By using this method, however, we allow any number of items to be used in each list, and unequal numbers as well, which is much more useful.

RESTORE, used in this way is a useful method of choosing from a number of lists which will be selected each time the program is used.

The law about order

We saw earlier in Fig. 4.10 how numbers can be compared. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign, =, means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 65. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, in order to compare complete words, character by character. Figure 5.14 illustrates this use of comparison using the = and > symbols. Line 20

```
10 CLS
20 A$="QWERTY"
30 PRINT:INPUT"Type a word, using capita
ls ";B$
40 IF A$=B$ THEN PRINT"SAME AS MINE!":EN
D
50 IF A$>B$ THEN Q$=A$:A$=B$:B$=Q$
60 PRINT"Correct order is ";A$;" then ";
B$
70 END
```

Fig. 5.14. Comparing words to decide on their alphabetical order.

assigns a nonsense word – it's just the first six letters on the top row of letter keys. Line 30 then asks you to type a word. The comparisons are then carried out in lines 40 and 50. If the word that you have typed, which is assigned to B$, is *identical* to QWERTY, then the message in line 40 is printed and the program ends. If QWERTY would come later in an index than your word, then line 50 is carried out. If, for example, you typed PERIPHERAL, then since Q comes after P in the alphabet and has an ASCII code that is greater than the code for P, your word B$ scores lower than A$, and line 50 swaps

them round. This is done by assigning a new string, Q$ to A$ (so that Q$ = "QWERTY"), then assigning A$ to B$ (so A$ = "PERIPHERAL"), then B$ to Q$ (so that B$="QWERTY"). Line 60 will then print the words in the order A$ and then B$, which will be the correct alphabetical order. If the word that you typed comes later than QWERTY – for example, TAPE – then A$ is not 'greater than' B$, and the test in line 50 fails. No swap is made, and the order A$, then B$, is still correct. Note the important point, though, that words like QWERTZ and QWERTX will be put correctly into order – it's not just the first letter that counts.

Put it on the list

The variable names that we have used so far are useful, but there's a limit to their usefulness. Suppose, for example, that you had a program that allowed you to type in a large set of numbers. How would you go about assigning a different variable name to each item? Figure 5.15 illustrates this. Lines 10 to 40 generate an (imaginary) set

```
10 CLS
20 FOR N=1 TO 10
30 A(N)=1+INT(RND(1)*100)
40 NEXT
50 PRINT
60 PRINT TAB(16)"MARKS LIST"
70 PRINT:FOR N=1 TO 10
80 PRINT TAB(2)"Item ";N;" received";A(N
);" marks."
90 NEXT
```

Fig. 5.15. An array of subscripted number variables. It's simpler than the name suggests!

of examination marks. This is done simply to avoid the hard work of entering the real thing! The variable in line 30 is something new, though. It's called a *subscripted variable*, and the 'subscript' is the number that is represented by N. The name that we use has nothing to do with computing; it's a name that was used long before computers were around. How often do you make a list with the items numbered 1,2,3 and so on? These numbers 1,2,3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names A(1), A(2), A(3) and so on, we can identify different items that have the common variable name of A. A member

of this group like A(2) has its name pronounced as 'A-of-two'.

The usefulness of this method is that it allows us to use one single variable name for the complete list, picking out items simply by their identity numbers. Since the number can be a number variable or an expression, this allows us to work with any item of the list. Figure 5.15 shows the list being constructed from the FOR...NEXT loop in lines 20 to 40. Each item is obtained by finding a random number between 1 and 100, and is then assigned to A(N). Ten of these 'marks' are assigned in this way, and then lines 60 to 90 print the list. It makes for much neater programming than if you needed a separate variable name for each number.

So far, so good, but one point has been omitted so far. Try altering the loop, so that it reads FOR N=1 TO 11, and then run the program. You'll get an error message that says 'Subscript out of range in 30'. The computer is prepared for the use of subscript numbers of up to 10, but no higher. The computer has to be prepared for the use of numbers greater than 10 – the preparation consists of getting some more memory ready to receive the data. When you use DIM (meaning dimension), the memory is allocated for the array. A line such as DIM A(11) actually allows you up to *twelve* items in an array, in fact, because we can use A(0) if we like, but you must not attempt to use A(12) or any higher number. You will get the error message again if you do so.

The important DIM instruction, then, consists of naming each variable that you will use for arrays, and following the name with the

```
10 CLS:DIM A(12),N$(12)
20 PRINT TAB(2)"Please enter names and m
arks."
30 FOR N=1 TO 12
40 INPUT"Name ";N$(N)
50 INPUT"Mark ";A(N)
60 NEXT
70 CLS:Total=0
80 PRINT TAB(16)"MARKS LIST":PRINT
90 FOR N=1 TO 12
100 PRINT TAB(2);N$(N);TAB(22);A(N)
110 Total=Total+A(N)
120 NEXT
130 PRINT
140 PRINT"Average is ";Total/12
```

Fig. 5.16. Using strings in one array, and numbers in another. The arrays have been dimensioned this time.

maximum number, within brackets, that you expect to use. You aren't forced to use this number, but you must not exceed it. If you do, and your program stops with an error message, you will have to change the DIM instruction and start again – which would be tough luck if you were typing in a list of 100 names! Note that you can dimension more than one variable in a DIM line, as Fig. 5.16 shows. Even though you don't have to use DIM when you use numbers of 10 or less, it's a good habit to do so. The reason is that it avoids wasting memory space, by making the most efficient use of the memory.

Figure 5.16 extends this use of array variables a step further. This time you are invited to type a name and a mark for each of twelve items. When the list is complete, the screen is cleared and a variable called Total is set to zero in line 70. The list is then printed neatly, and on each pass through the loop the total is counted up (in line 110) so that the average value can be printed at the end. The important point here is that it's not just numbers that we can keep in this list form. The correct name for the list is an *array*, and Fig. 5.16 uses both a string array (names) and a number array (marks).

Rows and columns

You can imagine an array as a list of items, one after the other, but there is a variety of array which allows a different kind of list, called a *matrix*. A matrix is a list of groups or items, with all the items in a group related. We could think of a matrix as a set of rows and columns, with each group taking up a row, and the items of a group in separate columns. Take a look at Fig. 5.17 to see how this works. We use here a variable N$ which has two subscript numbers. The first number is the row number and the second is the column number. We need two FOR...NEXT loops to read data into this matrix. This is

```
10 CLS:DIM N$(3,2)
20 FOR N=1 TO 3
30 FOR J=1 TO 2
40 READ N$(N,J)
50 NEXT J:NEXT N
60 FOR N=1 TO 3
70 PRINT TAB(5);N$(N,1);TAB(20);N$(N,2)
80 NEXT
100 DATA Horse,Foal,Cow,Calf,Dog,Puppy
```

Fig. 5.17. Making a matrix of rows and columns.

carried out in lines 20 to 50. The items are then printed in columns by the loop in lines 60 to 80. In this loop, the variable N is used as the row number and we use the column numbers 1 and 2. The rows contain animal names, and the columns separate the different names that we use for adult and for young animals respectively. This example has used a *string matrix*, but a number matrix is also possible. If your maths has progressed to A level or beyond, you probably know a variety of uses for number matrices, particularly in the solution of simultaneous equations.

Figure 5.18 shows a rather more ambitious matrix program. The idea is to store sets of names and telephone numbers which are fed in by you in the course of the loop in lines 20 to 50. Once the matrix has

```
10 CLS:DIM A$(50,2)
20 FOR N=1 TO 50
30 LOCATE 2,5:INPUT "Name ";A$(N,1)
40 LOCATE 2,7:INPUT "Tel. No. ";A$(N,2)
50 CLS:NEXT
60 CLS:LOCATE 14,1:PRINT"LIST COMPLETE"
70 LOCATE 10,4:INPUT"Pick an initial let
ter ";J$
80 FOR N=1 TO 50
90 IF J$=LEFT$(A$(N,1),1)THEN PRINT"Name
 ";A$(N,1);TAB(20)"Number ";A$(N,2)
100 NEXT
```

Fig. 5.18. Using a name and number matrix for a simple telephone directory application.

been filled, you can pick an initial letter for a name, and ask the computer to print out the name and number that it has located. I've left out mugtraps just to keep this example reasonably short, but you would certainly need some sort of mugtrap, even if only in the form of a message like:

```
PRINT" SORRY, CAN'T FIND ";J$;" ENTRIES"
```

Normally, having set up something like this, you would want to use it more than once. A loop is needed, so that once you have found one name, you can go back to line 70 to pick another. You might also want to think about a suitable way of dealing with it when the name is not found. This will be when the program has completed line 100 without ever having gone past the IF part of line 90. There will be clues about this later!

One rather useful and unusual string function is INSTR. This is used to find if one string is contained in another. It's used in the form:

X= INSTR (A$,B$)

to find if B$ is contained in A$. If it is, then X is the position number of the first letter of B$ that is found in A$. If B$ is *not* contained in A$, then X is zero. X will always be zero if B$ is longer than A$. You can, of course, use the form:

PRINT INSTR(A$,B$)

if you just want to see the number.

Figure 5.19 shows a simple example of this function in action. Lines 20 to 40 allocate names to strings, and lines 50 to 70 make the tests, so that you can see how they work out. Notice that the strings have to be exact for the function to work – it's no good looking for 'Bert' if what is contained in the string is 'bert' or 'BERT', for example. To leave you with a thought, suppose you had a string A$="YESyesYUPyupSUREsureOKok", and you asked for a yes/no answer. You could get INSTR to look through this. If the result of X=INSTR(Answer$,A$) is zero, then the answer wasn't any form of YES!

```
10 CLS
20 A$="Albert Hall"
30 B$="Richardson, Bertram"
40 C$="Sinclair, I"
50 PRINT"In A$, bert is located at";INST
R(A$,"bert")
60 PRINT"In B$,Bert is located at";INSTR
(B$,"Bert")
70 PRINT"In C$,BERT is located at";INSTR
(C$,"BERT")
```

Fig. 5.19. How the INSTR instruction is used.

Chapter Six

Menus, Subroutines and Programs

Figure 4.16 introduced the idea of making a choice, by pressing a key. In that example, the choice of keys was limited, Y or N. A choice of two items, isn't exactly generous, and we can extend the choice by a program routine that is called a *menu*. A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause a section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. Figure 6.1 shows what a typical menu of

---

MENU

1. Make a new list of names.
2. Add names to list.
3. Read names from list.
4. Delete names from list.
5. Select one name.
6. End program.

PLEASE CHOOSE BY NUMBER

Fig. 6.1. A typical menu as it would appear on the screen.

this type would look like on the screen. With a computer which was fitted with Stone Age BASIC, we could use a set of lines such as:

```
IF K =1 THEN 1000
IF K =2 THEN 2000
```

and so on. There is a much simpler method, however, which uses new CPC464 instruction words, ON...GOTO or ON...GOSUB. These represent two different ways of carrying out the same task, and we'll look at both closely, because this command is not available in some other versions of BASIC that you might have used.

To start with, suppose we want to pick from four items by typing

numbers 1 to 4 on the keyboard. The first thing that you have to ensure is that only the numbers 1 to 4 are accepted, not numbers like 0, 5, −10 and so on – in other words, a bit of mugtrapping. The second point is that the use of INPUT is a bit tedious, because you have to press ENTER after you have typed your number key. We'll use INKEY$, then, to get the reply (the number key), and it only remains to check that you have chosen a number that is in the correct range. Following this check, we'll use the number that was entered to find a line number and run the piece of program that starts at this line.

In the example of Fig. 6.2, lines 10 to 60 print a title and a menu, using TABs to make the printing reasonably neat. This is followed by

```
10 CLS:PRINT TAB(19)"MENU"
20 PRINT TAB(2)"1. Daily takings."
30 PRINT TAB(2)"2  Stock situation."
40 PRINT TAB(2)"3. Purchases."
50 PRINT TAB(2)"4. Summary."
60 PRINT:PRINT TAB(4)"Please select by n
umber 1-4."
70 K$=INKEY$:IF K$=""THEN 70 ELSE K=VAL(
K$)
80 IF K>4 OR K<1 THEN PRINT"Incorrect ra
nge- 1-4 only- please try again.":GOTO 7
0
90 ON K GOTO 110,130,150,170
100 END
110 PRINT"This is routine No. 1"
120 GOTO 100
130 PRINT"This is routine No. 2"
140 GOTO 100
150 PRINT"This is routine No. 3"
160 GOTO 100
170 PRINT"This is routine No. 4"
180 GOTO 100
```

Fig. 6.2. A menu choice which uses the ON K GOTO instruction.

advice on how to choose (it might be obvious to you, but not to any other user!). The action starts in lines 70 and 80. In line 70, there is an INKEY$ loop, which will keep looping while no key is pressed. If a key *is* pressed, however, K$ will have a value which is a string, and the ELSE part of the line converts this into a number. Line 80 tests this number, to make sure that it is in the range of 1 to 4 that comprises our menu. If the number is not in the correct range, you get a polite

message and a reminder of what the correct range is. If a letter key was pressed, incidentally, the K=VAL(K$) step will convert the letter string into the number 0, and this will be rejected by line 80. Never put a number choice of zero into a menu, because it's then a lot harder to distinguish between letter and number!

At line 90, with the value of K in the correct range, the choice is made by the statement on K GOTO 110, 130, 150, 170. This list *must* be in the order that will serve the menu choices. In other words, the routine which starts at line 110 should attend to the needs of menu item 1, the line which starts at 130 should attend to the needs of menu item 2, and so on. These lines need not be numbered 110, 130, 150, 170, of course. The numbering might have been 7000, 600, 150, 2010 *provided that these were the correct starting lines* for the routines. In this example, each 'routine' is just a printed message, and it is followed by a GOTO 60. This makes sure that the program ends after each routine. If we had used GOTO 10, we would have made the program return to the menu. This is very often what is needed. When the program always returns to the menu, one of the menu choices should be 'End this program', so that you can stop without having to switch off the machine.

Sectional programming

Many programs consist of a title, some instructions, and then a menu. Depending on what menu choice you make, some part of the program is run, and the program ends, or returns to the menu. The ON K GOTO type of menu selection is useful, but an even more useful method makes use of *subroutines*. A subroutine is a section of program which can be inserted anywhere you like in a longer program. A subroutine is inserted by typing the instruction word GOSUB, followed by the line number in which the subroutine starts. When your program comes to this instruction, it will jump to the line number that follows GOSUB, just as if you had used GOTO. Unlike GOTO, however, GOSUB offers an *automatic* return. The word RETURN is used at the end of the subroutine lines, and it will cause the program to return to the point immediately following the GOSUB. Figure 6.3 illustrates this. When the program runs, line 20 prints a phrase, with the semicolon used to prevent a new line from being selected. The GOSUB 1000 in line 30 then causes the word 'Yellow' to be printed, but the RETURN in line 1010 will send the program back to line 40, the instruction that immediately follows the

```
10 CLS
20 PRINT"This is a ";
30 GOSUB 1000
40 PRINT"subroutine":PRINT:PRINT
50 PRINT"Red light and Green light make
";:GOSUB 1000:PRINT"light."
60 PRINT:PRINT:END
1000 PRINT"Yellow ";
1010 RETURN
```

Fig. 6.3. Using a subroutine - this is the key to more advanced programming.

GOSUB 1000. This type of action will also occur even when the GOSUB is part of a multistatement line, as line 50 demonstrates. The GOSUB 1000 will cause the word 'Yellow' to be printed, but the return is to the PRINT instruction that follows GOSUB 1000 in line 50; it doesn't jump to line 60. This example is, of course, a yellow subroutine!

Now for something more serious. Figure 6.4 shows subroutines in use as part of an (imaginary) games program. Lines 10 to 80 offer a choice, and line 90 invites you to choose. The familiar INKEY$ and mugtrap actions follow, and then line 120 causes the choice to be carried out. This time, however, the program will return to whatever follows the choice. For example, if you pressed key 1, then the subroutine that starts at line 1000 is carried out, and the program returns to line 120 to check if you might also want subroutines 2000, 3000, 4000, or 5000. Since the value of K is still 1, the program then goes to line 130 to see if you want to return to the menu or end it. If line 1000 had altered the value of K to 2, 3, 4, or 5, however, you could find that a second subroutine was selected following the first one. If this is likely to happen (you shouldn't use K in such subroutines, should you?), then each subroutine should end with K=0, or some other way of preventing unwanted selection.

A subroutine is extremely useful in menu choices, but it's even more useful for pieces of program that will be used several times in a program. In Fig. 6.4, by way of an example, the INKEY$ routine has been written as a subroutine because it's one that you are likely to use many times in the course of any program. Putting the INKEY$ into a subroutine means that you need to type these program lines once only. Wherever you need the action, you simply type GOSUB10000 (or whatever line number you have used), and the routine will be inserted when the program runs. Notice that in each case, the subroutine is placed in lines that can't normally be RUN. If you

```
10 CLS:PRINT
20 PRINT TAB(11)"Choose your monster."
30 PRINT
40 PRINT TAB(2)"1. Vampire."
50 PRINT TAB(2)"2. Werewolf."
60 PRINT TAB(2)"3. Zombie."
70 PRINT TAB(2)"4. Sgt. Major."
80 PRINT TAB(2)"5. Flying picket."
90 PRINT:PRINT"Select by number, please"
:PRINT:PRINT
100 GOSUB 10000: REM inkey$ routine.
110 IF K<1 OR K>5 THEN PRINT"Faulty sele
ction - 1 to 5 only-":PRINT"Please try a
gain.":GOTO 100
120 ON K GOSUB 1000,2000,3000,4000,5000
130 PRINT:PRINT"Want another choice? Typ
e Y or N."
140 GOSUB 10000:IF K$="Y" THEN 10
150 END
1000 PRINT"Blood, blood, bootiful blood"
:RETURN
2000 PRINT"Howl, snarl, gnash":RETURN
3000 PRINT"I obey, master, I obey.":RETU
RN
4000 PRINT"You 'orrible little man, you":
RETURN
5000 PRINT"Blood, howl, I obey, smash.":
RETURN
10000 K$=INKEY$:IF K$=""THEN 10000 ELSE
K=VAL(K$)
10010 RETURN
```

Fig. 6.4. A menu choice that makes use of subroutines.

removed the GOSUB instructions from these programs, the subroutines couldn't run, because there is an END or STOP line before the computer could naturally get to the subroutine. This is important, because if the computer gets to a subroutine line by accident (when GOSUB has not been used), the program will stop with an error message 'Unexpected RETURN in 1010' (using whatever line number the RETURN is in). Getting into a subroutine the wrong way is called *crashing through*, and you must avoid it by placing an END at the end of your program, before the subroutine lines start.

Figure 6.5 shows an elaboration on the INKEY$ subroutine. The trouble with INKEY$ is that it doesn't remind you that it's in use – there's no question mark printed as there is when you use INPUT.

```
10 CLS
20 PRINT"Choose 1 or 2, please"
30 GOSUB 1000
40 PRINT"Your choice was ";K$
50 END
1000 K$=INKEY$
1010 IF K$<>""THEN RETURN
1020 PRINT"*";:GOSUB 2000
1030 PRINT CHR$(8);CHR$(16);:GOSUB 2000
1040 GOTO 1000
2000 FOR J=1 TO 200:NEXT:RETURN
```

Fig. 6.5. A flashing-asterisk subroutine. The asterisk flashes until you press a key.

The subroutine in lines 1000 to 1040 remedies that by causing an asterisk to flash while you are thinking about which key to press. The asterisk is flashed by alternately printing the asterisk and the delete step. CHR$(8) causes the cursor to move one step back, and CHR$(16) wipes out the character at the cursor position. To make the rate of flashing reasonably slow, I've added another subroutine, a delay in line 2000. Try this pair of subroutines in your programs, and see what a difference they make. Meantime, make friends with subroutines. They are not just a useful way of obtaining an action at several points in a program; they are an indispensable aid to program planning, of which there's much more just about to come.

Rolling your own

You can obtain a lot of enjoyment from your CPC464 computer when you use it to enter programs from cassettes that you have bought, or from plug-in cartridges. You can obtain even more enjoyment from typing in programs that you have seen printed in magazines. Even more rewarding is modifying one of these programs so that it behaves in a rather different way; making it do what suits you. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's one

hundred per cent your own work, and you'll enjoy it all the more for that. After all, buying a computer and not programming it yourself is like buying a Ferrari and getting someone else to drive it for you.

Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of vintage steam locomotives. Programs of this type are called *database* programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in games, colour patterns, drawings, sound, or other programs that require shapes to move across the screen. Programs of that type need instructions that we shall look at in quite a lot of detail in Chapters 8 to 10. What we are going to look at in this section is the database type of program, because it's designed in a way that can be used for all types of programs. Once you can design simple programs of this type, you can progress, using the same methods, to designing your own graphics and sound programs. Remember, though, that most of the very fast moving or elaborate graphics programs that you see are not written in BASIC. The reason is that BASIC is too slow-acting to allow fast movement or the control of lots of moving objects. These arcade-type programs that you can buy are written in *machine code*, a set of number-coded instructions issued direct to the microprocessor that is the heart of the computer. This bypasses BASIC altogether, and is very much more difficult. Many of the machine code programs are, in fact, written with the aid of other programs on much bigger computers, and then loaded into the home computers. If you learn how to design programs in BASIC, however, you will be able to learn machine code later. All you need is experience – a lot of it.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works after a bit of thought and modification, than from an elaborate program that never seems to do what it should. The second point is that program design has to start with the computer switched off, and preferably in another room! The reason is that program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!

Put it on paper

We start, then, with a pad of paper. I use an A4 'student's pad' which is hole punched so that I can put sheets into a file. In this way I can keep the sheets tidy and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Yes, sheets! Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a very simple program, but it will illustrate all the skills that you need.

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to do, it's odds on that the program will never do it! The reason is that you get so involved in details when you start writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time over it, and consider what you want the program to be able to do. If you don't know, you can't program it! What is even more important is that this action of writing down what you expect a program to do gives you a chance to design a properly *structured* program. Structured in this sense means that the program is put together in a logical sequence, so that it is easy to add to, change, or redesign. If you learn to program in this way, your programs will be easy to understand, take less time to get working, and will be easy to extend so that they do more than you at first intended.

As an example, take a look at Fig. 6.6. This shows a program outline plan for a simple game. The aim of the game is to become familiar with the names of animals and their young. The program plan shows what I expect of this game. It must present the name of an animal, picked at random, on the screen, and then ask the name of its young. A little more thought produces some additional points. The name of the young animal will have to be correctly spelt. A little trickery will be needed to prevent the user (son, daughter, brother, or sister) from finding the answers by typing LIST and looking for the

Aims
1. Present the name of an animal on the screen.
2. Ask what its young is called.
3. Reply must be correctly spelled.
4. User must not be able to read the answer from a listing.
5. Give one point for each correct answer.
6. Allow two chances at each question.
7. Keep a track of the number of attempts.
8. Present the score as the number of successes out of the number of attempts.
9. Pick animal names at random.

Fig. 6.6. A program outline plan. This is your starter!

DATA lines. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program in the way that an artist paints a picture or an architect designs a house. That means designing the outlines first and the details later. The outlines of this program are the steps that make up the sequence of actions. For example, we shall want to have a title displayed. Give the user time to read this, and then show instructions. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and make other such preparations. We then need to play the game. The next thing is to find the score, and then ask the user if another game is required. Yes, you have to put it all down on paper! Figure 6.7 shows what this might look like at this stage.

1. Display title, then instructions.
2. Display the name of animal on the screen.
3. Ask for the name of the young.
4. Use INPUT for reply.
5. Compare reply with correct answer.
6. If correct, add 1 to the score and ask if another one is wanted.
7. If incorrect, allow another try.
8. If the second attempt is also incorrect, select another question.
9. Ends when user types N in response to 'Do you want another one?'

Fig. 6.7. The next stage in expanding the outline.

Foundation stones

Now, at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have established how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 6.8 shows what you should aim for at this stage. There are only fourteen lines of program here, and that's as much as you want. This is a *foundation*, remember, not the Empire State Building. It's

```
10 CLS:GOSUB 1000
11 REM Title
20 GOSUB 1200
21 REM Instructions
30 GOSUB 1400
31 REM Setup
40 GOSUB 2000
41 REM Play
50 GOSUB 3000
51 REM Score
60 GOSUB 4000
61 REM Another?
70 IF INSTR("YESyes",K$)<>0 THEN 40
100 END
```

Fig. 6.8. A *core* or *foundation* program for the example.

also a program that is being developed, so we've hung some 'Danger – men at work' signs around. These take the form of lines that start with REM. REM means REMinder, and any line of a program that starts with REM will be ignored by the computer. This means that you can type whatever you like following REM, and the point of it all is to allow you to put notes in with the program. These notes will not be printed on the screen when you are using the program, and you will see them only when you LIST. In Fig. 6.8, I have put the REM notes on lines which are numbered just 1 more than the main lines. In this way I can remove all the REM lines later. How much later? When the program is complete, tested, and working perfectly. REMs are useful, but they make a program take up more space in memory, and run slightly slower. I always like to keep one copy of a program with the REMs in place, and another 'working' copy which has no REMs. That way I have a fast and efficient program for everyday use, and a

fully-detailed version that I can use if I want to make changes.

Let's get back to the program itself. As you can see, it consists of a set of GOSUB instructions, with references to lines that we haven't yet written. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of Fig. 6.7 exactly, and the only part that is not committed to a GOSUB is the IF in line 70. We shall write a subroutine which will use INKEY$ to look for a y or Y being pressed, and line 70 deals with the answer. What's the question? Why, it's the 'Do you want another game' step that we planned for earlier.

Yes, you're right, line 70 does look rather unfamiliar. By testing with INSTR("YESyes",K$), we will get 1 if Y is pressed and 4 if y is pressed. We will also get these numbers if the reply were to be YES or yes (not that it can be with INKEY$ used). If K$ is neither y nor Y, then INSTR gives 0, meaning that the string we are seeking is not contained in YESyes. Simple, but very useful.

Take a good long look at this fourteen-line piece of program, because it's important. The use of all the subroutines means that we can check this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the subroutines. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions last, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you'll have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
9 GOTO 30
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it.

The next step is to get to the keyboard (at last, at last!) and enter this core program. If you use the GOTO step to skip round the title and instructions temporarily, you can then put in simple PRINT lines at each subroutine line number. We did this, you remember, in the program of Fig. 6.2, so you know how to go about it. This allows you to test your core program and be sure that it will work before you go any further.

The next step is to record this core program and then keep adding to the core. If you have the core recorded, then you can load this into

your CPC464, add one of the subroutines, and then test. When you are satisfied that it works, you can record the whole lot on another cassette. Next time you want to add a subroutine, you start with this version, and so on. In this way you keep tapes of a steadily-growing program, with each stage tested and known to work. Again, this is important. Very often testing takes longer than you expect, and it can be a very tedious job when you have a long program to work with. By testing each subroutine as you go, you can have confidence in the earlier parts of the program and be able to concentrate on errors in the new sections.

Subroutine routine

The next thing we have to do is to design the subroutines. Now some of these may not need much designing. Take, for example, the subroutine that is to be placed in line 4000. This is just our familiar INKEY$ routine, along with a bit of PRINT, so we can deal with it right away. Figure 6.9 shows the form it might take. The subroutine is

```
4000 PRINT"Would you like another one?"
4010 PRINT"Please answer y or n."
4020 K$=INKEY$:IF K$=""THEN 4020
4030 RETURN
```

Fig. 6.9. The subroutine for line 4000.

straightforward, and that's why we can deal with it right away! Type it in, and now test the core program with this subroutine in place.

Now we come to what you might think is the hardest part of the job – the subroutine which carries out the Play action. In fact, you don't have to learn anything new to do this. The Play subroutine is designed in exactly the same way as we designed the core program. That means we have to write down what we expect it to do and then arrange the steps that will carry out the action. If there's anything that seems to need more thought we can relegate it to a subroutine to be dealt with later.

As an example, take a look at Fig. 6.10. This is a plan for the Play subroutine, which also includes information that we shall need for the setting-up steps. The first item is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't

1. Keep the answers as an array of ASCII codes in a string.
2. Keep list of animals in another string.
3. The number which selects the animal will also select the answer.
4. Use variable TR to record tries, SC to keep score.
5. Use variable GO to record the number of attempts at one question.

Fig. 6.10. Planning the Play subroutine.

deter the more skilled, but it will do for starters. I've decided to put one answer in each DATA line in the form of a string of ASCII codes, with each code written as a three-figure number. Why three figures? Well, the capital letters will use two figures only, the small letters three, so making them all into three figures simplifies things. You'll see why later – what we do is to write a number like 86 as 086, and so on. That's the first item for this subroutine.

The next is that we shall keep the names of the animals in an array. This has several advantages. One is that it's beautifully easy to select one at random if we do this. The other is that it also makes it easy to match the answers to the questions. If the questions are items of an array whose subscript numbers are 1 to 10, then we can place the answers in DATA lines (one set of numbers in each data line) and read these also as a string array.

The next thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using SC for the score and TR for the number of tries looks self-explanatory. The third one, GO, is one that we shall use to count how many times one question is attempted. Finally, we decide on a name for the array that will hold the animal names – Q$.

Play for today

Figure 6.11 shows what I've ended up with as a result of the plan in Fig. 6.10. The steps are to pick a random number, use it to print an animal name, and then find the answer. That's all, because the checking of the answer and the scoring is dealt with by another subroutine. Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time. As it is, I've had to put another subroutine into this one to keep things short.

We start the subroutine at line 2000 by 'clearing a variable'. The size of GO is set to 0, to make sure that this variable has the correct

```
2000 GO=0:V=INT(10*RND(1))+1
2010 CLS:PRINT"The animal is - ";Q$(V)
2020 PRINT:PRINT"The young is called - "
;
2030 INPUT X$:TR=TR+1
2040 GOSUB 5000
2041 REM Find correct answer
2050 RETURN
```

Fig. 6.11. The program lines for the Play subroutine.

size each time this subroutine is started. The second part of line 2000 then picks a number, at random, lying between 1 and 10. Lines 2010 to 2030 are straightforward stuff. We print the name of the animal that corresponds to the random number, and ask for an answer; the young of that animal. The last section of line 2030 counts the number of attempts. This is the logical place to put this step, because we want to make the count each time there is an answer. Now it's chicken-out time. I don't want to get involved in the reading of ASCII codes right now, so I'll leave it to a subroutine, starting in line 5000, which I'll write later. The REM in line 2041 reminds me what this new subroutine will have to do, and the Play subroutine ends with the usual RETURN.

Down among the details

With the Play subroutine safely on tape, we can think now about the details. The first one to look at should be one that precedes or follows the Play step, and I've chosen the Score routine. As usual, it has to be planned, and Fig. 6.12 shows the plan. Each time there is a correct

---

1. For a correct answer, increment SC.
2. For an incorrect answer, with GO=0, allow another try, and make GO=1.
3. For second incorrect answer, with GO=1, pass on to the next question, and make GO=0 again.

Fig. 6.12. Planning the Score subroutine.

answer, the number variable SC will be incremented, and we can go back to the main program. More is needed if the answer does not match exactly. We need to print a message, and allow another attempt. If the result of this next attempt is not correct, that's an end

to it. Later on, after you have read Chapter 11, you might want to include some sound in the program. We could have a short beep to announce a mistake, and a long one for a correct answer. Write it down!

Figure 6.13 shows the program subroutine that has been developed from this plan. Line 3000 deals with a correct answer. We need to print a message here, and to give us a bit more space, we use GOTO 3200 to finish the job. The GOTO 3040 in line 3210 ensures that if the

```
3000 PRINT:IF X$=A$ THEN SC=SC+1:GOTO 32
00
3010 IF GO=0 THEN GOTO 3300
3020 GO=0:PRINT"No luck- try the next on
e."
3030 FOR Q=1 TO 1000:NEXT
3040 RETURN
3200 PRINT"Correct- you score is now";SC
3210 PRINT"in"TR;" attempts. ":GOSUB 700
0:GOTO 3040
3300 PRINT"Not correct- but it might be
your"
3310 PRINT"spelling! You get another go
free.":TR=TR-1
3320 GOSUB 7000:GO=1:GOSUB 2010:GOTO 300
0
```

Fig. 6.13. The Score subroutine written.

answer was correct, the rest of the subroutine is skipped, and the subroutine returns. If the answer is not correct, though, line 3010 swings into action. This tests the value of GO and if it is zero causes a change to line 3300 to print its message and give further instructions. Line 3320 calls the subroutine at line 2010 again so that the user can make another answer entry. The GOTO 3000 at the end of line 3320 then tests this answer again.

There's a piece of cunning here. The number variable GO starts with a value of 0. When there is an incorrect answer, however, and GO is still 0, line 3010 is carried out. One of the actions of line 3320, however, is to set GO to 1. When you answer again, with GO=1, line 3000 will be used, and if your second answer is wrong, line 3010 cannot be used, because GO is not zero. The next line that is tried, then, is 3020. This puts GO back to zero for the next round, prints a sympathetic message, pauses, and then lets the subroutine return in line 3040.

Now that we've got the bit between our teeth, we can polish off the rest of the subroutines. Figure 6.14 shows the subroutine that deals with dimensioning and arrays. Line 1400 sets all the variables for the scoring system to zero. Line 1410 dimensions the array Q$ that will be used for the names of the animals, and A$ which will be used for the numbers that give the answers. Line 1420 then reads the names from a data list into the array Q$, and line 1430 reads the numbers into A$ – and that's it! We can write the DATA lines later, as usual.

```
1400 TR=0:SC=0:GO=0
1410 DIM Q$(10),A$(10)
1420 FOR J=1 TO 10:READ Q$(J):NEXT
1430 FOR J=1 TO 10:READ A$(J):NEXT
1440 RETURN
```

Fig. 6.14. The dimensioning and array subroutine.

Next comes the business of finding the answer. We have planned this, so it shouldn't be too much hassle. Figure 6.15 shows the program lines. The variable V is the one that we have selected at random, and it's used to select one of the strings of ASCII numbers, A$(V). Since each number consists of three digits, we want to slice this string three digits at a time, and that's why we use STEP 3 in the

```
5000 A$="":FOR J=1 TO LEN(A$(V)) STEP 3
5010 A$=A$+CHR$(VAL(MID$(A$(V),J,3))):NE
XT
5020 RETURN
```

Fig. 6.15. Checking the answer.

FOR...NEXT loop in line 5000. Line 5010 then builds up the answer string, which we call A$. Remember that A$, used alone, is not confused with the A$(V) array. A$ is set to a blank in the first part of line 5000 to ensure that we always start with a blank string, not with the previous answer, which would also be A$. The string A$ is then built up by selecting three digits, converting to the form of a number by using VAL, then to a character by using CHR$. This character is then added to A$, and this continues until all the numbers in the string have been dealt with. That's the hard work over. Figure 6.16 is the subroutine for the instructions, and Fig. 6.17 is the title subroutine. The title lines include a pause, and have been written with the MODE 0 type of display. This gives large letters easily, and is excellent, unless you use long words that fall off the end of the screen! Finally, Fig. 6.18 shows the DATA lines and the delay subroutine.

```
1200 CLS:PRINT TAB(15)"INSTRUCTIONS"
1210 PRINT:PRINT TAB(4)"The computer wil
l supply you with"
1220 PRINT"the name of an animal. You sh
ould type"
1230 PRINT"the name of its young - and m
ake sure"
1240 PRINT"that your spelling is correct
 and that"
1250 PRINT"you start each name with a ca
pital"
1260 PRINT"letter. The computer will kee
p score"
1270 PRINT"for you. You get two shots at
 each"
1280 PRINT"name."
1290 PRINT:PRINT"Press the spacebar to s
tart."
1300 IF INKEY(47)=-1 THEN 1300 ELSE RETU
RN
```

Fig. 6.16. The instructions - always leave these until you have almost finished.

```
1000 MODE 0
1010 PRINT TAB(4)"Young Animals"
1020 GOSUB 7000:MODE 1:RETURN
```

Fig. 6.17. The title program lines.

```
6000 DATA Dog,Cat,Cow,Horse,Hen,Fox,Kang
aroo,Goose,Lion,Pig
6001 DATA 080117112112121
6002 DATA 07505116116101110
6003 DATA 067097108102
6004 DATA 070111097108
6005 DATA 06710410509910710111 0
6006 DATA 067117098
6007 DATA 074111101121
6008 DATA 07111111510810511 0103
6009 DATA 067117098
6010 DATA 08010510310810111 6
7000 FOR Q=1 TO 3000:NEXT:RETURN
```

Fig. 6.18. The DATA lines that are needed, along with a time delay subroutine.

Now we can put it all together and try it out. Remember that the CPC464 allows you to record pieces of program, and then MERGE them together. Because it's been designed in sections like this, it's easy for you to modify it. You can use different DATA, for example. You can use a lot more data – but remember to change the DIM in line 1410. You can make it a question and answer game on something entirely different, just by changing the data and the instructions. You can create much more interesting sound effects, or add more interesting graphics. One major fault of the program is that once an item has been used, it can be chosen again, because that's the sort of thing that RND can cause. You can get round this by swapping the item that has been chosen with the last item (unless it was the last item), and then cutting down the number from which you can choose. For example, if you chose number 5, then swap numbers 5 and 10, then choose from 9. This means that the 10*RND(1)+1 step will become D*RND(1)+1, where D starts at 10, and is reduced by 1 each time a question has been answered correctly.

There's a lot, in fact, that you can do to make this program into something much more interesting. The reason that I have used it as an example is to show what you can design for yourself at this stage. Take this as a sort of BASIC 'construction set' to rebuild any way you like. It will give you some idea of the sense of achievement you can derive from mastering your CPC464. As your experience grows, you will be able to design programs that are very much longer and more elaborate than this one. By that time, you'll be thinking of adding a printer and a disk drive to your CPC464. Even before you get to that stage, you'll see how useful it is to keep recordings of useful subroutines which you can add to your programs by making use of MERGE.

There's just one more thing. Suppose your own program doesn't perform as you expect it to? How do you set about sorting out problems? For a brief look at this, read Appendix A, which is concerned with editing and trouble-shooting.

Chapter Seven

Cassette Data Filing

The CPC464, unlike most other microcomputers, comes with a built-in storage system for programs and data. Whereas most other computers use ordinary cassette recorders, the CPC464 uses its own built-in recorder, designed specially for use with the computer. Since the use of such a system for data recording will probably be quite new to owners of a CPC464, even if they have used a computer previously, this chapter is devoted to the use of the cassette system for storing data. Chapter 1 has already dealt with the use of the cassette system in connection with storing and loading programs. The most puzzling new names that cause difficulty to the CPC464 owner are *devices*, *streams* and *buffers*. Once you grasp what is meant by these words, and how they are applied, you will find cassette system operation much more interesting, and you will also be able to do much more with your CPC464. Let's start, then, by explaining these words.

A *device* is something that puts out or receives data. Your keyboard is a device, because each time you touch a key, a set of electrical signals is sent to the computer. The screen is another device, because every time the computer sends a set of electrical signals to the TV set, you will see something appear on the screen. The keyboard is a transmitting device, because it sends signals. The screen (or the TV) is a receiving device, because it accepts signals. The CPC464 screen, in fact, can behave as if it were several devices, because it can be used in separate portions. Later, in Chapter 8, we'll see how you can divide up the screen space as you wish.

Some devices can perform both operations. A *console* is the combination of keyboard and screen, which both sends and receives data. The cassette system is also a device which can be used in both directions. The disk system is a similar type of device.

We have talked of electrical signals passing from one device to another, and this is what actually happens. It's a lot more useful to think of what these signals represent. Each set of signals represents a

unit of data called a *byte*, and *data* is the stuff that computers are designed to deal with. One byte is the amount of memory that is needed to store one character in ASCII code, or one command in BASIC. Data may be numbers or names; it's anything that the computer has to work with. If you have a program that arranges the names of your friends in order of birthdays, then that program needs data. The data in this example is the set of names and birth dates. If you have a program that shows cookery recipes and shopping lists, then the data is the instructions, the names of the foodstuffs, and the quantities. Every computer that is designed to be used for anything more than the simplest games must be able to save and load data of this type, separately from the program that generates it or uses it.

There's a lot to be gained from this approach. The memory of the CPC464 is used for many purposes over which you have no control. A very long program which gathered data and then made use of it might not fit into your computer. It's a lot more sensible to have a short program which gathers the data, using INPUT lines, and which records the data as it is gathered. The data is then safe if anything should happen that scrambles the memory of the CPC464. Another program can then make use of this same data. By keeping the two programs and the data separate, you can deal with much more information than would be possible if you had to have the whole lot in the memory at one time.

What has all this to do with buffers, streams and devices? Well, devices are the parts of the computer which give out or receive data, as we now know. *Streams* are the paths which carry the data. Just think of what happens when you use your CPC464. When you press a key, something appears on the screen. The keyboard is one device, the screen is another, and there is a stream which links them. This is just a fancy way of saying that there is a path for data signals from the keyboard to the screen. The important point, however, is that these paths or streams can be controlled. Controlling them means that we can change the paths, breaking some and making others, as we please. It wouldn't be sensible to break some of the paths, of course, except for special purposes. You normally want to see on the screen the words that you type on the keyboard. If you were typing a special password, however, and you didn't want anyone who was watching the screen to see it, it would make sense to break the stream that connects the keyboard to the screen. Figure 7.1 shows this in action. Line 10 asks you to enter a four-letter password. In line 20, PRINT CHR$(21) will disconnect the stream that links the keyboard and the screen. You can now enter your password, using the INPUT line 30,

```
10 PRINT"ENTER YOUR 4-LETTER PASSWORD NO
W"
20 PRINT CHR$(21)
30 INPUT B$
40 PRINT CHR$(6)
50 IF LEN(B$)<>4 THEN PRINT"Incorrect- p
lease try agsin":GOTO 10
60 PRINT"Thank you- password acknowledge
d."
70 PRINT"Type PRINT B$ to see it!"
```

Fig. 7.1. How to conceal an entry by shutting off the screen stream!

but nothing will show on the screen! Line 40 links the stream again, so that the messages can be displayed. As usual, the entry is tested, so that you can try again if you have not entered four letters. If you want to check that you really did enter something, then a PRINT B$ will reveal it.

As you might expect by now, there are some streams which are connected to devices from the moment you switch on. It's obvious, for example, that there's a connection between the keyboard and the screen. This is normally a stream numbered 0, and you can force a PRINT instruction to use this stream by putting #0 following the PRINT. The hashmark (#) is the US sign for 'number', and is used as we would use 'No.'. The way that the CPC464 is designed enables ten streams to be used. Of these, three (0,8,9) have fixed jobs to do, and you should normally try to avoid changing these tasks. The devices are identified by number, using the hashmark (#) before each number.

As so often happens in computing, the ten streams are not numbered 1 to 10, but 0 to 9. Of these streams, 0,8 and 9 are set aside for the CPC464's own use. Stream #0 normally controls the screen, stream #8 normally controls the printer, and stream #9 controls the cassette unit. That leaves seven stream numbers for you to play with, and in this chapter and the next, we'll be looking at ways of using these spare stream numbers. This chapter is about storing data on the cassette, and in order to do this you have to know two things. One is how to select a buffer, the other is how to link it to the cassette recorder.

The way that you select a *buffer* and link the cassette recorder is by using one of the OPEN commands. There are two of them; OPENOUT and OPENIN. OPENOUT must be followed by a filename, just like SAVE. When this runs, it will link the cassette

recorder to the computer ready for data to be recorded. This is simply a preparation – no data passes when OPENOUT is used; it only gets things ready for data. OPENIN must also be followed by a filename, just like LOAD. Because of the use of the filename, only a file of the correct name will be selected, and the cassette system *will be operated* so as to find this filename on the tape.

Data filing techniques

What is a file?

The word *file* occurs many times in the course of this book. A file can mean any collection of characters which belong together. The characters of a BASIC program constitute a file, for example, because the program will not run if characters are missing. A set of names and addresses in ASCII code is a file because they form one group of information, such as our friends, or our suppliers, or debtors. A set of bytes of machine code is a file, and a collection of the numbers that are used by a financial program is a file. In this chapter, though, I'll take *file* to mean a collection of information which we can record on a cassette, and which is separate from a program. For example, if you have a program that deals with your household accounts, you would need a file of items and money amounts. This file is the result of the action of the program, and it preserves these amounts for the next time you use the program. Taking another example, suppose that you devised a program which was intended to keep a note of your collection of vintage 78 rpm recordings. The program would require you to enter lots of information about these recordings. This information is a file, and at some stage in the program, you would have to record this file. Why? Because if a program like this is going to be really useful, there will not be enough space even in the memory of your CPC464 computer to hold all of the information at one time. In addition, you wouldn't want to have to change the program each time you wanted to add items to the list. This is the topic that we're dealing with in this chapter; recording the information that a program uses. The shorter word is *filing* the information.

We can't discuss filing without coming across some words which are always used in connection with filing. The most important of these are *record* and *field* (Fig. 7.2). A record is a set of facts about one item in the file. For example, if you have a file about LNER steam locomotives, one of your records might be used for each

	FRIENDS FILE
RECORD 1	
FIELD 1	Name 1
FIELD 2	Address 1
FIELD 3	Phone No. 1
FIELD 4	Birthday 1
RECORD 2	
FIELD 1	Name 2
FIELD 2	Address 2
FIELD 3	Phone No. 2
FIELD 4	Birthday 2
RECORD 3	
etc.	

Fig. 7.2. The meaning of *record* and *field*.

locomotive type. Within that record, you might have the designer's name, firebox area, working steam pressure, tractive force, and anything else that's relevant. Each of these items is a field; an item of the group that makes up a record. Your record might, for example, be the SCOTT class 4-4-0 locomotives. Every different bit of information about the SCOTT class is a field; the whole set of fields is a record; and the SCOTT class is just one record in a file that will include the Gresley Pacifics, the 4-6-0 general purpose locos, and so on. Take another example, the file 'British motor-bikes'. In this file, BSA is one record, AJS is another, Norton is another. In each record, you will have fields. These might be capacity, number of cylinders, bore and stroke, suspension, top speed, acceleration, and whatever else you want to take note of. Filing is fun – if you like to arrange things in the right order.

Cassette system filing

In this book, because we are dealing with the CPC464 cassette system, we'll ignore filing methods that are based on DATA lines in a BASIC program. This is because cassette data filing keeps the data separate from the program, and is therefore much more useful. If it's all familiar to you, please bear with me until I come to something that you haven't met before.

To start with, there are two types of files, only one of which we can use with a cassette system. These are *serial files* and *random access*

files. The difference is a simple, but important one. A serial (or sequential) file records all the information in order on a cassette system, just as it would be placed on a cassette. If you want to get at one item, you have to read all of the items into the computer, and then select. There is no simple way in which you can command the system to read just one record or one field. A random access file does what its name suggests – it allows you to obtain from the recorded data one selected record or field without reading every other one from the start of the file. The difference between serial filing on tape and random access filing on disk is illustrated in Fig. 7.3. Random access filing is

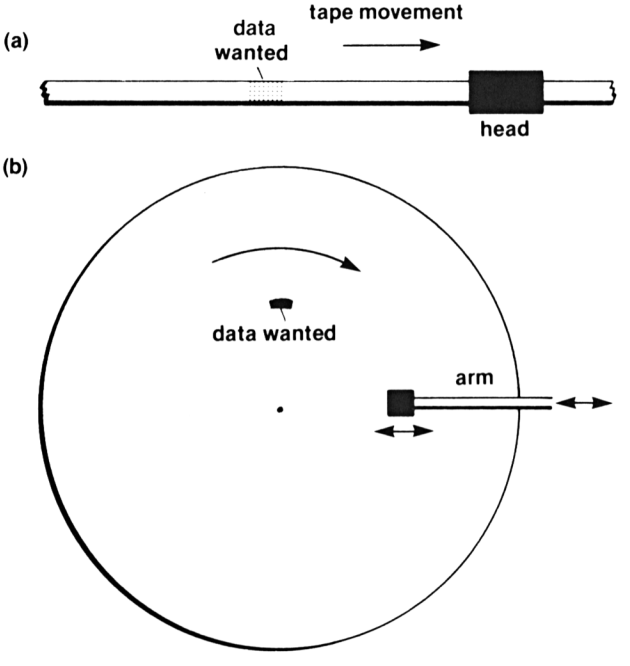

Fig. 7.3. Illustrating the difference between (a) serial files on tape and (b) random access files on disk. Any part of the disk can be reached by placing the head at the correct radius as the disk spins. To read a piece of tape, you have to pull all the preceding length of tape past the head.

something which requires the use of a disk system, but we can design programs which achieve something like the same effect by using serial files on the cassette system. We'll start, then, by looking at serial files, which are also the type of files that we record on a cassette.

Creating a file

When you are dealing with something new, it's always a good idea to start at the beginning and keep things simple at first. We'll start looking at filing by considering the way we make connections to the cassette system. Figure 7.4 shows a very simple example of how one

```
10 OPENOUT "TEST"
20 A=5
30 PRINT#9,A
40 CLOSEOUT
```

Fig. 7.4. Recording the value of a single variable. It's the value that's recorded, not the variable name.

item of data, the value of a variable called A, can be recorded on the cassette system, whose stream number is always 9. Because the steps are so important, we'll look at each of them in close detail. Starting with line 10, then, we *open a file*. This requires the command word OPENOUT, and we have to specify a name for this file of data. Line 10 uses OPENOUT "TEST" to open a new file on the cassette system with the name TEST. You must be careful about how you use the recorder here, because you need to select a cassette, and a position on the cassette, which has nothing else recorded on it. There is nothing to prevent you from 'wiping' a file by accidentally recording over it another one which has the same name, or a different name. Disk systems can protect you against that sort of error, but when you use a cassette for data storage, you have only the reading of the tape counter to help you.

The next step is to assign a value to the variable, A, in line 20. Line 30 is the important one now. The instruction PRINT#9,A means 'send out the value of A over stream 9'. Stream 9, however, is always connected to the cassette system, so that this line 30 will 'print' the value of A on to the cassette. You will see the usual messages that you get when you use SAVE, and the value will be filed under the name TEST so that it can be easily found again. Line 30 causes the recording to be made, but it's not quite so straightforward as it might seem. The cassette system records groups of characters, and the operating system is arranged so that the computer will gather data in the memory until it has enough. This part of memory is called a *buffer*, and when you open a stream, you also automatically allocate a buffer for that stream. The data that is to be recorded is shifted into the buffer, and is then recorded. If there is more data than the buffer can cope with (more than 2000 characters), then the buffer will have

to be filled and emptied more than once. At the end of such a process, you have to make sure that the buffer is cleared. This is done in line 40 by the CLOSEOUT command.

Now what happens when this runs? As far as you are concerned, it's just that the cassette system delivers its usual messages, spins for rather a long time, then stops.

Now we have to prove that this data was actually recorded, and show that we can recover it. Take a look now at the listing in Fig. 7.5. Line 10 in this listing uses OPENIN, not OPENOUT. We already

```
10 OPENIN "TEST"
20 INPUT #9,X
30 PRINT"X IS";X
40 CLOSEIN
```

Fig. 7.5. Recovering the variable value from the tape.

have a file on our cassette, and we want to read it, not to create a new file. The stream number is once again #9, and we have to specify "TEST", the name of the file. Having 'opened the file', meaning that a buffer will now be allocated for use with signals from the cassette system, the system finds the filename on the tape, if it exists. If the filename cannot be found, the system will keep on trying as long as there is tape to read! A disk system does this type of thing much better, because it can find at once if the filename is on the disk. Having found the file, we can then read the data. Line 20 does so, using INPUT #9,X. INPUT by itself always refers to the keyboard, but when we place the #9 after INPUT, it will cause the input to come from the cassette system. Because we have specified "TEST" in the OPENIN statement, what comes from the cassette system will only be whatever is in the file TEST. Line 30 prints what is read in, and line 40 closes the stream again. It all looks reasonably simple and straightforward, but take a close look at these two little pieces of programming, because they contain a lot you will need to get to grips with in the course of data filing. Notice, for example, that we can assign what is read to any variable name that we like. We used A when recording, but X when replaying. As far as the computer is concerned, reading a file from cassette is just another INPUT step like reading from the keyboard.

Now try something more ambitious with the creation of a file of numbers. Figure 7.6 shows a program which generates a file of numbers – the even numbers from 0 to 50 – and then records these numbers on the cassette system and also prints them on the screen. There are only six lines to this program, but three of them contain

```
10 OPENOUT "EVENS"
20 CLS:FOR N=0 TO 50 STEP 2
30 PRINT #9,N
40 PRINT#0,N;" ";
50 NEXT
60 CLOSEOUT
```

Fig. 7.6. Creating a file of numbers and recording the values.

these important commands that you need to understand. We'll start, reasonably enough, with line 10. This is one of those OPENOUT commands which connects a buffer to a stream. The stream is #9, the cassette stream, and the filename is EVENS. The next thing is to create a file, and in this case, it's being done by a loop which starts in line 20. This allocates variable N as each of the even numbers in turn, with the NEXT in line 50. How do we place the numbers into the buffer? Line 30 does this, using PRINT #9,N, and since there is a string of numbers to be recorded, the buffer accepts the numbers *before* the cassette recording begins. The cassette system is nothing like as fast as the computer. The FOR...NEXT loop in lines 20 to 50 could easily be completed before the motor of the cassette system could be started! Each time line 30 runs, the value of N is stored temporarily in the buffer. It's a good name, because its action is to connect the computer and the cassette system so that they work smoothly together. In this simple example, all the numbers that are generated by the action of the loop are simply passed into the buffer. Nothing is recorded this time; the cassette system doesn't start turning. The recording of data takes place when all of the numbers have been assembled. CLOSEOUT means close down the stream, and when a stream is closed, part of the action is to empty any buffer which is part of that stream. In addition, an *end-of-file* marker is recorded, so that the system can identify the end of a file even when several blocks have been recorded. That's it! If you now RUN this program, you'll see the numbers appear on the screen, showing that the whole loop is being run. You will get the message about pressing REC and PLAY only after the whole loop has ended.

There's just one more point before we get seriously into this data business. In some applications, you don't really want the ordinary messages about pressing the keys and recording the blocks. You can suppress these messages by making the first character of the filename an exclamation mark. For example, if you alter line 10 so that it reads OPENOUT "!EVENS", then when the program runs you will see the numbers, and recording will start with nothing more on the screen.

This is useful, but not if you then forget to press REC and PLAY! It does, however, give you the chance to control things for yourself, so that you can put in your own messages, as we'll see.

This program has shown you the buffer in action, and the size of the buffer is large enough for a lot of data. You can try it for yourself by altering the program so that it looks like Fig. 7.7. This program uses the exclamation mark in the filename to suppress messages, so

```
10 OPENOUT "!MOREVENS"
20 PRINT"Please press REC and PLAY keys
NOW."
30 PRINT"Press SPACEBAR to start."
40 WHILE INKEY(47)=-1:WEND
50 CLS:FOR N=0 TO 5000 STEP 2
60 PRINT #9,N
70 PRINT N
80 NEXT
90 CLOSEOUT
100 PRINT"End of recording- please press
 STOP key"
```

Fig. 7.7. A much longer number file, to display buffer action. The messages have been suppressed by using the ! mark in the filename.

lines 20 and 30 provide a suitable message to give you time to prepare the recorder. This time, the number of bytes of data will need several thousand bytes, and when you RUN the program, you will find that the numbers appear on the screen until 598 is reached. At this point, the cassette motor starts, and a block of data is recorded. The numbers start running again until 1160, when the cassette records another block, and the same happens at 1672, and every 512 numbers after that. What is happening is that the buffer is being filled because of the loop, and is being emptied by the cassette system.

These short programs have put data into a file, but so far you have had to trust me that there is actually something on the tape. Figure 7.8 shows how this can be done for the "EVENS" file. In this example, the filename of !EVENS has been used, to suppress the normal messages. This means that we have to provide our own, but as you can see from the program, you have to be careful where you put them. If you start the program with OPENIN "!EVENS", you will find that the program doesn't appear to work correctly. This is because when OPENIN "!EVENS" is carried out, the machine must find the filename of "EVENS" (even though this was not recorded as

```
10 PRINT"Press PLAY key, then SPACEBAR."
20 WHILE INKEY(47)=-1:WEND
30 OPENIN "!EVENS"
40 FOR N=0 TO 50 STEP 2
50 INPUT#9, A
60 PRINT A;" ";
70 NEXT
80 CLOSEIN
90 PRINT: PRINT"Press STOP key of record
er."
```

Fig. 7.8. Reading the EVENS file, with messages suppressed. This means we have to provide our own messages.

!EVENS) before anything else can be done. You therefore need to have your 'PRESS PLAY' message right at the beginning of the program, so that OPENIN can do its work. As usual, the program starts in line 30 with opening the file, using OPENIN, with the file name of !EVENS. We read the file with INPUT#9, in a loop to read and display the data. When this runs, then, you will hear the cassette motor run, the machine will locate the file, and read it, and you will see the numbers appear on the screen.

Now let's try the longer file, !MOREVENS, in Fig. 7.9. This time, we'll arrange a more tidy screen display by interrupting the display process. Line 70 performs the interruption. The condition is IF N/20=INT(N/20). This means the condition when N divides evenly by 20. If N/20 has a remainder, then it can't be equal to INT(N/20),

```
10 PRINT"Press PLAY key on recorder-"
20 PRINT"then spacebar to start."
25 WHILE INKEY(47)=-1:WEND
30 OPENIN "!MOREVENS"
40 FOR N=0 TO 5000
50 INPUT #9,K
60 PRINT K
70 IF N/20=INT(N/20)THEN FOR J=1 TO 2000
:NEXT:CLS
80 NEXT
90 PRINT"Press STOP key of recorder"
```

Fig. 7.9. Reading the MOREVENS file, with a screen display that gives you time to see the numbers.

which is the whole number part of N/20 only. Each time N has a value that divides evenly by 20, then, the second part of line 70 causes

a delay, and at the end of the delay, the screen is cleared before the main loop continues. Now when you run this one, you will hear the cassette motor start, find the file, and read it. The number 0 then appears under the text on the screen, and then the main loop displays the numbers in groups of 20. The cassette motor will start and stop at intervals as required to fill the buffer. You will hear this stop and start action going on all the time that data is being read back from the MOREVENS file. The process is a slow one, and it's better if data filing is carried out at the faster speed, which is obtained by using SPEED WRITE 1. If the data has been recorded at the slow speed, however, you have to read it also at the slow speed. Incidentally, you'll notice that CLOSEIN has been omitted in this example. You can get away with that if the program is being run, and then a NEW used to clear the machine. If other data were to be read, however, you could not use another OPENIN unless the CLOSEIN had first been executed.

More serial filing

Suppose that what we want to record is not a set of numbers that has been generated by a program, but a set of names that you have typed. As far as the cassette data system is concerned, this is just another set of data, and it's dealt with in exactly the same way. Each time you press ENTER at an INPUT step, the data is stored in a buffer, and it stays there until the buffer is full, or until the entry is complete and the file is closed. Once again, you can see the importance of using a buffer – you wouldn't expect the cassette data system to record each letter as you typed it, would you?

Figure 7.10 shows a short program of this type. Normally, if you were gathering information like this, you would store the names as an array. This, as you probably know, introduces complications like having to dimension the array. Unless you might want to look at a previous entry at some point when entering names, you don't need to use an array for recording a set of names. It can be a different matter when you play back, but that's something that we'll look at shortly.

Taking the program in more detail now, line 10 selects SPEED WRITE 1, the faster cassette writing speed. This does *not* mean that the cassette tape moves faster, just that the data is recorded on to it at a higher rate. We shall cancel this command at the end of the program, because if we don't, it will remain in force. You might want to SAVE another item at the slower rate, and this would not be

```
10 SPEED WRITE 1
20 OPENOUT "!NAMES"
30 CLS:PRINT TAB(18)"NAMES"
40 PRINT:PRINT"This program stores a fil
e of names for":PRINT"you on the cassett
e. Make sure that":PRINT"you have a cass
ette ready."
50 PRINT"Input X to end the entry."
60 PRINT"Press REC and PLAY on the recor
der,and":PRINT" then the spacebar when y
ou are ready."
70 WHILE INKEY(47)=-1:WEND
80 WHILE NAME$<>"X" AND NAME$<>"x"
90 INPUT"Name ";NAME$
100 PRINT#9,NAME$
110 WEND
120 CLOSEOUT
130 PRINT"Press STOP key of recorder now
."
140 SPEED WRITE 0
```

Fig. 7.10. Filing names. The faster cassette filing speed has been selected to make the program more efficient.

possible if the SPEED WRITE 1 command has been executed. Line 20 opens the file, suppressing messages as usual. Lines 30 to 60 then provide brief instructions, and line 70 is a 'Press spacebar to start' pause.

The main WHILE loop then starts in line 80, and the idea is to input and record each name until an X or x is typed. If you wanted to expand this into a more realistic name file, you would probably want more detailed instructions. Line 90 accepts the name that you type (no commas permitted with an INPUT step, remember.) The name is then recorded in line 100. If x has not been entered, then line 110 returns for the next word. If you type names fast and continually, you will find that the cassette motor spins every now and again, emptying the buffer. You can't enter data while this is going on.

More on reading

By this time, you have a few files of both numbers and names stored on your cassette data system, and it's time to pay rather more

attention to the methods that are used for reading files and using the information from them. Suppose, for example, that the numbers which we recorded in the file EVENS had been placed on the file by an accounts program. They might, for example, be the daily takings of a small shop. One thing that we might want to do with the numbers, then, would be to read them from the file and add them, showing only the total. This is a conventional and straightforward piece of programming, and one for which you will probably find a large number of uses. Figure 7.11 shows what is needed. It starts in line 10 by clearing the screen and then line 20 prints the title, followed by some instructions in lines 30 and 40. Line 50 causes the machine to wait for the spacebar to be pressed, and the real work starts in line 60,

```
10 CLS
20 PRINT TAB(18)"TOTALS"
30 PRINT:PRINT"Press PLAY key on recorde
r when the":PRINT"cassette is ready."
40 PRINT"Press SPACEBAR to start read-in
."
50 WHILE INKEY(47)=-1:WEND
60 OPENIN "!EVENS"
70 Total=0
80 FOR N=1 TO 26
90 INPUT #9,J
100 Total=Total+J
110 NEXT
120 PRINT:PRINT"TOTAL IS";Total
130 CLOSEIN
```

Fig. 7.11. Reading back the EVENS file using a FOR...NEXT loop.

which opens the !EVENS file. Now when we recorded the file "EVENS", we selected all the even numbers from 0 to 50, which is a total of 26 numbers. To read the same set back, then, line 80 uses a FOR...NEXT loop with 26 passes. The number total has been set to zero in line 70. Line 90 then inputs each number from the file, giving it the variable name of J, and line 100 adds this number to the total. At the end of the loop, line 120 prints the value of the total and the file is closed in line 130.

Suppose that you didn't know how many numbers were recorded? This makes the use of a FOR...NEXT loop impossible, because you wouldn't know what number of passes to use. As it happens, we can cope with this quite easily. The CPC464 filing system puts an end-of-

file code at the end of the last block of data that it records. Now this end-of-file code can be detected by using the word EOF. If EOF=0, the end is not yet nigh. If EOF=−1, then you have reached the end of the file. Our EVENS file can therefore be much more conveniently written as in Fig. 7.12. This time we use a WHILE...WEND loop in

```
10 CLS
20 PRINT TAB(18)"TOTALS"
30 PRINT:PRINT"Press PLAY key on recorde
r when the":PRINT"cassette is ready."
40 PRINT"Press SPACEBAR to start read-in
."
50 WHILE INKEY(47)=-1:WEND
60 OPENIN "!EVENS"
70 Total=0
80 WHILE EOF=0
90 INPUT #9,J
100 Total=Total+J
110 WEND
120 PRINT:PRINT"TOTAL IS";Total
130 CLOSEIN
140 PRINT"Press STOP key of recorder."
```

Fig. 7.12. A better method of reading back a file. This time, you don't have to know how many items are in the file.

place of FOR...NEXT. The WHILE condition is EOF=0, because while EOF=0, we have not yet reached the end of the data for this file.

Naming the names

Now that we have replayed and used a number file, it's time to start looking at some replaying methods for the file of names that we created earlier. When this file was created, each name was recorded, and the usual end-of-file marker would be placed on the tape. What we normally want to do is to place the names into an array, so that the computer can make use of the data. Using an array allows us to carry out tasks like placing the names into alphabetical order, for example. Not all uses call for an array, however. Suppose, for example, that you want to search the names for one beginning with the letter J. You could do this by using the program which is shown in Fig. 7.13.

The first few lines follow familiar patterns. When the program is run, it will allow you to find a name from the file simply by typing the

```
10 CLS:PRINT TAB(16)"NAMEFINDER":PRINT
20 PRINT"This program will find a name f
or you ":PRINT"from the NAMES file."
30 PRINT"Press PLAY when the cassette is
 ready":PRINT"and then the SPACEBAR."
40 WHILE INKEY(47)=-1:WEND
50 SPEED WRITE 1
60 INPUT"First letter of name ";Q$
70 OPENIN "!NAMES"
80 WHILE EOF=0 AND Q$<>LEFT$(N$,1)
90 INPUT#9, N$
100 WEND
110 PRINT:PRINT"Name is ";N$
120 CLOSEIN:SPEED WRITE 0
130 PRINT"Press STOP on recorder."
```

Fig. 7.13. Searching a file for one item – in this case, a name that starts with a given letter.

first letter of the name. You have to switch to SPEED WRITE 1 in line 50, because the NAMES file was recorded at this higher speed. When the faster speed has been used, you must be sure that the cassette is fully wound back to just before the start of the file. The faster system is less tolerant of starting the playback at a point where the tones have already been recorded! Line 60 asks you for a first letter of a name (don't forget the capital letter!). Line 70 opens the file for reading, and line 80 starts a loop that takes a name from the buffer and checks first for the end-of-file character, and then for the first letter of the name being identical to the letter that you requested. If the whole file has not been read, this part compares the first letter that you have selected with the first letter of the name. If you have searched through the whole list without finding this letter, then line 80 will respond by ending the program when the end-of-file marker is found. If, on the other hand, the name that you want is found, then line 110 will print it, and the search ends also. The WHILE...WEND loop is a very economical way of programming this type of search.

This works quite well with small amounts of data, but only if all the data is different. If one name is Mary and another is Margaret, then a request for M will give you whichever of these comes first on the file. You will have to alter the tests in the loop if you want the program to print every name which starts with a given letter. Many programs of this type can be dealt with more satisfactorily by using an array to hold all the data in the memory. The cassette data system is then being used as a store only, and all of the selection is being done in the

memory of the computer. You might wonder if there is any advantage here compared with having the data in DATA lines. There is, because the data can be created by one program, and used by several others. You can make the program that reads and uses the data a fairly short one, so that there is room for a lot of data in the large memory of the CPC464. The cassette data system, in other words, allows you to use short programs and lots of data.

Figure 7.14 shows how data from a program that has been put into the cassette data system can be read back into an array. If we are to

```
10 CLS:PRINT TAB(16)"NAMEFINDER"
20 PRINT:PRINT TAB(2)"Load the names fil
e as instructed."
30 PRINT"When the file is completely loa
ded,":PRINT"type the first letter of the
 names that"
40 PRINT"you want to see. Type 0 to stop
 the ":PRINT"action."
50 PRINT:PRINT"Wind the NAMES cassette t
o the start.":PRINT" Zero the counter."
60 PRINT"Press PLAY, then the SPACEBAR w
hen ":PRINT"ready to start"
70 WHILE INKEY(47)=-1:WEND
80 PRINT:PRINT TAB(10)"Please wait....."
90 SPEED WRITE 1
100 OPENIN "!NAMES"
110 J=0:WHILE EOF=0
120 INPUT#9,N$
130 J=J+1
140 WEND:CLOSEIN
150 CLS:PRINT:PRINT"PRESS STOP KEY."
160 PRINT:PRINT"Now rewind tape to start
 again, and":PRINT"press PLAY. Press SPA
CEBAR again"
170 PRINT"When ready."
180 WHILE INKEY(47)=-1:WEND
190 PRINT:PRINT TAB(10)"Please wait...."
200 OPENIN "!NAMES"
210 DIM Name$(J)
220 FOR N=1 TO J
230 INPUT #9,Name$(N)
240 NEXT:CLOSEIN:SPEED WRITE 0
```

Fig. 7.14. Reading a file into an array. The file is read once to find how to dimension the array, then again to obtain the items.

```
250 PRINT:PRINT"PRESS STOP KEY NOW"
260 FOR X=1 TO 2000:NEXT
270 CLS:PRINT:PRINT"Please type first le
tter of name.":Q$="X":M=0
280 WHILE Q$<>"0"
290 INPUT Q$:Q$=LEFT$(Q$,1):M=0
300 FOR X=1 TO J
310 IF Q$=LEFT$(Name$(X),1) THEN PRINT N
AME$(X):M=1
320 NEXT
330 IF M=0 AND Q$<>"0" THEN PRINT"Cannot
 find the name. Please try ":PRINT"anoth
er one."
340 WEND
350 PRINT"End of program"
```

Fig. 7.14. (cont.)

read items into an array, we find ourselves facing a problem. The problem, you see, is that we have to dimension the array so that it will hold all of the items. To do this we need to know how many items there will be. It's easy enough if we used an array to hold the items when we recorded. Suppose, for example, that we INPUT the items into an array Name$(J), instead of recording directly. We could then open the file, record the value of J with a PRINT#9,J, and then set up a loop to record the array items. If we didn't use an array and didn't count the items at the time when we recorded, what can we do? One way out is to keep a note of the number of items, and enter the number in response to a question in the replay program. For example, you could use:

INPUT"How many items ";N:DIM Name$(N)

to get your dimensioning. Another possibility is to use a counter in the recording program, such as:

INPUT Name$:N=N+1

and to record this number, using a filename that will relate it to the main program, on another piece of tape. For example, if the main program is filed as NAMESOFFRIENDS, the other one could be NUMBERFRIENDS. It may seem awkward, but it's a small price to pay for the sake of precise dimensioning. Precise dimensioning means that you will never get an error message because of an attempt to place too many items into an array. It also means that you are using the memory of the CPC464 in the most efficient way, and that

can make the difference between being able to use as many items as you need and being restricted to a lot fewer.

If, however, you have a file like our existing NAMES file, in which the number of names is not known, perhaps because names are being added as the file is updated, then we have to take drastic steps. This is another respect in which disk filing is very much superior to cassette filing, because a number can be recorded on a disk either before or after a set of data, and the number read before the other data if needed. Figure 7.14 shows one approach which is not too time-consuming even for fairly long files. In this program, all of the items are read, one by one, and counted, until the end-of-file marker is found. You are then asked to rewind the cassette. The number of items in the file is now known, and an array can be correctly dimensioned. The file can then be read again, this time placing the items into an array. The selection of names can then use a loop, because the file does not have to be read again.

Looking at the program in detail, lines 10 to 60 print instructions, and line 70 is the usual spacebar detector. In line 80, the 'Please wait' message is printed to remind you that you are waiting for the data to load. The fast speed is selected, because the original file was recorded at this speed, and variable J is zeroed. In the loop, each item is read, and the value of J is increased for each item. The file must be closed in line 140, because we shall want to open it again later. We then have to rewind the cassette to the zero mark again, and messages are printed as a reminder. The value of J is then used to dimension the array Name$ in line 210, just before the names are read from the file. Since we know the number of names, we can then use a FOR...NEXT loop in lines 220 to 240 to read the names into the array. Once the names are read, we can operate the NAMEFINDER action in lines 280 to 350. Note that we need a dummy value for Q$ before starting the finding loop. If a name is found, variable M is set to 1. If the name cannot be found starting with the specified letter, M=0 and the message in line 330 is printed. When you press the 0 key to end the finding action, this also causes M=0, and the message in line 330 is suppressed by making the IF condition M=0 AND Q$<>"0". If Q$="0", then the message is *not* printed. It looks a lot neater that way!

Making amendments

Let's be clear from the start that you cannot alter a file that is recorded

on the cassette data system. What you can do, though, is to read in a file, make some alterations to it, and then re-record it. Since the CPC464 uses a built-in cassette data system you can take advantage of this feature to record the new file *using the same name* on another part of the cassette or on a different cassette. Using the same name allows the updated file to be read by the same programs. This is not so easy to arrange with a disk system. The technique which is shown in Fig. 7.15 follows the program of Fig. 7.14 very closely.

```
10 CLS:PRINT TAB(10)"FILE UPDATE"
20 PRINT:PRINT TAB(2)"Load the names fil
e as instructed."
30 PRINT"When the file is completely loa
ded,":PRINT"follow the instructions abou
t":PRINT"re-recording it."
40 PRINT"You can now add items to the fi
le-":PRINT"Type 0 to end entry."
50 PRINT:PRINT"Wind the NAMES cassette t
o the start.":PRINT" Zero the counter."
60 PRINT"Press PLAY, then the SPACEBAR w
hen ":PRINT"ready to start"
70 WHILE INKEY(47)=-1:WEND
80 PRINT:PRINT TAB(10)"Please wait....."
90 SPEED WRITE 1
100 OPENIN "!NAMES"
110 J=0:WHILE EOF=0
120 INPUT#9,N$
130 J=J+1
140 WEND:CLOSEIN:J=J-1
150 CLS:PRINT:PRINT"PRESS STOP KEY."
160 PRINT:PRINT"Now rewind tape to start
 again, and":PRINT"press PLAY. Press SPA
CEBAR again"
170 PRINT"When ready."
180 WHILE INKEY(47)=-1:WEND
190 PRINT:PRINT TAB(10)"Please wait...."
200 OPENIN "!NAMES"
210 DIM Name$(J)
220 FOR N=1 TO J
230 INPUT #9,Name$(N)
240 NEXT:CLOSEIN
250 PRINT:PRINT"PRESS STOP KEY NOW"
```

Fig. 7.15. Updating a file. This has to be done by reading the file, making changes, and then recording again on a different part of the tape.

```
260 FOR X=1 TO 2000:NEXT
270 CLS:PRINT:PRINT TAB(5)"PLEASE PREPAR
E TO RE-RECORD THIS FILE."
280 PRINT:PRINT"Find a clear place on th
e cassette, or":PRINT"use a new cassette
. Zero the counter"
290 PRINT"again, and follow the instruct
ions."
300 PRINT:PRINT"Press REC and PLAY, then
 the SPACEBAR":PRINT"when you are ready.
"
310 WHILE INKEY(47)=-1:WEND
320 OPENOUT "!NAMES"
330 FOR N=1 TO J
340 PRINT#9,Name$(N):NEXT
350 CLS:PRINT:PRINT TAB(16)"NEW ENTRY"
360 PRINT:PRINT"Type the names that you
want to add":PRINT"to the file now. Type
 0 to end entry."
370 Name$="x":WHILE Name$<>"0"
380 INPUT Name$
390 PRINT#9,Name$
400 WEND
410 SPEED WRITE 0
420 CLOSEOUT
430 PRINT"END OF PROGRAM."
```

Fig. 7.15. (cont.)

Lines 10 to 260 are virtually the same as the corresponding lines in Fig. 7.14, except for the instructions. These lines dimension an array, and then fill it with names. Line 140, however, now contains J=J−1. This avoids using the last item in the array, which is the terminator, x or 0, whichever was used in the program that created the file. In this way, we have a continuous file with an x or 0 at the end, rather than a file which has these marks scattered through it at each place where the file had ended previously.

Lines 270 to 340 then prepare to re-record this file. I say prepare, because unless the names file is a very long one, nothing appears to happen in lines 320 to 340. This is because the file is simply delivered to the buffer, not recorded at this stage. You are then asked to type more names. While you are doing this, the buffer may fill, and names will be recorded. During this time, you will not be able to type more names. If you type only a few names, however, the buffer will not fill,

and you will not hear the cassette system recording names until you have typed the zero that ends the entry. This completes the new file. If you want to check it, then RUN again, and when you are asked to type more names, press ESC twice. Now type the line:

```
FOR Z=1 TO J:?Name$(Z);" ";:NEXT
```

and ENTER. You will then see all of the names being listed on the screen. If you want to see the names listed in a more orderly way, or selected by letter, or sorted alphabetically, then you will have to write a piece of program for yourself!

Chapter Eight

Windows and Other Effects

Working with a computer is never dull, but a lot can be done to make things more interesting. One of the special effects that is now available on modern computers is *windowing*, and in this chapter we'll take a look at this effect and a few others that produce screen displays that are rather more interesting than plain text or figures. Windowing looks like a good place to start, because unless you have used one of the few machines that allows this effect to be programmed easily, you probably haven't tried this for yourself.

A 'window' simply means a section of screen which can be used independently of other sections and even independently of the screen as a whole. A window can have text printed on it, and can be scrolled or cleared irrespective of what is happening on other parts of the screen. Try this out. Fill the screen with a listing, or any other text. Now type as a direct command:

WINDOW 15,25,2,6 (then ENTER)

Now use CLS. You'll see that only a small piece of the screen, which is the window, will clear. Try typing a short program into this window. You will find that it behaves as if it were the only screen you have. It automatically selects a new line when it needs to and it scrolls when it needs to, just like a full screen. Meanwhile, the rest of the screen remains unaffected. A CLS will clear only the window, not the rest of the screen. To get back quickly to normal, type MODE 1 (ENTER).

If that whetted your appetite, it's time to look at how we can control this window business for ourselves. WINDOW must be followed by four numbers. These are like the LOCATE numbers, and they specify, in order, the left, right, top and bottom positions of the window. When you use this type of WINDOW command, all of your printing will be to this window. It uses stream #0, which is the normal stream for PRINT and other commands of this type. You can, however, start the WINDOW specification with a stream number,

like #2. When you do this, the window is used only when you have an instruction like PRINT #2,"WINDOW".

Figure 8.1 shows this type of window in action. The whole screen is cleared, and then a window is connected to stream #2 by using the command:

WINDOW#2,10,30,1,5

This means that stream number 2 is being connected to a screen window. The numbers that follow #2 and the comma specify both the size and position of this window. Its left-hand side will be at column

```
10 CLS
20 WINDOW#2,10,30,1,5
30 WINDOW#3,12,28,10,15
40 PRINT#2,"This is window #2 in use"
50 GOSUB 150
60 PRINT#3,"Look at the text in window #
2 - this is window #3"
70 GOSUB 150
80 CLS #2
90 GOSUB 150
100 CLS#3
110 GOSUB 150
120 PRINT#2,"Window #2"
130 PRINT#3,"Window #3"
140 END
150 FOR N=1 TO 2000:NEXT:RETURN
```

Fig. 8.1. Creating a window on stream #2.

10, and its right-hand side at column 30. Its top is on line 1, and its bottom is on line 5. The effect of this command, then, is to allow you to use commands like CLS #2 and PRINT #2 to clear this window or print to it. This action is selective. Unless you close the stream or allocate it to something else, this window will be controlled by using stream #2. Another big difference is that the whole screen can still be cleared and printed on. Clearing the whole screen will also clear this window, and printing on the whole screen will print over the window, though you can clear the window again with a CLS#2.

In the example, then, the first window is created in line 20, and another window further down the screen is created in line 30. To prove that these windows exist, line 40 prints a phrase onto window #2. You can, incidentally, put LIST#2 into your program to list your program on this window, but the machine will not execute any

program lines following a LIST command in the program. Then there is a pause caused by the subroutine at line 1000. After the pause, more text is printed to the other window, #3. The rest of the program then illustrates how the windows can be cleared and printed on separately. Notice, however, at the end of the program, how the 'Ready' message and the cursor appears at the top left of the screen. The main screen, whose stream number is 0, will always override any windows.

Some more window magic can be worked by using another window command, WINDOW SWAP. Suppose, for example, that you are using a program which requires you to type in data and record it on cassette. This is the sort of thing that we were looking at in Chapter 7. You could use a window somewhere in the middle of the screen for your entry, and reserve another one for messages. You might want, for example, the usual cassette messages to appear somewhere else. If you define a window near the bottom of the screen, perhaps using #3, you can then carry out:

```
WINDOW SWAP #0,#3
```

to make everything that would normally go to the main screen, such as cassette messages, go to window #3 instead. Figure 8.2 shows this in (simulated) action. The actual full cassette routines have been

```
10 CLS
20 WINDOW#2,2,39,5,10
30 WINDOW#3,2,39,25,25
40 PRINT#3,"Please type names. Type X to
 end."
50 WHILE Name$<>"X" AND Name$<>"x"
60 INPUT Name$
70 WEND
80 WINDOW SWAP 0,3
90 OPENOUT "NAMES"
100 PRINT#9,"Name$"
110 CLOSEOUT
120 REM: USE ESC TO STOP
```

Fig. 8.2. Using WINDOW SWAP to prevent messages from appearing on the full screen.

omitted because we don't want to waste time on them at the moment. You can place a cassette in the recorder, and use PLAY only when you are prompted to record – this will avoid wasting tape on a recording. There's another way of doing this sort of thing,

incidentally. You will find that you normally can't press the REC key when there is no cassette in the machine. If you push your finger hard against the back left-hand corner of the cassette slot, however, you will find that you can then push the REC key down. There's a small lever at this position which is used to detect whether or not a cassette is present, and your finger can push it enough to fool it. It's a useful tip if you want to experiment with programs that record on cassette.

Getting back to the program, the two windows that are defined are #2 and #3. The #3 window is of just one line, near the bottom of the screen. In this way, each message is seen separately, with no confusion from earlier messages. You have to make sure, of course, that none of your messages needs more than one line! The program is straightforward until you get to line 80. This swaps windows 0 and 3 – note that you don't have to use #0 and #3, and you will get a syntax error message if you do. From then on, all of the screen messages, which otherwise would appear at the top of the main screen, appear on this window. It's a very neat and useful trick for keeping your messages away from your other text.

Write it in colour

It's time now to look at the colour instructions of CPC464. There are four particularly important ones, which use the instruction words BORDER, PAPER, PEN and INK. We'll start with the simple one, BORDER. BORDER can be used with one number, or with two. If you use one colour number, you will get a border round your main screen area which is of one steady colour. The colour numbers are listed in Fig. 8.3. If you use *two* colours with BORDER, the border will flash between these two colours.

Figure 8.4 gives you a taste of all this. The program first of all defines a window that is going to be used to display the colour numbers on the right-hand side of the main screen. It then starts a loop which will run through all of the available BORDER colours. Each colour is held on the screen by using a time delay. Unlike previous time delays, this is obtained by using the built-in timer of the machine. TIME is a number which starts from zero when you switch the machine on, and which is incremented 300 times per second. Now if you set up the subroutine that is shown in lines 150 to 170, you are setting a variable equal to the value of TIME at some instant. Each second later, TIME will have increased by 300, so a WHILE... WEND loop that waits for TIME to become 600 more than START

Number	Colour
0	Black
1	Blue
2	Bright blue
3	Red
4	Magenta (red light + blue light)
5	Mauve
6	Bright red
7	Purple
8	Bright magenta
9	Green
10	Cyan (blue light + green light)
11	Sky blue
12	Yellow
13	White
14	Pastel blue
15	Orange
16	Pink
17	Pastel magenta
18	Bright green
19	Sea green
20	Bright cyan
21	Lime green
22	Pastel green
23	Pastel cyan
24	Bright yellow (gold)
25	Pastel yellow
26	Bright white

Fig. 8.3. The colour numbers. The numbers are arranged in order of increasing brightness, as they would appear on a black and white receiver or monitor.

is going to cause a delay of 2 seconds. It's simple and elegant, and easier to get precise times than a FOR...NEXT loop. Meanwhile, back at the program, after running through the single colours, the program goes through the flashing colours. The rate of flashing is set by another variable, and you can alter it for yourself. While the border continues flashing at the end of the program, try typing SPEED INK 10,10. When you press ENTER, you will soon see the flashing rate speed up. Now type SPEED INK 20,100, and watch the effect now. The first number measures the time for one colour, the

```
10 REM LIST BEFORE RUNNING
20 WINDOW#2,35,40,5,5
30 FOR N=0 TO 26
40 PRINT#2,N
50 BORDER N
60 GOSUB 150
70 NEXT
80 GOSUB 150
90 FOR N=1 TO 26
100 PRINT#2,N;",";27-N
110 BORDER N,27-N
120 GOSUB 150
130 NEXT
140 END
150 START=TIME
160 WHILE TIME<START+600:WEND
170 RETURN
```

Fig. 8.4. Displaying the BORDER colours, with TIME used to make a delay.

second number measures the time for the second colour. The numbers are fiftieths of a second (for the UK machine, sixtieths for the US version), so that a number 50 (UK) should give you a one-second burst of one colour. Be careful how you use this command – *some flashing rates of about 8 per second can be harmful to anyone who is subject to epilepsy.*

Quite apart from being unpleasant, a flashing border can also be a nuisance. You will see that as the border flashes, the size of the picture on your monitor or TV will change. This is because these units are not designed to cope with flashing pictures – a monitor which could do this would be too expensive for most of us (allow something like £800 if you are thinking of saving for one!). To stop your border flashing, type BORDER 1 and ENTER this. A MODE change does not affect the BORDER.

The next items are PAPER, PEN and INK. The names tell some of the story but not all. In particular, if you have come to the CPC464 after serving an apprenticeship on a Spectrum, you will find these items slightly confusing. If you are completely new to these terms, you still find them confusing – so stick with me! The one to start with is INK.

INK means a colour and, for each mode, you have a limited range of INKs that you can use. MODE 0 allows you to use up to 16 different colours of INK on the screen. This is *any sixteen selected*

from the list of 27 possible colours in Fig. 8.3. MODE 1, the normal text mode, allows you to use up to four INK colours. Once again, of course, you can choose which four of the 27 available colours you want to use. MODE 2, the high resolution, 80 characters per line mode, allows only two INK colours, any two of the twenty-seven that you may like to use. A good way to think of INK is that you have a paint box with a limited number of pots. For MODE 1, for example, you have four pots. You can put any of your twenty-seven colours in each pot – but you are allowed to paint only with the colours in the pots. If you want to use other colours, you have to refill the pots!

Now this is where the CPC464 starts to be very different from some other machines. In MODE 1, your INK *numbers* are 0,1,2 and 3. I've used *numbers*, not *colours*, here, because these numbers are *not* the colour numbers. Coming back to our comparison with a paintbox, these numbers are just the numbers of the pots. We can use the INK command to decide which colours to put into the pots. The form of the command is, for example, INK2,7. This will fill pot 2 with ink whose colour is 7 (purple). When you start up in MODE 1, the pots are filled for you. Pot 0 is filled with blue, and this is the pot that decides the colour of your screen background. Pot 1 is filled with gold paint (actually called bright green), colour 18. This is the pot that is used for text colour. Pot 2 is filled with colour 20, bright cyan, and pot 3 is filled with colour 6, bright red. Figure 8.5 reminds you of these settings, which are useful to know.

Mode 0 and Mode 1:

Inkpot 0	...	colour 1	...	blue
Inkpot 1	...	colour 24	...	bright yellow
Inkpot 2	...	colour 20	...	bright cyan
Inkpot 3	...	colour 6	...	bright red

(In Mode 2, only colours 1 and 24 are used initially).

Fig. 8.5. The standard settings for INK.

How do you change these colours? The answer is by using the INK command. In MODE 1, you can use INK colours of 0 to 3 (you can use numbers greater than 3, but you will just get the same range used over again). To allocate ink pot number 1 to colour 15, for example, you simply type: INK 1,15. Don't forget the space between the 'K' of INK and the number. If you type *two* colour numbers, separated by a comma, your pot contains flashing ink! The colour that you use will be flashing between the two colour numbers that you have specified.

For example, INK 1,3,7 will give you an ink that flashes between colours 3 and 7. You can alter the rate of flashing by using SPEED INK as before. It's rather like these old jokes about striped paint!

When you switch on, the computer puts you into MODE 1, and allocates the 'default' ink pot colours as I have shown above. It also decides which of the pots shall be used for background and which for text. The background, or PAPER, uses INK number 0, and the text, or PEN, uses INK number 1. Now I must emphasise yet again, particularly if you are a former Spectrum owner, that these INK numbers *are not colour numbers*, but just the inkpot numbers. The colours that appear on the screen depend on which pot you use, and what colour of ink was put into it. By using the PAPER command, you can alter the number of the inkpot you use for background. By using PEN, you can alter the number of the inkpot that you use for text.

Figure 8.6 illustrates PAPER and PEN in use. An instruction such as PAPER 2 does not, by itself, cause colour to appear. If we print on

```
10 BORDER 0
20 FOR N=0 TO 3
30 PAPER N:CLS
40 GOSUB 140
50 NEXT
55 PAPER 0:CLS
60 FOR N=0 TO 3
70 PEN N
80 GOSUB 170
90 GOSUB 140
100 NEXT
110 GOSUB 140
120 PEN 1
130 END
140 START=TIME
150 WHILE TIME < START+600:WEND
160 RETURN
170 PRINT"This is some text to show the
effect of colour of PEN";N
180 RETURN
```

Fig. 8.6. Using the PAPER and PEN commands.

the screen following a PAPER 2 instruction, the background for our printing will appear in bright cyan, but only for the part on which we have printed. To make PAPER 2 colour a complete screen, we have

to follow it with a CLS instruction. That's illustrated in line 30. The second part of the program uses PAPER 0, which is dark blue, because that's the colour of paint in pot number 0, and prints in different PEN colours. You can see from the results of this program that if you want to keep your text clear, you have to use contrasting colours. Colours whose brightness values are very close to each other will never give enough contrast to make a good display. My own preference is to have the PAPER colour dark, and the PEN colour light, because the opposite, a light screen with dark paint, can appear to flicker very irritatingly. Don't expect the letters to appear in very good colours if you are using a TV receiver, because colour TV sets are not very good at displaying colour in small chunks. Add to that the fact that 90% of the male population is partially colour blind, and you'll see that the most impressive colour displays are the ones that use strong colours in big areas, displayed on a monitor.

When you run the program, you'll see that the text line for PEN 0 never appears. That's because inkpot 0 is the one that has been used for background, paper, colour. If you change the background colour before you write the text, however, this line will appear, and one of the other ones will not. I have used MODE 1 for illustrations here because the choice of four inkpots is a convenient one. Remember, however, that you have the choice of two inkpots, 0 and 1, in MODE 2, and of 16 inkpots numbered 0 to 15 in MODE 0.

```
10 BORDER 0
20 WINDOW#1,1,40,3,5
30 WINDOW#2,1,40,7,10
40 WINDOW#3,1,19,11,24
50 WINDOW#4,20,40,11,24
60 A$="This is a test phrase to show off
 the windows."
70 PAPER#1,0
80 PAPER#2,1
90 PAPER#3,2
100 PAPER#4,3
110 PEN#1,3
120 PEN#2,2
130 PEN#3,1
140 PEN#4,0
150 CLS
160 FOR N=1 TO 4
170 PRINT#N,A$:NEXT
```

Fig. 8.7. PAPER and PEN used in different windows.

The PEN and PAPER commands can be used to specify the colours that are used in windows. Figure 8.7 illustrates this, and also shows that not all colour combinations are good ones! When you want to use different PEN and PAPER colours in the windows, take a good look at the colours first, and check that they are really compatible. Sometimes the combination of two shades of one colour can be quite effective, but two light colours or two dark colours will never be very satisfactory, especially if you are using a colour TV for display. Some of these combinations are not satisfactory even on a monitor.

Some prettier printing

The best place to start on our next bit of exploration of special text effects is with INK again. Take a look for starters at the program in Fig. 8.8 which illustrates flashing text. The flashing is obtained by

```
10 CLS
20 INK 2,6,8
30 INK 3,12,19
40 PEN 2
50 PRINT:PRINT"ATTENTION PLEASE!"
60 PRINT:PRINT:PRINT
70 PEN 3
80 PRINT"WARNING- DANGER!"
90 PEN 1
```

Fig. 8.8. Flashing text obtained by using flashing ink!

using 'flashing ink' in pots 2 and 3 in lines 20 and 30. When we use PEN to dip into these inks, the result is flashing text as you can see when lines 50 and 80 run. Line 90 switches back to PEN 1, the normal inkpot for text, so that the 'Ready' prompt isn't flashing as well. Note that the letters keep flashing for as long as they are on the screen. You can change the flashing colours instantly, or change to a non-flashing colour, by altering the INK. Figure 8.9, for example, shows how a title can be made to flash for a few seconds, and then revert to another steady colour. This is a good way of making sure that flashing achieves its effect of drawing attention to something without boring the user.

Another useful feature CPC464 allows you to carry out effects like underlining, foreign accents, and so on. This, and a lot of other

```
10 INK 2,24,6
20 CLS
30 PRINT:PEN 2
40 PRINT TAB(14)"FLASHING TITLE"
50 FOR N=1 TO 3000:NEXT
60 INK 2,20
70 PEN 1
80 PRINT:PRINT"Now we can place text on
the screen":PRINT"without being distract
ed!"
```

Fig. 8.9. Changing from flashing to steady display by changing INK.

special effects that we shall look at later, is enabled by one of the 'non-printing' codes of the CPC464. These are the codes 1 to 31, and we have already used code 21 (turn off screen), 6 (turn on screen) and 8 (move cursor one step back) already. Your manual carries a full list of these effects in Chapter 9, page 2. To underline text, as illustrated in Fig. 8.10, we need to alter the way that printing is done. Normally,

```
10 CLS
20 PRINT:PRINT:PRINT
30 PRINT"THIS IS IMPORTANT";
40 S$=STRING$(18,8)
50 PRINT S$;
60 PRINT CHR$(22)+CHR$(1);
70 PRINT"____ __ _________"
80 PRINT:PRINT
90 PRINT CHR$(22)+CHR$(0)
```

Fig. 8.10. Underlining, using CHR$(22) to prevent one character from erasing another. Note also how STRING$ can be used with an ASCII code in place of a character in quotes.

when we print, the new print completely replaces the old. By using CHR$(22)+CHR$(1), we can alter this, superimposing new text on to old. In the program, then, the phrase in line 30 is printed, and the string S$ is defined as being of 18 back steps. This will place the cursor back at the start of the phrase. We then set the superimpose with CHR$(22)+CHR$(1), and print a set of underlines. In this state, the underlines are added – they do not replace the text. If we didn't cancel this, however, it could lead to odd effects if we wanted to type new text, or edit, and we have to turn it off using PRINT CHR$(22)+CHR(0). A lot of the commands of this type need two (or more) CHR$ terms like this, and this is how they are joined up to make the command complete. A lot of these commands carry out the

same actions as BASIC commands, and the point of having them in this form is to allow the actions to be carried out by the use of numbers. If any of you have used the BBC Micro, you might recognise this as equivalent to the BBC's 'VDU' commands.

Odds and ends

There are a lot of commands on this machine which don't fit into neat categories, but which are very useful. Take for example, LOWER$ and UPPER$ (Fig. 8.11). Each of these has to be followed by the

```
10 CLS:PRINT:PRINT TAB(10)"UPPER-LOWER"
20 PRINT:PRINT
30 FOR N=1 TO 5
40 INPUT "NAME- ";A$
50 PRINT UPPER$(A$),LOWER$(A$)
60 NEXT
```

Fig. 8.11. Converting case with UPPER$ and LOWER$.

name of a string variable, within brackets. When you use UPPER$, all the letters of the string variable will be converted to upper-case letters. When you use LOWER$, all of the letters are converted to lower-case. This is useful if you have entries which are mixed, some lower-case and some upper-case, and you want to convert them all to the same case. Why would you want to do that? Well, one good reason is alphabetical sorting. As we have seen, sorting puts words in order of ASCII codes. The ASCII codes for lower-case letters are all larger than the codes for upper-case letters, however. The computer would arrange the words ALL, act, BIN, bell as ALL, BIN, act, bell; whereas we would probably prefer to have act, ALL, bell, BIN. By changing all of the cases, a more logical sort arrangement can be achieved. Incidentally, while we're on the subject of string sorting, there's a useful command, FRE. If you type, or have in your program, PRINT FRE(0), then the computer will tell you how much memory you have left. I wish my other computer allowed me as much! If you use FRE(""), the computer will tidy up its string storage. This is important, because during a string sort, the memory becomes cluttered with unused spaces where strings have been created for swapping purposes. FRE("") releases this space, and its use during a long sort program can make the program run much faster and more efficiently.

Want a space to be printed? There's a useful string quantity, SPACE$, which will make spaces for you. For example, S$= SPACE$(40) will give you a line of 40 spaces when you print S$. It's easier than forming a loop to fill a string with spaces. A more 'way-out' command is WRITE. If you use WRITE "PAPER" in place of PRINT "PAPER", then the *quotemarks are printed as well as the word*! The same happens if you use A$="PAPER" and then WRITE A$. It's useful if you want to have quotes appearing in a phrase, because you can't normally do this when you use PRINT.

One last one. The keys of the CPC464 repeat when you hold them down. For many purposes, the speed that comes as standard is ideal, but you sometimes want to alter it. You might, for example, find that the rate is too slow for some games purposes, or too fast in some data programs. In some cases, you might want the rate of repeat to be so slow that no repeat would normally happen. If, for example, you have two INKEY$ steps, one after the other, then holding down a key too long at the first step might cause a choice to be made in both steps, getting you to the wrong part of a menu. You can choose your key repeat speed by using SPEED KEY, followed by two numbers. The first number measures how long you have to hold the key down before it starts repeating. The second number decides how long it will be between repetitions. The range of numbers is 0 to 255, and if you make the numbers too short, you may find odd effects, like double letters appearing when you have hit a letter just once. Try typing, first with SPEED KEY 2,2, then with SPEED KEY 255,255 (if you can type it!). The first example makes it almost impossible to type normally, the second just about cuts out repetition for all practical purposes.

Chapter Nine
Starting Graphics

Any modern computer is expected to be able to produce dazzling displays of colour and other special effects. The CPC464 is no exception, and in this chapter, we'll start to look at some of the graphics effects that are possible. To start with, we have to know some of the terms that are used. The first, *graphics*, means pictures that can be drawn on the screen, and all modern computers have instructions that allow you to draw such patterns. In connections with these patterns, you'll see the words 'low resolution' and 'high resolution' used. Resolution isn't such an easy term to explain. Imagine that you are creating pictures on a paper sheet of about eleven inches across by eight inches deep – that's roughly the size of a TV screen that is described as being a 14-inch screen (it's about 14 inches diagonally!).

Now if you are asked to create the pictures by using rectangles of coloured paper, you are dealing with picture-making in a way that is very similar to the way that the computer operates. Suppose that you are allowed only 936 pieces of paper, of such a size that all 936 will fill the screen. You couldn't draw very finely detailed pictures with so few large pieces, and this is what we mean by low resolution. Several computers provide little better than this. On the other hand, if you were provided with pieces so small that you would need 32000 of them to fill an entire screen size, you could produce very much more detailed pictures. This is what we mean by high resolution. The CPC464 has high resolution graphics commands available, and the figure of 32000 pieces that I have used corresponds to the size of the blocks that the CPC464 can use in its *lowest resolution mode*, Mode 0. In its highest resolution mode, the CPC464 can work with 128000 pieces on the screen. To create these graphics pictures, a computer has to make use of memory. A lot of computers make use of the same memory as the owner uses for programs. Because of this, a lot of computers that are said to have 64K or 32K of memory can turn out

to have very little left for you to use, perhaps as little as 6K! This is because so much of the memory is being used in servicing the graphics display and the other operating systems. The CPC464 does not steal your program memory for graphics. Instead, it has a reserved (*dedicated*) piece of memory that is used for graphics, and the 43533 bytes (42.5K) of program memory is not grabbed for graphics use.

Up until now, we have been working with what is called the 'text screen', which is the normal arrangement of screen that we use. This text screen can be used only for text, meaning characters that are put into place by the PRINT instruction. As we shall see, there are other instructions that can be used with 'graphics screens', and we'll be looking at these later.

Keyboard graphics

Some graphics shapes, that are illustrated in Fig. 9.1, can be obtained by pressing keys on the keyboard. The difference is that you have to

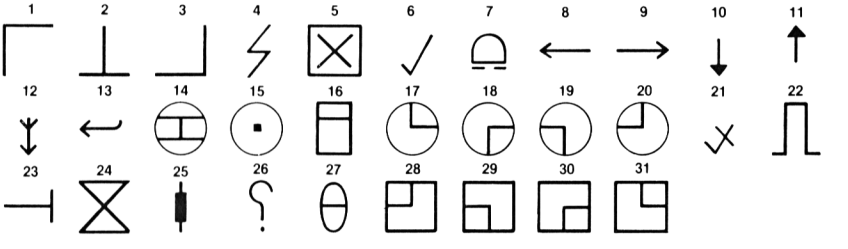

Fig. 9.1. The graphics shapes that can be obtained by pressing keys, with their ASCII codes.

press the CTRL key as well. These graphics characters *cannot*, however, be printed in the same way as you print words, by using the PRINT command, followed by a quote, then typing the graphics characters, then ending with another quotemark. If you want to use the characters to make fancy underlining, or to provide shapes to identify menu choices, you have to use a different method to achieve this. The alternative way is by using character code numbers.

The character codes

The alternative method, which allows us a lot more scope for illustration is to use the ASCII codes for the characters. Now there

are two sets of ASCII codes that produce graphics shapes. One set uses the ASCII numbers 128 to 255. These are shown in Appendix III of the CPC464 manual as well, but the program in Fig. 9.2 will remind you of them. There are several letters of the Greek alphabet in

```
10 CLS:N=127
20 FOR J=1 TO 128
30 PRINT CHR$(N+J);" ";
40 IF J/20=INT(J/20) THEN PRINT
50 NEXT
```

Fig. 9.2. A program that will show you the graphics characters for codes 128 to 255.

this set, which will be useful for scientific and technical use. The program is arranged to print the shapes separated from each other, and you might like to work out how this has been achieved. The other graphics shapes use ASCII codes in the range 1 to 31. The snag here is that these codes are also used to achieve other effects, and you have to know how to switch them from one use to the other. The secret is CHR$(1). If you precede any of these codes with CHR$(1), then the shape will be printed *without* the other effects that you might expect. For example, you know that PRINT CHR$(8) will move the cursor back; but PRINT CHR$(1)+CHR$(8) will simply print a shape. The program in Fig. 9.3 reveals these shapes, several of which are useful in business and educational programs as well as in games.

```
10 CLS
20 FOR N=1 TO 31
30 PRINT N;" ";CHR$(1)+CHR$(N);"  ";
40 IF N/5=INT(N/5)THEN PRINT:PRINT
50 NEXT
```

Fig. 9.3. A program which reveals the graphics characters for codes 1 to 31. Note that a simple PRINT CHR$(X), when X is between 1 and 31, does *not* produce one of these shapes.

You can do some ornamental work with these CHR$ shapes if you use one of the grids in the Manual for planning. The grids for Mode 1 (Appendix VI, page 2 in the Manual) shows 40 squares across the screen and 25 down because this is the number of character positions on the Mode 1 screen. If you use the Mode 0 screen, the characters will be larger, because there are only 20 characters per line in this Mode. If you use Mode 2, you will be able to squeeze 80 characters into a line. Note, however, that the number of lines stays constant. This means that the characters always have the same size top to

bottom, only their width changes. The shapes therefore look rather different on the three Modes. Mode 0 shows 'Cinemascope' characters, Mode 2 shows slimmed-down characters! Each square in the planning grid is the position for a character, and if you draw what you want on a piece of tracing paper placed over this grid, then you can plan what the shape will look like on the screen. There are three ways of programming this. One is to print each line of shapes separately. Another way is to print in a loop, using code numbers that are stored in a DATA line. A third way is to place all of the characters into a string, just as you can type words into a string.

Yes, an illustration would help. Figure 9.4 shows a design for a 'logo', an identifying mark for a firm, perhaps. It consists of the letter

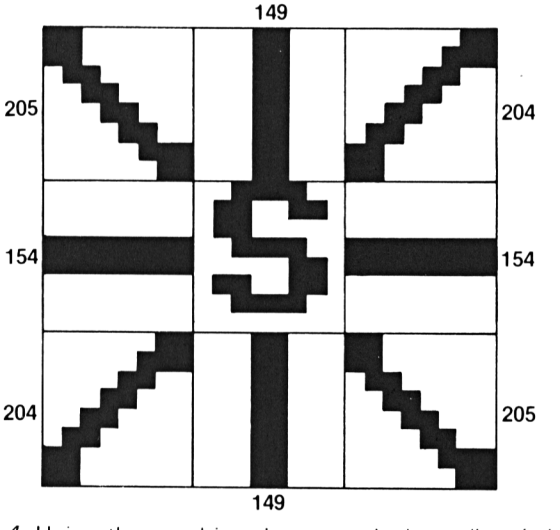

Fig. 9.4. Using the graphics shapes to design a 'logo' shape.

'S' with radiating lines. The letter 'S' is easy enough, it's CHR$(83), but the other characters are taken from the set shown in the manual. The plan has been drawn by tracing over the shapes in the Manual. Now you can simply write a program which prints each CHR$ value in the right place, as Fig. 9.5 shows. This works, but it's clumsy programming. Now take a look at Fig. 9.6. This may not look neater to you – it needs more lines, for example, but it is much better. There is only one PRINT CHR$(D%) instruction, instead of three lines of them. Two loops are used, one for each line of characters, and another loop for each column. All of the number variables are integer variables (using the % sign), so that the program can run fast. In addition, the position of the logo is decided by using the LOCATE

```
5 PAPER 3:PEN 2
10 CLS:PRINT:PRINT
20 PRINT TAB(5) CHR$(205)+CHR$(149)+CHR$
(204)
30 PRINT TAB(5) CHR$(154)+CHR$(83)+CHR$(
154)
40 PRINT TAB(5) CHR$(204)+CHR$(149)+CHR$
(205)
50 PRINT:PRINT
```

Fig. 9.5. A simple program to print the shape of Fig. 9.4.

```
10 PAPER 3:PEN 2:CLS
20 FOR J%=1 TO 3:LOCATE 18,J%+10
30 FOR N%=1 TO 3
40 READ D%:PRINT CHR$(D%);
50 NEXT:PRINT:NEXT
60 WINDOW#2,1,40,24,25
70 WINDOW SWAP 0,2
100 DATA 205,149,204
110 DATA 154,83,154
120 DATA 204,149,205
```

Fig. 9.6. Using loops to read graphics characters.

instruction, also in the loop. The advantage of this method is that you can see the data clearly, and it's easy to alter the data, while keeping the program the same. Note that I have used a window, swapped with #0, to make the 'Ready' prompt and the cursor appear at the bottom of the screen, instead of just under the logo. You will have to do another WINDOW SWAP 0,2 after running this program to get your listing on the full screen again.

Figure 9.7 illustrates an even better method, however. It starts by defining a string called B$. This consists of three characters whose

```
10 PAPER 2:PEN 3:CLS
20 GR$="":B$=STRING$(3,8)+CHR$(10)
30 FOR J%=1 TO 3:FOR N%=1 TO 3
40 READ D%:GR$=GR$+CHR$(D%):NEXT
50 GR$=GR$+B$:NEXT
60 LOCATE 18,12:PRINT GR$
70 DATA 205,149,204,154,83,154,204,149,2
05
```

Fig. 9.7. Assembling graphics characters into a string, which can be printed with one command.

code is 8, the back-space, followed by code 10. If you look this up in the Manual, you'll see that 10 is the 'cursor down' character. The effect of printing B$, then, will be to put the cursor three places to the left and one line down. In line 20, also, the string GR$ is equated to a blank. The next thing is to start two loops, one for the lines, another for the columns. After a line of data has been read and added to GR$ in line 40, B$ is added. This will cause the cursor to move down one line and three spaces left. The total effect, then, is to print three characters, and then move the cursor to the correct position in the next line. Each line is added to the string, and then the complete string is printed in line 60.

The great advantage of the method that is illustrated in Fig. 9.7 is that the shape can be printed anywhere on the screen without anything special having to be added to the program. Any PRINT GR$ instruction will print the shape, placed wherever the cursor starts out. You have to be careful, of course, that you don't place the cursor too far over to the right, or too near the bottom of the screen. Armed with this ability to produce patterns, let's see now how we can make them appear with some animation. This is illustrated in Fig. 9.8. The method is to print the shape at some position which has been

```
10 PAPER 2:PEN 3:CLS
20 GR$="":B$=STRING$(3,8)+CHR$(10)
30 FOR J%=1 TO 3:FOR N%=1 TO 3
40 READ D%:GR$=GR$+CHR$(D%):NEXT
50 GR$=GR$+B$:NEXT
60 A$=STRING$(3,32):SP$="":SP$=A$+B$+A$+
B$+A$
70 FOR N=1 TO 18
80 LOCATE N,12
90 PRINT GR$:GOSUB 1000
100 LOCATE N,12
110 PRINT SP$:GOSUB 1000
120 NEXT:LOCATE N,12:PRINT GR$
130 DATA 205,149,204,154,83,154,204,149,
205
140 END
1000 FOR K=1 TO 20:NEXT:RETURN
```

Fig. 9.8. Simple animation on the text screen.

determined by LOCATE. The shape is left on the screen for a short time, and then erased. After another short interval, the LOCATE

position is then shifted to the next place, and the process is repeated. If the time delay is short, as it is in this example, and the movement small, which in this case it is *not*, then the illusion of movement is good. Animation is not, however, ideally suited to the text screen, in which the positions of the character blocks are so far apart.

Create your own characters!

The CPC464 offers an interesting way of producing graphics, however, on the ordinary text screen, using only the PRINT instruction to place the patterns. These could be classified as 'low resolution' in the sense that they use the same limited number of PRINT positions on the screen, but they offer much more scope for dazzling effects. As this title suggests, we can create our own character shapes. There are two parts to this – the planning of the shapes, and how we place the shapes on the screen. Let's take these two in easy stages.

We'll start, logically enough, with planning. The size of the shape we're talking about is one screen character, the size of the cursor block. Now this, and every other CPC464 character, is made out of up to 64 dots that are arranged on an 8 by 8 grid. Figure 9.9 shows the

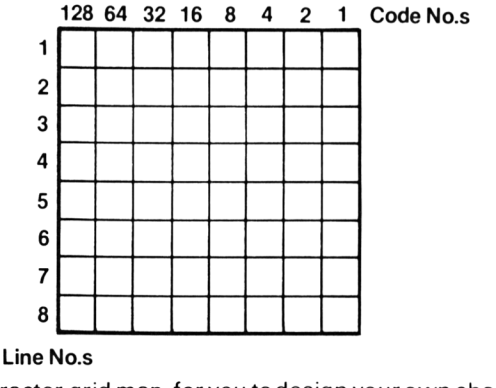

Fig. 9.9. The character grid map, for you to design your own character shapes.

shape of this grid – you can redraw it for yourself on a sheet of graph paper if you want more copies. The important point is that the small squares of the grid represent dots on the screen that can be in either INK or PAPER colour, according to the value of code numbers that we use to instruct the computer. When a character is designed on this 8×8 grid, we normally use only 7 dots across and 7 down, leaving the right-hand column and the bottom row unused. This is because we

want to have a space between any two characters, and also between any two lines of text on the screen. If you are designing your own graphics shapes, however, you might want to make them fill the whole 8 × 8 block, and so join on to each other when you print them on to the screen.

Now the CPC464 manual shows you very briefly how to design these shapes, but unless you have done it before, you may be rather puzzled. The key to it is the numbers that are printed on top of each column of squares in Fig. 9.9. Each number is a code for any square in the column underneath it. Use the number, and the square will be in INK colour. Use 0 instead, and the square is in PAPER colour, which means invisible. An example will help to make this clearer, and it appears in Fig. 9.10. I've used a simple shape of a 'space-walker' to illustrate the principle.

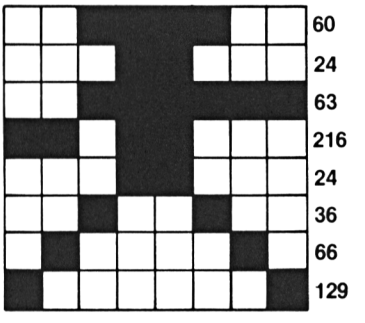

Fig. 9.10. How to use the grid map – for a 'space-walker' character plan.

The first line of squares has just four squares shaded in. I usually work on tracing paper clipped over the grid pattern, but in this example I've shown what it will look like on the graph paper itself. The shaded squares in the top line are the ones that we want to appear in INK colour, and they are under the code numbers 32, 16, 8 and 4. There's nothing else shaded in this line, so we add the numbers 32, 16, 8 and 4, to get 60, which is the number we note at the side. Similarly, in the second line down, the squares in the 16 and 8 positions are shaded, and so the number that we use is the sum of these, 24. We continue in this way until all the eight lines have been dealt with.

There are a few points to note. One is that if none of the squares in a line is shaded, the code number is zero. The other point is that you can save a lot of arithmetic by remembering that a complete set of shaded squares in a line adds up to 255. This is the maximum size of number that you can use as a code number for a text character. You

must always end up with eight numbers, no matter what shape you are trying to produce.

The next matter is how we instruct the computer to produce the shape. What we have to do is to store the code numbers in the CPC464's memory, along with an ASCII code number that we will use to obtain the shape on the screen. This makes use of a new instruction, SYMBOL. SYMBOL has to be followed by *nine* numbers. The first number is the ASCII code that we want to use. In this way, pressing a key or using this ASCII code will give our own pattern. The next eight numbers in SYMBOL are just the pattern numbers that we have already found. All of the numbers are separated by commas, there must be a space between the 'L' of SYMBOL and the first number, and the pattern numbers are read from top to bottom of the shape. The effect of SYMBOL is therefore to place the code number into the memory.

Now that may seem all very well, but what ASCII numbers can we make use of. The answer is that you can use ASCII codes 240 to 255, a total of 16 codes for this purpose. Twelve of these codes are normally provided by the cursor keys at the top right-hand side of the keyboard – there are 12 because the SHIFT and CTRL keys can be pressed as well as the cursor keys. Using the codes for your own purposes, however, *does not affect the action of these keys*, so that you do *not* get a space-walker pattern when you press a cursor key! Let's suppose that we want to make our space-walker pattern appear when CHR$(244) is used. Figure 9.11 shows what is needed – and it's not much. The ASCII code number is 244, and it's followed by the set of eight numbers that map out the shape. We can then produce the shape wherever we like by using PRINT CHR$(244).

```
10 CLS
20 SYMBOL 244,60,24,63,216,24,36,66,129
30 PRINT:PRINT TAB(12)CHR$(244)
```

Fig. 9.11. Making the space-walker appear.

Now you aren't limited to creating characters for use by ASCII codes 240 to 255. You can use any starting number you like from 32 onwards. This means that you can redefine *any* key code, and so make characters that will appear when you press the ordinary keys! Figure 9.12 shows what is needed to do this. In line 10, we have now added SYMBOL AFTER 90. This means that we can make our own shapes ('redefine') for any character that has an ASCII code of 90 or

```
10 CLS:SYMBOL AFTER 90
20 SYMBOL 92,60,24,63,216,24,36,66,129
30 PRINT:PRINT TAB(12);"\"
```

Fig. 9.12. Redefining a key code, so that the key gives the space-walker shape.

more. We have chosen 92, which is the ASCII code for the forward slash sign, the key next to the right-hand SHIFT. When you RUN this program, you'll see the space-walker symbol printed in response to the PRINT TAB(12);"\" instruction this time. You can, of course, use CHR$(92) if you like. More important, though, you will see the space-walker character appear each time you press the \ key from now on. To get back to normal, save your program, and press SHIFT CTRL ESC to reset the machine.

There's an important point here, however. None of the other codes greater than 90 are affected by this change, only the one which you have redefined. Of course, if you want to, there's no reason why you shouldn't redefine each key to give a different symbol – but it's hard work. Just as a bit of fun, take a look at Fig. 9.13 which redefines all

```
10 CLS:SYMBOL AFTER 32
15 FOR N=32 TO 127
20 SYMBOL N,60,24,63,216,24,36,66,129
25 NEXT
```

Fig. 9.13. Redefining all keys – it makes further programming difficult! Press CTRL-SHIFT-ESC to restore normal action, but save your program first.

of the keys as space-walkers. When this program has run, anything on the screen, apart from the cursor, is a space-walker! Even messages like 'Syntax error' and 'Ready' come out as a string of space-walkers! It's a good way of ensuring that no-one messes about with your computer when you leave it around! You will need to use SHIFT CTRL ESC to restore normal service, and remember that this will remove your program from the memory.

You are not confined to creating just one character in this way, and you can also create characters that fit together to form a shape! Figure 9.14 shows a multi-character map, which allows you to plan for shapes that are made out of up to 9 squares. Suppose, for example, that we made new characters to fit the codes 240 to 242. We could then place these three characters into one string, GR$, then the instruction PRINT GR$ would print the whole shape! Figure 9.15 shows a shape drawn on to three grids stacked together. Figure 9.16

then shows this in action, and since you now know what most of it's about, I don't need to say much more. I have used the three character

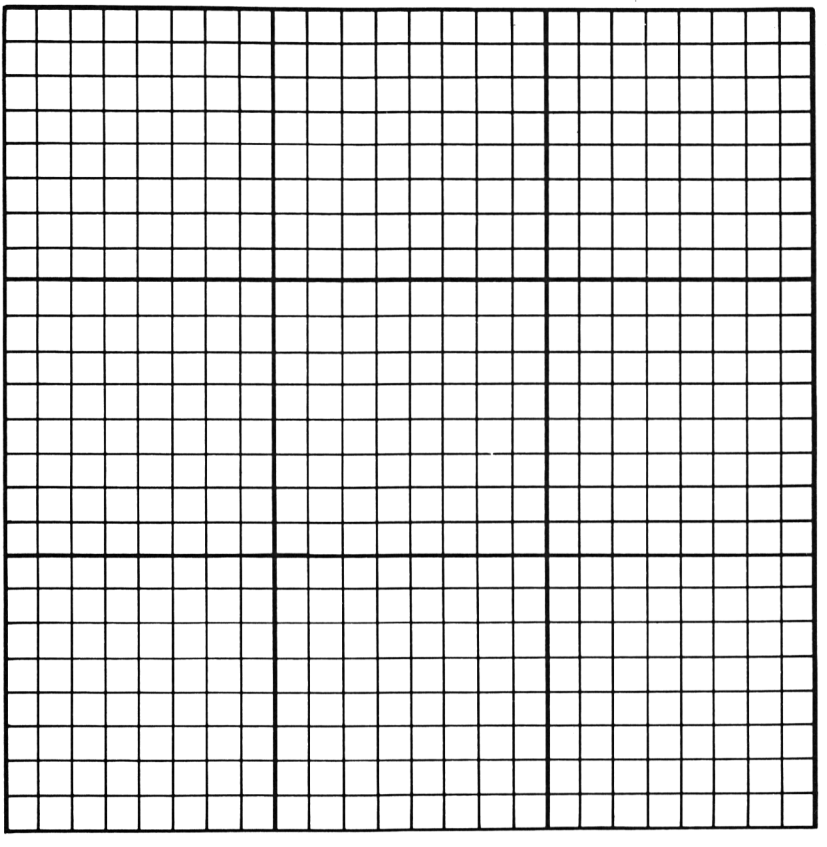

Fig. 9.14. A multi-character map, so that you can make shapes from several character blocks.

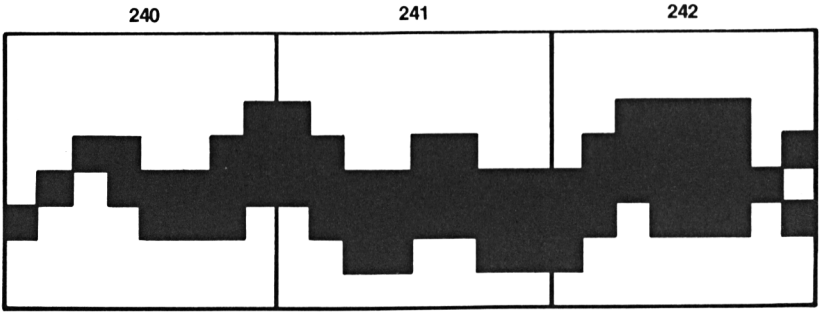

Fig. 9.15. A snake shape drawn on to three character blocks.

```
10 CLS
20 SYMBOL 240,0,0,1,51,95,142,0,0
30 SYMBOL 241,0,0,128,204,255,127,51,0
40 SYMBOL 242,0,0,60,125,254,221,128,0
50 GR$=CHR$(240)+CHR$(241)+CHR$(242)
60 X=2:Y=12
70 EVERY 10 GOSUB 1000
80 FOR N=1 TO 25:LOCATE 5,N:PRINT"HISSIN
G SID"
90 FOR J=1 TO 1000:NEXT:NEXT
100 GOTO 100
110 END
1000 LOCATE X-1,Y:PRINT" ":LOCATE X,Y
1010 PRINT GR$;
1020 X=X+1
1030 IF X>38 THEN LOCATE X-1,Y: PRINT CH
R$(18);: X=2
1040 RETURN
```

Fig. 9.16. Animating the snake, using EVERY.

codes 240 to 242 inclusive, and allocated shapes to each. Because the whole 8 × 8 grid has been used, the parts of the 'snake' will join up when you see them printed. Printing is done by combining the characters into a string and printing the string. It's simple, hard work, and fascinating to look at!

It also illustrates, however, an entirely new instruction, EVERY. This is one which is very important for animation, and we'll look at this and others of its type in more detail again. EVERY allows you to make use of one of the CPC464's timers, of which there are four. The idea is that you can specify something to happen every so often. How often depends on the number that follows EVERY. For the UK versions of the CPC464, a figure of 50 here specifies an action once per second – the US version will need a figure of 60. At this interval, for example, you can specify that a subroutine will run. The important thing is that the subroutine will run every interval, *even when the computer is doing other things*! You will see in the example that a loop is running, printing the words 'HISSING SID' while the snake makes its way across the screen. When the loop has ended, there is still an endless loop running in line 100. This keeps the program, and hence the timer, going. The timer works independently of the other parts of the program, and can interrupt the program action to carry out its subroutine at the interval that has been

specified. In this example, the interval is 10, one fifth of a second in the UK machine. A shape that can be animated independently of other program events in this way is called a *sprite*. A lot of machines nowadays allow you to use sprites, but few give you such simple and effective control over them as the CPC464!

Since there are four timers, you can have at least four sets of independently moving objects – you can have each timer controlling as many sprites as you like, of course. To specify which timer you want to use, the timer number, 0 to 3, is placed following the interval number, with a comma as separator. You can, for example, use commands such as EVERY 10,0 GOSUB 1000 or EVERY 20,3 GOSUB 2000. While we're on the subject of moving objects on this 'text screen', the CPC464 offers you a way of finding where the cursor is. This requires the words POS and VPOS, and each *must* use a window number. For the whole screen, this is #0, but it can't be omitted. Figure 9.17 shows a simple illustration of how POS and VPOS operate. POS reports the horizontal position of the cursor,

```
10 CLS
20 FOR Y=1 TO 25
30 PRINT TAB(RND(1)*35);POS(#0);",";VPOS
(#0)
40 FOR N=1 TO 1000:NEXT
50 NEXT
```

Fig. 9.17. Using POS and VPOS to find and report on the cursor position.

and VPOS reports the vertical position. The numbers that they use are the same as the numbers that the LOCATE command uses. POS and VPOS are particularly useful when you are working with windows, because it isn't always easy to figure out where the cursor will be. You can use the numbers from POS and VPOS in any sort of test that you like to devise.

Chapter Ten

Guide to Greater Graphics

The graphics abilities of the CPC464 are very much greater than those of many competing machines. In this chapter, we are going to look at the commands that apply specifically to graphics, meaning that we create shapes without the use of PRINT or CHR$. The first point to note is that the resolution of these CPC464 graphics is very high. If you look closely at the screen of a colour TV, you will see that it is divided into a set of lines or dots. These are the bits that glow and give out light, and you can't have anything displayed on the screen which is smaller than one screen dot or the distance across a line. In fact, unless you use a monitor, you can never get anywhere near displaying dots so small or lines so narrow. The 'dots' that the computer can work with are called *pixels* (*pic*ture *el*ement*s*). Each pixel that a computer can control corresponds to the size of several dots on the TV screen. The CPC464 allows you three choices of resolution, corresponding to the three modes. In Mode 0, your graphics resolution is 160 pixels across the screen and 200 down. For the other modes, the number of pixels down the screen is constant, always 200, but Mode 1 allows 320 pixels across the screen, and Mode 2 allows 640 pixels across the screen. This highest resolution isn't really well illustrated on a TV receiver, and you must use a colour monitor if you want to see really good results in Mode 2. This is a very high resolution by any standard, and the price which has to be paid is a restricted number of colours – just two.

The ability to 'paint' each of 64000 dots (Mode 1) would not, by itself, be very useful if the only way to make pictures were to specify the position and colour of each pixel. There *are* computers which allow no other way of going about high resolution graphics, but the CPC464, as you might expect, does rather better than this. There is, in fact, an excellent variety of graphics commands which allow you to draw in a more sensible way. All of these commands make use of the X and Y co-ordinate system that we have already come across used

with LOCATE. The difference now is the numbers that you can use. For all of the graphics modes, the range of X numbers is 0 to 399. This is very convenient, because it means that if you have developed a graphics program for one mode, and you want to run it in another mode, you don't have to go through the program changing all of the numbers. Figure 10.1 shows how you can draw a graphics map for yourself, using graph paper.

---

1. Take an A4 size sheet of graph paper, scaled in mm and cm. Chartwell and Guildhall make suitable graph papers.
2. Write scales, numbering the long side in steps of 25 up to 600. Number short side in steps of 25 up to 400.
3. Use tracing paper over this graph to draw your outlines and read co-ordinates.

Fig. 10.1. How you can construct your own graphics map, using graph paper.

There are important differences between working with text and working with graphics, however. When you work with the graphics commands, you are operating directly on the pixels, so that the distances that the units of X and Y numbers represent are much smaller. In addition, the origin of the graphics is different. The 'origin' is the place on the screen that is referred to by X=0 and Y=0. For the text screen, you can't use zero, but LOCATE 1,1 means left-hand side, top of screen. For graphics, the point X=0, Y=0 (usually referred to as the point 0,0) is at the left-hand side *bottom* of the screen. The Y-distances are measured *upwards*, and the X-distances across from left to right. This is the same place as is used as the 'origin' on most graphs. If you have been accustomed to drawing graphs, you will find graphics commands relatively simple to learn. Since we're on the topic of graphs, we'll start with a graph drawing command, PLOT.

Plotting it out

Drawing graphs is one very important aspect of graphics which is essential to have in any machine that can seriously be used for business or educational purposes. A graph is a set of points which can be joined and which should convey some information. The CPC464 will plot graph points for you, using the PLOT or PLOTR

commands. The colour of the point that is plotted will be whatever INK colour happens to be in use at the time. The difference between these two commands is that the PLOT command uses absolute co-ordinates and the PLOTR command uses co-ordinates measured *relative to the cursor position*. For example, if you use PLOT 0,0 the plot will be at point 0,0, the bottom left-hand corner of the screen. If you use PLOTR 0,0, the plot will be wherever the cursor happens to be. The cursor, throughout any graphics commands, is not the usual block that you see when you are using text. The graphics cursor is invisible, so that you don't see any evidence of it until you use a command that leaves pixels in a colour which is different from the background colour. The ordinary text cursor is restored when a graphics program ends.

Figure 10.2 shows the CPC464 being used to plot three graphs at the same time, and doing so at a reasonable speed. The range of X in

```
10 MODE 0
20 INK 0,0
30 FOR X=1 TO 639
40 Y=200+200*SIN(2*PI*X/639)
50 PLOT X,Y,1
60 Y=200+200*(SIN(2*PI*X/639)^2)
70 PLOT X,Y,2
80 Y=200+200*(SIN(2*PI*X/639)^3)
90 PLOT X,Y,3
100 NEXT
110 GOTO 110
```

Fig. 10.2. A graph-drawing program using high resolution. Three separate graphs are drawn in different colours.

line 30 is set so as to cover the whole width of the screen, and for each value of X, three values of Y are calculated. One is obtained from the sine of the angle whose value is X/639, in radians. Another is obtained from the square of the sine of this angle, and the third is obtained from the cube of the sine of the angle. Each point is plotted by the PLOT commands in lines 40, 60 and 80, using different INK colours for the three different graphs. The reason for the numbers that are used is that we want the graphs to fill the screen. The sine of an angle has a value that lies between −1 and +1. Now a value of 1 in the Y-direction does not make much impression, so we multiply the value by 200. This makes the quantity vary between −200 and +200. We can't plot −200 on this screen, however, so we add 200 to each

value, making the size vary between 0 and 400, the full range of Y values. The other point is that we use 2*PI*X/639 as the angle. This is to allow for radian measure. These angle functions go through a range of values as the angle goes from zero radians to 2*PI radians. When X=0, then 2*PI*X/639 is zero, which gives us zero radians at the left-hand side of the graph. When X=639, at the right-hand side of the graph, then the angle is 2*PI*639/639, which is 2*PI radians – just what we want. If you don't like working with angles in radians, just have the command DEG in a line somewhere before the angle functions are used. After using DEG, all angles will be in degrees. To reset to radians, reset the machine or use RAD.

Now, after running that program, take a look at the graphs. They show the pixel size of Mode 0 very effectively. They also show that the CPC464's 'low resolution' looks better than some computers' high resolution! Try altering line 10, first to MODE 1, and then to MODE 2, to see how small the pixels are in the other modes. It's less easy to see the colour in these very small dots, unless you are using a monitor, but the graph lines look smoother and more joined-up than in Mode 0. You see only two graphs in Mode 2. Why? Because you can have only two colours in this mode! For most purposes, Mode 1 is the ideal compromise between number of colours and high resolution.

We aren't quite finished with graph drawing yet, though. Try the slightly amended program in Fig. 10.3. This starts with ORIGIN 0,200. Now the effect of this is to shift the origin of all graphs to the point 0,200 – the left-hand side, halfway up the screen. The origin is always taken as being the point that you get to with 0,0, however, so PLOT 0,0 will now put a point halfway up the screen, left-hand side.

```
10 MODE 0:ORIGIN 0,200
20 INK 0,0
30 FOR X=1 TO 639
40 Y=200*SIN(2*PI*X/639)
50 PLOT X,Y,1
60 Y=200*(SIN(2*PI*X/639)^2)
70 PLOT X,Y,2
80 Y=200*(SIN(2*PI*X/639)^3)
90 PLOT X,Y,3
100 NEXT
110 GOTO 110
```

Fig. 10.3. Using ORIGIN to shift the origin of the graphs. This is the simplest way of using ORIGIN.

Shifting the position of the origin like this allows us to make the graph drawing instructions simpler, because we don't have to use the '200+' part any more. We can now do things like PLOT 50,−50, because −50 just means 50 pixels down from the mid-point, just as +50 means 50 pixels up. The ORIGIN command can also be used to specify a window size, as the Manual illustrates.

Drawing the line

Plotting graphs is one very useful part of high resolution graphics, but we can make use of commands which do much more than this. Two of these are MOVE and DRAW. The MOVE commmand, as you might expect from the name, moves the graphics cursor. Now since the graphics cursor is invisible, you don't see anything happen when you use MOVE. It's like moving a paint-brush – but with the brush kept clear of the paper. The DRAW command is for painting a line – the brush this time is definitely on the paper. Each of these commands uses the same system of X and Y numbers ('co-ordinates') that you have already met in connection with PLOT. In addition, the DRAW command can use a third number, the inkpot number.

It's time to look at an example, in Fig. 10.4. This is nothing elaborate, simply a program that draws a square. The colour of the

```
10 ON BREAK GOSUB 1000
20 MODE 0
30 MOVE 150,100
40 DRAW 150,300
50 DRAW 450,300
60 DRAW 450,100
70 DRAW 150,100
80 GOTO 80
1000 MODE 1
1010 LIST
```

Fig. 10.4. A drawing program which uses DRAW to produce a square.

square will be 'gold' against a blue background if you have just switched the machine on, but it will probably be red on black if you run this just after running the graph program. The reason is that if you don't issue any paintpot number following a DRAW X,Y instruction, the colour that will be used will be the colour that was

used last time a DRAW or PLOT was carried out. They don't forget, these machines! You will also find that the square is drawn very quickly, faster than the eye can follow. Compare this with the time some other machines take on even this simple drawing.

Oh, yes, I slipped in another new instruction as well. Line 10 uses ON BREAK GOSUB 1000. What this means is that if you tap the ESC key twice (which BREAKS the program), then a subroutine starting at line 1000 will be executed. The reason that I have used this is because listings do not look good in Mode 0. The 'subroutine' at line 1000 converts back to Mode 1, then LISTs. Since a LIST command *always* comes back to 'Ready', there's no point in having a RETURN for this 'subroutine'. When you press the ESC key twice, then, to get the program out of its endless loop in line 80, the screen goes back to Mode 1, and the program is listed for you. It's a very useful thing to have when you are getting a program running.

While we're on a simple drawing, we might as well see how DRAWR can be used. As I said earlier, this means DRAW with RELATIVE co-ordinates. DRAWR 10,100, for example, means DRAW a line 10 pixels to the right and 100 up. This is *not* the same as drawing a line to the point 10,100, which is what DRAW would do. Figure 10.5 illustrates our square drawing with DRAWR this time. If

```
10 ON BREAK GOSUB 1000
20 MODE 0
30 MOVE 150,100
40 DRAWR 0,200,2
50 DRAWR 300,0,2
60 DRAWR 0,-200
70 DRAWR -300,0
80 GOTO 80
1000 MODE 1
1010 LIST
```

Fig. 10.5. Using DRAWR to draw a square. Note the difference in the numbers.

you want to move to the right or up, the distances are positive. If you want to move to the left or down, then the distances are negative. If you want one co-ordinate, X or Y, to stay the same, then the number to use is 0.

Just to rub in the differences between DRAW and DRAWR, Fig. 10.6 shows a set of lines drawn from the centre of the screen to random positions, using DRAW (line 60). After a pause, a quite

```
10 MODE 1
20 CLG
30 FOR N=1 TO 50
40 MOVE 320,200
60 DRAW RND (1)*639,RND(1)*399,RND(1)*4
70 NEXT
80 AFTER 200 GOSUB 1000
90 GOTO 90
1000 CLG
1010 MOVE 320,200
1020 FOR N=1 TO 50
1030 A=RND(1):IF A>0.5 THEN A=1 ELSE A=-
1
1040 DRAWR A*INT(RND(1)*100),A*INT(RND(1
)*100),RND(1)*4
1050 NEXT
1060 RETURN
```

Fig. 10.6. Random line patterns which illustrate the differences between DRAW and DRAWR.

different effect is achieved using DRAWR. This time, each new line is attached to the end of the previous one. By using line 1030 to generate a value of A which is either −1 or +1, the direction of each line is made random, and its size is also made to be random by line 1040. This produces a pattern which is called a 'random walk' – you might like to think of it as the path of a demented fly. The pause has been organised by another new command, too. AFTER 200 GOSUB 1000 means that after the number 200 has been counted out by the timer, the program will go to the subroutine at line 1000. As before, the timer counts at a rate of 50 per second (UK), so that 200 represents a time delay of four seconds. For US readers, use 240 in place of 200. After this time, the GOSUB 1000 carries out the DRAWR routine, and then returns to the endless loop in line 90.

Planning it!

Drawing in high resolution graphics is all very well, but to do the job thoroughly, you need some planning. The best planning aid is a sheet of graph paper. An A4 sheet of cm/mm graph paper, such as is supplied by Guildhall, is ideal. If you make four centimetres equal 100 pixels, you can mark in 0 to 400 up the sheet and 0 to 650 along

the long side. You can then plan your drawings either on this directly or onto tracing paper clipped over the graph paper. When your drawing (straight lines only at the moment, I'm afraid) is complete, mark each point where one line meets another, and pencil in the X and Y numbers for these points. You can now start writing your program. As an illustration of this, Fig. 10.7 shows a shape that has

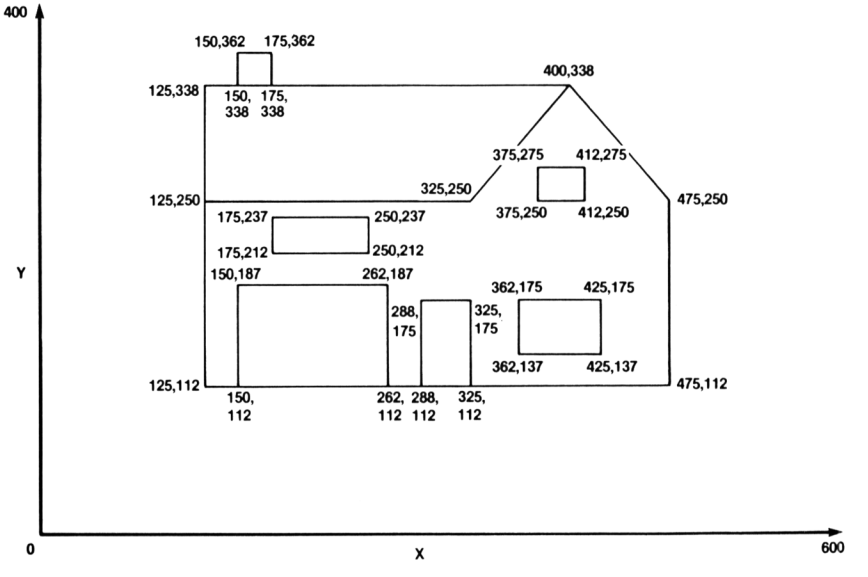

Fig. 10.7. A house shape planned by placing tracing paper over graph paper.

been planned in this way. The house has been drawn on tracing paper, with the graph paper underneath. The rough sketch has been redrawn so as to follow the lines of the graph paper, and then the X and Y numbers for each point have been read off from the graph paper.

This leads to the program in Fig. 10.8. This makes as much use as possible of DATA lines, so that if there is anything that looks as if it

```
10 MODE 0
20 MOVE 125,112
30 PEN 2
40 FOR N=1 TO 7
50 READ X,Y
60 DRAW X,Y
70 NEXT
80 MOVE 150,112:FOR N=1 TO 3
```

```
90 READ X,Y:DRAW X,Y:NEXT
100 MOVE 288,112
110 FOR N=1 TO 3
120 READ X,Y:DRAW X,Y:NEXT
130 MOVE 362,137
140 FOR N= 1 TO 4:READ X,Y
150 DRAW X,Y:NEXT
160 MOVE 375,250
170 FOR N=1 TO 4:READ X,Y
180 DRAW X,Y:NEXT
190 MOVE 175,212
200 FOR N=1 TO 4:READ X,Y
210 DRAW X,Y:NEXT
220 MOVE 150,338
230 FOR N=1 TO 3:READ X,Y
240 DRAW X,Y:NEXT
250 MOVE 125,338:DRAW 400,338
1000 DATA 475,112,475,250,400,338,325,25
0,125,250,125,338,125,112
1010 DATA 150,187,262,187,262,112
1020 DATA 288,175,325,175,325,112
1030 DATA 362,175,425,175,425,137,362,13
7
1040 DATA 375,275,412,275,412,250,375,25
0
1050 DATA 175,237,250,237,250,212,175,21
2
1060 DATA 150,362,175,362,175,338
```

Fig. 10.8. The program which draws the house shape.

needs improving, it's only the data lines that will need to be changed, not the whole program. Each piece of drawing starts with a MOVE. I always move to the bottom left-hand corner of anything I draw. Following the MOVE, a set of DRAWs in a loop makes the pattern. It's very fast and effective in action, but the time that you would spend on this without planning would be criminal!

Making circles

Drawings that consist of nothing but straight lines are all very weli, but for many types of drawings, we need circles, ellipses and arcs.

This is a weakness of the CPC464, one of the very few, because it contains no CIRCLE command, nor anything to fill in a shape in colour. We have to accept this, and make use of subroutines to carry out these tasks. The Manual is very helpful in this respect. Figure 10.9 shows one circle-drawing program. Line 10 sets the mode, just in case it had been changed, and this clears the screen. Line 20 then sets the centre of the circle as the origin, and also sets the radius, R. The

```
10 MODE 1
20 ORIGIN 320,200:R=100
30 GOSUB 1000
40 ORIGIN 0,0
50 GOTO 50
1000 DEG
1010 FOR N=1 TO 360
1020 PLOT R*SIN(N),R*COS(N),2
1030 NEXT
1040 RETURN
```

Fig. 10.9. A circle-drawing program. This method is rather slow.

circle-drawing subroutine is then called, and the program shifts the origin back, and loops endlessly until you press ESC ESC. In the subroutine, DEG sets the angles that are used to units of degrees. Line 1010 then starts a loop, plotting each point around the edge of the circle. If you did (and remember) co-ordinate geometry at school, you'll understand what is happening. If you don't or didn't, then take it on trust rather than getting mixed up in mathematics – the subroutine *paints* a circle.

It's not the only way of painting a circle, and it's very slow, but at least it's reasonably simple. Figure 10.10 shows a much faster

```
10 MODE 1
20 ORIGIN 320,200:R=100
30 GOSUB 1000
40 ORIGIN 0,0
50 GOTO 50
1000 DEG
1005 MOVER 0,R
1010 FOR N=0 TO 360 STEP 10
1020 DRAW R*SIN(N),R*COS(N)
1030 NEXT
1040 RETURN
```

Fig. 10.10. A much faster method of drawing a circle.

alternative. This does not paint a true circle, but instead draws a shape with a lot of straight sides. By choosing a large number of sides (36), however, the shape looks quite reasonably circular, and the speed makes up for a lot! This subroutine is the one to go for if you don't want to hang about waiting for a circle to be drawn.

While we are working with the CIRCLE command, it's a good time to take a quick look at how to fill in a circle. Figure 10.11 gives

```
10 MODE 1
20 ORIGIN 320,200:R=100
30 GOSUB 1000
40 ORIGIN 0,0
50 GOTO 50
1000 DEG
1010 FOR N=0 TO 360
1020 MOVE -R*SIN(N),R*COS(N)
1030 DRAW R*SIN(N),R*COS(N)
1040 NEXT
1050 RETURN
```

Fig. 10.11. How to fill a circle with colour. Rectangular shapes are much easier to fill!

you a taste of this, with lines 10 and 20 setting up the origin and radius, and the subroutine that starts at 1000 drawing and filling the circle. The idea is to draw a set of lines close together, with the ends of the lines drawing out the circle. The fact that the lines are in colour fills the circle with colour. The filling effect is achieved in lines 1020 and 1030. The MOVE step goes to a set of points on the left-hand side of the circle, and the DRAW draws a line to the corresponding point on the right-hand side. Try it – it looks good, and it's almost as fast as the FILL commands that a few computers can use.

Detective work

A lot of interesting effects are possible if you can arrange for the program to detect, automatically, where the cursor is. The graphics cursor, remember, is invisible, so some method of detecting where it is or what it has just hit must rely on some sort of command. There are, in fact, several commands that report what the graphics cursor is up to. Two of these are XPOS and YPOS. Each comes up with a number. If you use J=XPOS, then the value of J will be the X co-ordinate number at the time when the test was made. If you use

K=YPOS, then, similarly, you get the value of the Y co-ordinate number. These commands carry out for the graphics cursor what POS and VPOS do for the text cursor. The difference is that XPOS and YPOS do not have any window number following the command word. What we need to look at now is how we make use of these actions.

One very useful thing that we can do is to ensure that the graphics cursor never goes off the screen. We can, for example, achieve 'wrap-around'. This means that if the cursor moves off the screen on one side, it appears on the other side. Figure 10.12 illustrates this, in a

```
10 CLS
20 X=320:Y=200
30 MOVE X,Y:RANDOMIZE TIME
40 WHILE X<>0
50 J=RND(1):IF J>0.5 THEN X=20*RND(1) EL
SE Y=20*RND(1)
60 DRAWR X,Y,1
70 IF XPOS>630 THEN MOVE 10,YPOS
80 IF XPOS<10 THEN MOVE 630,YPOS
90 IF YPOS>390 THEN MOVE XPOS,10
100 IF YPOS <10 THEN MOVE XPOS,390
110 WEND
```

Fig. 10.12. Using XPOS and YPOS to achieve 'wrap-around'.

program that draws ragged lines. Line 20 sets a position in the middle of the screen, and line 30 puts the graphics cursor there. Line 30 also contains the new instruction RANDOMIZE TIME. The reason for this is that RND does not generate truly random numbers, nothing like as random as the spin of a roulette wheel. The RND numbers are calculated by the computer, and like anything else that is calculated, they can't be truly random. The result is that the RND numbers, if you print out enough of them, will start to repeat their sequence. You can stop this by using RANDOMIZE, and the effect is even better if the number that follows RANDOMIZE is itself a number that won't repeat except by chance. TIME is a number of this sort, so RANDOMIZE TIME is a pretty cast-iron way of ensuring that you will never get two displays from this program that are exactly alike.

Line 40 then starts an endless loop, because X can never be 0. Line 50 then decides on new X or Y values. One or the other is changed to a random number, according to the value of J. In line 60, the new

values of X and Y are used to draw a line, but with relative values, not absolute. We then test the position of the cursor in lines 70 to 100. These tests make sure that if the cursor is getting to the edge of the screen, it will appear on the opposite side. Note the use of VPOS and YPOS in the MOVE parts of these lines. You can't use X or Y because these are *relative* position numbers, not absolute.

There's another way of checking where the invisible graphics cursor is. The command TEST (X,Y) gives an INK number, which is the INK number for the position X,Y. You can therefore put TEST (X,Y) into IF... statements to find if you have background colour, or the colour of some obstacle. You can also use TESTR, which takes relative co-ordinates. Using TESTR (1,0), for example, will test the pixel just to the right of the cursor in Mode 2. You can use TESTR (2,0) in Mode 1, and TESTR (4,0) in Mode 0 for the same purpose.

Tagging along

Our high resolution graphics so far have been rather static, confined to drawing patterns and shapes. It doesn't have to be like this, because there's a very useful command, TAG. TAG connects the graphics cursor to the text cursor. Now the graphics cursor is small, just one pixel in size, and the text cursor is 8×8 pixels. The point of attachment is the top left-hand side of the graphics cursor. If you PRINT a character at the position that is given by the graphics cursor, then, the character will occupy a space which extends to some seven pixels to the right of the graphics cursor and seven down. I'm assuming a text character which is constructed of 7×7 pixels – you may be using a 'defined' character of 8×8, of course. The effect of TAG is that we can print shapes such as we get from CHR$ number on the screen *at the position of the graphics cursor*. Why should this be important? Well, if you think of the MODE 1 screen as an example, we have only 40 positions for a character shape on a line. If we want to move the shapes along the line, we have to move it in 40 steps, which looks very jerky. The graphics cursor can move in much finer steps. For MODE 1, the graphics cursor can move in 320 steps, using X numbers of 0 to 639. This is a much smaller movement, and it can make animation look a whole lot better.

As a sample, take a look at Fig. 10.13. The TAG command in line 30 does this task of attaching the text cursor. Anyone who has programmed the BBC Micro will recognise this action – it's the equivalent of the BBC's VDU5 command. We can now print at the

```
10 MODE 1
20 Y=200
30 TAG
40 FOR X=0 TO 639 STEP 2
50 MOVE X,Y
60 PRINT" ";
70 MOVER 2,0
80 PRINT CHR$(243);
90 CALL &BD19
100 NEXT
110 TAGOFF
120 END
```

Fig. 10.13. Using TAG to make graphics shapes appear at the graphics cursor position.

graphics cursor position. Line 40 sets up a simple loop, which will take the graphics cursor across the screen. Line 50 moves the graphics cursor to the position that has been set by the values of X and Y. Note that we use STEP 2 in the loop. This is because the loop takes the usual values of 0 to 639 to control the cursor, but in MODE 1 there are only 320 positions. Each position of the graphics cursor can have two X numbers. X=0 and X=1 are the same position, the next one is X=2 or X=3 and so on. In MODE 2, each number from 0 to 639 is a separate position – try the program in this MODE. In MODE 0, there are only 160 separate positions across the screen, so each one can use any of four numbers. X=0,1,2 or 3 is the first, X=4,5,6 or 7 the next and so on. In a loop, we could use STEP 4. These STEP numbers are important, because they eliminate another cause of jerkiness. If the shape spends (in MODE 0) four passes of the loop in each position, then moves, it is going to move much more jerkily than if it moves on each pass.

At the cursor position, then, we print a blank. This is done to remove the previous drawing, and it has no effect first time round. Line 70 then moves the cursor two steps on, and line 80 prints the character shape. It's at this point that you need to be careful. Try running the program with these semicolons omitted, and look at the strange results! When you are using the graphics screen, *everything prints*, including the carriage return and line feed codes. When you use TAG, you can suppress these codes by using the semicolon, which does not print a shape of its own. The next mystery is in line 90. This invokes a bit of 'machine code'. The best way to see why is to remove this line and run the program. You will see the shape move faster, but

with some peculiar effects now and again. The reason for the odd distortions in the shape is that the TV picture is formed by drawing lines across the screen and down, and brightening the spot wherever there is something to be put on the screen. When the object on the screen is moving as well, there are times when the two lots of movements conflict. To avoid this type of problem, we have to move the object at a speed which is synchronised with the rate at which the screen spot traces out the pictures. This is fifty times per second in the UK, 60 times per second in the USA. A routine in the ROM of the CPC464 can do this, and we call up this routine by using CALL &BD19. '&BD19' is actually a number, in a form of number code that is called *hexadecimal*. If this arouses your curiosity, look out for my book on Amstrad machine code (*Introducing Amstrad Machine Code*) some day!

Of course, you might feel that slowing down the movement is the last thing you want to do. How do we speed it up? One way is to make X and Y both integers, X% and Y%. If this isn't fast enough for you, try STEP 4, or even STEP 8. Don't be tempted to omit the CALL &BD19, however, because the effects are nothing like so good – the lack of synchronisation is much more noticeable when the object is moving fast. Note that in this program, the MOVER in line 70 is used to keep part of the shape on the screen, which also helps to reduce jerkiness. For other types of shapes, you might want to omit this, or use a different number.

INK animation!

A quite different, and very cunning, method of animation is possible by making use of the INK command. Suppose that what you want to animate is not a CHR$ shape (which, remember, can be a shape that you have designed), but a complete drawing which has been created using DRAW and MOVE? You could, of course, draw the shape, wait, erase the shape, wait, draw in a different position, and so on. The trouble with this is that it's slow, and the animation is jerky. A much better method is achieved if you make use of these incredible INK commands. Suppose you make a set of drawings, each in a different position, and each with ink taken from a different numbered pot. Suppose also that the ink in each pot is of the same colour as the background. The result, of course, is that all of your drawings are invisible!

Now imagine a loop in your program. In each pass of the loop, you change one of the ink pot colours that you have used to a new colour, a colour that is *not* background colour. This will have the effect of making one of your drawings visible. Wait a moment, and then make this ink pot contain background colour again. Your drawing disappears, and you can then change the ink in another pot to make another drawing appear, then disappear. In this way, you can make a set of drawings *that have been put on the screen beforehand* appear and disappear in turn, giving an illusion of animation. This, of course, is best done in MODE0, in which you have a lot more ink pots to play with.

Figure 10.14 shows the kind of thing I have in mind. The first part of this program sets ink colours 2 to 15 to the background colour, which is colour number 1. Lines 40 to 90 then draw four rectangles on

```
10 MODE 0
20 FOR CL=2 TO 15
30 INK CL,1:NEXT
40 FOR NR=1 TO 4
50 READ X,Y:MOVE X,Y
60 FOR LN=1 TO 4
70 READ X,Y
80 DRAW X,Y,NR+1
90 NEXT:NEXT
100 WHILE INKEY(47)=-1
110 FOR NR=2 TO 5
115 CALL &BD19
120 INK NR,24
130 FOR X=1 TO 30:NEXT
140 INK NR,1
150 NEXT
160 WEND
170 DATA 300,300,350,300,350,100,300,100
,300,300
180 DATA 385,288,415,252,270,113,235,148
,385,288
190 DATA 225,225,425,225,425,175,225,175
,225,225
200 DATA 230,255,275,293,410,148,377,110
,230,255
210 MODE 1:END
```

Fig. 10.14. Animation which is achieved by using 'invisible ink'!

the screen. One rectangle is vertical, and the next three are each at 45 degrees to the one before. The whole set shows the successive positions of a rectangle as it turns from vertical to horizontal to vertical again, in a clockwise direction. If you want to examine what is going on in this part of the program, make the INK number in line 30 equal to 24 instead of 1, and put in a line 95 STOP.

Once the drawings are done, with the screen still blank, you can make them appear by changing INK colours. The loop that starts in line 100 will continue until the spacebar is held down. The FOR...NEXT loop then changes each INK colour in turn for a short time and then returns it to background colour. The CALL &BD19 is added to make the animation smoother – try omitting it to see the effect. The overall impression is of a rectangle which rotates clockwise.

Nothing in graphics is achieved without a certain amount of sweat and toil. The co-ordinates for the successive drawings have to be found, and put into the DATA lines, and this involves hard work – it's easier if you have a drawing board! It's very likely, however, that someone will bring out a program that will produce these co-ordinate numbers for you. You produce the original shape, and the program will then give you co-ordinates for each point when the shape is turned through an angle which you specify. It's a piece of geometry that the machine can solve a lot quicker than you can. This method of animation, incidentally, is used also on the BBC Micro, though in a less simple way.

Chapter Eleven
Sounding Out the CPC464

The ability to produce sound is an essential feature of all modern computers. The sound of the CPC464 comes from its built-in loudspeaker, which has a volume control situated on the right-hand side of the casing of the computer. In addition, a socket on the back of the CPC464 allows you to connect to a hi-fi amplifier unit, so that you can hear the sound at greater volume, and with better bass (low notes). The difference is quite astounding if you have only ever heard the sound delivered from the tiny loudspeaker of the computer itself. In addition, the sound that can be obtained from the socket can be in stereo. If you have one of the popular makes of pocket stereo players (mine is a Sanyo), you can plug in the stereo earphones to this socket which is at the rear right-hand end of the CPC464. The volume of sound that you get on these headphones is not very great, however, and a stereo amplifier is really needed to get the best from this effect.

Now if all that you need in a program is a short beep to remind you of something, like making a selection, then you need look no further than PRINT CHR$(7). This will give you your beep – and Fig. 11.1 shows one application of such a note. This is the start of an imaginary program which demonstrates sounds, and the bit we're interested in is the subroutine in line 1000. This uses the K$=INKEY$ to detect a key, and if a key is pressed, J=VAL(K$) gets its number value for the rest of the program to use. The interesting bit is what follows if no key has been pressed. Line 1010 sounds a short beep, using CHR$(7). You need to put a semicolon following the CHR$(7) to prevent the screen scrolling. Though PRINT CHR$(7) does not print anything on the screen, the cursor will move down one line after each PRINT, and if you wait for a time without pressing a key, you will see the screen scroll. This is not what is wanted, and the semicolon prevents it. Try removing the semicolon, and you'll see what I mean. After the beep, there is a short delay, and then the GOTO 1000 carries out the INKEY$ test again. The net result is that the machine honks at you

```
10 CLS:PRINT TAB(19)"MENU"
20 PRINT:PRINT TAB(3)"1. Music"
30 PRINT TAB(3)"2. Explosions"
40 PRINT TAB(3)"3. Water noises"
50 PRINT TAB(3)"4. Engines"
60 PRINT:PRINT"Please select by number"
70 GOSUB 1000
80 IF J<1 OR J>4 THEN PRINT"Incorrect- p
lease try again":GOTO 10
90 ON J GOSUB 500,500,500,500
100 END
500 PRINT"You'll be able to put a progra
m in here":PRINT"later!"
510 RETURN
1000 K$=INKEY$:IF K$<>""THEN J=VAL(K$):
RETURN
1010 PRINT CHR$(7);:FOR N=1 TO 500:NEXT
1020 GOTO 1000
```

Fig. 11.1. Producing a beep with CHR$(7).

until you press a key – it's a useful alternative to the flashing asterisk. You can even combine this with the flashing asterisk if you like, to make sure that no-one can ignore the menu choice!

For anything more than just a beep, however, you need to understand a little about sound itself. What we call sound is the result of rapid changes of the pressure of the air round our ears. Everything that generates a sound does so by altering the air pressure, and Fig. 11.2 shows how the skin of a drum does this. Air pressure is invisible, and we don't notice these pressure changes even with our ears unless the changes occur fairly fast. We measure the rate in terms of cycles per second, or *Hertz*. A cycle of a wave is a set of changes of pressure, first in one direction, then in the other and back to normal, which we can illustrate by the graph in Fig. 11.3. The reason that we talk about a sound 'wave' is because the shape of this graph is like the shape of a wave of water. What the graph shows is how the pressure of the air alters during the time that the sound is being generated.

The *frequency* of sound is its number of *Hertz* – the number of cycles of changing air pressure per second. If this amount is less than about 20 Hertz, we simply can't hear it, though it can still have disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 Hertz, going up to about 15000 Hertz. The frequency of the waves corresponds to what we sense as the 'pitch' of

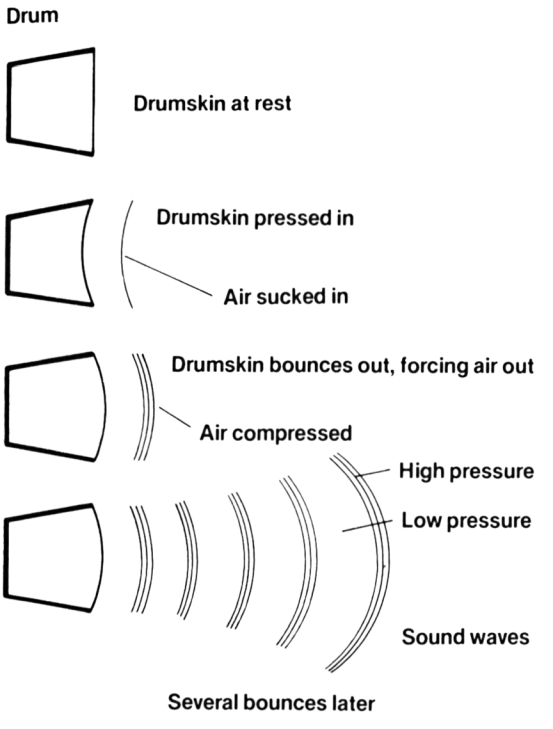

Fig. 11.2. How a drum-skin produces sound by alternately compressing and decompressing air.

a note. A low frequency of 80 to 120 Hertz corresponds to a low-pitch bass note. A frequency of 400 or above corresponds to a high-pitch treble note.

The amount of pressure change determines what we call the *loudness* of a note. This is measured in terms of *amplitude*, which is the maximum change of pressure of the air from its normal value. For complete control over the generation of sound, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds.

For the purposes of writing music, a composer has to specify for each note how loud it will be, for how long it has to be played, and its pitch. In written music, loudness is indicated by letters such as *f* (loud) and *p* (soft), and by using more than one letter if necessary. For example, *fff* means very loud, and *ppp* means very soft. The duration of a note is indicated in two ways. One of these ways is a metronome reading. A metronome is a sound generator which ticks at precise intervals, and a metronome reading indicates how many 'beats' (units

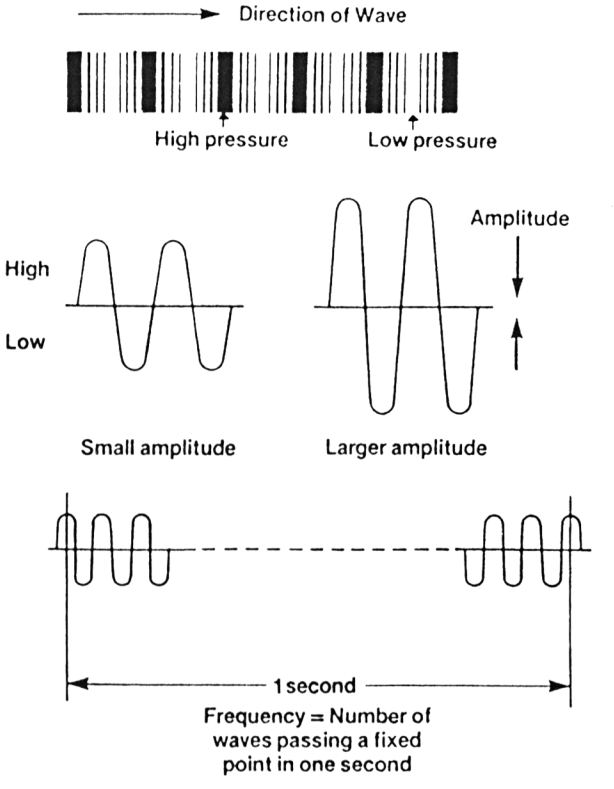

Fig. 11.3. The frequency and amplitude of a sound wave.

of notes) are sounded per minute. The unit duration of note is called the *crochet*, so the metronome reading decides on the time of a crochet by measuring the number of crochets that would be sounded per minute. The durations of the other notes are then measured in comparison to this. A *minim* sounds for twice as long as a crochet, and a *semibreve* sounds for twice as long as a minim, and is therefore four times as long as a crochet. The *quaver* is allowed half the time of a crochet, and a *semiquaver* a quarter the time of the crochet. These differently timed notes are indicated by the shape of the symbols that are used for the notes (Fig. 11.4). In addition, there are symbols for different durations of silence in the music, and these are illustrated in Fig. 11.5. Some music scores do not show a metronome reading, but rely on the use of Italian words like lento, moderato or allegro, to indicate the time of a crochet less precisely.

The pitch of a note is indicated in written music by placing it on a kind of musical 'map' that is called the *stave*. Piano music shows two of these staves, each consisting of five lines and four spaces. The set of

Symbol	Time	Name
𝅘𝅥𝅯	1/8	Demisemiquaver
𝅘𝅥𝅰	1/4	Semiquaver
𝅘𝅥𝅮	1/2	Quaver
𝅘𝅥	1	Crotchet
𝅗𝅥	2	Minim
𝅝	4	Semibreve

Fig. 11.4. The symbols that are used in written music to indicate the time of each note.

Rest Symbol	Time
𝄾	1/4
𝄽	1/2
𝄼	1
𝄻	2
𝄺	4

Fig. 11.5. The symbols for silences in written music.

lines which is marked with the sign like the ampersand (&) is the *treble stave*, used for the higher notes, and the stave below it is the *bass stave*. The bass stave is also marked with a distinctive symbol, like a reversed 'C'. When music is written for instruments which do not use a keyboard, then only one stave is used. Piano and organ music always shows two staves, with a place for a note placed between them. This note is called *Middle C* and on a piano, this note is played by pressing the key which is almost exactly at the centre of the keyboard. Figure 11.6 shows the staves with the notes marked.

The notes that are shown in Fig. 11.6 are arranged in groups of eight, counting inclusively. This group is called an *octave*. It is possible to make notes whose pitch falls between the pitches of any two notes in this set of eight, and these 'in-between' notes are called semitones. Music from the Western hemisphere traditionally uses a total of twelve distinct notes, tones and semitones in an octave, and this full range is illustrated in Fig. 11.7, which shows the appearance of part of the piano keyboard. The semitones are marked on the

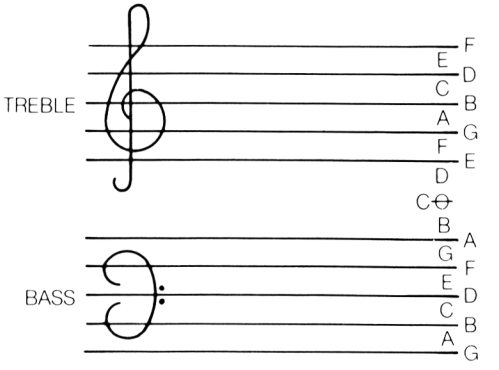

Fig. 11.6. The staves of written music, with the names of the notes written in. (Note the position of Middle C.)

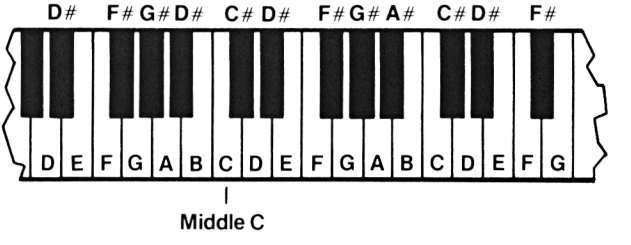

Fig. 11.7. Part of the piano keyboard, showing Middle C. There is only one semitone between B and C, and between E and F.

piano mainly by the black keys, though two semitones (between B and C, and between E and F) are not marked in this way. On written music, semitones are indicated by using the signs # (sharp) and ♭ (flat). A sharp indicates that the pitch has to be raised one semitone above the written note, and a flat indicates that the pitch is to be lowered one semitone below the written note. On the piano, the semitone above one note is the same as the semitone below the next note, so that C# is the same as D♭. This is not necessarily true on other instruments.

The CPC464 SOUND

To work, then. The CPC464 provides a sound command which can be used either as a simple command, or as a complicated one. How you use it depends on what you want of it, so that you don't have to do a lot of planning just to produce a warning note or to play a melody. When you are experienced in the use of the sound command,

however, you can make use of it in much more interesting ways. Of all the computers I have used, the CPC464 has by far the best sound system from the point of view of being able to control the sound with BASIC instructions. We shall start with the straightforward production of notes.

This is provided by the SOUND instruction. The SOUND instruction has to be followed by at least two numbers. Of these, the first number is a 'channel select number'. The CPC464 can produce three notes of sound at the same time, and all three notes can be separately controlled. This is done by allocating each note to a separate 'channel'. These channels are labelled A, B and C, and one or more can be selected by using the channel select number. Figure 11.8

Select No.	Channel(s) selected
1	A
2	B
3	A and B
4	C
5	A and C
6	B and C
7	A and B and C

Fig. 11.8. The numbers which can be used to select the channels.

shows how the numbers 1 to 7 can be used to select one or more of the channels. The CPC464 goes further than any previous home computer in allowing stereo sound to be generated. Channel A is fed out to the left-hand stereo connector, and Channel C to the right-hand connector. Channel B connects to both, and anything on this channel will appear at equal volume on both stereo outputs. Numbers greater than 7 produce more complicated effects which we shall look at later.

The second number that follows the SOUND instruction is the 'pitch number'. This controls the pitch of the note that is produced by the CPC464, and its range is from 1 (highest pitch note) to 4095 (lowest note). A more practical range is 10 to 1000, because the notes above 1000 are more like a series of clicks unless you have a really excellent loudspeaker system. The number 0 can be used, but only for other purposes, which we'll look at later. The notes that you get for pitch numbers less than 10 are too high to be very effective. The musical equivalents of the pitch numbers are clearly shown in the

Manual, in Appendix VII. Page 2 of this Appendix shows the most familiar set of notes, the scale of C Major from Middle C to one octave above. The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play this one particular series of notes, which is called the 'scale of C Major'. The scale starts on a note that is called Middle C, and ends on a note that is also called C, but which is the eighth note above Middle C. A group of eight notes like this forms an octave, so that the note you end with in this scale is the C which is one octave above Middle C.

Let's start our investigation of the SOUND instruction with a simple single note. This is illustrated in Fig. 11.9, which has its first

```
10 SOUND 1,478,50
20 FOR N=1 TO 2000:NEXT
30 SOUND 1,478,100
40 FOR N=1 TO 2000:NEXT
50 SOUND 1,478,200
```

Fig. 11.9. A simple single note.

SOUND instruction in line 10. Channel A is selected by using the number 1, and the pitch of the note is selected by the number 478, which gives a note very close to Middle C. Line 10 turns the sound on, and the action of this simple SOUND command is that the sound will continue for about one fifth of a second. Now at this point, it's *very* important to understand what is going on. When the computer executes a SOUND command, it extracts the numbers, and passes them to a separate system, the sound system. This part of the computer operates on its own, and generates the sound. In the meantime, however, the computer gets on with its job of carrying out the next instruction. Here again, two things can be done at once. If you have a lot of SOUND instructions, one after the other, the computer will zip through them, sending the numbers to the sound system, and getting on with its own job. Each note takes a time to sound, and the computer may have sent the data on five more notes to the sound system by the time the first note has finished playing! This is what the manual means by the 'sound queue', and unless you understand this idea, you may find that the more complicated SOUND instructions will never produce the effects that you expect of them. In this example, each SOUND has been followed by a delay loop which is so long that each note ends before the computer has

been able to get to the next SOUND. The duration of a note is fixed by the number that follows the pitch number, the third number of the set. In line 30, this number is 50, and the time is 50/100 second, which is half a second. Values of zero, and negative numbers, have special uses which, once again, we'll come to later.

Figure 11.10 shows a variation on this previous program,

```
10 SOUND 1,478
20 WHILE SQ(1)>128:WEND
30 FOR N=1 TO 100:NEXT
40 SOUND 1,478,50
50 WHILE SQ(1)>128:WEND
60 FOR N=1 TO 100:NEXT
70 SOUND 1,478,200
```

Fig. 11.10. Sounding notes with a fixed rest between them. SQ is used to find when a note ends.

improving on the space between notes. The novelty here is SQ, which reports on the sound queue in the way that XPOS reports on the cursor position. While there is something in a queue, SQ has a value which will be at least 128. By testing this value in a WHILE...WEND loop, we can make the computer wait for the sound queue, so that the next command does not get added until the first one has finished. In this way, we can, once again, separate the notes. The advantage is that the same FOR...NEXT loop will always give the same interval between the notes now, rather than depending on the length of the notes. If you use a simple FOR...NEXT loop, timing starts from the instant that the note is put in the sound queue. By using the WHILE...WEND with SQ, you time from the end of one note to the start of the next. The quantity SQ has to be followed by the channel select number in brackets. Other uses of SQ, like so many of these sound commands, deserve a manual all to themselves!

The next step is to try a musical scale. The table of numbers and frequencies that is shown in the Appendix of the CPC464 Manual is your guide to musical notes. Making use of this table, we arrive at the program in Fig. 11.11, which plays the scale of C Major. This is a scale which starts with Middle C, and goes up to the C above it. In this scale of notes, the frequencies are such that the C above Middle C has exactly double the frequency of Middle C itself. The same is true of all the other notes in the scale – doubling the frequency is equivalent to going up one octave in pitch. Halving the frequency corresponds to going down one octave in pitch. The CPC464 pitch numbers operate

the opposite way round, so that an octave rise in pitch is achieved by halving the CPC464 number. You can see from Fig. 11.11 that Middle C uses a number of 478, and the C above it uses 239, which is exactly half. We can't always achieve this when we use only whole numbers, but we keep as close as possible to these ratios.

```
10 CLS:PRINT TAB(13)"SCALE OF C MAJOR":P
RINT:PRINT
20 FOR N=1 TO 8
30 READ note
40 SOUND 2,note,100
50 NEXT
60 PRINT"Now all of the notes are in the
 queue!
100 DATA 478,426,379,358,319,284,253,239
```

Fig. 11.11. The scale of C Major, by CPC464.

Figure 11.11 also brings home the point about sound queues to you. The CPC464 sound system can handle a total of five notes in a queue. Of these one will be playing and the other four will be waiting in the queue. When you run this program, then, you will see the title, and hear the scale start. When five notes are waiting, and there are more notes to read, the computer simply has to wait as well. Whenever all of remaining notes are placed in the queue, however, the computer can get on with its work, and the message then appears.

The next step is to investigate the use of more than one channel of sound at a time. Figure 11.12 shows a chord being sounded with three

```
10 SOUND 1,478,300
20 SOUND 2,379,300
30 SOUND 4,239,300
```

Fig. 11.12. A chord which uses three channels.

channels. We haven't looked at control of volume yet, and the same volume has been used on each channel for simplicity, but for the best musical effects you might want to use different volumes for different channels. Because the human ear is less sensitive to low notes and high notes compared with notes around Middle C, it's often useful to sound these notes at higher volume numbers than the notes in the middle range. The chord is not quite as harmonious to the musical ear as it ought to be, because the numbers that determine the pitches of

the notes are only approximate. If you have a trained musical ear, you can experiment with small alterations in the numbers here. For example, I find a number of 380 more acceptable (line 20) than 379.

The fourth number that you can use in the SOUND instruction is one that sets the relative volume of the note. Relative volume means that if you use different values of this number with the same setting of the volume control of the CPC464 (or the stereo amplifier), you will hear different volumes of sound. You can set the *absolute* volume as you wish by using the volume control as usual. The range of values for this number is whole number from 0 (silence) to 7 (full volume). Numbers ranging from 0 to 15 can be used when the volume is being controlled by an *envelope* – that's yet another topic that we'll be looking at later. If you try to use volume numbers greater than 15, however, you'll get the 'Improper argument' error message.

Figure 11.13 shows the volume number in use. This starts by using a loop in which the volume number in a sound instruction is increased

```
10 FOR N=0 TO 7
20 SOUND 2,339,50,N
30 NEXT
40 FOR N=7 TO 0 STEP -1
50 SOUND 2,339,50,N
60 NEXT
```

Fig. 11.13. Using the volume command number.

as the loop is repeated. In the second part, the volume number is decreased as the loop runs. The effect is of a note that gets louder in noticeable steps, then softer again. The notes that are produced by musical instruments of the traditional type always change in volume like this. A lot of instruments produce notes that start fairly loud and then fade. The piano is typical of this class, because the note is loudest when a string is struck, and then fades as the 'damper' is held against the string. The CPC464 allows you to imitate this type of behaviour (an 'envelope') in other ways, however, so we don't have to form loops to control amplitude.

Music, music, music

The ability of the CPC464 to produce more than one channel of sound allows you to compose music which has harmony. Unless you are an accomplished composer, or want to be, it's best to use sheet

music as a guide. The best music to use, if you want three channels, is violin and piano or soprano voice and piano, or flute and piano. These provide a range of notes that a small loudspeaker can cope with reasonably well. Music for the double-bass does not sound good on a small speaker! Avoid music for instruments like the clarinet and bassoon, because these are 'transposing instruments', on which the notes that are actually sounded are not the same as another instrument would sound from the same music.

The best technique to use is a loop for the data of pitch number and duration for each note. Later on, we'll look at other data that you might want to use, but it's best to start simply. When you try this at first, it's advisable to have one line of DATA for each note. If you want to keep a note playing in one channel while other notes change, you will have to use methods of synchronising that we'll look at in a moment. You'll find that piano music usually sounds better if you have short pauses between notes, but that organ music doesn't need this. Experience is the main thing that you need once you have acquired the programming skills.

As an example, look at Fig. 11.14, which illustrates a bit of three-part harmony. This was not written with the aid of sheet music, and I

```
10 FOR N=1 TO 9
20 READ C1,C2,C3,P
30 SOUND 1,C1,P,7
40 SOUND 2,C2,P,5
50 SOUND 4,C3,P,5
60 NEXT
70 END
500 DATA 478,319,956,100
510 DATA 506,319,851,70
520 DATA 478,319,956,210
530 DATA 426,319,851,70
540 DATA 379,253,758,140
550 DATA 319,268,638,70
560 DATA 358,284,716,70
570 DATA 379,319,758,100
580 DATA 426,319,851,140
```

Fig. 11.14. A touch of harmony, using three channels.

had to operate by writing one channel at a time. This is comparatively straightforward, because the main program consists of a loop, with C1, C2, and C3 used for the channel notes in channels A, B and C

respectively, and P used for the duration. Using this form makes it easy to change the tune by altering the DATA lines. I wrote the number for channel 0 first, with zeros for the other channels. This was adjusted as necessary, and the duration numbers set to values that gave reasonable timing. Having got the tune as I wanted it, I added the channel 1 sound, and when that was sorted out, I added the channel 2 sound. It's not as good as it would be if I had used a score. Beethoven wrote the score, I only programmed the computer.

Special effects department

The SOUND instruction, even in this simple form, can produce a large range of useful sound effects. Let's start with a series of notes of rising pitch, which makes a useful warning, or a 'something about to happen' note. This is illustrated in Fig. 11.15. The loop that starts in

```
10 FOR J=1 TO 10
20 FOR N=200 TO 1 STEP -10
30 SOUND 2,N,1,7
40 NEXT
50 NEXT
```

Fig. 11.15. Programming a note of rising pitch, using a loop.

line 10 uses values of J which will cause the rising note to be repeated ten times. The next loop, using N, has a range from 200 to 1 in steps of −10. This will have the effect of covering a good part of the sound range, with a rapid change of pitch. These numbers are then used as pitch numbers in the SOUND instruction in line 30. This uses a very short duration, and full volume. Try it, and listen carefully. One of the problems of sound instructions is that it's almost impossible to describe the results! Unlike a graphics program, in which you can often get an idea of what will happen by looking at the listing, a sound program is nearly always a surprise until you have had a lot of experience. It's important, then, to try each of these program samples so that your own experience will grow in the right way. One good idea is to keep recordings of the actual sounds that you can replay on an ordinary cassette recorder.

Figure 11.16 shows a program that produces a warbling note. This is particularly useful for attracting attention, or for announcing an event in a game. For some reason, a warbling note attracts our attention more than a single note, which is why a warbling note

```
10 FOR J=1 TO 50
20 SOUND 2,239,5,7
30 SOUND 2,244,5,7
40 NEXT
```

Fig. 11.16. A warbling note program.

was chosen for the later types of telephones. The warble in this program uses the loop that starts in line 10. This sounds 50 pairs of notes, which are short with a duration set at 5 for each note. Remember that these durations are in units of 1/100 second. The two pitch numbers that have been chosen in this example are 239 and 244. Higher pitches are even more effective, and values like 90 and 95 give very effective attention-getting warbles.

Adding noise

You can specify a different type of sound for any of the three channels, different because what you get from it is not ordinary musical notes, but *noise*. What we call noise is a random mixture of frequencies, but very often you find that one frequency or frequency range is much louder than others. A noise source is useful because it can be used to provide effects like surf, drumbeats, gunshots, and other sounds that are otherwise impossible to produce with the ordinary SOUND command. To produce noise, we must shut off the ordinary tone source by using a zero as the pitch number. The 'noise pitch' number can take values from 0 to 15, and it must be the seventh number in the SOUND command. I know that we've skipped the fifth and sixth, but there are reasons for that, and we'll come back to them. For the moment, then, we'll make these numbers equal to zero.

Figure 11.17 illustrates noise on channel B. There is only one noise source, so you can't have different noises in each of three channels, but you can, of course, place the noise to whichever channel you like. With a stereo amplifier (take a look at the Amstrad range!) and speakers in front of you, for example, you can have a showdown at the O.K. Corral, with shots from the left and shots from the right! The number that we use for 'noise period' does not have the same effect as it has in the other channels, nor the same range. The range is 0 to 15, and the effects simply repeat if you use numbers greater than 15. You'll find that all of the numbers are useful for noise effects, but there is not much difference between the effects that are produced by

```
10 CLS
20 FOR J=0 TO 15
30 PRINT"Noise Period";J
40 SOUND 1,0,20,7,0,0,J
50 FOR N=1 TO 2000:NEXT
60 NEXT
```

Fig. 11.17. Producing noise on one channel.

numbers 1 to 5. For a better appreciation of the noise channel, however, try the program in Fig. 11.18. This gives two splendid effects that are caused by programming noise in a loop, varying different things each time. In the first set, ten 'surf' noises are produced. This is done by changing the noise period number in the loop that uses variable J. The noise starts with lower frequencies predominant, and shifts to the higher frequencies. The second part of

```
10 CLS:PRINT:PRINT"Surf on the shore....
."
20 FOR N=1 TO 10
30 FOR J=15 TO 1 STEP -1
40 SOUND 2,0,20,7,0,0,J
50 NEXT:NEXT
60 PRINT:PRINT"Now hammer that tinware..
."
70 FOR N=1 TO 10
80 FOR J=15 TO 0 STEP -1
90 SOUND 2,0,5,J,0,0,1
100 NEXT:NEXT
```

Fig. 11.18. How the noise generator can be used in producing sound effects.

the program shows how hammering noises can be produced. In this case, the volume is altered in the loop, and the noise period is fixed. The small noise period gives a tinny hammering note; larger numbers give lower pitches of noise which sound quite different.

Waveshapes unlimited

It's time now to investigate the effect of extending the SOUND command so that it controls the 'envelopes' of the sound that you hear. The effect is anything but simple, and because it's difficult to

describe in words what a noise sounds like, you simply have to try the programs and listen! First of all, I have to explain what is meant by an 'envelope'. The sounds that the simple SOUND command produces have a constant amplitude and constant frequency. In simpler words, their loudness is constant and so is their pitch as the notes play. Musical instruments, however, produce notes in which the loudness varies in each note. A piano note, for example, is loud at the instant it is struck, because that's when the hammer strikes the strings. This dies away rapidly, and it's the way in which the note's loudness dies away that makes a piano note so distinctive. Other instruments produce notes which behave quite differently, and all instruments produce notes which consist of a mixture of frequencies. That's why you can tell a piano from a violin from a flute from an oboe, even if they are all playing the same note. A graph of the amplitude of a note, plotted over the time that the note takes, is called the 'volume envelope' of the note.

In addition, the pitch of the note is never constant either. If you have ever seen a violinist in action, you will have noticed how the hand which is used to hold the string down is shaken to and fro. This gives an effect of changing pitch which is called 'tremolo'. Adding tremolo to a note can make it sound more interesting than a note whose pitch is constant. A note which has tremolo, or any other variation of pitch, will have a 'pitch envelope'.

Take a look at Fig. 11.19. This shows a volume envelope which is typical of many musical notes. It has a sharply rising amplitude at the start, which we call the 'attack'. This is followed by the 'decay' portion, in which the amplitude drops. The amplitude then remains steady for a time, in the 'sustain' section, and then finally drops to zero in the 'release' section. The CPC464 gives us the chance to synthesise even a waveform like this by using an extended version of the SOUND command. The principle is to use a command which can create a number of different envelope shapes. Each of these envelopes is numbered, and we can use this number in the SOUND command – it's the fifth number in the command. When an envelope is used, it must be allowed to control both the volume and the time of a note. If a duration number of zero is used, the envelope will take control. Whatever volume number is used will decide what the starting volume of the note is, but the envelope will take command from then on.

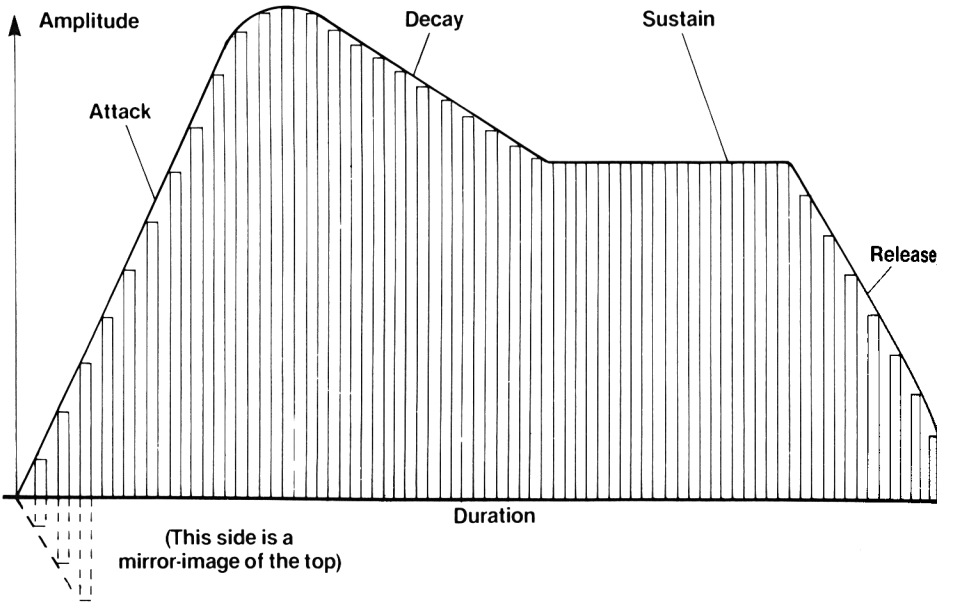

Fig. 11.19. A typical sound volume 'envelope', which shows how the volume of a note changes during the time that you hear it.

Sealing the envelope

To create a volume envelope, we have to use the ENV command, and follow it by at least 4 and not more than 16 numbers. At first sight, this looks a bit fearsome, but like any other aspect of this superlative sound system, it's well worth getting to grips with. Let's illustrate how, using simple examples. To start with, we need a planning grid and Fig. 11.20 shows a suitable one. The steps of volume use numbers 0 to 15, which is the range of volume numbers that is permitted for this command. The range of times is shown as 0 to 50. With time units of one hundredth of a second, this gives envelopes of up to a half-second long. For longer sounding envelopes, you can draw your own graph, using a different scale. The principle is that you draw your envelope shape on this graph, and try to get as close to the shape as possible, using steps of volume. You cannot have fractional volume numbers, so any sloping straight line has to be represented by a set of steps. The only thing that you can do here is to decide how many steps you can use – it can't be more than fifteen, because that's the number of steps of volume.

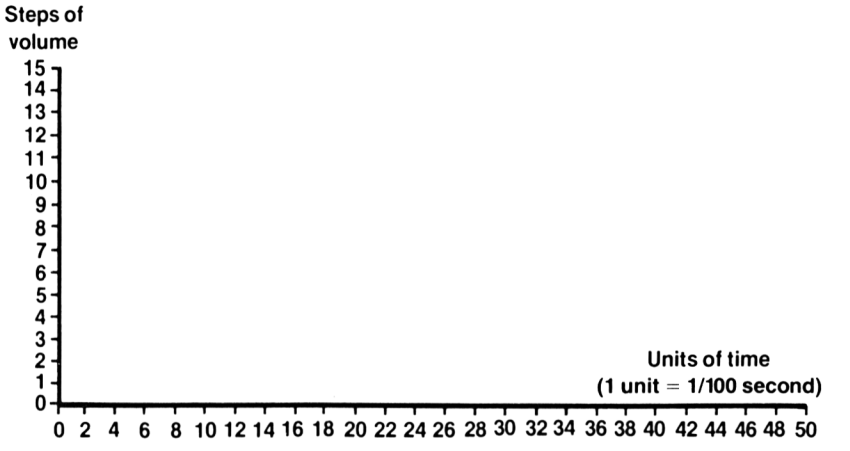

Fig. 11.20. A planning grid for the ENV command. You can draw these grids for yourself, using graph paper.

To show an example, the simplest possible case is of a sound that changes from maximum volume to zero, with a straight line shape, as the thick line in Fig. 11.21 shows. Now we have to see how many steps we can use. Just to keep things simple, the plan shows five steps, each

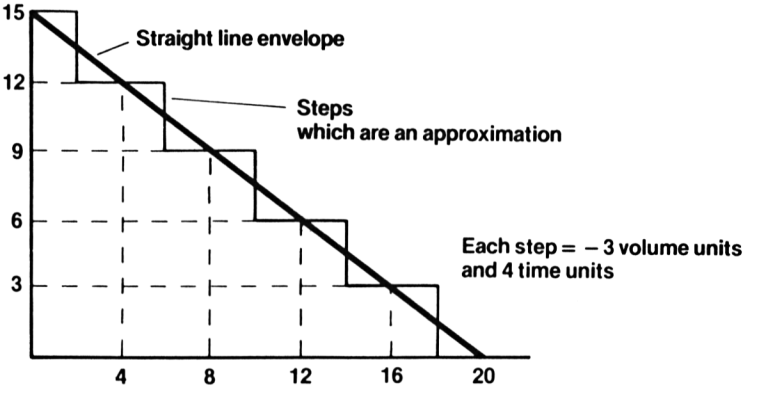

Fig. 11.21. A simple one-step envelope shape.

of volume change −3 units, and each taking a time of 4 units. The time that is measured here is the time that is spent on the level part of each step – the Manual calls this the 'pause time'. Let's call this envelope number 1. Now this has to be programmed by using ENV, then the number, which is 1, and then the number of steps, step size, and step time. This gives us ENV 1,5,−3,4. We can put this into a program, and listen to it (see Fig. 11.22).

```
10 CLS
20 ENV 1,5,-3,4
30 PRINT"Zero duration number"
40 SOUND 2,239,0,15,1,0,0
50 FOR N=1 TO 2000:NEXT
60 PRINT"Negative duration number= 10"
70 SOUND 2,239,-10,15,1,0,0
```

Fig. 11.22. The envelope opened – listen to it!

This program specifies our simple envelope, ENV 1,5,−3,4, and then a SOUND which uses this envelope. The SOUND channel select number is 2, and its tone period is 239. The duration number is 0, which allows the envelope to choose the duration. Our design of envelope has allowed a total time of 18 units, making the total time equal to 18/50 second. The starting volume number is 15, maximum volume. When we use an envelope, volume numbers from 0 to 15 are permitted. After the volume number, which sets the starting volume, we place the volume envelope number, which is number 1. The tone envelope is 0, because we aren't using that yet, and the noise period is 0, because we aren't using noise. All of this is programmed in line 40, and you can hear the effect when you run the program. Line 70 then illustrates the effect of using a negative number as the duration number. This causes the envelope to repeat for the number of times that has been specified, ignoring the sign. A duration number of −10, for example, makes the envelope repeat ten times.

Even with this very simple envelope, which uses only four numbers in the ENV command, there is a lot of scope for experiment. Just to take one rich seam, set the tone number to 1, and put in a noise number – try 12. This gives a 'sudden shot' noise, and there are many possible variations on this, and mixtures of tone and noise. For more effect, though, you can have *up to five sections* of envelope, and you can use steps that are much finer than the crude five that we used – it's best to aim at as many steps in each stage as you can have volume changes as the time permits, which means up to 15. Let's try a more ambitious envelope, which contains four sections, attack, decay, sustain and release.

This is illustrated in Fig. 11.23. Once again, the thick line shows what we are aiming at, and the steps show what can be achieved. This will need an ENV instruction with four sections, and the numbers that will be used in each section are shown at the right-hand side of the diagram. Now we have to try it out, in Fig. 11.24. The ENV

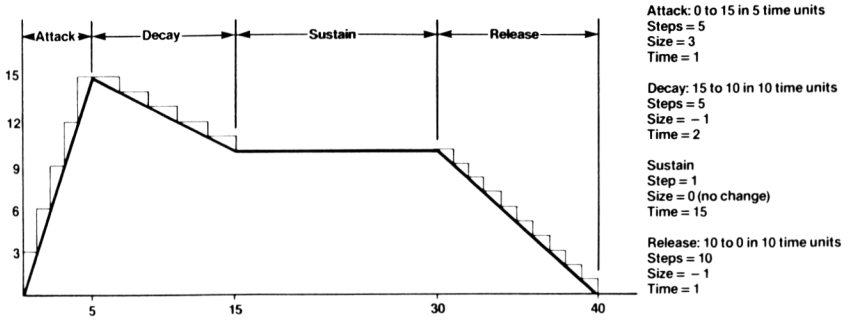

Fig. 11.23. A plan for a four-section envelope.

```
10 ENV 1,5,3,1,5,-1,2,1,0,15,10,-1,1
20 SOUND 2,478,-10,0,1,0,0
30 FOR N=1 TO 2000:NEXT
40 FOR J=1 TO 8
50 READ note%
60 SOUND 2,note%,0,0,1,0,0
70 NEXT
100 DATA 478,379,426,638,638,426,379,478
```

Fig. 11.24. The SOUND program which uses the envelope.

command starts by specifying the envelope number, then lists the step number, step size and step time for each section. Having done that, we create the sound, using a duration number of −10 to give ten strokes of one pitch. In lines 40 to 70, we try a 'clock chime' with this envelope. A bit fast, is it? That's easy to remedy – just add a fifth section of silence to the envelope. Add the numbers 1,0,40 to the end of the ENV command, and listen to the difference!

Working with these envelopes takes a bit of planning, a good ear, and a lot of practice. Though the rules for using the envelope numbers are really quite straightforward, and a lot simpler than the methods on other machines, you can make life simpler for yourself by using some hints. Try to make your steps of a sensible size. Have a reasonable number of steps. If you make the time of your envelope too short, then there won't be much to hear, unless you are trying to create noises like pistol shots. Certainly for musical notes, short duration envelopes will prove to be very disappointing. As a general rule, try to aim at musical notes which will last for more than a second. Another useful hint is to try to draw changes of amplitude along the diagonals of the squares in the planner. This ensures that

you have whole numbers for your number of steps and step size.

Tremulous notes

We saw earlier that we can also use a tone envelope for a note. The tone envelope is programmed in very much the same way as the amplitude envelope, and a tone envelope planner is illustrated in Fig. 11.25. Once again, the shape that is wanted is approximated to in a series of steps. These steps are tone number steps, however. A

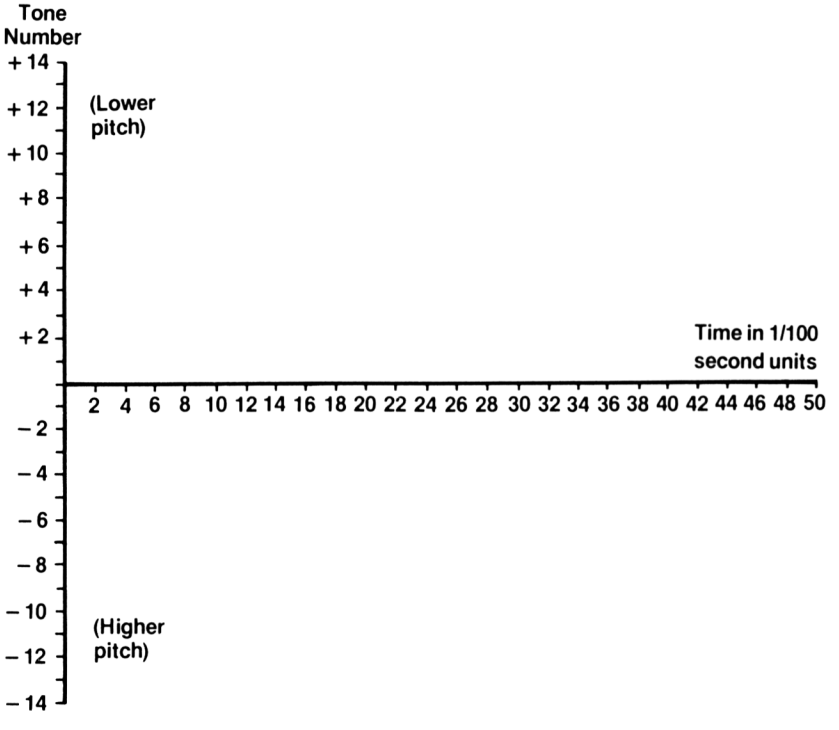

Fig. 11.25. A tone envelope planner.

positive change of tone number means a lowering pitch, a negative step means a rising pitch. As usual, an example will help, and Fig. 11.26 shows a simple plan for a tone envelope. This envelope will cause the pitch to rise for the first 8/100 second, and fall again for the next 8/100 second. As usual, drawing the lines along diagonals of the graph squares makes the plan easier to achieve. Figure 11.27 then illustrates the type of sound that we get from this. It's a good one to use for bells and springs and other odd noises!

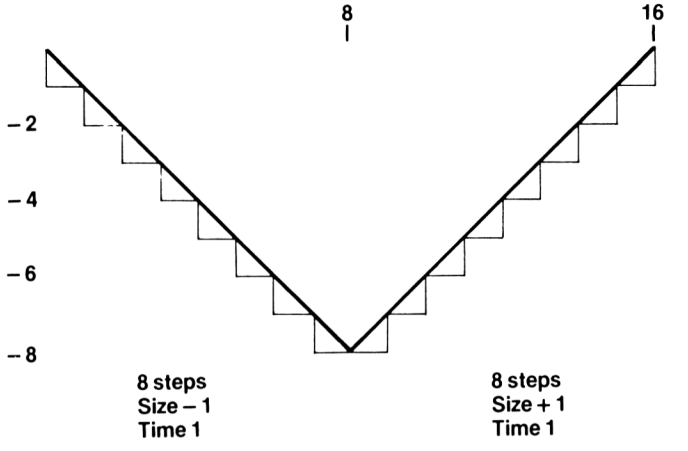

Fig. 11.26. A simple plan for a tone envelope.

```
10 ENT 1,8,-1,1,8,1,1
20 SOUND 2,478,50,7,0,1,0
30 FOR N=1 TO 2000:NEXT
40 FOR J=1 TO 8
50 READ note%
60 SOUND 2,note%,40,7,0,1,0
70 NEXT
100 DATA 478,379,426,638,638,426,379,478
```

Fig. 11.27. What this tone envelope sounds like.

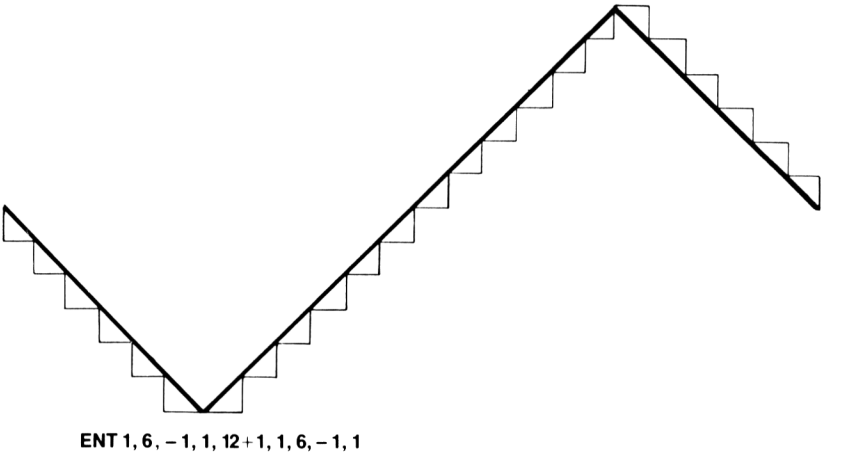

Fig. 11.28. A tremolo pattern for tone envelope.

For musical notes, a tremolo effect is obtained by making the tone number rise and fall, using a pattern like the one in Fig. 11.28. This is rather a slow tremolo, and for a lot of purposes, a faster variation would be better. In the program, Fig. 11.29, the ENT number for this displays a novelty, however. The envelope number in the ENT command has been typed as −1, though in the SOUND command, 1

```
10 ENT -1,6,-1,1,12,1,1,6,-1,1
20 SOUND 2,478,200,7,0,1,0
```

Fig. 11.29. Using the tremolo envelope. Note the use of ENT −1 to make the envelope repeat.

has been used. The effect of the minus sign in the ENT command is to make the effect repeat. If we used 1, then there would be one wobble of the pitch at the start of the note, and no more. By using ENT −1, we ensure that the tremolo keeps on going for the duration of the note.

The combinations of noise with tone envelopes are as fascinating as the combinations of noise with volume envelopes. Just to give a lovely example, take the SOUND in Fig. 11.29; make the tone number equal to 1 and the noise number equal to 7, then change the duration number to 1200. Run this one, and listen to the authentic sound of a 4-4-0 steam loco pulling hard up a slope! Needless to say, you can combine volume envelopes with pitch envelopes, to make practically any sort of note that you like. The important thing is to use some sort of orderly system for investigating sounds. Always record the notes that you obtain, and keep a note of envelope shapes. In this way, you can build up a library of sound effects which you will find extremely useful for your own programs, or even to put into other people's programs. Don't forget, also, that some of the magazines are hungry for program material, and pay (at the time of writing) from £60 per page for your listings.

Synchronisation

Space is running out. To do justice to the sound system of the CPC464 would need another book of this size. Some things just have to be omitted – particularly the details of the SQ command. In this last part, however, I would like to glance at the subject of *synchronisation*. When you pile a set of notes into the sound queue for each channel, it becomes more difficult to ensure that they are

played correctly. In fact, it's almost an impossible task on some computers. In particular, you may want two notes to be sounded together at one time, then silence in one channel when another note sounds in the first channel and so on. The CPC464 uses a scheme of synchronisation numbers in order to force this to happen. The synchronisation is achieved by using the channel select numbers, and Fig. 11.30 shows the full range. As well as the numbers which compel sound signals to be sent to the three channels, there are three numbers

Select No.	Effect
8	Synchronise with Channel A
16	Synchronise with Channel B
32	Synchronise with Channel C
64	Hold
128	Clear buffers of sound queues

Fig. 11.30. The full range of channel select numbers, including synchronising.

which cause synchronisation, one which will cause a note to be held waiting in the queue, and one which removes everything from the queues – this last number is a quick way of getting silence. You can achieve more than one effect at a time by adding the numbers together.

Figure 11.31 illustrates the principle. Line 10 is a sound command. In the ordinary way, this would sound at once but, by making the channel select number equal to 33, it does not. This channel select number consists of 1, which is the normal 'sound to channel A' number, plus 32, which is the 'synchronise with channel C' number. As a result, the sound is held in the queue, and you hear nothing until channel C sounds. After the time delay, we come to the SOUND

```
10 SOUND 33,478,300
20 FOR N=1 TO 2000:NEXT
30 SOUND 12,379,300
```

Fig. 11.31. A brief illustration of synchronisation in action.

command for channel C. This uses a select number of 12. This is made up from the usual 4, which selects channel C, plus 8 which causes synchronisation with channel A. Note that *both select*

numbers must contain synchronisation numbers if the two are to synchronise with each other. The effect, when you play this one, is that nothing sounds until the end of the time delay, then you hear the chord.

If all this seems rather daunting, remember that for a lot of effects, you will not need to use it. You can just as easily read a set of channel select numbers and tone numbers from a data list, using tone number 0 for a silence. Unless you are really strongly into musical composition, the problems of synchronisation can be overlooked! Nevertheless, to have this facility on a computer of this price range is quite excellent, and like all good talents, it doesn't fade away just because you aren't using it yet.

Appendix A
Editing

Editing means changing something that has already appeared on the screen. You can, of course, delete a character by using the DEL key, or you can retype a faulty line. You can also delete a range of lines by using the DELETE command (such as DELETE 10–100). Editing, however, means changing one feature of a line without having to change anything else. Any feature of a line – including its line number – can be changed by editing. The editing process can be carried out:

(a) while a line is being entered, before ENTER has been pressed;
(b) at a later stage, after a line has been entered, but before the program is run;
(c) when an error is signalled during running.

Dealing with these in order:

(a) While a line is being entered, all of the line-editing commands below can be used. Editing is completed by pressing the ENTER key.
(b) When the line has been entered but the program has not been run, you can use either of the editing methods that are detailed below.
(c) When the program stops with an error message, the line in which an error has been traced may appear on the screen, with the cursor on its line number. You can edit the mistake using the line editing method.

Editing methods

A unique feature of the CPC464 is that editing can be carried out in two different ways. Most computers use either line editing or screen editing. Line editing means that you have to call up the line that you want to change, using a command like EDIT 200 (to edit line 200). The cursor can then be moved over the line to locate and correct the

mistake. In screen editing, any line that you see on the screen can be edited simply by moving the cursor to it, and correcting the mistake. Most people have strong likes and dislikes about editing methods, and will accept only one of these systems. By using both, the CPC464 pleases everybody!

Editing commands

1. Line editing

If you don't already have the line on the screen with the cursor on it (as when an error is signalled), then use the EDIT command to get the line in place. Remember that there must be a space between the 'T' of EDIT and the first digit of the line number. The result will be to place the line on the screen with the cursor over the first digit of the line number. You can now move the cursor, using the arrowed keys. You can delete a character which the cursor is over by pressing the CLR key. By holding this key down, you can clear everything that is to the right of the cursor. Alternatively, you can delete whatever is *to the left of the cursor* by using the DEL key. Typing a character will insert that character *at the cursor position*. Press ENTER to complete editing a line. Remember that you can edit a line in this way while you are entering it. If you have typed

```
PRINT THIS IS THE END"
```

and you suddenly notice that you have left out the first quotes, then use the left-arrow cursor key to get the cursor over the 'T' of THIS, press the quotes key, and then ENTER. It's important to note about line editing like this that you *can press ENTER whenever you have corrected the mistake*; you don't have to shift the cursor to the end of the line.

2. Copy editing

Copy editing is the CPC464 version of screen editing. This method makes a copy of a line, but enables you to omit or replace parts of it. Any line that you can see on the screen, and even direct commands, can be copy edited. Press the SHIFT key down and hold it. Now press the cursor keys so that the cursor moves to the line that you want to edit. If you now release the cursor keys (but hold the SHIFT key down) and press the COPY key, you can make a copy of the line at the bottom of the screen. The copying will start from where the cursor was put. If you want to skip a part of the line, use the cursor

key instead of the COPY key. Press ENTER when you have copied as much as you want. If you want to copy all the rest of the line, *you must complete the copy before you press ENTER*. If you press ENTER midway along a line, only part of the line will be copied. This is a very important difference between copy editing and line editing.

This is a very versatile editing method. You can, for example, change a line number. Suppose you have line 200 on the screen, and you want to make it 1000. Type 1000, then use SHIFT and the cursor keys to send the cursor to the first command in line 200, not to the line number itself. Now copy the line, using the COPY key. Press ENTER, and you have a line 1000, identical to line 200. Type 200, and ENTER, and line 200 disappears, leaving its new copy.

You can even copy something which never had a line number. Suppose that in the middle of entering a program, you type FOR N=1 TO 200:NEXT, forgetting the line number. You don't have to type it all over again. Just type the correct line number, then use the cursor keys to place the cursor over the 'F' of FOR, and COPY the line. Press ENTER at the end of the line, and it's in place! Another excellent feature of this method, which is also used by the BBC Micro, is that you can copy a bit of one line into another. If you decide that the time delay that you have used in line 400 is also needed in line 700, then it's simple. Ensure that both lines are on the screen. Copy down line 700 until it reaches the place where you want the time delay. Shift the cursor to the place in line 400 where the routine exists, and copy it. Then go back to copying the rest of line 700, if there is any. Press ENTER when finished.

Digging out the bugs

In computing language, a fault in a program is called a *bug*, and someone who puts the faults there is called, of course, a programmer. Your programs can exhibit many kinds of bugs, and these are indicated by the error messages that you get when you try to run a program. Some of these messages are pretty obvious. 'Line does not exist', for example, means that you have used a command like GOTO 1000 or GOSUB 1000 and forgotten to write line 1000. It can also appear if you have tried to DELETE 1000 with no line 1000, or if you have a THEN 1000 ELSE 2000 following an IF somewhere.

The most common fault message is 'Syntax error'. This means that you have wrongly used some of the reserved words of BASIC. You might have spelled a word incorrectly, like PRIBT instead of PRINT.

You might have missed out a bracket, a comma, a semicolon, or put a semicolon in place of a colon. Machines can't tell what you meant to do; they can only slavishly do exactly what you tell them. If you haven't used BASIC in exactly the way the machine expects, you'll find a syntax error being reported. Another common error is 'Improper argument'. This usually means that something silly has happened involving a number. You might, for example, have used TAB(300). Of course, having read this book, you wouldn't write TAB(300) in a program, but you might have TAB(N), and the value of N has got to 300 in some sneaky way. Anything that makes use of numbers, like MID$, LEFT$, RIGHT$, INSTR, STRING$, and others can have an incorrect number used – and this will cause the 'Improper argument' error. You will also find that using a negative number in SQR(N), a negative or zero value in LOG(N), and other mathematical impossibilities will cause this error message. The cause shouldn't be hard to trace because the machine tells you which line caused the trouble.

A lot of errors can find their way into programs, even when you are entering a program that has been printed in a magazine. In general, the programs that you find in the monthly computing magazines are pretty reliable, but some are printed in a way that makes it difficult for you to enter them correctly. The main problems arise when the author of the program has used I (capital I) or l (small L) as a variable name, or has used a printer which does not have slashed zeros. Of these, confusion between O and 0 is the worst. A line like:

```
IFM=10ORJ=4ORD=2
```

can cause a lot of trouble, and one magazine seems to specialise in lines like this! If it had been printed as:

```
IF M=10 OR J=4 OR D=2
```

all would have been clear. Sometimes this has to be done just so as to be able to get a program to fit into the memory of a computer, but this is the only real excuse.

Even when you have eliminated all of the syntax errors and improper arguments, you may still find that your program does not do what it should. The CPC464 does what any machine of the Eighties should do – it gives you a lot of ways of finding out exactly what has gone wrong. One of the most powerful of these is the ESC key. This, as you know, stops the action of the machine when you press it, and restarts it when you press any other key. This, as we'll see later, can be very useful for graphics bug-hunting but, for other

programs, pressing ESC twice is more useful. This stops the program, and prints the line number in which the program stopped. What you probably don't know, however, is that you can print out the values of variables, and even alter values while the program is stopped, and then you can make the program resume by using a GOTO. Suppose, for example, that you are running a program which uses a slow count, and you press ESC twice at some early stage. The program stops, and you get a message like 'Break in 40'. That line number, 40, is important, because you can start the program again at that line *providing* you don't edit, delete or add to any of the lines of the program. Suppose that the counter variable is N. Try typing ?N, and ENTER. This will give you the value of N. Now try N=998 (say) and press ENTER. Type GOTO 40 (or whatever line number you stopped in). You will then see the count start again – but at 998! This is an excellent way of testing what will happen at the end of a long loop. Testing would be a rather time-consuming business if you had to wait until the count got there by itself. You can even make this testing process automatic! We have already looked at the command ON BREAK GOSUB. This will make sure that a subroutine is run whenever the ESC key is pressed twice. In this case, you can make the subroutine print the values of whatever variable you want to see, allow you to input others, and then return!

ESC used alone can be most useful when you have a graphics program that has gone wrong. When you press ESC, the graphics action (apart from flashing characters) will be frozen, and you can use this to see in what order things happen. Pressing another key then resumes the action, and there is no limit to the number of times that you can press the ESC key to check on how the picture is changing. If this alone isn't enough, add a delay loop temporarily to your graphics program, and run it in slow motion, using ESC to check the tricky parts.

Tracing the loops

One way in which a program can be baffling is when it runs without producing any error messages – but doesn't run correctly. This is really a sign of faulty planning, but sometimes it's an oversight, and the ON BREAK GOSUB method of tracing a fault can be very useful, because it allows you to print out the state of the variables at any stage in the program, and then carry on. Sometimes you want a simpler form of tracing, though. If your program contains a lot of

IF...THEN...ELSE lines, it often happens that one of these does not do what you expect. In such a case the CPC464 provides help for you in the form of two commands, TRON and TROFF.

TRON (how did you think the film got the name?) means TRACE ON, and its effect is to print on the screen the line number of each line as it is executed. The line numbers are put between square brackets and they are printed at the start of a screen line, in front of anything the program prints. Try typing TRON and then running a slow-acting program. TRON is particularly useful if you aren't sure what a program is doing, and it can be very handy in pointing out when something goes wrong with a loop. Remember that you can combine TRON with other debugging commands. You can, for example, stop the program, alter the variables, and then continue, with TRON showing you which lines are being executed. Typing TROFF (then ENTER) switches off this tracing process.

Appendix B
Printing

The CPC464 is delightfully easy to connect to any of the popular printers that use the Centronics parallel connection. You will need a suitable connecting cable, which you can buy at the store which sold you your CPC464. One end of this cable plugs into the PRINTER outlet of the computer, and the other slips easily into the socket on the printer. The great advantage of this system is that it gives you a choice of whatever printer you want to use. You can choose, for example, the popular and versatile Epson RX-80, or for the highest quality output the Juki daisywheel, or for graphics plotting, the amazing little Tandy CGP-115 printer/plotter. You are not confined, as you are with some makes of computer, to 'own-brand' printers, and by using the Centronics connection, you have the choice of printers in every possible price range.

According to the Manual, the CPC464 sends out two codes at the end of each line. One is the line feed code, ASCII 10, and the other is the carriage return code, ASCII 13. Most of the popular printers can be arranged so that they advance the paper when either the line feed *or* the carriage return character is used. This is usually done by a switch inside the printer. If you find that your computer prints everything on one line, or makes two line spaces between lines, then the usual remedy is to flick this switch over. I found with all of my printers, however, that I obtained double line spacing in either position of this switch, indicating that the CPC464 was sending two line feed characters. If you use an Epson RX-80, however, this can be remedied simply by setting the line spacing to the lower than normal value of $\frac{7}{72}$ inch. This is done by sending the command

```
PRINT#8,CHR$(27);"A";CHR$(7)
```

while the printer is switched on. This eliminates the problem *providing the CPC464 and the printer use the same number of characters per line*. If you set the number of characters per line to 40

for the printer, as has been done in this book, then you must do the same for the computer by using WIDTH 40. If this is not done then, when a line spills over on to another line of paper, the spacing will be too small. Fortunately, most printers allow this kind of adjustment to line spacing, but you will have to consult your printer manual if you use another type of machine.

Appendix C
KEY Antics

An essential feature of a modern computer is the provision of 'soft keys'. This means that a range of the keys can be programmed to carry out actions, so that pressing the key provides the action. Instead of typing LIST (ENTER) each time you want a listing, for example, you can have one key do this job, or you can have a key to provide PRINT TAB(2)" each time you need this you are writing text.

Though a number of computers provide special keys for this purpose, very few provide any simple way of programming. Some users of other machines may well suspect that their 'programmable keys' are little more than ornamental. The CPC464 allows you to define an action for up to 32 keys, using simple BASIC commands. There are, however, practical restrictions. The instruction codes that are produced by the action of each key must not exceed 32 characters per key. The total of these characters must not exceed 120. In practice, it's most unlikely that you will find either of these restrictions to be a handicap in any way.

Suppose, then, that you want to redefine a key. It makes sense if you redefine one that you are not going to use while you are working. You can make use of ASCII codes 128 to 159 for this purpose, and the first thirteen of these codes are already allocated to keys, the keys on the number-pad at the right-hand side of the keyboard. Appendix III, page 15, of your CPC464 manual shows the codes for these keys, and you will see that the 0 key is allocated the code 128. To make this produce LIST (ENTER) you have to program as follows:

```
KEY 128,"LIST"+CHR$(13)
```

and press ENTER. The CHR$(13) in the definition provides an ENTER at the end of the command. When you press the 0 key on the number pad, you will now see a listing!

Suppose you pick an ASCII code, such as 156, which has no key attached to it? Try it – type

```
KEY 156,"PRINT:PRINT"+CHR$(13)
```

You can now make *another key* provide ASCII 156. Take the square bracket key which is the upper one next to the ENTER key. Type

```
KEY DEF 17,1,156
```

and press ENTER. Now when you press this key, your PRINT:PRINT appears and is obeyed. The '17' in this command is the INKEY number for the key, so that any key on the board can be redefined in this way. All of these definitions will disappear when the machine is reset with SHIFT CTRL ESC. If you don't want the command that is entered by a key in this way to be executed at once, just omit the CHR$(13). When you use PRINT TAB(, for example, you will always want to add the tab number, then the closing bracket, then some text.

Appendix D
Error Trapping

Earlier in this book, we came across the idea of *mugtrapping*. This is a way of checking data that has been entered at the keyboard, to see if it makes sense or not. The mugtrapping is carried out by using lines such as:

```
60 IF LEN(A$)=0 THEN GOTO 1000:GOTO 50
```

and you need a separate type of mugtrap for each possible error. This can be fairly tedious, and it usually turns out that there is one other error that you haven't spotted. The CPC464 is one of those exclusive few machines that offers you another mugtrapping command, ON ERROR GOTO. Figure D.1 gives a very artificial example – a real-

```
10 ON ERROR GOTO 1000
20 PRINT"Type a word please"
30 INPUT A$
40 L=LEN(A$)
50 PRINT 1/L
60 END
1000 PRINT"Word has no letters!"
1010 RESUME 20
```

Fig. D.1. Using the ON ERROR GOTO command.

life example would involve too much typing. In this example, the length of a word is measured, and the number inverted (divided into 1). This is impossible if the length is zero, and the ON ERROR GOTO is designed to trap this. You could get a zero entry, for example, by pressing ENTER without having pressed any other keys. Now normally, when this happened, you would get an error message, and the program would stop. The great value of using ON ERROR GOTO is that the program does not stop when an error is found; instead it goes to the subroutine. In this example, the subroutine

prints a message, then resumes on line 20. It's delightfully simple, but it's something that calls for experience. You see, if your program still contains things like syntax errors, these also will cause the subroutine to run, and this can make the program look rather baffling as it suddenly goes to another line.

Figure D.2 shows another example. Line 20 ensures that any error will take the program to line 1000. The program then reads a set of

```
10 CLS
20 ON ERROR GOTO 1000
30 FOR N=1 TO 5
40 READ X:Y$=STR$(SQR(X))
50 PRINT"Number";X;"  Square root";Y$
60 NEXT
70 DATA 5,4,3,-2,2
80 END
1000 Y$=STR$(SQR(ABS(X)))+"J"
1010 RESUME 50
```

Fig. D.2. Another Example of error checking, with an automatic resumption.

numbers and forms a string version of the square root of each number. The catch is that one of the numbers is negative. Now the computer can't find the square root of a negative number, because this is an 'imaginary' quantity. Squaring a positive or a negative number always gives a positive value, so no real number has a square root that is negative. In a lot of applications, however, we assign a meaning to such a quantity – such as to mean a length measured at right angles to another length. When −2 is read, then the effect of line 20 is to make the program jump to line 1000 whenever the computer finds an error. In line 1000, the absolute value of −2 is found, and its square root taken. The letter 'J' is then added, to indicate that this was a negative number. Line 1010 then causes normal service to be resumed, so that the message, with the new value of Y$ is printed.

Figure D.3 shows a variation on this, which lets you know which error occurred and in which line. In this line 1005, ERL means the error line number, and ERR is the error code number. You will find a list of the error code numbers in the CPC464 manual. This is a handy line to use when you are testing the program, because if the error is one which you didn't think about, it will be noted on the screen.

```
10 CLS
20 ON ERROR GOTO 1000
30.FOR N=1 TO 5
40 READ X:Y$=STR$(SQR(X))
50 PRINT"Number";X;"   Square root";Y$
60 NEXT
70 DATA 5,4,3,-2,2
80 END
1000 y$=STR$(SQR(ABS(x)))+"J"
1005 PRINT"Line ";ERL;" -- error ";ERR
1010 RESUME 50
```

Fig. D.3. Printing the error number line in an error subroutine.

Using your PRINT

Another command which was too complicated to deal with earlier can now be looked at. This is PRINT USING. The aim of the instruction is to force printing to take a certain pattern. Following the USING part there must be a string, which can be declared beforehand. This string must take a form which suits what you want to print.

It all sounds very mysterious, so take a look at an example in Fig. D.4. In this example, A$ is defined as ###.##. This means that any

```
10 CLS
20 PRINT TAB(15)"PRINT USING"
30 PRINT:PRINT
40 N=140.2716
50 PRINT"N IS ";N
60 A$="###.##"
70 PRINT "With PRINT USING, it's ";USING
 A$;N
80 PRINT"The VAT on #";N;" is #";15*n/10
0
90 PRINT"It looks neater as #";USING A$;
N*15/100
```

Fig. D.4. A simple PRINT USING example.

number which is printed USING A$ will be printed with three figures before the decimal point and two following. In computer language, A$ is a 'formatting' string. This is by far the most useful action of

PRINT USING, but the Manual lists some others. Some of these are of interest only if you are printing quantities in dollars (yet the BASIC was written in the UK!).

Index

HOW TO MASTER THE AMSTRAD!

The Amstrad CPC464 is a giant leap forward in computing – a machine that is readily available, uses parts of proven reliability, with a design which incorporates much new thinking, and which above all comes with an excellent manual.

However, just as you would find it difficult to learn a foreign language from a dictionary, you will probably find it difficult to learn to master your Amstrad CPC464 from the manual alone. This book takes you step by step, in detail, through the commands. You are encouraged to try out your ideas, write your own programs and extend your capabilities. Soon you will see how much more your computer can do for you. You will find how business calculations are worked out, how vivid displays are obtained, how attention-catching sounds can be arranged. This book, hand in hand with the encyclopaedic manual, will lead you to greater enjoyment and a better return from your investment in the Amstrad CPC464.

The Author

Ian Sinclair has regularly contributed to journals such as *Personal Computer World, Computing Today, Electronics and Computing Monthly, Hobby Electronics* and *Electronics Today International*. He has written over forty books on aspects of electronics and computing, mainly aimed at the beginner.

More books for the Amstrad

40 EDUCATIONAL GAMES FOR THE AMSTRAD CPC464
Vince Apps
0 00 383119 1

SENSATIONAL GAMES FOR THE AMSTRAD CPC464
Jim Gregory
0 00 383121 3

ISBN 0-00-383120-5

Computer photograph courtesy of AMSOFT
Background illustration by Mike Masters

COLLINS
Printed in Great Britain

£6.95 net

9 780003 831207

SINCLAIR **AMSTRAD COMPUTING**

COLLINS

Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/