AMSTRAD CPC664 COMPUTING

INCLUDING THE DISC DRIVE AND CP/M

IAN SINCLAIR

Amstrad CPC664 Computing

Other books for Amstrad users

Sensational Games for the Amstrad CPC464 and CPC664
Jim Gregory
0 00 383121 3

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

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

Amstrad CPC664 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
Collins Professional and Technical Books 1985

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

Copyright © Ian Sinclair 1985

British Library Cataloguing in Publication Data
Sinclair, Ian R.
Amstrad CPC664 computing.
1. Amstrad CPC664 (Computer)
I. Title
001.64'04 QA76.8.A9

ISBN 0-00-383184-1

Typeset by V & M Graphics Ltd. Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system or transmitted,
in any form, or by any means, electronic, mechanical, photocopying,
recording or otherwise, without the prior permission of the
publishers.

Much of the material in this book has previously
been published in *Amstrad Computing* and
The Amstrad CPC464 Disc System both by Ian Sinclair.

Other books for Amstrad users

Sensational Games for the Amstrad CPC464 and CPC664
Jim Gregory
0 00 383121 3

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

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

Contents

Preface		vii
1	Setting up the CPC664	1
2	Putting It All On The Screen	26
3	A Few Variations	40
4	Repeating Yourself	58
5	Strings and Other Things	72
6	Menus, Subroutines and Programs	89
7	The Disc System	107
8	The CP/M Operating System	117
9	BASIC Filing Techniques	133
10	A Database Example – Filing Cabinet	148
11	Windows and Other Effects	164
12	Starting Graphics	179
13	Guide to Greater Graphics	192
14	Sounding Out the CPC664	209
15	Printers	234
Appendix A: Editing		252
Appendix B: The ASCII Codes in Hex		258
Appendix C: KEY Antics		260
Appendix D: Error Trapping		262
Appendix E: Use of the CTRL Key in CP/M		266
Appendix F: Dotted Lines		267
Appendix G: XOR, AND, OR		269
Index		272

Preface

The Amstrad CPC664 has been designed as the serious computer user's dream machine. It comes complete with disc drive and monitor at a price which is excellent by the standards of home computers, and almost unbelievable by the standards of the 'business' machines with which it can also compete. Many buyers of the CPC664 may never have owned a computer previously, others may have used an early model of the machine which did not have a disc drive, others may have used a machine which was very differently designed. All categories of prospective buyer of the CPC664 will, I hope, find much that is useful in this book.

The manual that comes with the CPC664 is one of the best, but it is always an impossible task to cater for every need, and the manual caters best for the user who has had a reasonable amount of experience in computing and programming. Very few computer owners are likely to have used the CP/M operating system on three inch discs, and former users of home computers will not have encountered CP/M at all. With this in mind, I have devoted a considerable amount of this book to the disc system of the CPC664 and to the way in which it uses CP/M. I have also included a section on printers, because most users of a machine in this class will wish to make use of a printer.

Practically all users of the CPC664 will find at some time that they have applications for the machine which cannot be solved, or which can be solved only in a clumsy way by off-the-shelf software. When this type of problem arises you will need to learn how to program the CPC664 for yourself. After all, a large portion of the price of any computer is attributable to the fact that it is designed to be programmed by the user. If you own a computer and don't program it for yourself, it's rather like owning a Poggenpohl kitchen and then taking all your meals at the pub! If you have never programmed a computer before, then this book will show you how, step by easy step. If you have programmed some of the earlier models of home computers, then this book will open a completely new world of

programming to you, with the advantages of using discs stressed throughout. The Amstrad CPC664 does not use the primitive version of the BASIC programming language that you may have seen on the old home computers, so its BASIC really has to be learned from scratch if you have previously used an older type of machine.

The listings in this book have all been obtained directly from the computer, using an Epson printer. This might seem unimportant, but there are still books appearing in which all of the listings have been retyped, with errors appearing in almost every example. If you have problems with any of the listings in this book, you should check your typing rather than wonder if there is a mistake in the listing. All of the screen displays that are described were obtained on the Amstrad colour monitor.

Several users, mainly educational users, need to connect computers to large-screen TV receivers in order to demonstrate to large numbers of viewers. This can be done using the Amstrad power pack for the CPC464 and connecting the video output of this adaptor to the TV receiver. I have therefore included details of how to set up and tune a TV receiver for the CPC664 signals, though most buyers will use the monitor which is part of the CPC664 package.

As always, I am greatly indebted to the many people who made this book possible. I must congratulate Amstrad Consumer Electronics plc for yet another excellent machine, and for the open way in which they release information. At Collins Professional and Technical Books, Richard Miles and Janet Murphy have worked wonders with my manuscript, and the most efficient team of typesetters and printers in the business have collaborated to produce this book in record-breaking time. I am certain that you will agree that the effort was worthwhile.

Ian Sinclair

Chapter One

Setting up the CPC664

The CPC664 package contains the CPC664 itself, a system disc 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 CPC664. 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 CPC664 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 disc. For that reason, it's very important to be able to use the disc system 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 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 CPC664 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.

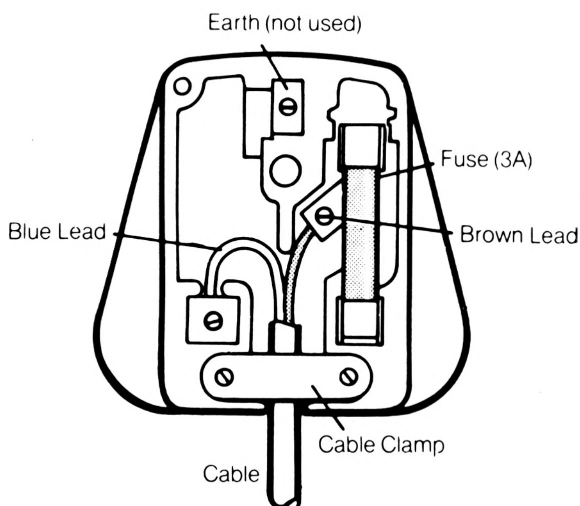


Fig. 1.1. The connections to a three-pin mains plug. Only the live and neutral leads are used. If you haven't done this sort of thing before, play safe and hand it to an electrician.

Once over that hurdle, you are almost ready to put the CPC664 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 CPC664 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 CPC664 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 CPC664, you will find that you have to keep plugging and unplugging the aerial cable and the CPC664 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 CPC664 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 CPC664 to the TV or to the adaptor using the special cable that is provided with the CPC664 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 CPC664's signals need not be a colour type, not to start with at least. The skills of

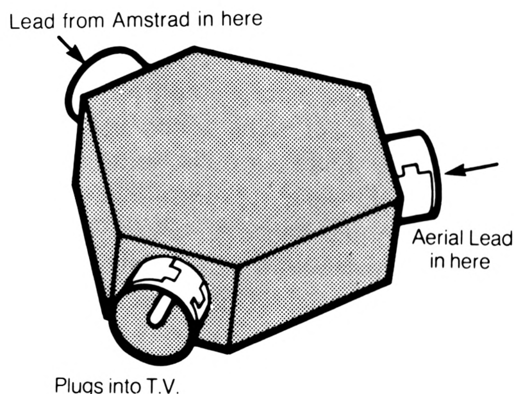


Fig. 1.2. A TV aerial cable adaptor, which allows you to keep both the computer and the aerial connected.

programming a CPC664 do not require you to see the results in colour until you come to the colour instructions of the CPC664 in Chapter 11. When you use a black and white TV or monitor to show the CPC664 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 CPC664 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 CPC664 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 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 CPC664 is such an inviting computer that you will probably want to add a printer and perhaps a modem 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 CPC664 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.

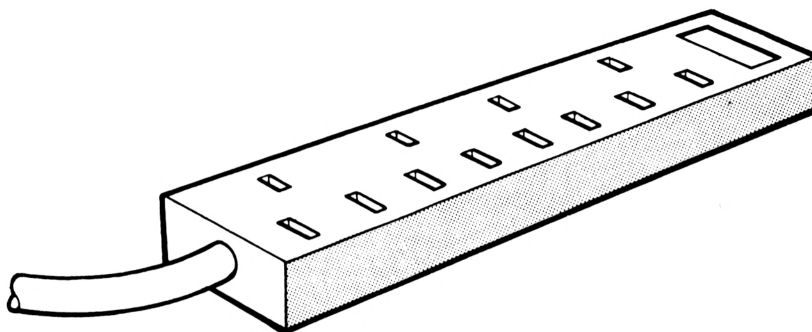


Fig. 1.3. A four-way socket strip which avoids the use of the old-style adaptors.

The CPC664 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 CPC664 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 a modem 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 CPC664, unlike cheaper home computers, makes use of a disc system 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 disc system can record signals on a magnetic disc, and these remain recorded after 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, the CPC664 has its own built-in disc system. This has several advantages. For one thing, you have no fussy connections to make, as you often have when using a separate system. The other point is that this built-in disc system is very conveniently situated and is a much more reliable way of storing information. Even a disc is not perfect, though, and it's wise always to make two copies of any computer program that

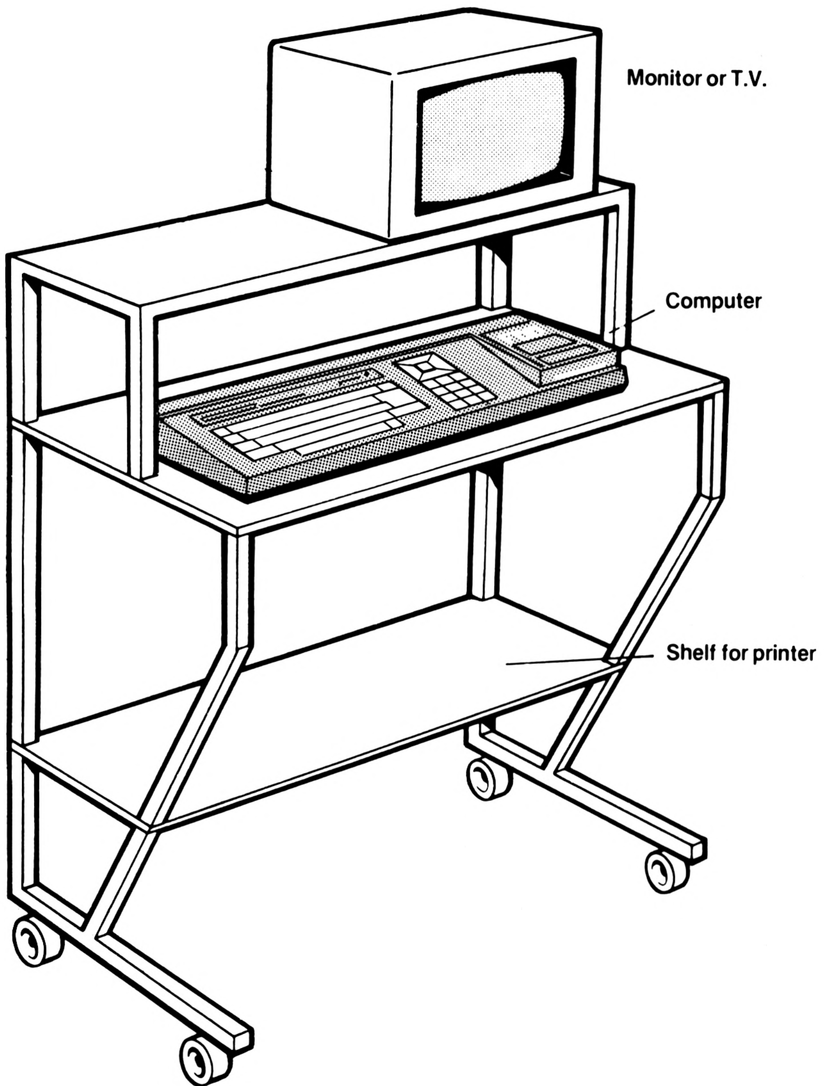


Fig. 1.4. Using a Selmor stand to house all the bits and pieces of a CPC664 computer system.

is specially valuable to you – the second copy is called a *back-up*. The CPC664 has a socket to which you can connect an ordinary tape recorder if you need to use one.

The next step is to switch on the TV receiver and the CPC664, or monitor and CPC664. 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 CPC664. 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 CPC664 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 CPC664'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. 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 CPC664'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 CPC664 signal appear.

What you are looking for, if the CPC664 hasn't been touched since you switched it on, is the screen display that is illustrated in the manual. The second line of this display is a copyright notice. 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 CPC664's signal during this time, you'll see the message 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 CPC664 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 CPC664 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

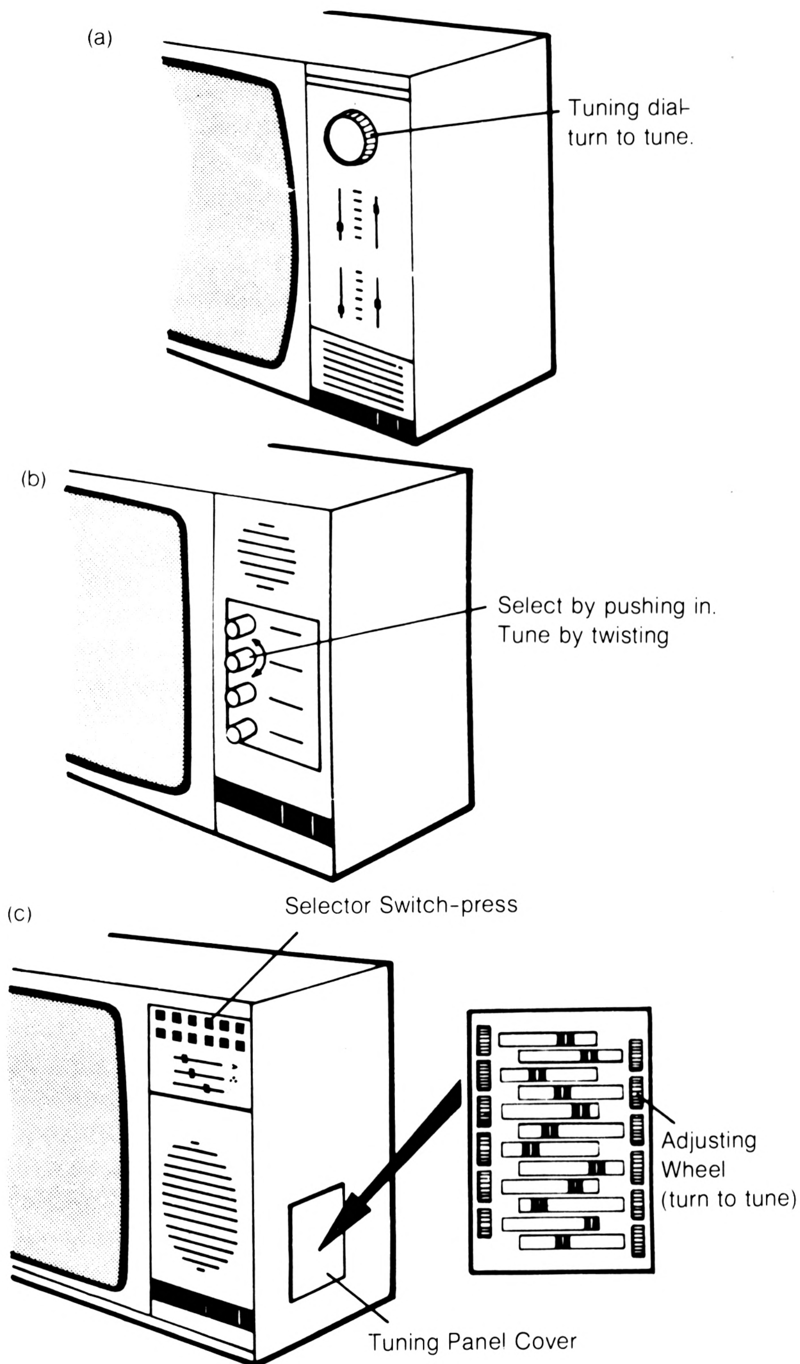


Fig. 1.5. TV tuning controls. (a) Single dial, as used on black and white portables, (b) four-button type, (c) the more modern touch-pad or miniature switch type.

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 CPC664 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 CPC664, the business of mastering the use of the CPC664 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 CPC664 machine – the worst you can do is to lose a program that was stored in the memory. You can, however, damage the CPC664 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 discs if you attempt to take them out while they are still running, or if you switch the machine on or off with a disc 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 disc in place, or remove a disc while the disc drive is being used. Keep your discs 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 CPC664. If we ignore the groups of keys at the right-hand side, then most of the CPC664 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 CPC664 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 CPC664 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 CPC664 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 CPC664 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.

Disc recording introduced

Unlike tape, which is pulled in a straight line past a recording/replay head, a disc spins around its centre. When you insert a disc into a drive, the protective shutter is rotated so as to expose part of the disc. When the drive is activated a hub engages the central hole of the disc, clamps it, and starts to spin it at a speed of about 300 revolutions per minute. The disc itself is a circular flat piece of plastic which has been coated with magnetic material. It is enclosed in a hard plastic case to reduce the chances of damage to the surface. The hub part of the disc is also built up in plastic to avoid damage to the disc surface when it is gripped by the drive. The surface of each disc is smooth and flat, and any physical damage, such as a fingerprint or a scratch, can cause loss

of recorded data. The jacket has slots and holes cut into it so that the disc drive can touch the disc at the correct places. The slot and one hole (one of each on each side) are covered by a metal shutter when the disc is withdrawn from the drive. You can see the disc surface if, holding the 'A' side uppermost, you insert your thumbnail into the slot at the right-hand side of the disc casing, near the front. By sliding you nail back, you engage the sliding peg which acts on the shutter, and you can turn the shutter until the disc surface is visible. Do *not* touch the disc surface, sneeze on it, or do anything which could leave any marks on the surface.

Through the slot that is cut in the casing (Fig. 1.6), the head of the disc drive can touch the surface of the disc. This head is a tiny electromagnet, and it is used both for writing data and reading. When the head writes data, electrical signals through the coils of wire in the head cause changes of magnetism. These in turn magnetise the disc surface. When the head is used for reading, the changing magnetism of the disc as it turns causes electrical signals to be generated in the coils of wire. This recording and replaying action is very similar to that of a cassette recorder, with one important difference. Cassette recorders were never designed to record digital signals from

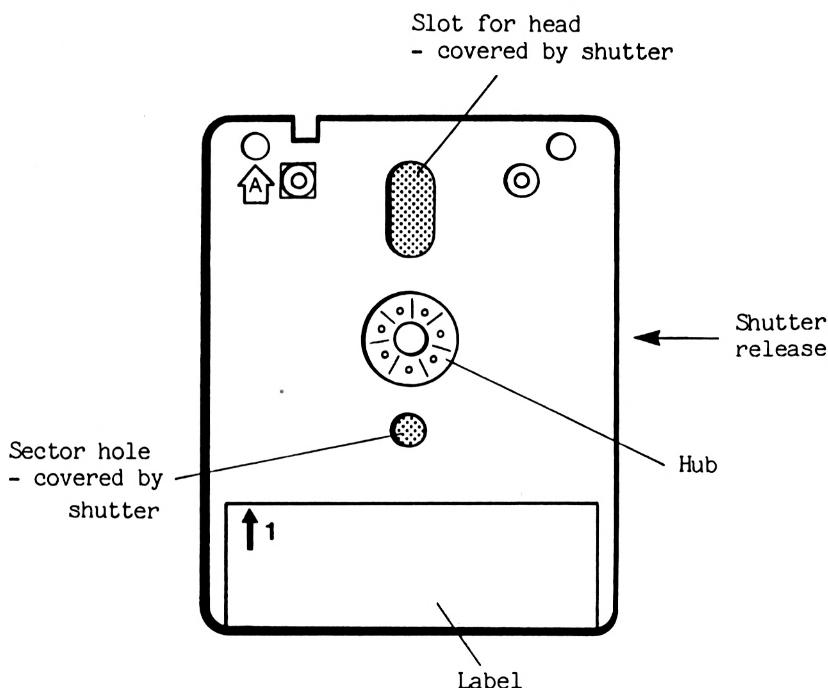


Fig. 1.6. The slot in the casing of the disc is there to allow the head of the drive to touch the disc surface.

computers, but the disc head is. The reliability of recording on a disc is therefore very much better than you can ever hope for from a cassette.

Unlike the head of a cassette recorder, which does not move once it is in contact with the tape, the head of a disc drive moves quite a lot. If the head is held steady, the spinning disc will allow a circular strip of the magnetic material to be affected by the head. By moving the head in and out, to and from the centre of the disc, the drive can make contact with different circular strips of the disc. These strips are called 'tracks'. Unlike the groove of a conventional record, these are circular, not spiral, and they are not grooves cut into the disc. The track is invisible, just as the recording on a tape is invisible. What creates the tracks is the movement of the recording/replay head of the disc drive. A rather similar situation is the choice of twin-track or four-track on cassette tapes. The same tape can be recorded with two or four tracks depending on the heads that are used by the cassette recorder. There is nothing on the tape which guides the heads, or which indicates to you how many tracks exist.

The number of tracks therefore depends on your disc drives. The vast majority of disc drives for other machines use larger discs with either 40 or 80 tracks. Forty-track drives use 48 tracks per inch, and 80-track drives use 96 tracks per inch. The CPC664 disc drive uses a 3-inch disc with 40 tracks, but with 96 tracks per inch. This forces you to find a source of these special discs, however, which are by no means common at the time of writing. In fact, at the time of writing these discs were in *very* short supply, and of the independent suppliers, only Disking were advertising them. Be very careful when you send for discs that you specify the 3-inch type. Several business computers, such as Apricot, have standardised on the Sony 3½-inch disc, and this size is quite easy to find. It is, however, *not* suitable for your CPC664 disc drive.

Once you have accepted the idea of invisible tracks, it's not quite so difficult to accept also that each track can be divided up invisibly. The reason for this is organisation – the data is divided into 'blocks', or sectors, each of 512 bytes. A byte is the unit of computer data; it's the amount of memory that is needed for storing one character, for example. Each track of the disc is divided into a number of 'sectors', and each of these sectors can store 512 bytes. Conventional 40 or 80-track discs use ten sectors per track, but the CPC664 system uses 9 sectors per track, allowing 512×9=4608 bytes to be recorded on each track. Two tracks are reserved on the Master disc (the *system* disc) for holding essential data, leaving 38 tracks for your use. This

corresponds to a total of 175104 bytes free, which is 171K. It is possible to make use of the reserved tracks if you are, for example, storing only word processing text on a disc. In this way, 180K can then be stored on the disc.

The next thing that we have to consider is how the sectors are marked out. Once again, this is not a visible marking, but a magnetic one. The system is called 'soft-sectoring'. Each disc has a small hole punched into it at a distance of about 14 mm from the centre. There is a hole cut also through the disc jacket, so that when the shutter is swung aside and the disc is turned round, it is possible to see right through the hole when it comes round. When the disc is held in the disc drive, and spun, this position can be detected using a beam of light. This is the 'marker', and the head can use this as a starting point, putting a signal on to the disc at this position and at eight others, equally spaced, so as to form sectors (Fig. 1.7). This sector marking has to be carried out on each track of the disc, which is part of the operation that is called 'formatting'.

Formatting discs

Formatting discs, as we have seen, consists partly of the action of 'marking out' the sectors on a disc. The formatting action, however, should also test the disc. This is done by writing a pattern to each sector, and checking that an identical pattern is read back later. Failure to do so indicates a faulty sector, and a disc with such a fault should be returned to the supplier with a request for a replacement. These small discs cost three times as much as a conventional floppy disc, and they *ought* to be perfect. The prices will probably come down from their present levels, and some of the large suppliers are already quoting prices of around £39.90 for a box of ten.

Formatting, then, consists of marking out sectors and testing them. This takes about half a minute, and will normally end with a message which asks you if you want to format another disc. Any fault which is found at the formatting stage will be reported. This does not necessarily imply a *disc fault* however. All 3-inch discs make use of a small sliding shutter (Fig. 1.8) which exposes or covers a hole at the front left-hand side of the disc casing. If this hole is exposed, the disc is 'write-protected' which means that it cannot be formatted. Now if the disc is protected in this way, it's probably because you wanted to preserve some information that has been recorded on it. Since formatting will wipe the disc clean, the message is a warning to you

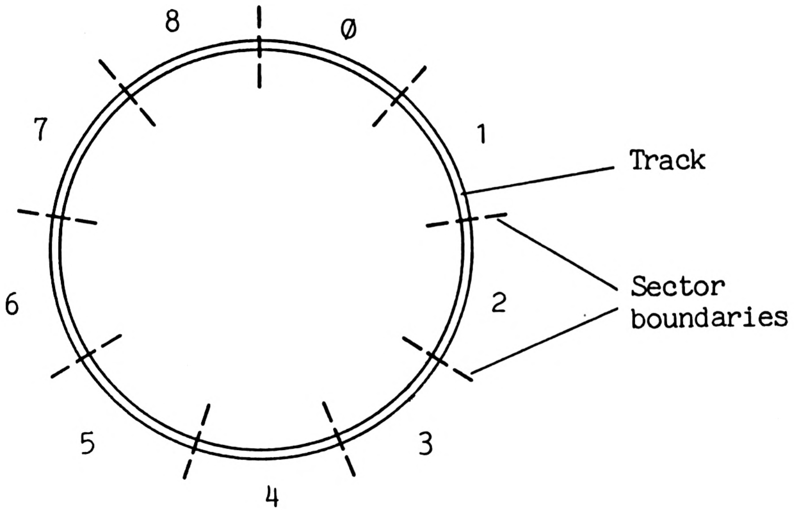


Fig. 1.7. How the disc sectors are arranged. These are not visible, because they consist only of magnetic signals.

that you might want to think again. If you really want to format, then you have to slip the plastic cover over the write-protect hole. Note that if you have been used to ordinary floppy discs of 5¼-inch size, that this protection action works in the *opposite* way. On the ordinary floppy, a slot has to be covered to protect the disc; on the 3-inch disc, the hole has to be *open* for protection.

The formatting action is carried out when a set of instructions has been typed and entered. The CPC664 disc system uses two controller programs, called AMSDOS and CP/M 2.2 respectively. Of the two, AMSDOS is intended for use with the Locomotive BASIC language

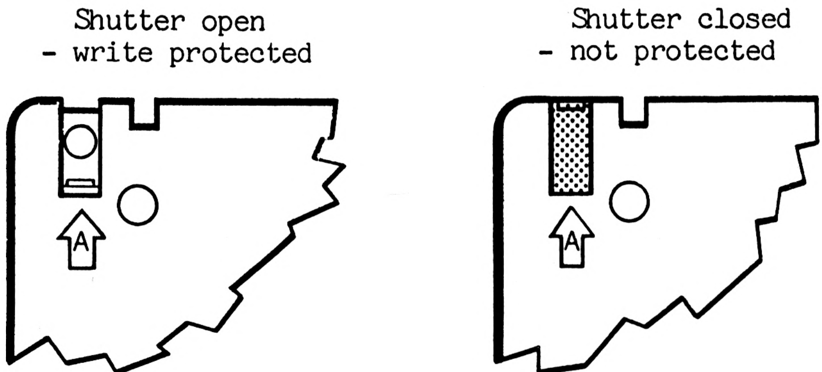


Fig. 1.8. The write-protect hole and its shutter. You will need the tip of a ball-pen to slide this shutter in and out.

of the CPC664, and it's the system that you will use along with programs that are written in BASIC. CP/M 2.2 is a more general disc system, which handles some of the tasks that are common to all disc operations, like formatting. In other words, if you want to format a new disc or reformat an old one, you have to make use of the CP/M operating system.

First of all, you will need the Master disc. Make sure that the write-protect hole is exposed, because you cannot take risks with this disc – it is *valuable*. If you don't believe me, then just try getting a replacement! Insert the Master disc in the drive, making sure that it is the correct way round. The discs are double-sided, with the sides labelled as 'A' or 'B' at the front left-hand side, next to the write-protect hole. The actual recording and reading is carried out on the *under* side of the disc, but the labelling is placed so that side A or 1 is the one being used when this number is *uppermost*. The flap on the front of the drive unit is opened when you push the disc into the slot. Hold your Master disc with the 'A' side facing up. On the label of the Master disc, incidentally, side A is labelled as '1' and side B as '2', and this system is used on other labels. Slide the disc into the slot. Don't use any force to do this, because you can jam the disc if you do so. Press the disc in firmly until it stays put and clicks into place. The disc is now ready for formatting.

You then have to type the command which will allow the CP/M system to take control of the disc drive. This command is:

    |CPM (or |cpm)

then press ENTER. As in Amstrad BASIC, commands can be in upper or lower case. To avoid confusion with other text, however, all command words will be printed in upper case in this book. We'll look later in more detail at the effect of this and similar commands. You will see the red 'drive working' light brighten, and hear the disc drive motor whirring. Soon the screen will clear to a bright pale-blue colour, change to 80-column mode, and display the message:

    CP/M 2.2 Amstrad Consumer Electronics plc

followed on the next line by the prompt A> and the cursor. This distinctive screen layout is used to remind you that you are using CP/M and not AMSDOS. I, for one, find the letters hard to read in this format, and numbers *very* difficult to distinguish. If you find the same difficulty, then be particularly careful when you are using this mode.

You must now type the word 'format', and press ENTER. This will bring up the message:

Please insert the disc to be formatted into drive A
then press any key:

followed by the cursor. You must now remove the Master disc, and insert the disc which you want to format. Make sure that this disc is the correct way up for the side that you want to format (1 or 2 uppermost), and then press the spacebar or the ENTER key. Unless the disc is write-protected, you should then find that the formatting action takes place. If the disc is write-protected and you change your mind about formatting it, then press CTRL-C (CTRL key and C key together). This will abort the formatting program, and produce the message:

Please insert a CP/M system disc into drive A then press any key:-

You should then insert the Master disc, A side up, into the drive and press the spacebar or ENTER key. The drive will spin briefly, restoring the action of the main CP/M program. This same message will also appear if formatting has been successful, and you have answered 'N' to the question:

Do you want to format another disc (Y/N):

Finally, you will need to return to the Amstrad operating system if you want to make use of the disc with BASIC programs. With the Master disc in place, either side up, type AMSDOS (or amsdos) and press ENTER. This will restore the familiar screen with 40 characters per line, and the 'Amstrad BASIC 1.0' message.

The formatting action reserves some sectors on the third track of the disc. This portion is reserved as a way of storing information about the contents of the disc. To put it crudely, the disc system reads the first few sectors of this track to find if the filename for a program is stored on the disc, and then to find at which sector the program starts. With this information, the head can then be moved to the start of the program, and loading can begin. This part of the track is known as the directory, which keeps a record of what tracks and sectors have been used, and which are free for further use. The directory entries consist of filenames and numbers which indicate which track and sector is used for the start of each program or other file that is stored on the disc. The filename for a disc consists of up to eight letters for the main name, and (optionally) three letters for the 'extension'. The 'extension' may be a novelty to you, and we'll deal with it later. To wipe a program or some data from the disc, simply

remove its directory entry – the data remains stored on the disc until it is replaced by new data. This can *sometimes* allow you to recover a program that you thought you had erased.

Storage space

How much can you store on a disc? The Amstrad system uses 9 sectors on each track, and a maximum of 40 tracks; 38 tracks on discs which contain the CP/M system program. This makes a maximum of 360 sectors, if the system tracks are used as well. Of these, up to 4 sectors will be used for the directory entries, and this leaves 356 sectors free for you to use.

Each of these sectors will store 512 bytes, which is half of a kilobyte. If you take 356/2, giving 178, you end up with a figure of 178K on a single side of a 40 track drive. Not all of this will normally be usable, however, because data is not stored at every possible point on the disc. This is because the disc operating system works in complete sectors only. Suppose you have a program that is 1027 bytes long. The disc operating system will split this into groups of 512 bytes, because it can record 512 bytes on one sector. When you divide 1027 by 512, you get 2 and a fraction – but the DFS does not deal with fractions of a sector. Three sectors will be used, even though the last sector has only 3 of its 512 bytes recorded. When the next program is saved it will start at the next clear sector, so that the unused bytes are surrounded, and there is no simple way of making use of them. If you save many short programs on the disc you will find that a lot of space may be wasted in this way. A set of short BASIC programs, for example, will use one sector each. There is another way in which space can be wasted if you keep a large number of very short programs on a disc. Each program will have a separate directory entry, and when the directory track is full no more entries can be accepted. The system, however, allows up to 64 directory entries on a disc, so you would have to be very fond of short programs to run out of directory space!

The large amount of storage space, theoretically up to 180K, on a disc contrasts with the 41K or so which you have available for BASIC programs on the CPC664. For long programs, then, a disc system can be used as a form of extra memory. If a long program is split into sections, the sections can be recorded on a disc, and a master program entered into the computer. This master program can then call up different sections from the disc as needed, giving the impression that

a very large program is, in fact, operating. The use of a disc system therefore allows you not only to load programs more quickly and store a lot of data, but also to use the computer as if it had a very much larger amount of memory.

Care of discs

1. Keep discs in their protective boxes when they are not inserted in the drive. If you drop a box, it will chip, but this is better than chipping the disc jacket.
2. Buy discs from a reputable source, such as Amsoft or one of the large disc suppliers. At these prices, you can't afford to take risks.
3. Never pull back the protective shutter unless you need to – which is normally never!
4. Never touch any part of the inner disc.
5. Keep your discs away from dust, liquids, smoke, heat and sunlight.
6. Avoid at all costs magnets and objects that contain magnets. These include electric motors, shavers, TV receivers and monitors, telephones, tape erasers, electric typewriters, and many other items which have surfaces that you might lay discs onto.
7. Label your discs well. If the label on the disc is not large enough, use self-adhesive labels in addition – but don't cover any of the shutters,
8. Remember that the disc is read and written from the *underside*.

Fig. 1.9. Taking care of your discs. They are not as fragile as this might suggest, but remember that each disc can hold a lot of valuable programs.

Figure 1.9 lists some precautions on the care of discs. These may look rather restrictive, but remember that a disc is precious. It can contain a lot of data, perhaps all your programs. An accident to one disc, then, can wipe out all your work at the keyboard, or all the programs you have bought over the course of a year! Always make a backup copy, and always take good care of your discs. If you leave a fingerprint on a piece of tape, you may cause some loading difficulties on that piece of tape, but it's unlikely that you will lose a whole program. A fingerprint on the surface of a disc could make the directory impossible to read, so that the whole disc is useless. Similarly, a disc can be demagnetised by strong magnetic fields. These fields can be around loudspeakers, TV receivers or monitors, headphones, and electric motors. All of these should be regarded as potential disc-killers and avoided.

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 CPC664, but if you have been pressing keys at random then it's a good idea to switch off again, then on, with any disc 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 CPC664. 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. 1.10, 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. 1.10. A program for testing the disc 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. 1.10.

Now make sure that you have a disc ready. The disc must have been formatted before you can use it, and it's a good idea *always* to format a disc when you unwrap it. If you attempt to use an unformatted disc you will get an error message, and no program can be recorded on the disc.

Place the disc in the drive, and note which number or letter is uppermost. It will save you time later when you want to find a program if you know which side has been used. Now type SAVE"TEST" (or save"test") and press the ENTER key. The disc will spin briefly, with the red light indicating action, and then stop. By this time your program will have been recorded. In addition, the part of the disc which is used as a 'catalogue' will have a note of the filename TEST and at what part of the disc this program is stored. It's this catalogue action of a disc system which makes discs so useful, because it means that the program can be recovered at any time just by using its filename of TEST. It also means, however, that you can't record another program with this same filename.

To load (i.e. replay) a program that is on disc, you can type LOAD "TEST" (or whatever filename you have chosen) and press the ENTER key. If the disc is correctly inserted in the drive, it will spin, and the 'Ready' prompt will reappear shortly to indicate that the program is loaded and ready. If you used the wrong filename, you will either get the wrong program or an error message, depending on whether a file of that name exists. If there is no program called TEST on the disc, for example, you will get the error message:

```
TEST    . not found
```

Not all programs will load in this way. You will find, for example,

that the ROINTIME demonstration program on the Master disc will not load in this way. That's because, like most games programs for the CPC664, it is written in protected machine code rather than in BASIC, which is the language which you would normally use for programming. A protected machine code program has to be RUN rather than just loaded, and if you try to use a LOAD command, you will get some form of error message. If, for example, you use LOAD"ROINTIME", you will get the 'not found' message. If you notice that the full title of the program is ROINTIME.DEM, and you use this filename, you will get the 'memory full' error message. You can make use of the program only by typing RUN "ROINTIME.DEM", and then ENTER. This loads and runs the program, but you will find that you need to press CTRL SHIFT ESC to leave the program. Once again, you can copy such programs only if you are reasonably proficient in machine code, or have purchased an unlocking program.

The ordinary LOAD command fails also to load the LOGO program which is on the B side of the Master disc. (LOGO is another computing language which is designed for beginners.) This is because LOGO is in machine code and has been saved by using CP/M. To load it, then, you must switch to CP/M, and because of the way that DR. LOGO has been recorded, this will also load the LOGO. Switch to CP/M by typing |CPM (ENTER), and wait. You will see the A> prompt, then the name LOGO and a copyright message. The more normal way of loading a CP/M program is to type |CPM, wait until the disc is ready, then type the name of the program, and then press ENTER. This is all that you need to load normally when CP/M is in use. Unless you use some of the (very expensive) business programs which have been transferred on to 3-inch discs, however, LOGO might be the only program that you are likely to load from CP/M!

Loading is generally much faster than storing (recording on the disc), because the DFS (disc filing system) carries out a check on data when it records, but not when it replays. If you get any sort of error message when you are saving a program, then it's wise to assume that the program has not been saved, and to save it again. When you have saved a program on disc, it's time to take a look at the way the disc keeps track of your program. This is done by reading the directory of the disc. Make sure that you are using AMSDOS (dark blue background, orange-yellow print), and then type CAT and press ENTER. If you happen to be using CP/M, then type DIR instead of CAT. The AMSDOS CAT command gives you a list, in alphabetical order, of the filenames, file types and size of each stored file. It also shows under this list how many kilobytes of storage remain unused and

available on the disc. This is a more useful display than the one you get by using DIR in CP/M, because the DIR display does *not* show the file sizes, nor does it show the remaining space. On CP/M, however, you can find the size of a file by using STAT. Typing, for example, STAT ED.COM will produce the file size and arrangement on the disc of the file called ED.COM. When you use STAT, you have to follow it with the *full* filename, including the three letters which follow the dot.

You can also print out the DIR display, assuming that you have a printer connected. If you type DIR, and then follow it with CTRL P before pressing ENTER, the directory will be printed on paper as well as appearing on the screen. It's very convenient to keep printouts of your directories because you can then find what is on each disc without having to insert the disc and use CAT or DIR. With over one hundred discs in use, I wouldn't want to be without this facility!

Remember that each disc has two sides, but only one side can be read by a drive. You will have to turn the disc over to CAT or DIR the other side. If you have only one drive, the commands that we have looked at are all you need, but with two drives, you will have to select the drive that you want to use *before* you attempt to LOAD or SAVE or CAT that drive. The first drive is labelled 'A', and is the one which is attached to the connector at the *far* end of the cable. The cable has another connector, and the drive on this connector is labelled 'B'. At switch-on, drive A is always selected. If you want to select drive B, then type | B if you are using AMSDOS, or B: if you are using CP/M. In CP/M, you can add the drive letter to a filename. This means that you can be using drive A normally, but load a file called ADDFILE from drive B by using:

```
B:ADDFILE
```

which will load this file from drive B, and then switch back to using drive A again. You should *not* attempt to use B: if you have only one drive, because this can cause the operating system to become jammed. To recover, press CTRL SHIFT and ESC together.

You *must* keep a careful record of the filenames that you use. This is because the disc drive will quite happily replace one program with another of the same name. If you have just completed a program and you want to save it, then always use CAT to read the directory to find if you have used a filename already. The old program is *not*, however, wiped from the disc. Instead, it is renamed with the extension label BAK. For example, suppose you wanted to save a BASIC program called "EFFORTS". If there is another BASIC program of this name on the disc, it will appear in the catalogue as EFFORTS.BAS – the

extension name of BAS has been placed there *automatically* by the action of the system. When you record your new program using the same name, it gets the name of EFFORTS .BAS, and the other program is renamed EFFORTS .BAK. You can do this only once, though. If you save yet another program with the file name EFFORTS, then the first one will be wiped from the disc, the second one will be renamed EFFORTS .BAK, and the latest one will be labelled EFFORTS .BAS.

If you really want to prevent the replacing of a program, however, this can be done. For details, see the chapter on CP/M operations (Chapter 8). Though the protection is carried out by using CP/M, it is recognised by AMSDOS, and in a catalogue the filename is marked with an asterisk. Any attempt to save a file with the same name will then bring up an error message which shows the filename followed by the message 'is read only'. This rather cumbersome but useful protection should be applied to all valuable files. When you have a number of valuable files on a disc, you should write-protect the whole disc by flipping back the small shutter with the end of a ball-point pen or a nail-file. This will protect all of your files on that disc from being written over. It can't protect them from coffee or from stray magnetic fields, though!

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. We'll look in Chapter 7 at more details of what can be achieved with the disc system other than loading and saving programs in BASIC.

Chapter Two
Putting It All On The Screen

Chapter 1 will have broken you into the idea that the CPC664, 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 CPC664 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 CPC664 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 CPC664, 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 CPC664 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 CPC664 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 CPC664, 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 CPC664 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 CPC664 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 CPC664 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.

```
10 PRINT"This is"
20 PRINT"the excellent"
30 PRINT"Amstrad CPC664 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 CPC664"
```

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 CPC664"

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 CPC664 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 CPC664 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 CPC664"
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 CPC664 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 CPC664 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.

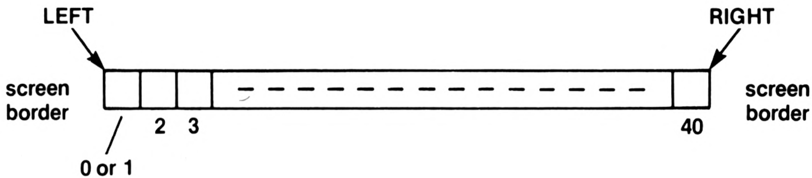


Fig. 2.7. The TAB map, which shows how the TAB numbers correspond to positions in a line.

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 CPC664 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)"CPC664"TAB(16)"COMPUTER"
20 PRINT TAB(2)"CPC664"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 CPC664 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 CPC664 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 CPC664 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 CPC664. 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

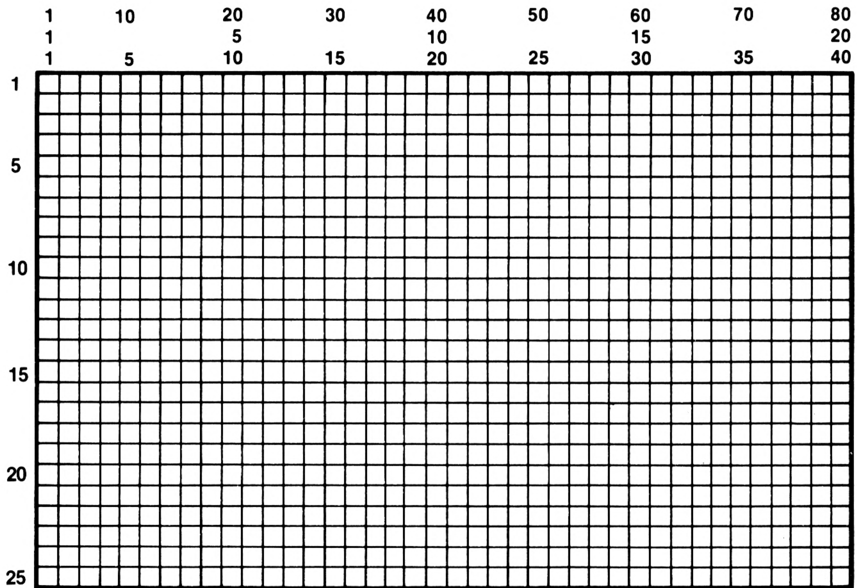


Fig. 2.12. A LOCATE map to help you to make the best use of this command.

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 CPC664, 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 CPC664 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 CPC664 *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 CPC664 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 CPC664, however, is more tolerant, and will allow you to use any word which is not identical to a reserved word. If you ask the CPC664 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 CPC664 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$="CPC664"
20 FIRST$="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 CPC664- "
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$. The 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$="THE NEW CPC664"
30 CLS
40 PRINT TAB(8)A$+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 CPC664'. 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. 11.11 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 then 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 CPC664 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 CPC664. 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 CPC664 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 CPC664! Figure 3.18 shows how the CPC664 copes with fractions like $\frac{1}{3}$, $\frac{2}{3}$, $\frac{1}{11}$ and $\frac{10}{11}$. 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 $\frac{1}{11}$ 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 CPC664, like most small computers, will accept and print numbers in this form. The conversion is automatic, and the CPC664 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 CPC664 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 CPC664 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. \ and MOD come into their own when you get involved in more advanced programming methods than 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 CPC664 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 CPC664 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 CPC664 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 CPC664, 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. 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!

Incidentally, it's often useful to ensure that the INKEY or INKEY$ instruction is not affected by a key that was pressed *earlier*. You can do this by having the instruction CLEAR INPUT in a line just before your INKEY or INKEY$.

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 CPC664 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 CPC664 string – a maximum of 255 characters per string, which is a longer string than most of us will ever need. Like other computers, the CPC664 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 blank square, but on the CPC664 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$="You can expand the CPC664"
30 PRINT"There are";LEN(A$);" characters
 in the phrase":PRINT"'";A$;"'"
40 INPUT"Try a phrase for yourself ";B$
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$="Amstrad CPC664"
30 L=LEN(A$)
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 CPC664, 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("CPC664"), 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$="CPC664 Computing"
20 CLS:PRINT
30 FOR N=1  TO LEN(A$)
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 could 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 conversion back to ASCII codes made in the program. This will deter all but really persistent de-coders! 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 66. 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!

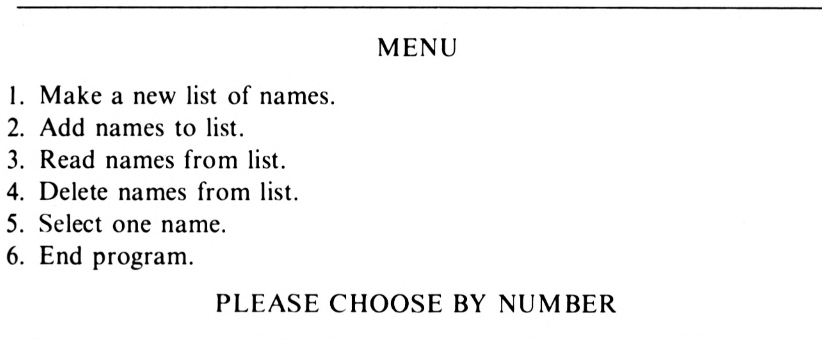
```
10 CLS
20 A$="Albert Hall"
30 B$="Richardson, Bertram"
40 C$="Sinclair, I"
50 PRINT"In A$, bert is located at";INST
R(A$,"bert")
60 PRINT"In B$,Bert is located at";INSTR
(B$,"Bert")
70 PRINT"In C$,BERT is located at";INSTR
(C$,"BERT")
```

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 CPC664 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 CPC664 computer when you use it to enter programs from discs that you have bought. 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 11 to 14. 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 CPC664, add one of the subroutines, and then test. When you are satisfied that it works, you can record the whole lot on another disc. Next time you want to add a subroutine, you start with this version, and so on. In this way you keep a disc 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 14, 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 075105116116101110
6003 DATA 067097108102
6004 DATA 070111097108
6005 DATA 067104105099107101110
6006 DATA 067117098
6007 DATA 074111101121
6008 DATA 071111115108105110103
6009 DATA 067117098
6010 DATA 080105103108101116
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 CPC664 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 CPC664. 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 to your CPC664. 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
The Disc System

Retitling and erasing

As your use of discs increases, you may find that you want to group files that are related in some way on to one disc. It would then be very helpful if you could give this disc a title which would remind you of what it contains. This is possible on a number of other disc systems, but not alas on the AMSDOS. You can, however, delete and rename individual files and groups of files. The system for doing this is not exactly simple or straightforward compared, for example, with that of the BBC disc system, so a bit of practice with it will be useful.

Starting with deleting files, the keyword here is ERA (ERAse). You might think that this could be used in the form ERA "FILENAME", but it can't. Instead, you have to pass to the disc system the address in memory of where this name is placed. The BASIC of the CPC664 has provided for this by the command @. If you precede a variable name, number or string, with the @ sign, then the effect is to locate whereabouts in the memory that variable value is stored. For example, if you type X$="FILE" (ENTER) and then follow it with ?@X$, you will see a number (such as 374) appear on the screen. This is a memory address; the number of the first of a set of bytes in memory that 'points to' the variable name. By 'points to', I mean that the contents of these bytes contain information on the length of a string and its location in memory. This is the way in which such information *must* be passed to the disc system when the | commands are used.

Suppose, for example, that we want to erase a file which is called ALLWORK. This has to be done in two steps. The first step is to assign "ALLWORK" to a variable name, such as X$. The second is to apply ERA to @X$. The two lines of commands are therefore:

```
X$="ALLWORK"  (ENTER)
|ERA,@X$  (ENTER)
```

and the file will be erased from the directory after the second command has been executed. This is possible only if the file has not been protected. If the file has been made 'read only' by using the CP/M STAT command (see later), or if the whole disc is write-protected because of the write-protect shutter being pulled back, then the erasure cannot take place. If you want to erase a number of files which have very different names, then you can use a loop in BASIC which reads each filename from a DATA list, assigns each in turn to X$, and then carries out the ERA action.

ERA, however, is one of the many commands that can make use of the 'wildcard' character, *. The asterisk can be used to mean any collection of characters, so that if you assigned X$="E*", then this would mean *any name* which started with E. The asterisk can be used in various parts of a filename. For example, *.BAK would mean any 'old version' file, because it would refer to any filename followed by .BAK, like ERROR.BAK or PASSIT.BAK and so on (see page 24). A 'name' such as *.* would mean *any* file. This wildcard system can be useful but you have to be careful with it, especially when you are erasing files.

Renaming a file makes use of the REN command. The form of the command is |REN,@N$,@X$, with the @ sign placed before each variable name. N$ in this case means the new name that you want to use, and X$ is the 'ex-name', the one you want to replace. Using X$ is preferable to using O$, because O and 0 are too easily confused. A renaming command needs three lines, with two assignments. For example:

```
N$="NEWFILE"  (ENTER)
X$="OLDFILE.BAS"  (ENTER)
|REN,@N$,@X$  (ENTER)
```

will rename the file that was called OLDFILE to the name NEWFILE. Of course, the disc must not be write-protected, and the name NEWFILE must not already exist on the disc. The *full name* of the old file, including extension, must be used in the assignment of the old string, X$. If this is not done, the change of name will not take place and you will get an error message such as:

```
OLDFILE . not found
```

to show that the extension was omitted. Putting a 'wildcard' into the old filename will cause a 'Bad command' error message. Because this command is so fussy about its syntax, you should always type it carefully and check each line before entering it. Remember that you

won't have to repeat each part of the command if you want to try again. You will probably need to repeat only one of the assignments and the REN part if you have slipped up somewhere. If you want to rename a number of files, then set up a loop in BASIC, reading the old names and the new names from DATA lines.

Hexadecimal codes

Unless you program in machine code, you probably haven't encountered the hexadecimal scale. If you use your disc system only as a convenient way of storing BASIC programs and data, and you have no intention of trying to read data from damaged discs or write machine code disc routines, altering CP/M routines, or transferring to disc programs from tapes which are copy-protected, then you can skip what follows, and reserve it for later. At some stage, however, you will probably want to make use of this information, and this is as good a place for it as any other.

Hexadecimal means scale of sixteen, and it's a way of writing numbers that is much better suited to the way the computer uses number codes. Our ordinary number scale is denary, scale of ten. This means that we count numbers up to nine, and the next higher number is shown as two digits, 10, meaning one ten and no units. Similarly, 123 means one hundred, two tens and three units. This counting scale, invented by the Arabs, replaced the Roman numbering system many centuries ago (except, oddly enough, for writing the dates of films and TV programs!). The unit of memory in the CPC664 and all other machines in its class is the byte, which can store a number between 0 and 255 (inclusive). A denary number for a byte may therefore be one figure (like 4) or two (like 17) or three (like 143). Hex (short for hexadecimal) is a much more convenient code for these numbers, and for address numbers. All single-byte numbers can be represented by just two hex digits, and any two-byte address by four hex digits.

One hex digit, then, can represent a number which, written in ordinary denary, can be between 0 and 15. Since we don't have symbols for digits higher than 9, we have to use the letters A,B,C,D,E, and F to supplement the digits 0 to 9 in the hex scale, as Fig. 7.1 illustrates. The advantage of using hex is that we can see much better how address numbers are related. For example, consider the address for the start of BASIC in the ROM of the CPC664. This is the address which is used when you type RUN and press ENTER. In

hex, this is 0170, whereas in ordinary denary numbers it is 368. Similarly, the address which is the start of the set used for screen memory is C000 in hex, 49152 in denary.

Denary	Hex	Denary	Hex
1	∅1	9	∅9
2	∅2	1∅	∅A
3	∅3	11	∅B
4	∅4	12	∅C
5	∅5	13	∅D
6	∅6	14	∅E
7	∅7	15	∅F
8	∅8	16	1∅

Fig. 7.1. How numbers 1 to 16 are written in hex.

The hex scale

The hexadecimal scale consists of sixteen digits, starting as always with 0 and going up in the usual way to 9. The next figure is not 10, however, because this would mean one sixteen and no units, and since we aren't provided with symbols for digits beyond 9, we use the letters A to F. The number that we write as 10 (ten) in denary is written as 0A in hex, eleven as 0B, twelve as 0C and so on up to fifteen, which is 0F. The zero doesn't have to be written, but programmers get into the habit of writing a data byte with two digits and an address with four even if fewer digits are needed. The number that follows 0F is 10, sixteen in denary, and the scale then repeats to 1F, thirty-one, which is followed by 20. The maximum size of byte, 255 in denary, is FF in hex. The maximum size of address in the memory of the computer, 65535, is hex FFFF. This is the number that we refer to as 64K. The K means 1025 in denary, #400 in hex.

When we write hex numbers, it's usual to mark them in some way so that you don't confuse them with denary numbers. There's not much chance of confusing a number like 3E with a denary number, but a number like 26 might be hex or denary. The convention that is followed by machine code programmers of the Amstrad CPC664 is to mark a hex number with the hash sign (#) placed *before* the number. For example, the number #47 means hex 47, but plain 47

would mean denary forty-seven. The BASIC of the CPC664 will *not* recognise the use of # to mark a hex number, so you cannot enter numbers like #2B or #028A. It will, however, work with hex numbers if you prefix them with '&' or '&H'. The machine code program-writing pack, called DEVPAC, will require the use of the hashmark, and another such pack, the very popular ZEN assembler, needs hex numbers to be followed by the letter H, and will reject hashmarks or ampersands (& signs).

If you are using some types of utility programs that recover data from damaged discs, or which alter the machine operating system, you may have to enter numbers in hex. These utility programs usually contain routines for the conversion of numbers between hex and denary scales, so that you never need to carry out hex arithmetic for yourself. In addition, the CPC664 will carry out conversions for you. To find the equivalent of a denary number you use HEX$(number, digits). The 'number' is the denary number that you want to convert, and 'digits' means the number of hex digits that you want to use. This will normally be two for a byte and four for an address. For example, ?HEX$(210,2) will give the correct hex conversion to #D2, and ?HEX$(23540,4) will give the conversion to #5BF4.

Backing up

One feature of a disc storage system which is less pleasant is that an accident to a disc can result in the loss of a lot of information. If you break a cassette tape, it's possible to splice the tape, and with some juggling, lose only a part of one program. If you damage a disc, it's likely that all of the information on the disc will be lost as far as conventional LOAD commands are concerned. This does not mean that the information cannot be recovered from the disc, but this is a desperate measure, not to be undertaken lightly. It makes sense, then, if you have a disc full of valuable programs or data, to make a backup copy as soon as possible.

One sensible measure is to make a second copy of each program as you put in on disc. If you have bought programs on disc, however, you will need to make a backup copy , or two copies if the disc is a valuable one. The CPC664 system also allows you to copy the whole of a disc surface. Note that I mean *surface*, not disc. The discs are two-sided, and any backup method will copy only from one side. If you want to back up an entire disc, you will have to back up each side separately, turning the disc over at some stage so as to read from the

other side. For many purposes, however, copying a file is enough, because you may only have one valuable program or data file on the disc. The operating system of the disc drive provides very well for copying a named file from one disc to another. If you use AMSDOS only, with programs in BASIC, this involves separate LOAD and SAVE steps. In other words, you will have to load the file into memory from one disc, and save it to another. This is straightforward enough when the files are BASIC programs, but the task is a lot more difficult when the files are machine code programs or data files. Fortunately, the utility programs which are part of the CP/M system are available for carrying out this essential task. We'll look at the CP/M FILECOPY backup utility later. A 'utility' is a program which aids you in some useful task like backing up a disc, printing what's on the screen, and so on.

Backing up is particularly easy when you have twin drives. With two drives in use, you can use a utility program to cause *everything* on one side of the disc in drive A to be copied to the disc in drive B, or the other way round. The process is accompanied by a lot of clicking and whirring, as one disc is read and the other written, but at least you don't have to attend to the process. You can make yourself a cup of coffee while it is all happening. A low-cost alternative, if you have a lot of spare cassettes, is to keep backup copies on cassettes. For business software, however, it's much safer to backup on to another disc, and to keep this backup disc in a cool safe place well away from all the hazards to discs, such as loudspeakers, TV receivers, electric motors and anything else that uses magnets of any kind.

Making back-ups

Of all the topics in disc use, that of making backup copies of individual files and of complete discs is the most important. The method which we have looked at so far, of loading into memory and saving on to disc, is simple enough for BASIC programs. It is, in fact, the only method that is available in the AMSDOS operating system. It can cater for machine code programs also providing that you know the essential information about the program. For example, if you have a machine code program from which you can break out into BASIC, then you can save this on disc. You need to know the address at which this program starts, and the length of the program in bytes. For example, suppose you have a machine code program which

starts at address #03E8 (denary 1000) and which consists of 9060 bytes (denary). You can save this on disc by using:

SAVE"MCODE",B,1000,9060 (ENTER)

but this is no great help if you don't know where the start and end addresses happen to be.

Other backups

For backing up programs which are not BASIC, and about which nothing is known, we have to turn to the other operating system, CP/M. The CP/M package contains several utility programs which are stored on the CP/M Master disc, and the two which are of special interest to us at the moment are FILECOPY and DISCCOPY. You should by this time have followed the instructions in the Amstrad manual about making a backup copy of your Master disc. If you have not, then this is the time to do it! DISCCOPY is the utility that allows such a backup copy to be made, and we shall therefore look at it first. Before we do so, however, *you must make sure that the Master disc is write-protected.* This is to prevent any muddle. If you copy the contents of the Master disc on to a blank disc, all is well. If, by some terrible blunder, you copy the contents of the blank disc on to the Master disc, you will have lost the Master disc programs, and you will probably not have the use of the disc system until you can replace it. Such blunders are easy to make, especially in the wee sma' hours of the morning.

To copy the whole disc, as you would for a Master disc, insert the Master disc in the drive. The label should show side 1 uppermost, and you should then engage CP/M by typing |CPM. When you press ENTER, the disc will spin, and after a short time (when CP/M has been loaded) you will see the screen change to 80-character mode. This is when you have to be careful, because some numbers can be very hard to read in this mode, and a mistake can have unfortunate consequences. Fortunately, for the DISCCOPY program, no numbers have to be read. You simply type DISCCOPY and press ENTER. This is the universal CP/M method of loading a named program – just type the name and (ENTER). You will be asked to insert the source disc into drive A, and press any key. If you are copying the CP/M Master disc itself, you don't need to do anything at this point. If you are making a complete backup copy of any other disc, you should insert it now. It's a good idea to label your discs as

'SOURCE' and 'DESTINATION' before you start, so as to avoid confusion. This is less likely when you are copying the Master disc, but when you are backing up another disc, it may be less clear which is which. As long as you always write-protect the SOURCE disc, you are not likely to come to grief, but a lot of time can be saved by using clear labels. The DESTINATION disc need *not* have been formatted if you are using DISCCOPY, because DISCCOPY automatically carries out formatting as it operates.

Since a disc can hold much more data than will fit into the memory of the computer, the complete backup will involve reading data from the SOURCE disc, storing it in the memory, and then writing it to the DESTINATION disc. This has to be done five times in all to transfer all the data from the SOURCE to the DESTINATION. When each read is complete you will be prompted by a screen message to insert the DESTINATION disc into the drive, and press any key. As usual, it's best to press the spacebar or, if you have a slightly sticky spacebar, the ENTER key. After the last chunk of data has been read and written, the program repeats, asking if you want to copy another disc. If you do, you answer with the 'Y' key, and the process repeats. If you don't, then press the 'N' key, and you will be instructed to place the CP/M Master disc into the drive and press CTRL C. This replaces the DISCCOPY utility with the normal CP/M operating system again. Fig. 7.2 shows the error messages that you might encounter while using this utility, and their causes. The alternative copying program, COPYDISC, can be used *only if you have two drives*. For this reason, it's often useful to make a backup copy of the Master disc that is *not* complete, but which contains only the utilities that are relevant to your own system. If you use COPYDISC in place of DISCCOPY (not a difficult mistake to make), you may find yourself wondering why it doesn't work, and how you can get out of it. Individual files are copied by using the other utility, FILECOPY.

Using FILECOPY

FILECOPY is a utility that allows files to be copied from one disc to another on a single drive. Unlike DISCCOPY, which was loaded simply by typing the name, the name FILECOPY must be followed by information about the file that is to be copied. If, for example, you want to copy the program BOOTGEN.COM from the Master disc onto a spare disc, you will have to use the command FILECOPY-

'You must insert the source disc into drive A'
No disc in the drive.
'You must insert a CP/M system disc into drive A'
A system disc must be in the drive in order to return to CP/M.
'You must insert the destination disc into drive A'
There has been no disc in the drive to receive the copy.
'The destination disc in drive A must be write-enabled'
You have used a disc which has its write-protect shutter drawn back.
'**WARNING**: Failed to copy disc correctly. The destination disc should not be used until it is successfully copied onto'
The program has been abandoned mid-way, and must be started again.
'The source disc has an unknown format'
Disc cannot be read, program has been abandoned.
'Failed to read source disc correctly: track x sector y'
Fault in the source disc at the named track/sector location. You *may* be able to correct this with a disc editor.
'Failed to write destination disc correctly: track x sector y'
Bad sector(s) in destination disc, which should be replaced.
'Failed to read destination disc correctly: track x sector y'
Faulty sector(s) or magnetic interference. Check position of disc drive and try again.
'Failed to verify destination disc correctly: track x sector y'
As above.
'Failed to format destination disc correctly: track x'
Disc fault or magnetic interference again.
'^C... aborted'
You typed CTRL-C while the program was waiting for an instruction. Program abandoned.
Two other messages, 'Illegal message number' and 'Insufficient space' should not appear unless you have modified your CP/M system.

Fig. 7.2. The error messages that you can encounter when using DISCCOPY.

BOOTGEN.COM (ENTER). FILECOPY, used by itself, is meaningless. The DESTINATION disc *must* be formatted.

Though the FILECOPY command requires a filename, there is nothing to stop this from containing the 'wildcard' asterisk character. If, for example, you want to copy all the files which are CP/M utilities with the .COM extension, then you can use FILECOPY *.COM to do this. If you want to copy all files whose filenames start with B, then you can use FILECOPY B* to copy these files. You can even copy all files by using FILECOPY *.*, but this is rather pointless if the disc is fairly full, because it would normally be easier

just to use DISCCOPY for this job. Fig. 7.3 shows the error messages that you can encounter when using FILECOPY.

'No SOURCE file present on input line'
You did not type a filename.
'Syntax error in options'
Incorrect user number.
'Failed to open SOURCE file correctly'
File not found, or reading failure.
'Failed to close DESTINATION disc correctly'
Usually means that disc directory is full, or disc fault.
'DESTINATION disc directory full'
No room for file directory entry, start again with another disc.
'DESTINATION disc full'
No storage space on disc, use another disc.
'The DESTINATION disc has an unknown format'
Usually means an unformatted disc, or one formatted by a different make of computer. Can also mean a disc corrupted by a magnetic field.
'The SOURCE disc has an unknown format'
Disc corrupted, or not an Amstrad disc.
'SOURCE disc missing'
You didn't put it in! This can also mean that the disc is not *quite* fully home in the drive.
'DESTINATION disc missing'
As above.
'DESTINATION disc is write protected'
You forgot to slide the protection shutter forward, or have used the wrong disc side.
'Incorrect DESTINATION disc'
You started copying the file with one destination disc and have changed to another.
'Failed to read SOURCE disc correctly'
Possible disc failure, or incomplete file being read.
'Failed to write DESTINATION disc correctly'
Failure of destination disc – try reformatting and start again.
'**WARNING**:DESTINATION file (name) is incomplete'
The copy will not work because the file is not correctly closed. This is unusual.
'^C... aborted'
You pressed CTRL-C when the program expected an input.

Fig. 7.3. The error messages that you can encounter when using FILECOPY.

Chapter Eight
The CP/M Operating System

So far, we have looked at the use of the AMSDOS operating system in more detail than the CP/M system. This is because anyone who programs the CPC664 in BASIC is more likely to use the AMSDOS system almost exclusively. CP/M is a system which is designed much more for the *user* of expensive business-biased software than the writer of programs in BASIC. It is a well-established system (first designed in 1973!), which is used mainly on computers which have no BASIC in ROM. When you use CP/M, *BASIC is switched out* so you cannot expect to load and run a BASIC program when you use CP/M. All of the programs which you use with CP/M will be in machine code, and unless you write machine code, or make use of a language (like Pascal) which can be compiled to machine code, you are unlikely to write programs using CP/M. In any case, the aids to machine code programming which are included with the CP/M package are designed for the old 8080 chip rather than for the Z80 which is used in your CPC664. This does not make these packages entirely useless, because 8080 code is compatible with Z80 code, but there are many better ways of writing true Z80 code. One of them is the DEVPAC set of two programs, available from Amsoft; another is the ZEN package, available from Kuma Computers. In this chapter, we shall look mainly at the commands of CP/M which do not require extensive knowledge of machine code. First of all, though, we need to look at the filename extensions. Remember that, although all the commands are shown here in upper-case (capital) letters for the sake of clarity, they can be typed in lower-case.

From using cassettes, you should be familiar with the idea of filenames. The main difference, if you use only AMSDOS, is that the filenames can have no more than eight characters. In addition to the main name, however, you are permitted an 'extension' of three characters following the main name. There *must* be a dot (period) between the main name and the extension. These extensions are

necessary for CP/M, and so they are used in the AMSDOS system as well. Their purpose is to identify a file type correctly, and if you do not specify an extension for yourself, the operating system will supply one. The most common extensions are listed in Fig. 8.1. This is a small selection, and if you use CP/M programs extensively, you could come across thirty or more of these extension names, all of which will be automatically assigned.

.ASC	File of ASCII text.
.ASM	Assembly language program file.
.BAK	Backup file.
.BAS	BASIC program file.
.COM	CP/M transient program file.
.DAT	Data file.
.DOC	Document or text file.
.HEX	Machine code file.
.LIB	Library file.
.OBJ	Machine code (object code) file.
.PRN	Assembly language listing file.
.REL	Relocatable machine code file.
.SUB	SUBMIT file.
.TEX	Text file.
.TXT	Text file.
.$$$	Temporary file. Also used for an unreadable file.

Fig. 8.1. Commonly used CP/M extension codes.

You can also assign extensions for yourself. Suppose, for example, that you are working on a program for yourself – we'll assume for the moment that it is a BASIC program, using AMSDOS. When you first try it out, you might want to save it under the filename of MYPROG.001, using the extension of 001 to mean that this is version 1. After some work on the program, you might want to save another version before running it, and this one could be saved as MYPROG.002 . The use of the extension identifies this as a separate program, so that MYPROG.001 is not wiped out. When you save the final version of the program, you can type MYPROG as its filename, with *no* extension, and the system will automatically supply the extension of BAS, because this is a BASIC program. You can, of course, use these extensions as you please, but this use for numbering program versions is probably the most handy application. There are other extensions which can be used *preceding* the main filename,

separated by a colon. The most important type of extension preceding a filename is the drive letter, A or B. You might, for example, have two programs, both called TESTIT.BAS but on different drives. One could then be called by typing A:TESTIT.BAS, the other by typing B:TESTIT.BAS, using the drive letters preceding the filename. These preceding letters are not likely to be of much interest to you if you are using only one drive, however. The other 'extension' at the front of a filename is the user number, which will be mentioned briefly later.

The CP/M commands

CP/M commands are of two types, built-in and transient. The difference is important. The built-in commands are contained in the ROM, and when you make use of them, you don't replace anything that is in the memory of the computer. The transient commands are carried out by loading a program (a 'command' file) into the memory and running it. This will replace anything else that happens to be in the memory at the time, so that you have to be rather careful about how you use these commands. In particular, you should not have a BASIC program in the memory and then switch to using a CP/M transient command unless you have a copy of the BASIC program on the disc. Since you will not normally use BASIC along with CP/M transient commands, this is not quite such a problem as it might seem.

We'll look at the transient commands of CP/M later in this chapter, but for the moment we'll concentrate on the built-in commands, which are listed in Fig. 8.2. Each of these will be obeyed at once when its name (and any other data that is needed) is typed, followed by ENTER. Of these commands, you will by this time have used DIR to obtain the CP/M directory. You can obtain the same

DIR	Print directory.
TYPE	Print out named file.
ERA	Erase named file.
REN	Rename file.
SAVE	Save machine code in a block of memory.
n:	Select drive n.
USER	Enter user number.

Fig. 8.2. Built-in commands of CP/M.

directory from AMSDOS by using |DIR, and rather more information from CAT. Staying with the CP/M DIR, however, you can also use this command to select directory entries. Suppose, for example, that you want a listing of all the BASIC programs on a disc. You can then type DIR*.BAS(ENTER), using the wildcard character to force the system to list any file which has the extension letters BAS. It is likely, if you are working in CP/M, that you will require a list of the .COM files, or possibly .ASC or .DAT files. Once again, you can obtain these selective directories by using the asterisk wildcard character.

Of all the built-in commands, DIR is the one that you are likely to use most of all. If you have two drives, then you will find that you need to use A: and B: along with DIR. As you would expect, A: switches to drive A, and B: to drive B. If you use B: when you have only one drive (or if there is a connector fault in drive B), you will get the error message:

Retry, Ignore, or Cancel?

which requires you to press the R, I or C key. While the system is waiting for you to make up your mind, the disc keeps running. The appropriate response is to press the C key. You then find another error message:

Bdos Err On B: Select

A close study of CP/M manuals suggests that you should be able to get out of this one by pressing CTRL-C. I could *not* break out this way; the only way I could find of escaping from the endless loop of error messages was by using the CTRL-ESC-SHIFT set of keys. If you have only one drive, then, you should take care that you *never* use the B: command.

Using ERA and REN

By contrast with the rather clumsy way that AMSDOS uses the commands ERA (ERAse) and REN (REName), CP/M uses these built-in commands in a simple way. You must, however, be quite sure of what file, or files, you want to erase. The best method of proceeding is to use DIR first, and to copy the filename of the file that you want to delete. If there is no file of the name that you have typed, you will get the ‘No file’ message. Suppose, for example, that you want to delete MYPROG.001. If you type ERA MYPROG, you will

find that this is *not* sufficient. You may get the curious error message 'Bdos Err On A:R/O' which normally means that the file is write-protected. In this particular case (unless the file really *is* write-protected), you need to start again by pressing ENTER (or any other key), then ERA MYPROG.001. When you specify the full filename with its extension, the erasure will be carried out.

You can use the wildcard character, *, but *not* to replace the whole of the extension. You can, for example use ERA *.BAS to erase all BASIC files, but you cannot specify ERA MYPROG.* to erase all files called MYPROG which might have different extensions. What you can use, however, is ERA MYPROG.00* to erase all versions of MYPROG from 001 to 009. If you get a 'Bdos Err' message in the course of using ERA, then press the spacebar or the ENTER key to get back to normal. It's very important to use DIR *after* an ERA command just to make sure that the erasure has been carried out as you wanted it.

The REN command works rather as the |REN command in AMSDOS does, but with straightforward filenames. The syntax of REN is:

REN NEWNAME.XXX=OLDNAME.XXX.

For example, if you have a file called REMS.BAS, you can rename it TEST.BAS by using REN TEST.BAS=REMS.BAS. If the file REMS.BAS is locked, then you will get the Bdos Err message 'File R/O' to indicate that the file is read-only, and its name cannot be changed until you have removed the write-protection. Unlike ERA, no wildcard characters are permitted *anywhere* in the REN filenames, because it would be ridiculous to rename a set of files to the same name. The error message which you get in this case is the filename reprinted on the screen with a question mark following it. Another error is to try to rename a file with a name that is already in use on the disc. This also will be refused, with the 'FILE EXISTS' message.

USER, SAVE and TYPE commands

The USER and SAVE commands are much less likely to be used by the CPC664 owner than the other CP/M built-in commands. USER is a way of identifying files which have been saved by one user of the system. The idea is useful when several users share a disc system, with each user having an identity number between 0 and 15. By typing

USER 8, for example, you identify yourself as being entitled to load files which have been stored when this user code was in action. Since all the files on your disc have been stored using USER 0, there won't be any USER 8 files. If you type USER 8 (ENTER) and then DIR (ENTER), you will get the message 'NO FILE', meaning that there are no files with this USER number attached. Unless you are one of a set of disc users who write and save machine code files, this command is of little interest. If you want to keep your files secure, however, when strangers are around, typing USER 8 will prevent a DIR listing of the files. It might be useful when you are showing off the system on Club night! An alternative method is to use the number in the filename, as a pre-extension. For example, saving with the filename of 8A:MYPROG.BAS will alter the directory so that only user 8 will see this directory.

SAVE is a peculiar and rather old-fashioned CP/M command which is useful only if you are saving blocks of machine code or text directly from CP/M. The CP/M SAVE action is quite unlike the normal BASIC SAVE action, and requires a different syntax – it is closer to the SAVE"Name",B action of Amstrad BASIC. The command was originally intended for machine code programmers, but nowadays, anyone who uses machine code will make use of an assembler program which will incorporate its own version of a SAVE command. So that you know something about it, however, here's how it's used.

SAVE is used to write on to the disc any bytes that are stored in the 'transient program area', which means the memory addresses from #0100 (256 denary) upwards. A SAVE will *always* start at this address of #100, and you have to specify how much of the memory is to be saved. The amount is specified in units of #0100 (256 denary) bytes. If, for example, you need to save a short program of just a few bytes, the minimum you can SAVE is #0100 (256) bytes, which is one block or 'page'. You would therefore use SAVE1FILENAME.COM to save this under the name of FILENAME.COM. No wildcards are permitted in this filename. Since you are most likely to use SAVE along with the transient program DDT, we'll take no further interest in it at the moment.

The TYPE command is one which is useful only for ASCII files. These might be BASIC programs which have been stored from AMSDOS by using the SAVE"Name",A command, or they might consist of data (names and addresses, for example?) which has been saved by using PRINT#9 in a database program. The reason for specifying ASCII files is that any other files are likely to cause odd

effects. TYPE allows the contents of files to be displayed on the screen. Now if you send ASCII codes to the screen, you'll see normal characters appear, but codes in the range 0 to 31 can cause effects like turning off the cursor, setting a new MODE, clearing the screen and so on. Machine code programs are likely to contain such characters, and so are BASIC programs which have been saved in the normal way, without using the 'A' modifier. Try it for yourself – return to AMSDOS, and write a short BASIC program. Now save the program, using the filename "TESTXT",A. Return to CP/M, and type: TYPE TESTXT (ENTER). You will see your program appear on the screen. If you use TYPE on any other BASIC program which has been saved in the ordinary way, you will probably see a line of text, a few graphics symbols, and nothing else. You may find a mode change or other disturbing effects which may require you to 'reboot' (load CP/M all over again). The moral is to keep TYPE for ASCII files only.

The transient commands

As we saw earlier, a transient command is one which is kept stored as a file on disc, and it will be loaded and run only when called. All transient programs are stored in the memory of your CPC664 starting at address #0100 (denary 256), and will wipe out anything else which has been stored starting at this address. Fig. 8.3 is a list of the commands which come under this heading. Of this list, you will probably have used DISCCOPY, FORMAT and FILECOPY already, and we'll look at some of the other transient programs in this section. Of these, STAT is probably the most important.

STAT, short for STATISTICS, exists to give you more information about files stored on a disc than you get from DIR. It can also be used to change the STATus of files, however, such as write-protecting, or removing write-protection. *You cannot use STAT unless there is a copy of STAT on your disc* as well as the files you want to examine with it. This means that STAT is one of the utilities that you may wish to copy on to other discs from the Master disc. If you have a set of files on a disc, but no STAT, you can usually transfer a copy of STAT, using FILECOPY, from the Master disc, or any copy of the Master disc. This is *not* always possible, however. If your disc has been used with AMSDOS only, and has no copy of the CP/M tracks (the system tracks) on it, then STAT cannot be loaded on to it. Similarly, if the disc is nearly full, there may be no room for

MOVCPM.COM	Configure a new CP/M system.
ED.COM	Edit assembly language files.
STAT.COM	Print information on files.
FILECOPY.COM	Copy file, single drive.
CHKDISC.COM	Check disc copy, needs two drives.
FORMAT.COM	Format new disc.
PIP.COM	Transfer data.
ASM.COM	Run 8080 assembler.
DUMP.COM	Show file content in hex.
SYSGEN.COM	Copy CP/M system to another disc.
DISCCOPY.COM	Copy complete disc using single drive.
SUBMIT.COM	Make a list of files run in succession.
DDT.COM	Monitor for 8080 machine code.
BOOTGEN.COM	Put CP/M tracks on program disc.
DISCCHK.COM	Checks that a copy is valid, single drive.
CLOAD.COM	Reads (unprotected) cassette file, transfers to disc.
CSAVE.COM	Reads from disc, transfers to cassette.
XSUB.COM	Places direct commands into a file.
LOAD.COM	Converts a .HEX file into a .COM file.
AMSDOS.COM	Returns to AMSDOS.
COPYDISC.COM	Copies disc, needs two drives.
SETUP.COM	Changes CP/M configuration.

Fig. 8.3. The transient commands of CP/M.

STAT, which needs at least 6K of storage space. As so often happens, users of twin disc units have an advantage here. They can place the disc which contains STAT in drive A, and use the B: specifier for the filenames on a disc in drive B, so that STAT loads from drive A but works on files in drive B.

STAT has several varieties of syntax, some of which are useful even if you use mainly AMSDOS and BASIC. To start with, if you type STAT(ENTER) with no filename, you will get the statistics on the disc in drive A. These 'statistics' will be rather sparse, a reminder of the drive letter, an indication of protection, and the number of K of free space on the disc. The protection message will be R/O (read-only, meaning write-protected) or R/W, meaning no protection. When you use STAT in this way, the protection indication is meaningless, because a disc may be listed as R/O or R/W regardless of the position of its write-protection shutter. This is because STAT does not write on the disc, and the presence of the write-protect

shutter is detected *only when the disc is written to.* Even the 'bytes free' number has to be regarded with some suspicion – I found that storing STAT on a disc did not alter the number that appeared here!

STAT really comes into its own when you apply it to individual files or groups of files. To apply STAT to a file, you follow STAT by a space, then type the filename with extension, and ENTER. For example, STAT TESTFIL.BASENTER will provide the statistics on the BASIC file called TESTFIL. These statistics will consist of the full filename, with the status (R/O or R/W) and a note of the number of records, bytes, and extent of the file. None of these figures will show you *exactly* how large a file is. The 'Recs' number shows the whole number of 128-byte units that the file uses. If the file consists of 129 bytes, for example, its Recs number will be 2. The Bytes number shows the whole number of Kilobytes assigned to the file. Once again, no fractions are shown, and a file of only 12 bytes will still produce a Bytes number of 1K. The Ext is an even less useful figure, which remains at 1 until the file is really long, more than 16K. We'll see later how we can find the exact size of a file, using the transient command DDT.

The use of STAT with filenames can take wildcards, such as STAT *.BAS which will give statistics on all BASIC files, and STAT G*.* which will give STATS on all files of any type which start with the letter G. You can also use STAT *.* which will give statistics on all files. Be careful to include the extension to the filename, even if it consists only of a wildcard. Note that a file will be shown as R/W in the 'Acc' (access attribute) column *unless* it has been protected. Using the write-protect shutter on the disc does *not* cause individual files to be shown as write-protected for the reason given earlier.

The most important use of STAT for our purposes is in altering the protection of files. This is done by following the filename with a space, and then a set of 'attribute' characters. These attribute characters must start with the $ sign, and can then be R/O (read-only), R/W (read/write), DIR (appear in directory) or SYS (not in directory). For example, if you have a file called TRY.BAS, you can write-protect it by typing:

```
STAT TRY.BAS $R/O   (ENTER)
```

and if you have a file called REMS.BAS which is write-protected, you can release the protection by typing:

```
STAT REMS.BAS $R/W   (ENTER)
```

You can use STAT *.BAS to confirm that these alterations have been

carried out. It can be very useful to have some files which do not appear in the directory, like the CP/M system file, and this can be done by using SYS. For example, a file called PRINTSET.BAS can be kept out of the directory by typing:

STAT PRINTSET.BAS $SYS (ENTER)

and from then on it will be concealed from the user who types DIR. The file will *still appear* if STAT is used, so that, for example, typing STAT*.* will list this file with the others. It is marked out by having brackets around the filename. To restore a file like this to the directory, having found its name from STAT*.*, just type:

STAT PRINTSET.BAS $DIR (ENTER)

which will restore the directory entry for this file. All of these varieties of the STAT command can make use of the wildcard characters in the filenames, so that a whole set of files can be write-protected or removed from the directory and so on. A less important use of STAT is to list 'device names'. These are the names that CP/M allocates to external devices such as the keyboard, screen, printer and so on. By typing STAT DEV: (ENTER), these can be listed. The listing is not of particular interest or use to you unless you are designing machine code programs for CP/M.

Other transient commands

When you look through the list of transient commands found by using DIR on the Master disc, it's rather discouraging to find how many of them are of little use to you. Another disappointment is to find how many useful-looking names conceal programs that are, in fact, only of specialist interest. In this section, we'll look at some of the other files on the Master disc, and comment on their usefulness or otherwise to the ordinary CPC664 user. Remember always that CP/M transient commands are designed for use with CP/M programs in machine code, and are of little or no interest if you program only in BASIC. Even if you make extensive use of CP/M programs, many of the transient programs are of little use to you.

Of the files that remain, PIP is one of the more useful-looking ones, but it is less useful than might appear. PIP is a general-purpose program which deals with transferring data, but most of the data transfers that you want to make are already covered by other commands. The most useful transfer, for example, is to send to the

printer characters that normally go to the screen only. This is done simply, without using PIP, by pressing CTRL-P before starting the screen listing. Pressing CTRL-P again stops the process. This action, plus the FILECOPY and DISCCOPY programs, supersedes much of what PIP can do. You have to remember that CP/M is a very old system, designed for use by professionals, and incorporating features which are not quite so necessary on modern computers. In addition, because the system has grown over the years, PIP is a command which can take dozens of optional forms, some of which are useful, others not. The simplest syntax of PIP is:

PIP Destination=source

which will transfer data from destination to source. You have to specify what you want to use as destination and what you want to use as source. Both might be filenames, in which case PIP is being used to make a copy of a file under a new filename. This can sometimes be useful. A much more common use of PIP is with 'device' labels. A list of these is shown in Fig. 8.4, and each has to end with a colon (:). Of

CON:	Entry and display device.
RDR:	Receive information device.
PUN:	Send information device.
LST:	Print information device.
NUL:	A source of 40 null characters.
EOF:	A source of end of file mark, CTRL-Z.
PRN:	A printer output which formats copy.

Fig. 8.4. A list of device names that can be used with PIP.

the seven or so device labels that can be used in standard CP/M, the CPC664 system suggests four, of which CON: and LST: are the really useful pair. CON: can mean the keyboard when you are using this as source, or the screen if you are using it as destination. CON is short for console, another term from the dim past. LST means the listing device, in other words the printer. If you have no printer, many of the applications of PIP are rather pointless for you. Looking at a file on disc, for example, is more easily done by using TYPE, and the file copying actions are better performed by using FILECOPY.

One useful function, however, is to join two files into one large file. This can be very useful if you keep program subroutines for attachment to a main program. If you program in BASIC, of course, this applies only to subroutines which have been saved using the 'A'

option to create ASCII coded files. If, for example, you have a program MAIN.BAS and a subroutine set called SUBS1.BAS, then you can join them in a file called LARGE.BAS by using:

PIP LARGE.BAS=MAIN.BAS,SUBS1.BAS (ENTER)

using the comma to separate the filenames. You don't have to confine yourself to combining just two files in this way, because as long as you use the comma to separate the sections you can combine as many as you can type in one command; about a line and a half of typing. If you want to carry out several PIP actions, one after the other, you can use a rather different syntax of PIP. You can type PIP (ENTER), and this will load the PIP program and print an asterisk on the screen. You can then type a PIP command, such as NEW.BAS= OLD1.BAS,OLD2.BAS (ENTER), and this will be obeyed. After this has been done the asterisk will appear again, waiting for another command, but you don't have to type PIP again. When you have no more commands, typing (ENTER) by itself will return you to the normal CP/M commands.

The other uses of PIP are rather specialised. You *can* use PIP to accept text from the keyboard and store it in a file. This is done by using:

PIP WRITING:TXT=CON: (ENTER)

at which the disc finds the PIP program, then opens a file. The disc must not be write-protected – if it is, you will get the usual error message with the options of Retry, Ignore or Cancel, followed by the Bdos Err message (use CTRL-C to escape, with a Master or system disc in place). This is *not* a very useful method for entering text, however, because you have to learn a new set of keys. Instead of using the DEL key, for example, you have to use CTRL-H. After each use of (ENTER) to start a new line, you also need to use CTRL-J. The text is ended by using CTRL-Z. When you have such a piece of text, you can send it to the printer by using LST: or PRN:. LST: is noted in the DDI-1 Manual, but PRN: also works and is very useful. When, for example, you type:

PIP PRN:=WRITING.TXT (ENTER)

the text is sent to the printer, the lines are automatically numbered, and the output is broken up into pages of 60 lines. Any TAB characters in the text will cause the printer to leave an 8-column gap.

Any of the PIP commands can be modified by following them with optional extension letters. These letters *must be placed between*

square brackets, and they can be selected from the set that is shown in Fig. 8.5. Once again, this is rather specialised, but it can be turned to some advantage if you have a printer. For example, by using:

PIP LST:=CON:[FT8P60] (ENTER)

you can force the printer to type what you are typing on the screen. You have to press ENTER to get a line typed, and then press CTRL-J to get the line feed. You should *not* use CTRL-H to rub out a mistake, because the printer will take this as a signal to print. To rub

Note: n means number needed, $ means string needed, ^ means CRTL key.

[B]	Block transfer. Was used in paper tape reading.
[Dn]	Delete any character after number n.
[E]	Echo characters to screen.
[F]	Filter out form feeds.
[Gn]	Copy from other user number n.
[H]	Hex format for paper tape.
[I]	Ignore nulls in paper tape inputs.
[L]	Convert upper-case to lower-case.
[N]	Add line numbers
[O]	Transfer machine code files, or non-ASCII files.
[Pn]	Put out form feed after line n.
[Q$^Z]	Copy file up to this string.
[R]	Copy system files.
[S$^Z]	Copy data starting with specified string.
[Tn]	Set tab stops every n characters.
[U]	Convert lower-case to upper-case.
[V]	Verify copy.
[W]	Copy to R/O file.
[Z]	Zero parity bit during transfer.

Fig. 8.5. The extra commands of PIP which can be used within square brackets.

out, you use the DEL key as usual, but what you see on the screen is a checker square for each DEL key use. This prevents the error character from reaching the printer, however, unlike CTRL-H. The F modifier prevents the printer from reeling out a whole sheet of paper before starting, and the T8 sets the tabulation stops to 8 columns. The P60 makes the printer take a new page after 60 lines.

PIP has a very wide variety of uses, as you can see from the lists and the few examples, but most of these applications are for the

professional who is working with a much larger machine and with ASCII and machine code files. Most of the other transient programs are aimed at the machine code programmer, and detailed description would be pointless. Of these, though, DUMP.COM can be useful. The command DUMP must be followed by a full filename, and its effect is to display on the screen the contents of the file in *hex* codes. This is not something that would be useful to the BASIC programmer, and even the machine code programmer would normally prefer to have a disassembly. Nevertheless, for a quick check of a short machine code program, DUMP can be quite useful.

Other commands

Two commands of the CP/M transient set look rather familiar to anyone who has ever used Microsoft BASIC. These are CSAVE and CLOAD, and as the 'C' suggests, they are concerned with cassette use. Since you are not very likely to use the CPC664 with cassettes, we'll omit these commands which are, in any case, not straight-forward to use.

DISCCHK and CHKDISC

These utilities are for checking that a copy of a disc is an exact and perfect copy. CHKDISC is for use with twin drives only – you put a disc in drive A and its copy in drive B, then use CHKDISC to compare the discs, byte by byte. DISCCHK does the same more slowly, and will instruct you when to insert the source disc and when to insert the copy disc. If you find, incidentally, that for some reason these utilities refuse to load even when you have spelled the name correctly for the fourth time, then try loading another utility, such as DISCCOPY, and then aborting by pressing CTRL-C when the program runs. You should find then that normal action is restored. The cause is probably corruption of some of the CP/M operating system in the low RAM memory of the CPC664.

DDT, ED and ASM

These three are specialised programs for machine code programmers. Because some of the commands make use of the instruction set of the

old 8080 chip, they are likely to have little appeal for programmers of today. DDT is a 'debugging' utility for machine code, a monitor as it would now be called. It is called into action by typing DDT (ENTER) or DDT filename (ENTER), depending whether you want to use it to search through itself or another file. The main sub-commands of DDT are shown in Fig. 8.6 – the most interesting ones are D and L. **L** allows a disassembled listing, and can be followed by a start number and end number, both in hex. The disassembly is, however, in 8080

Note: DDT uses hex numbers. s means start address, e means end address, b means breakpoint address, d means destination address, c means constant byte, name means filename.

As	Assemble from address s.
Ds,e	Display memory from address s to address e.
Fs,e,c	Fill memory from s to e with constant c.
Gs,b	Start program running at s, break at b. More than one breakpoint can be used.
Iname	Insert filename.
Ls,e	List disassembled code from s to e.
Ms,e,d	Move block of memory from s to e so as to start at d.
Ra	Read file to address.
Ss	Read and alter memory starting at s.
Tn	Trace n steps of the program.
Un	Execute n steps, no trace.
Xx	Examine and alter Z80 state. The 'x' refers to register code letters.

Fig. 8.6. The main commands of the DDT utility. These are of interest mainly to machine code programmers.

code, and not everyone remembers it nowadays! The **D** command gives a hex dump, but with the added refinement of a set of columns of ASCII characters that correspond to any codes in the correct range. This allows you to read copyright notices, error messages and other items which might not otherwise come to your attention.

The DDT program extends from address #0100 to #13A0, and any program that you load in with DDT (by having its filename following DDT), will be loaded in from this address onwards. When you make use of this additional load, the last address that is used will be displayed on the screen as well as the 0100 starting address. As you might expect by now, all addresses are in hex. Note that you can write in assembly language by using the 'A' option of DDT. The snag, as

usual, is that this uses 8080 assembly language only, and there is no direct provision for saving. For writing 8080 assembly language, the ED utility is rather more useful. Once ED has been used to write assembly language, ASM can be used to assemble it into machine code, and record a file of this code. This file can then be converted into a .COM file by the LOAD command. If it looks very tedious, you're right, it is. If you are serious about writing in machine code, it makes more sense to use a good modern Z80 assembler, save the machine code with the extension .HEX, and then use LOAD to convert into a COM file.

Chapter Nine
BASIC Filing Techniques

What is a file?

I have used the word 'file' in the course of this book to mean a collection of information which we can record on a disc. Programs in BASIC are one type of file, and the only type, incidentally, which permits the use of LOAD and SAVE in a straightforward way, with no 'modifying' letters like A, B or P. As you will know by now, the best type of file for BASIC programs is the ASCII file, since this is the one which can be used both by AMSDOS and by CP/M routines.

In this chapter, however, I shall use the word 'file' in a narrower sense. I'll take it to mean a collection of data that 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 data-gathering action of the program, and it preserves these amounts for the next time that you use the program. Taking another example, suppose 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, such as title, artists, catalogue number, recording company, date of recording, date of issue and so on. This information is a file, and at some stage in the program, you would have to record this file. Why? Because when you load a BASIC program and RUN it, it starts from scratch. All the information that you fed into it the last time you used it will have gone – unless you recorded that information separately. 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. In this chapter, we are dealing with filing programs that are in BASIC, so we shall not make any use of CP/M.

Knowing the names

You 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'. A *record* is a set of facts about one item in the file. For example, if you have a file about vintage steam locomotives, one of your records might be used for each locomotive type. Within that record, you might have wheel formation, 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 motorbikes'. 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, gear ratios, suspension system, top speed, acceleration... and whatever else you want to take note of. Filing is fun – if you like arranging things in the right order. The importance of filing is that all of the information can be recovered very quickly, and that it can be arranged in any order, or picked out as you choose. If you have a file on British motorbikes, for example, it's easy to get a list of machines in order of cylinder capacity, or in order of power output, or any other order you like. You can also ask for a list of all machines under 250 cc, which ones used four-speed gearboxes, which were vertical twins. Rearranging lists and picking out items is something which is much less easy when the information exists only on paper.

Disk filing

In this book, because we are dealing with the CPC664 disc system, we'll ignore filing methods that are based on DATA lines in a BASIC program, or on the use of cassettes. Though you may be experienced at filing with cassette systems, I'll explain filing from scratch in this chapter. 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 that we can use with a disc system; serial files, and random access files. The differences are

simple, but very important ones. A serial (or *sequential*) file places all the information on a disc in the order in which the information is received, 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 from the beginning of the file into the computer, and then select. There is no way in which you can command the system to read just one record or one field. More important, with cassette files you can't change any part of a record, or add more records in the middle of such a file. Files of this type on disc are much more useful, because records can be read and checked much more quickly, but adding or changing items still presents problems. A random access file does what its name suggests – it allows you to get from the disc one selected record or field without reading every other one from the start of the file. You might imagine that, faced with this choice, no-one would want to use anything but random access files. It's not so simple as that, though, because the convenience of random access filing has to be paid for by more complication. For one thing, because random access filing allows you to write data at any part of the disc, it would be very easy to wipe out valuable data, or even the directory, with a program that was badly designed. Also random access filing on the CPC664 is possible only by using machine code programs.

We'll start, then, by looking at serial files which are the easiest to deal with.

Serial filing on disc

We'll start by supposing that we have a file to record, called CAMERAS. On this file we have records (such as Nikon, Pentax, Canon, Yashica and so on). For each record we have fields like Model, Film size, Shutter speed range, Aperture range (standard lens), Manual or Automatic, and so on. How do we write these records? First of all, we need to arrange the program that has created the records so that it can output them in some order. The usual procedure will be to take the records in some chosen order, and output the fields of the record in some order as well. Fig. 9.1 for example, shows how we might arrange this part of a BASIC program so as to input a number of records, with five fields to each record. The number of fields is five, so the fields are input from the keyboard using a FOR N%=1 TO 5 loop. The number of records isn't fixed, so we use a GOTO loop, which keeps putting out records until it finds one called "X" or "x", which is the terminator. I haven't used a

```
100 RC$="":X%=0:DIM Field$(5)
110 CLS:PRINT TAB(15)"DATA ENTRY":PRINT:
PRINT"Type X to end entry.":PRINT
120 INPUT "Record name - ";RC$:IF RC$="X
" OR RC$="x" THEN 190
130 REM NEED TO RECORD THIS ON DISC
140 X%=X%+1:FOR N%=1 TO 5
150 PRINT"Field item ";N%;" ";:INPUT Fie
ld$(N%)
160 REM NEED TO RECORD THIS ALSO
170 NEXT N%
180 GOTO 120
190 REM END OF FILE
200 PRINT"There are ";X%;" records on th
e file."
```

Fig. 9.1. How to organise data for disc writing. The example uses five fields in a record.

WHILE...WEND loop, because this forces you to enter a lot of dummy field information in the last record. Note that we haven't used an array for holding these items, because an array has to be dimensioned, and we don't know in advance how many items we will have. Instead of storing the items in an array for future use, they will be recorded on disc. The points where the disc recording routine would be fitted are shown in the REM lines 130 and 160. Each item, field or record, is treated as a string. This is because strings are easier to work with – you will not, for example, get any error messages at the INPUT stage because of a mismatch of variable type. The other good reason for using strings is that a string is a set of ASCII characters, and these files are *always* recorded as ASCII files.

That deals with the organisation of the data for putting on to disc, but how do we actually put it on the disc? There are several stages, and the first one is to open up the 'data channel', which is assigned with the number 9. This means assigning a filename which will be recorded on the disc, and sending data to the disc with the PRINT#9 command. The 9 is a number code that the machine will use to distinguish the disc drive from the screen windows or the printer. Later, in Chapter 11, we'll look at how the hashmark (#) is used with other numbers. Each time you want to make use of a file, then, you must have a filename, and this has to be used to prepare for recording on the disc by using the OPENOUT command.

The purpose of using the filename and the channel number 9 is to

organise data. The disc stores all data in units of 512 bytes. It wouldn't make sense to spin the disc and find a place on the disc just to record one byte at a time, so when you record or read a disc, it's always one complete sector, or as much of a sector as possible, at a time. In fact, the operating system of the CPC664 uses 2K blocks of data, which would fill four sectors. Some of the memory of the CPC664 has to be used to hold data which is being gathered up for recording, or which is being replayed. The channel number 9 is an identifying number for the piece of memory that is being used, so that the machine finds the correct data in the correct part of the memory. Using this channel number avoids the need for you to have to allocate parts of the memory to use in this way as buffers. The memory which is used for this purpose lies at the top of the usable range. To see this in action, switch on from cold, and type ?HIMEM. On my machine, this gave the number 42619. If you now type OPENOUT "test" (ENTER), and then ?HIMEM, you will see that the number is now 38523. The difference is 4096, equal to 4K, or two buffers each of 2K. If you now type CLOSEOUT(ENTER), you will see that ?HIMEM gives the original figure of 42619 again. HIMEM means the top of usable memory, and it is shifted down when files are to be read or written, and restored at the end of filing commands.

Opening the file

After that short diversion, back to our filing program. Before we start to gather the data together for filing, we need to 'open a channel' for the data. This is done using the OPENOUT command. OPENOUT has to be followed by a filename, and if you have used cassette files previously you will have to remember that disc files must use no more than eight characters. In addition, it helps if you can give the filename some useful extension label. The 'standard' extension for data is .DAT, so it makes sense to use this unless you have some pressing reason for using something else. You can, of course, use numbers like .001, .002 and so on, to show different batches of data by the different extensions, or you can include these numbers in the main name, as, for example, CAMERA01.DAT. To take another example, the line:

```
OPENOUT "AIRCRAFT.DAT"
```

prepares to write a file called AIRCRAFT. When this line is executed, the disc will spin for a short time, preparing for the file, *but the filename will not appear on the directory/catalogue* because no

data has been put into the file. The buffer space will also be prepared in the memory. As always, you can place the drive letter ahead of a colon if you have more than one drive.

The use of the OPENOUT command opens a file – which means that we can make use of the file for writing data on to the disc. It also means that the disc is prepared for the file. Any file that exists on the disc already and has the same name of AIRCRAFT will *not* prevent you from opening this file. This means that you have to be rather careful about how you use files, because one file will wipe out another of the same name. This, however, makes it very easy to update and modify files, as we shall see. If you want to lock a file you will have to make use of the CP/M STAT command *after* your BASIC data filing program has ended.

Printing to the file

It's at this stage that we need to make use of the loops in the writing program. Within these loops, we need to have a line something like:

```
160 PRINT#9,Field$(N%)
```

PRINT#9 means put the information out on channel 9, the channel for the disc system, so that PRINT#9 will *eventually* put out to the disc system the data that follows. In this example, it's Field$(N%). N% is the number in the FOR...NEXT loop, so that as the loop goes round, we will put on to the disc Field(1), then Field(2), then Field(3)... and so on. We also need to write the record name, and this is done within the loop, by using a line such as:

```
130 PRINT#9,RC$
```

without using an array (because of the unknown amount of dimensioning). Fig. 9.2 shows an example of a very short and simple program of this type which has been adapted from the first example. You can enter anything you like into this, but it makes more sense to enter something that you can easily check. Since the file is called AIRCRAFT, you could make each record name an aircraft type, and each field some feature of the aircraft, like wingspan, engine details, number of crew, and so on. You can, of course, easily change this program so that it has another title that suits the information that you might want to use.

Before we move on, consider what this program has done. It has created a file called AIRCRAFT.DAT, and allocated a channel

```
10 OPENOUT "AIRCRAFT.DAT"
100 RC$="":X%=0:DIM Field$(5)
110 CLS:PRINT TAB(15)"DATA ENTRY":PRINT:
PRINT"Type X to end entry.":PRINT
120 INPUT "Record name - ";RC$:IF RC$="X
" OR RC$="x" THEN 190
130 PRINT#9,RC$
140 X%=X%+1:FOR N%=1 TO 5
150 PRINT"Field item ";N%;" ";:INPUT Fie
ld$(N%)
160 PRINT #9,Field$(N%)
170 NEXT N%
180 GOTO 120
190 REM END OF FILE
200 PRINT"There are ";X%;" records on th
e file."
210 CLOSEOUT
```

Fig. 9.2. A program which writes to a serial file.

number of 9 to this file. It has then stored the data as it came along, in the sequence of RECORD, then FIELDS. Finally, the file has been recorded and closed by using CLOSEOUT. This last step is *very* important. For one thing, you don't actually record on the disc *any of the information* in this short program until the CLOSEOUT statement is executed. That's because it would be a very time-consuming business to record each item of a file at a time. What the DFS does, remember, is to gather the data together in memory. This is the 'buffer' piece of memory, placed just above HIMEM, and it will be written to the disc only under one of two possible circumstances. One is that the buffer is full, so that there are four sectors full of data (2048 bytes) to write. The other is that there is a CLOSEOUT type of statement in the program. For a large amount of data, the disc will spin and write data each time the buffer is full. The CLOSEOUT command then writes the last piece of data, the one which doesn't fill the buffer. It also writes a special code number, called the end-of-file code (EOF). This can be used when the file is read, as we'll see later. If you forget the CLOSEOUT statement, you'll leave the buffer unwritten, with no EOF – and cause a lot of problems both in your programs and possibly with your disc system. Forgetting the CLOSEOUT is called leaving your files open, and you wouldn't like to be seen like that, would you?

The biggest danger is when you are testing a program. If there is an

error, such as a syntax error, which stops the program from running, there will be no CLOSEOUT carried out, and the files will be open. If you had typed a lot of data, you would lose it if you then went on to correct the program and run it again. The correct procedure is to close all of the open channels. In this example, it's easy – you only have to type CLOSEOUT and press ENTER. For a large program you would probably find it better to write an ON ERROR GOTO line which, when an error occurs, closes files and ends. This automatically ensures that files are never left open. The CLOSEOUT ensures that your data will be recorded.

When you use an INPUT statement to gather up the data, you can find that with a lot of data you will hear the disc start and stop at intervals. That's an indication of the buffer transferring data to the disc. You can't use the keyboard while the transfer is taking place, but the time that's needed to write a sector is fairly short. You will find that the keyboard cannot be used during this time. In this example, there is nothing like enough data to fill a buffer. You will hear the disc spin when the OPENOUT command is executed, and again when the CLOSEOUT command is executed, but not at any time between these two unless you enter a huge amount of data.

Getting your own back

Having created a file on disc, we need to prove that it has actually happened by reading back the file. A program which reads a file must contain, early on, a command which opens the file for reading. This is OPENIN, and it must use the same filename as was used to write the file. If we recorded a file using the name 'AIRCRAFT', then we must not expect to be able to read it if we use 'CAMERAS' – or any other name. Misspelling can haunt you here! Once the channel has been opened, we can read data with INPUT#9, which will be followed by the variable name that we want to assign to each item. This command reads an item from the disc, and will allocate it to a variable name for printing the item or other use, according to what we have programmed. The number of reads can be controlled by a FOR...NEXT loop if the number is known, or it can make use of the EOF marker if the number is unknown. By testing for EOF, then, we can make the program stop reading the file at the correct place.

The example of Fig. 9.3 shows both methods in use. The number of fields has been five, so that a FOR...NEXT loop can be used to control the input of the fields. The number of records, however, has

```
100 DIM FX(5)
110 OPENIN "AIRCRAFT.DAT"
120 WHILE EOF=0:CLS:PRINT TAB(12)"AIRCRA
FT DETAILS":INPUT#9,Name$
130 PRINT"Type is ";Name$:RESTORE
140 FOR N%=1 TO 5
150 INPUT#9,Gen$(N%)
160 READ Field$:PRINT Field$;" ";Gen$(N%
)
170 NEXT
180 PRINT"Press spacebar for next record
"
190 K$=INKEY$:IF K$=""THEN 190
200 WEND
210 CLOSEIN:PRINT"END":END
220 DATA Wingspan,Length,Crew No.,Engine
s,Range
```

Fig. 9.3. A program which reads the serial file.

not been settled by a FOR...NEXT loop, so we have to keep reading the file until the EOF byte is found. This is done in line 120 by testing EOF in a WHILE...WEND loop. If EOF is not zero, then the file is closed, and the program ends. As you can see, this has been put into the WHILE...WEND loop, because EOF needs to be tested *before* another item is read. If you read again from a file like this, you will get the 'EOF Found' error message, and the program will stop. Unless you have arranged for an ON ERROR GOTO line to close files for you, the files will still be open. Leaving a reading file open is not quite such a disaster as leaving a writing file open, but it's still very undesirable. Note that the disc does *not* spin each time you press a key to get another record. This is because a complete sector or set of four sectors is read each time, and if the information that you want is all in one buffer load, the disc need not be used. Sorry if I seem to be labouring this point, but a newcomer to discs sometimes finds it difficult to remember.

Now this simple example shows a lot about serial filing that you need to know. When you use discs, then, the name that is used with OPEN (IN or OUT) is the filename for the file on the disc. Any other file that is later recorded with the same name will not overwrite this file, because the old file changes to a .BAK file. The system therefore provides for easy file replacement, *and* for reasonably good file security. This is an important point to emphasise if you have been

using cassettes, because you have more control over where a file is recorded on a cassette. You can write a file called INDEX at the start of a tape, for example, then wind the tape on slightly and record another, different, file with the same name. You certainly can't record two files with *identical* names on one disc. Even if the files have the same main name, the older one will have the extension .BAK to distinguish it. In addition, a file is closed by writing the EOF character. How, then, can you update a file, particularly if you want to add more items to the end of the file?

Updating the file

There are two answers, if we stick to serial filing. One possibility, which is the simplest one for short files, is to load the whole file into the memory of the computer, make the alterations (your BASIC program will have to be written so as to provide for this), and then write the file again, wiping out the earlier version. The other possibility is to open two files, one for reading and the other for writing. You don't need to have dual disc drives for this, though it makes life much simpler if you do. This means that the computer will maintain two buffers. You read one record from the reading file and you can, if you wish, display it. If it's all right, it's then written (to the buffer initially). If the record has to be modified, you can do so. If extra records have to be added, this is equally simple. Each time a buffer empties, the disc will spin and a read or write will take place. This 'simultaneous' operation is possible because of the use of different OPEN commands, which control different buffers. In practice, it's a matter of writing your program to suit.

Figure 9.4 shows a simple program which allows you to extend the file that was created by the program of Fig. 9.3. Note, however, that the files use *the same* names, even though I have assumed that both files will be on the same disc. This is because the OPENOUT file and the OPENIN file are treated separately, using different buffers. This saves any problems of deleting the old file and changing the name of the newly created file. The operating system will see to it that the new file is recorded as AIRCRAFT.DAT, and the old file is renamed AIRCRAFT.BAK. One point we have to be *very* careful about, however, is closing files. The CLOSEIN command is used whenever the program has finished reading the old file, and the CLOSEOUT command is used whenever the last of the new files has been added.

Looking at the program in detail, line 110 opens two files *with the*

```
100 X%=0
110 OPENIN "AIRCRAFT.DAT":OPENOUT"AIRCRA
FT.DAT"
120 CLS:LOCATE 14,11:PRINT"PLEASE WAIT":
WHILE EOF=0: INPUT#9,Name$:PRINT#9,Name$
130 FOR N%=1 TO 5
140 INPUT#9,Gen$:PRINT#9,Gen$
150 NEXT N%:WEND:CLOSEIN
160 CLS:PRINT TAB(15)"ADDITIONS":PRINT:P
RINT
170 INPUT"Aircraft name ";Name$:IF Name$
="X" OR Name$="x"THEN 220
180 X%=X%+1:PRINT#9,Name$:FOR N%=1 TO 5
190 PRINT"Field item ";N%;" is ";:INPUT
GEN$
200 PRINT #9,Gen$:NEXT N%
210 GOTO 170
220 PRINT:PRINT"You have added ";X%;" it
ems."
230 CLOSEOUT
240 PRINT"END":END
```

Fig. 9.4. Extending a serial file by reading, and rewriting.

same names. One, however, is an input file, and the other is an output file. The input file will be used by INPUT#9, and the output file by PRINT#9, so there should be no conflict between them, since they use separate buffers. Line 120 clears the screen and issues a PLEASE WAIT notice. If your files are long, it may take the disc some time to do all the reading and writing, and this notice is a reminder that it's all happening. Never leave a user with a blank screen, even if the user is always yourself! In these lines 120 to 150, data will be read in from the old file and written out to the new one until the EOF marker is found. When this happens, the WEND in line 150 takes effect, and the next command is CLOSEIN, which shuts down the reading file. The writing file is still open, however, with its buffer containing data that has been read so far. You can now add more data, using the same lines as you used in the program of Fig. 9.2. When an X or x is typed in response to the request for a record name, then the program displays the number of added items, closes the write file (so recording the file), and stops. Quite easy, really, but in this program, no provision has been made for altering any of the records that are read from the old file. This is a routine which we can easily add – and that's the next thing to look at.

Changing a record

It's not difficult to find how to alter a record on a file. You read the item, print it, and then change the item before rerecording the file. The main problem is finding a neat way of doing this. The program of Fig. 9.5 shows one approach which I use in my own file programs. This is to read the whole of one record, display it on the screen, and give the user the chance to edit or leave as need be. The editing is visual, rather than by the old-fashioned method of numbering the entries and asking for a number to be entered. When the record is displayed, a flashing arrowhead points at the first field. If you want to leave the record as it is, then press the COPY key, and this will bring up the next record. If you *do* want to change a record, move the arrowhead to the record using the cursor-up and cursor-down keys, the arrowed keys above and below the COPY key. When the arrowhead points to the field that you want to change, press the spacebar. This wipes out the field name on the screen, but not in the memory. You can now type a new field name or number, and terminate it with the ENTER key. Only when you have pressed the ENTER key is this new field entered, and if you want to change your mind, you can delete your entry and type the old entry again.

When a change has been made in this way, the arrowhead still points to the same field, and you can make another change to this or, by shifting the arrowhead, to any other field. When you have finished editing the record, you can press the COPY key to bring up the next record. The process will continue for as long as there are records to read. The amended file will be recorded as AIRCRAFT.DAT, and the old file will be renamed AIRCRAFT.BAK. Note, however, that the visual editing system is useful only if the fields are short – a field which spreads over more than one line will cause problems.

How it works

Lines 100 to 140 follow the pattern which should be familiar to you by now. The files are opened, one for reading and the other for writing, and the WHILE EOF=0 loop will load in each record until the end of the file. The record name is assigned to Gen$(0), however, to make it easier to work with the fields in one array. The new items start with the GOSUB 1000 in line 150. This carries out the editing, and when editing of a record is complete, this subroutine will return. The

```
100 DIM Gen$(5):X%=0
110 OPENIN "AIRCRAFT.DAT":OPENOUT"AIRCRA
FT.DAT"
120 CLS:LOCATE 14,11:PRINT"PLEASE WAIT":
WHILE EOF=0:INPUT#9,Gen$(0)
130 FOR N%=1 TO 5
140 INPUT#9,Gen$(N%)
150 NEXT N%:GOSUB 1000
160 FOR N%=0 TO 5:PRINT#9,Gen$(N%):NEXT
170 WEND:CLS:LOCATE 14,11:PRINT"PLEASE W
AIT":CLOSEIN:CLOSEOUT
180 PRINT"END.":END
1000 CLS
1005 LOCATE 1,23:PRINT"Press arrow key t
o move cursor, space   to alter item, CO
PY to end edit."
1010 PX%=5:PY%=4
1030 FOR N%=0 TO 5
1040 LOCATE PX%,PY%+N%
1050 PRINT Gen$(N%):NEXT
1060 PX%=1:PY%=4
1070 LOCATE PX%,PY%
1080 PRINT">";:FOR J=1 TO 100:NEXT
1090 LOCATE PX%,PY%
1100 PRINT" ";:FOR J=1 TO 50:NEXT
1110 IF INKEY(47)=0 THEN GOSUB 2000
1115 IF INKEY(9)=0 THEN RETURN
1120 IF INKEY(0)=0 THEN PY%=PY%-1
1130 IF INKEY(2)=0 THEN PY%=PY%+1
1140 IF PY%<4 THEN PY%=9
1150 IF PY%>9 THEN PY%=4
1160 GOTO 1070
2000 CALL &BB03:PX%=5:LOCATE PX%,PY%:PRI
NT SPC(34);
2010 LOCATE PX%,PY%:INPUT;Gen$(PY%-4)
2020 PX%=1:RETURN
```

Fig. 9.5. A file editing program which uses a visual menu choice.

amended or unamended record is then put into the new file by line 160, and the WEND in line 170 brings up the next record.

In the subroutine, the screen is cleared, and a message about the editing commands is printed at the bottom of the screen. Lines 1010 to 1050 then print the record name and each field on to the screen.

Note that this works only if each field is of less than 35 characters, because the whole method depends on using one line for each entry. In line 1060, X and Y positions for the cursor are assigned to PX% and PY%, and a loop starts in line 1070. In the loop, the '>' character is printed at the cursor position, held for a short time, then removed. Four keys are then tested by using the INKEY command. INKEY(47) tests the SPACEBAR, and if this is pressed, the GOSUB 2000 calls up the replacement routine. INKEY(9) tests for the COPY key, and causes a return from the subroutine if this key is pressed. The other two tests are for the cursor movement keys, and if one of these keys is pressed, then the value of PY% is altered. Lines 1140 and 1150 then test the value of PY%, to ensure that it does not stray outside the line limits, and line 1160 completes the loop.

When the SPACEBAR is pressed, the first action in line 2000 is CALL &BB03. This is a machine code call to the operating system, and its effect is to remove any codes from the keyboard buffer. This is essential, because when any key is pressed, its code is stored in memory until it is read. When the cursor movement keys are used, their codes accumulate in this keyboard buffer despite the fact that the program is using the keys differently. As a result, when the program leaves the loop, these codes are read and acted on. For example, if you have used the down-cursor key twice to shift to the second field, there will be two 'shift down' codes in the buffer. There will also be a 'space' code because you pressed the SPACEBAR. The effect would be to make the '?' prompt for the INPUT stage appear at the line you have selected, but the cursor would appear two lines down and one space across. This can be avoided if the buffer is emptied before the INPUT step, and this is done by the CALL &BB03. If you are curious about this and other calls to the operating system, they are documented in an Amsoft publication, the *Concise Firmware Specification.* An alternative is to use the CLEAR INPUT command.

With the buffer flushed out like this, the PX% number is changed so as to locate the first character of the entry, and the SPC(34) clears most of the line. The next LOCATE instruction places the cursor back at the start of the field name, and the INPUT then allows you to make the change. By using Gen$(PX%−4), you assign the new entry to its correct place in the array. Line 2020 then restores the value of PX% to place the arrowhead correctly, and the routine returns. The effect is quite impressive, though the key actions in the loop are slightly 'sticky' because of the time delays. If longer fields are used, it would be possible to modify the routine to make use of, say, two lines

per field. In any case, this and the previous routine demonstrate how serial filing can be used with advantage on a disc system. In many cases, you will find this type of filing more useful than random access filing, which is never easy with any disc system.

Chapter Ten

A Database Example - Filing Cabinet

This chapter consists mainly of one long listing (Fig. 10.1) for a database type of program. The program is called Filing Cabinet, and it allows you to specify four heading titles for the fields of your records. These field names are recorded on the disc, and will be used from then on. They might, for example, be Name, Address, Age, Telephone Number. You can then enter information, add, delete or change information, read all of the data or select items as you please. These are the normal actions of a simple database. Looking at the length of the program, you might wonder how long a complicated program would be, but this *is* a simple version. There is no facility, for example, for printing records of any field in alphabetical order. This is, you see, a skeleton database, which has been included to illustrate

```
10 OPENOUT"DUMMY":MEMTOP=HIMEM
20 MEMORY HIMEM-1:CLOSEOUT
30 ON ERROR GOTO 1420:ON BREAK GOSUB 280
40 NL$=CHR$(10)+CHR$(13)
50 REM FILING CABINET by Ian Sinclair 19
85
60 DIM H$(4),E$(4)
70 M$="Please make sure that the disc is
 in"+CHR$(10)+CHR$(13)+"the drive, corre
ct way up."
80 J%=1:Y$="Please answer Y or N.":Z$="P
ress any key..."
90 RESTORE:FOR N%=1 TO 4:READ H$(N%):NEX
T
100 REM Place in lines 60 to 80 any prin
ter setup instructions that you need.
110 REM
120 REM
```

Fig. 10.1. The database program, "FILING CABINET".

```
130 DATA First,Second,Third,Fourth
140 CLS:T$="Filing Cabinet":GOSUB 290
150 T%=2:GOSUB 1430:PRINT:PRINT"Do you n
eed instructions? ";Y$
160 GOSUB 300:IF K$="Y"OR K$="y" THEN GO
SUB 310
170 CLS:T$="MENU":GOSUB 290
180 PRINT:PRINT "1.Start NEW type of fil
e.":PRINT"2.Start ENTRY in file.":PRINT"
3.DELETE,CHANGE or ADD items.":PRINT"4.L
IST complete file.":PRINT"5.PICK one ite
m.":PRINT"6.END Program."
190 PRINT:PRINT"Please choose by number.
":GOSUB 300:K%=VAL(K$)
200 IF K%<1 OR K%>6 THEN PRINT"1 TO 6 ON
LY- PLEASE TRY AGAIN.":GOTO 190
210 IF K%=1 THEN GOSUB 440:GOTO 260
220 K%=K%-1:IF F$=""THEN GOSUB 1390
230 GOSUB 1300
240 ON K% GOSUB 540,650,690,790,1050
250 GOSUB 1310
260 CLS:PRINT"Do you want to return to t
he menu?"
270 PRINT:GOSUB 300:IF k$="Y" OR K$="y"
THEN 180
280 CLOSEIN:CLOSEOUT:MEMORY MEMTOP:PRINT
"END.":END
290 PRINT TAB(20-(LEN(T$))/2);T$:RETURN
300 K$=INKEY$:IF K$=""THEN 300 ELSE RETU
RN
310 CLS:T$="INSTRUCTIONS":GOSUB 290:PRIN
T
320 PRINT TAB(2)"This program allows you
 to set up and":PRINT"use a serial file
database. You will":PRINT"be asked to pr
ovide four titles, which"
330 PRINT"will be recorded along with a
filename.":PRINT" You can then use the o
ther options to":PRINT"put entries into
the file, with your":PRINT"headings appe
aring as prompts. You can"
340 PRINT"add to the file, change or del
ete items":PRINT" and list the file as y
ou wish."
```

Fig. 10.1. contd.

```
350 PRINT:PRINT"The main restriction is
that you must":PRINT"NOT enter anything
which contains a":PRINT"comma. You need
Menu Item 1 only when":PRINT"you start a
 new file for the first"
360 PRINT"time. For the rest of the time
 that":PRINT"this file is in use, the ot
her options":PRINT"apply. Keep one disc
side for each":PRINT"different file- you
 can keep a copy of":PRINT"this program
on each disc side as well."
370 PRINT
380 PRINT M$
390 PRINT:PRINT Z$
400 GOSUB 300
410 RETURN
420 INPUT Q$:IF LEN(Q$)>38 THEN PRINT"To
o long- please change now.":GOTO 420
430 RETURN
440 CLS:T$="New File Specification":GOSU
B 290:T%=2:GOSUB 1430
450 PRINT:PRINT"Now select your four tit
les for this":PRINT"new file, using ENTE
R after each title.":PRINT"Only four tit
les can be used.":PRINT
460 FOR N%=1 TO 4:PRINT H$(N%);" is- ";:
GOSUB 420:PRINT Q$
470 E$(N%)=Q$:NEXT
480 PRINT"End of titles specification- n
ow we ":PRINT"need a filename of up to e
ight ":PRINT"characters- no more.":PRINT
490 INPUT"Filename is- ",F$
500 IF LEN(F$)>8 THEN PRINT"Too long- ei
ght characters only.":PRINT"Please try a
gain.":GOTO 490
510 F$=F$+".DAT":E$(0)=F$
520 OPENOUT"HEADS.DAT":FOR N%=0 TO 4:WRI
TE#9,E$(N%):NEXT:CLOSEOUT
530 RETURN
540 CLS:T$="Entry of Items.":GOSUB 290:T
%=2:GOSUB 1430
550 PRINT"Items can now be entered until
 you ":PRINT"enter X as the first of a s
et."
```

Fig. 10.1. contd.

```
560 PRINT"Entry No. ";J%
570 PRINT E$(1):INPUT Q$:IF Q$="X" OR Q$
="x"THEN 630
580 PRINT E$(2):INPUT R$
590 PRINT E$(3):INPUT S$
600 PRINT E$(4):INPUT U$
610 WRITE#9,Q$:WRITE#9,R$,S$,U$
620 J%=J%+1:GOTO 560
630 J%=J%-1:PRINT"End of entry.":T%=2:GO
SUB 1430
640 RETURN
650 CLS:PRINT:PRINT"Do you want to - ":P
RINT:PRINT"1. ADD to the file.":PRINT"2.
 CHANGE an item.":PRINT"3. DELETE an ite
m.":PRINT"4. RETURN to the main menu."
660 PRINT:PRINT"Please select by number.
":GOSUB 300:K%=VAL(K$):IF K%<1 OR K%>4 T
HEN PRINT"1 to 4 only- please try again.
":GOTO 660
670 ON K% GOSUB 1070,1110,1210,1290
680 RETURN
690 CLS:T$="FILE LISTING":GOSUB 290:T%=2
:GOSUB 1430
700 PRINT:PRINT"Do you want to use the s
creen or the":PRINT"printer for your lis
ting?"
710 PRINT:PRINT"Please press P or S key.
"
720 GOSUB 300:IF K$<>"P"AND K$<>"p"AND K
$<>"S"AND K$<>"s" THEN PRINT"P or S only
 - please try again.":GOTO 720
730 Z%=0:IF K$="P" OR K$="p" THEN Z%=8
740 C%=1:WHILE NOT EOF: GOSUB 1320
750 PRINT#Z%,"Item ";C%;"_"
760 GOSUB 1330:PRINT#Z%,E$(1);": ";Q$+NL
$+E$(2)+": ";R$+NL$+E$(3);": ";S$+NL$+E$
(4);": ";U$+NL$
770 C%=C%+1:IF Z%=0 THEN PRINT Z$:GOSUB
300
780 WEND:RETURN
790 PRINT:T$="PICK AN ITEM":GOSUB 290
800 PRINT:PRINT"You can pick by number (
N) or by ":PRINT"letter (L)."
```

Fig. 10.1. contd.

```
810 PRINT:PRINT"Please press N or L key
now."
820 GOSUB 300:IF K$<>"N" AND K$<> "n" AN
D K$<>"L" AND K$<> "l" THEN PRINT"N or L
 only _ please try again.":GOTO 820
830 IF K$="N" OR K$="n" THEN GOSUB 860
840 IF K$="L"OR K$="l" THEN GOSUB 940
850 RETURN
860 PRINT"What number item do you want?"
:PRINT"Type number, then press ENTER key
."
870 INPUT X%:N%=1
880 WHILE NOT EOF:GOSUB 1320
890 GOSUB 1330:IF N%=X% THEN CLS:PRINT E
$(1);": ";Q$+NL$+E$(2);": ";R$+NL$+E$(3)
;": ";S$+NL$+E$(4);": ";U$:GOTO 920
900 N%=N%+1:WEND
910 PRINT"Item not found"
920 T%=1:GOSUB 1430:PRINT:PRINT"Press an
y key to return.":GOSUB 300
930 RETURN
940 CLS
950 PRINT:PRINT"Type first few letters o
y key to return.":GOSUB 300
930 RETURN
940 CLS
950 PRINT:PRINT"Type first few letters o
f the first ":PRINT"entry. Don't forget
capital letters.":PRINT"If you use one l
etter only you will":PRINT"get all entri
es which start with that":PRINT"letter."
:PRINT
960 PRINT"Press ENTER key after typing l
etters."
970 INPUT T$:Y%=LEN(T$):FD%=0
980 WHILE NOT EOF:INPUT#9,Q$
990 INPUT#9,R$,S$,U$:IF LEFT$(Q$,Y%)=T$
THEN PRINT E$(1);": ";Q$+NL$+E$(2);": ";
R$+NL$+E$(3);": ";S$+NL$+E$(4);": ";U$+N
L$:FD%=-1
1000 WEND
1010 IF FD%=0 THEN PRINT"Item not found"
1020 T%=2:GOSUB 1430
```

Fig. 10.1. contd.

```
1030 PRINT Z$;" to return.":GOSUB 300
1040 RETURN
1050 REM Sub-menu routines
1060 REM start with add to file.
1070 J%=0:WHILE NOT EOF
1080 GOSUB 1320:J%=J%+1:GOSUB 1330
1090 GOSUB 1340:WEND
1100 J%=J%+1:GOSUB 550:RETURN
1110 CLS:T$="CHANGE ITEM":GOSUB 290
1120 GOSUB 1350:GOSUB 1360:GOSUB 1320:GO
SUB 1330
1130 PRINT Q$+NL$+R$+NL$+S$+NL$+U$
1140 PRINT E$(1);:INPUT Q$:PRINT E$(2);:
INPUT R$
1150 PRINT E$(3);:INPUT S$:PRINT E$(4);:
INPUT U$
1160 WRITE#9,Q$,R$,S$,U$
1170 WHILE NOT EOF
1180 GOSUB 1320:GOSUB 1330
1190 GOSUB 1340:WEND
1200 RETURN
1210 CLS:T$="DELETE ITEM":GOSUB 290
1220 GOSUB 1350:GOSUB 1360
1230 PRINT:PRINT:T$="PLEASE WAIT...":GOS
UB 290
1240 GOSUB 1320:GOSUB 1330
1250 D$=Q$
1260 WHILE NOT EOF:GOSUB 1320:GOSUB 1330
1270 GOSUB 1340:WEND
1280 PRINT:PRINT:T$=D$+" DELETED!":GOSUB
 290:T%=2:GOSUB 1430:RETURN
1290 RETURN
1300 OPENIN F$:OPENOUT F$:RETURN
1310 CLOSEIN:CLOSEOUT:RETURN
1320 INPUT#9,Q$:RETURN
1330 INPUT#9,R$,S$,U$:RETURN
1340 WRITE#9,Q$:WRITE#9,R$,S$,U$:RETURN
1350 PRINT"Please type number of item.";
:INPUT Z%:N%=Z%-1:RETURN
1360 FOR J%=1 TO N%:GOSUB 1320:GOSUB 133
0
1370 GOSUB 1340:NEXT:RETURN
1380 REM Get headings and filename
```

Fig. 10.1. contd.

```
1390 OPENIN "HEADS.DAT"
1400 FOR N%=0 TO 4:INPUT #9,E$(N%):NEXT
1410 F$=E$(0):CLOSEIN:RETURN
1420 PRINT"ERROR ";ERR;" IN LINE "; ERL:
GOTO 280
1430 START=TIME:WHILE TIME < START+300*T
%:WEND:RETURN
```

Fig. 10.1. contd.

the use of the disc drive for this type of program. Once you have this program up and running, *and have completed reading this book*, you should be able to add whatever extra trimmings you need.

First principles

We shall start by looking at how the program works in outline. Two files are used, both of which are serial files. One short file, called HEADS.DAT, is used to keep a record of your four headings and of the filename for the main file. The purpose of this file is to make the action of the program automatic, so that you don't have to remember a filename or heading names. When you first use the program for a new variety of file, you will format a new disc side, record the program on it, run it so as to type these titles, which stay with the file from then on unless you start another type of file on the same disc side. If you want to use more than one Filing Cabinet, you must keep them on separate discs, or different sides, with a copy of the program on each disc side. The main serial file will carry a filename that you will specify when you first start a file, and it can be of as large a size as will fill a disc. Each record uses either its position number or its first field as a 'key' to finding that record. In other words, you can locate a record by knowing that it is number 58, or by knowing that the first field is surname, and you want to find Carruthers. This scheme is fairly flexible without being too difficult to implement. As I said, this is a skeleton program, and it's yours to trim to shape and pad out as you please. Because of the way that the disc system operates, there is always a backup copy of each file on the disc, with the .BAK extension, which makes the system quite safe to use.

When the program runs, some set-up work is done and you are presented with the main menu. The first time that you use the program on a disc, you should go for the 'Start NEW type of file' option. This allows you to choose four titles for the fields of your

records. These names will be recorded and used forever after, so you should plan them carefully. Figure 10.2 shows a typical display. After entering the fields and lengths, you are prompted for a filename. You can choose anything you like, as long as it has eight characters or less, and is not HEADS.DAT. Perhaps you might like to add a line 495 which rejects this name as a filename, and asks again?

```
           New File Specification


   Now select your four titles for this
   new file, using ENTER after each title.
   Only four titles can be used.


   First is- ? Name

   Name

   Second is- ? Address

   Address

   Third is- ? Age

   Age

   Fourth is- ? Phone No.

   Phone No.

   End of titles specification- now we
   need a filename of up to eight
   characters- no more.



   Filename is- friends
```

Fig. 10.2. A typical screen display during the entry of titles.

Once the filename has been typed and ENTER pressed, the HEADS file is opened and the headings, along with your filename for the main file, will be recorded. At this point, and at several other places in the program, the disc will be busy, and you may have to wait for it. When the file has been created, the program returns to the menu. You do *not* have to use this option again unless you decide to keep another different file of data on the same disc. There's nothing

wrong with this if your data files are fairly short, but it avoids confusion if you can keep one disc side for each file.

You can now type the first group of data into your file by choosing the 'Start ENTRY in file' option. You don't have to do so at once, of course, because the headings and filename are by now safely on the disc, and you can end the program and switch off if you like. When you go for the entry option, you will be prompted by the field names (such as Address, Name, etc.) to type in data. The data will be restricted by line 420 to 38 characters per entry. This has been done to make the screen appearance slightly neater, though data can still spill from one line to the next if both titles and data items are long. Once again, this is something that you can change as you wish. You will find if you enter a lot of data that the disc spins at intervals while you enter data, and you have to wait until the screen cursor is visible again before you can continue typing. To end the entry, you type X, in lower-case or upper-case. If this is inconvenient, change it (line 570)! The whole file will be recorded on the disc, using the filename that you supplied originally. Once again, you can leave the program by selecting option 6 of the menu after you have recorded as many items as you want.

If you select the DELETE, CHANGE or ADD option from the main menu, you will be presented with another menu. This time, the choice is to add to the file, change an item, delete an item, or just return to the main menu. You must not use any of the first three options unless a file has already been created, which is why you have the 'cop-out' option. If you choose to add to the file, the disc drive will read the whole of the file, rerecord it, and then stay open for additions to the file. The items will be correctly numbered, so that you know how many items were on the file, and the number of each item that you add. If you take the CHANGE option, you will be asked to identify which item you want to change, and the identification is by number only in this case (once again, this *could* be changed). When you enter a correct item number, the item is found by reading all of the file up to this point and rewriting it. The item whose number you have requested is then printed on the screen. This allows you to check whether you really want to change the record. There is, however, no drop-out option at this point, and you have to enter items for each field. Perhaps it might be useful to allow pressing ENTER to leave the original item unchanged? If that's what you want, then use an INPUT with a temporary assignment here, test it, and then assign to Q$, R$, S$ or U$ only if ENTER has not been pressed. When the change has been made, the changed item, and all the rest of the file, is

rerecorded. If you choose to delete an item, then once again the item is selected by number, and the file is read and rerecorded as far as the preceding item. The item to be deleted is then read, its title assigned to a variable name, and the rest of the file read and rewritten. The title of the deleted file is then displayed with the information that it has been deleted. Perhaps in this choice you might like to add a piece of routine that lets you read the record and decide whether or not to delete it? If you don't want to delete, then it can be rewritten along with the rest of the file.

Getting back to the main menu, option 4 allows you to list the complete file. When you choose this option, you get another choice, of listing on screen or on the printer. This is done by pressing the S or P keys. For business purposes, this might *always* be a printer option, but if you are using this program for hobby or household interests, you might use only the screen option. Obviously, if you have no printer, you might want to delete the P option. If you take the screen display option, each record is displayed and held for you to inspect. You can press the spacebar to get the next record until the end of the file. You might want to dispense with this, simply using the CPC664's ESC key to prevent the listing running away from you. If the listing is to the printer there are no pauses, the whole listing is printed. Lines 100 to 120 of the program, incidentally, have been kept clear for you to insert any special printer set-up instructions. As we shall see in Chapter 15, the very popular Epson range of printers require some set-up codes if they are to work properly with the CPC664.

Item 5 on the main menu allows you to pick one record for examination. You are then asked if you want to choose by item number or by letters. If you choose by item number, then the program reads the file to the correct place and prints the record whose number you have requested. If you choose to select a name, you are asked to type the name, or the first letter or letters of the name. If you type the whole name, the file will be located only if your typed name agrees *exactly* with the name in the file. You can, however, type just the first letter, and this will result in a list of all the records whose first field starts with this letter. If you type more than one letter, you will get all names which start with these letters - it's rather like using a wildcard in a disc filename. There are no options here for using the printer, nor for looking at one record at a time, but you can add these facilities as you choose.

The program in detail

Now for the hard work. There are many points in this program which are important. If you try to design your own database programs, you will need to know what the disc drive does, and this listing reveals much that isn't exactly made clear by the manual, and which is not so easy to illustrate by short examples. No matter how much you may hate looking at other people's programs, then, it will be useful to study this one, so that you can appreciate reasons for some of the lines. Unless you do so, you can waste a lot of time in your own programming looking at inscrutable error messages and wondering why they arise.

The program is built round a core and a set of subroutines. Much of the programming is straightforward BASIC, and I have made no use of fancy colour or screen presentation effects – there's quite enough to type as it is. Programs for business purposes use the printer for anything important, and the screen is used only for messages to the operator like 'Try putting a disc in the drive'. I'll concentrate on the explanations that relate to the use of the disc drive, rather than explaining everything in detail. In other words, I'm assuming that you knew a reasonable amount of BASIC before you started on this program.

Lines 10 to 130 are concerned with initial values of constants and setting up the system. The first two lines need more explanation than you will find in any of the manuals. Normally, when you open and close buffers, the CPC664 creates space by shifting the top limit of memory. This can create some odd effects in your program, and one of the most puzzling is the corruption of filenames. In this program, the filename for the data is held as a string variable F$. Commands such as OPENIN F$ are then used, but if you follow the normal course that we have outlined so far, you run into trouble with this command. This crops up when you open a file some time after making a menu choice. You may, for example, have defined F$ as 'MYFILE.DAT', but if a choice number 1 on the menu has been made, followed by OPENIN F$, you find that the disc system has been asked to find a file called '1MYFILE.DA', and it can't find it. A second attempt, using CONT, will always succeed, but this can be done automatically only by using ON ERROR GOTO in conjunction with ERR and DERR. Clearing the keyboard buffer does not solve the problem in this case, and the only remedy that I have found is to allocate the buffers permanently, using a set of steps which is mentioned in the *Concise BASIC Specification Manual.*

Lines 10 and 20 allow a dummy OPENOUT to shift the memory, and then allocate this limit permanently by using the MEMORY command. The dummy file is then closed again. This method of reserving space permanently for buffers solves all of the curious problems that arise when files are opened and closed.

Line 30 uses an ON ERROR GOTO statement to trap any errors in the BASIC, and ensure that an error will close all files and end the program. This includes syntax errors, so if the program ends when you start to run it, you know you have made a typing error! Line 1420 will analyse the error for you before the shutdown. Always check your data files if you have found an error message, because if this happens in the middle of a change-item action, you will have a data file which is only half full. You will then have to recover the old data file, which will have the .BAK extension. Once the program is up and running, with all mistypings removed, however, you should not find that the error trap ever operates. Lines 70 and 80 contain messages which are used in places, and which you might want to use considerably more. You might also want to use these messages in windows of different colour, to draw more attention to them. Line 90 reads the words which number the headings, and the program starts in earnest in line 140. This prints a title , centred by the GOSUB 290, and asks if you need instructions. The GOSUB 1430 is a time delay routine which will give you a delay of as many seconds as you assign to the integer T%. You are then asked if you need instructions. I have not written *very* detailed instructions, because these just involve another lot of typing. There is enough to remind you of what to do if you have not used the program for some time. You can type your own instructions if you have modified the program for your own use. By allowing the choice of skipping the instructions you can get into the program faster if you use it frequently.

Line 170 clears the screen and then prints the menu. You are asked to select by number, using the GOSUB 300 subroutine and converting to number form with VAL. This number is assigned to K% (an integer) and tested. If the range is acceptable, then lines 210 to 240 carry out the choice. This is not completely straightforward, because choice 1 is very different from the others. It is used *only when a new type of file is to be created*, and it opens the file HEADS.DAT. Because of this, it has to be treated separately. This is done by line 210, and because one choice has been removed from the list in this way, the value of K% has to be reduced by one. If the choice is to be any other item, the program must then check that the data which is contained in file HEADS.DAT is present. This is done in line 220 by

testing for the filename F$. If this is a blank, then the file HEADS.DAT must be read, using GOSUB 1390. If you have been using other parts of the program and have returned to the menu, this file will already have been read, and it won't have to be read again. When you reach line 230 you are definitely choosing one of the actions that will need the data file to be opened, so the subroutine which opens the files is used. Line 240 then makes the choice of subroutines, and when the subroutine is finished, line 250 closes the files again. This ensures correct file use unless the program is stopped within a subroutine. Lines 260, 270 give you a chance to return to the menu unless you have picked the 'END program' option. The subroutines carry out all of the main actions. This is important, because it makes the program very easy to change. Practically all the subroutines that you might need for your own 'custom' version are listed, so if you know in detail what each subroutine does, making your own version is relatively easy.

The creation subroutine

The subroutine that starts in line 440 creates a completely new file. This will wipe out any other file that has been created with this program on the same disc side, which is why it is useful to have several copies of the program on different disc sides. The loop that starts in line 460 gets title names for each field. The INPUT stage for this is handled by a separate subroutine in line 420, which allows the length of title to be tested. You might want to use this subroutine also to test changes to the file. Each title and length is assigned to an array variable E$, with four items. As always, you can change this to suit yourself. If all is well, then you are asked for a filename in lines 480, 490. Once again, it might be useful to test this to make sure that the name HEADS was not used, and that no extension is placed on this name. Line 510 adds the extension of DAT, and assigns this also to E$(0). This allows the headings and the filename to be recorded by one loop in line 520. When this has been done, the program returns in line 530 – in this case, the return will be to the GOTO 260 command in line 210. You can then either return to the menu, or end the program. Your titles and filename are now recorded, and the program is now ready to make use of this new file. You will *not* make use of this menu option again until you come to choose another subject for filing.

Writing to the file

Selecting the 'START entry in file' option in the main menu leads to the subroutine which starts in line 540. If this option is selected, this will replace any existing file. It is used, therefore, just after a new file has been created by the use of option 1. If you want to add items to a file, then option 3 should be used. Line 550 gives brief instructions, and the titles of each field are printed from the array E$ as reminders. No attempt has been made to restrict the size of each entry, and you might want to do this for yourself. For a new file, the record number J% has been assigned with starting number 1 in line 80. The test at the end of line 570 checks for the entry of the letter X as the first field, because this terminates the entry. Line 630 will then correct the count number, and the subroutine returns. When it returns, the action of closing files will ensure that all data has been recorded. The write line is in 610, and this uses WRITE#9 rather than PRINT#9. The reason is that WRITE#9 allows strings to be separated more easily when the output is in a form such as WRITE#9,R$,S$,U$. If PRINT#9 is used in such a case, then the read (using INPUT#9,R$,S$,U$) will *not* separate the items correctly. Our examples so far have used PRINT#9 in a loop, which has side-stepped this problem. Note that when X is used to terminate an entry, this letter is *not* recorded.

Once a file has been opened with menu choice 1, and written to with menu choice 2, it can be further used by choices 3, 4 and 5. You would not normally use choices 1 and 2 again, but the remaining choices will come in for heavy use from now on. We'll start with the 'heavyweight' item, choice 3. This allows addition to a file, alteration of a record, or deletion of a record. So that all of these actions can be catered for, this menu choice leads to another menu in line 650. This in turn leads to three more subroutines.

Choosing addition to a file leads to the subroutine in line 1070. Since the files are serial, adding to a file really means reading and rewriting the whole of the existing file, and then leaving the file open so that more items can be added. The reading and rewriting of the existing file is done by the WHILE...WEND loop in lines 1070 to 1090. In this loop, the counter variable J% is used to count the records as they are read and rewritten. Line 1100 then increments J%, and calls the original entry subroutine so that more items can be added. As before, entering the letter X as the first field of a record will terminate the addition of records. The rest of the file is written, and the files are closed when the routine returns.

When you take the 'change' option of this extra menu, you have to

specify which item is to be changed. The subroutine starts in line 1110 with its title, and then calls the subroutine at line 1350 to find the number of the item. This is a convenient method from the point of view of easy programming, but if you want to find the item by letter, then a method will be studied later. When the number is specified, Z% is set to a value of one less than this chosen number. The subroutine at line 1360 is then used to read and rewrite all records up to this item. The chosen record is then read, using the GOSUBs at the end of line 1120. Line 1130 then displays the record, and lines 1140 and 1150 are used to change each field of the record. You could use a more elaborate routine here, allowing the ENTER key with no entry to mean that the field should be unchanged. You might also want to add the option of leaving the whole record unchanged if you decide that you don't want to change it after all. Line 1160 writes the changed record to the buffer, and lines 1170 to 1190 write the rest of the file.

Listing the file

The file listing option in the main menu leads to line 690, prints the title of the choice, and asks for the options of screen or printer. This choice is made by pressing the S or P keys, and the choice is tested in line 720. The result causes Z% to be assigned with 0 for screen listing, or 8 for printer listing. This allows the expression PRINT#Z% to be used to give either type of output. The items are printed one on each line, with the item number. This item number can be used also in picking individual items for changing or deleting, using option 3. In line 770, if Z%=0, meaning screen output, the listing stops after each item to give you time to read it. You could, if you like, modify this so that you have the choice of paused or continuous listing. Listing to the printer is *always* continuous.

When the fifth option, to pick an item, is chosen, the subroutine which starts at line 790 is used. This prompts for a choice of number or letter selection, which is made in lines 830, 840. These are dealt with by separate subroutines, with the number choice routine starting in line 860. Taking this option first, the number is entered, and the two subroutines at 1320 and 1330 are used to input the fields, using a WHILE...WEND loop rather than a FOR...NEXT because of the ease of detecting an impossible record number. If the item is found, then line 890 will print it and break the loop. If it is not found, then the EOF marker operates the WEND and the message in line 910 is printed.

If the letter selection method is chosen, the subroutine starts at line 950 with brief instructions. The letter or group of letters is entered in line 970, and the number of characters is assigned to Y%. The 'marker' variable FD% is also set to zero. A loop starts in line 980 which will input each field name and compare a number of letters equal to Y% with the letters that you have entered. If these two are identical, then all of the fields of the record are printed and FD% is changed to −1. The loop continues over the whole file, because there may be more than one entry which uses the same letter or group of letters. Line 1010 will print the message only if no item has been found, using the value of FD% to indicate this.

The last menu option is simply to end the program. This is necessary, because it provides a way of ending without having to carry out any of the other menu actions which would open files. You should *never* have to end a program by pressing the ESC key twice, because this can result in the program leaving files open, and so leaving you with an incomplete data file on the disc. This has been avoided in this case by having ON BREAK GOSUB 280 in line 30. This ensures that pressing ESC twice will cause the program to end by carrying out line 280, rather than by breaking, and it therefore safeguards the files *to some extent*. The safeguard is not perfect, however, because it depends where you break. If you are deleting an item, for example, part of the file will have been read and passed to the output buffer. If you break at this point, the first part of the file will be saved, but the second part will not. You will then have to use the .BAK copy to restore your file. To do so, delete the new copy, and rename the .BAK file with the name of the file which should be the new file. You still have the .BAK copy as your backup.

That's all there is to it. Taken as a whole, it looks rather intimidating, but when you split it into core and subroutines, as it was when it was written, it looks a lot simpler. It's by no means a polished piece of programming. You'll find, for example, that more use could be made of subroutines in some sections. You'll certainly find that you will want to modify parts of the program to suit your own needs. It's yours now, so modify it as you wish, but please don't sell it or publish it as your own work!

Chapter Eleven
Windows and Other Effects

Devices and streams

As you progress with programming your CPC664, you'll increasingly come across the words 'device' and 'stream'. 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 monitor (or TV), you will see something appear on the screen. The keyboard is a transmitting device because it sends signals. The screen is a receiving device because it accepts signals. The CPC664 screen, in fact, can behave as if it were several devices because it can be used in separate portions. Later in this chapter 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 disc system is also a device which can either record or replay data.

I 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*, which is the amount of memory that is needed to store one ASCII character, or one command word in BASIC. Tap a key on your keyboard, and one byte of data is transferred, in the form of electrical signals, from the keyboard to the screen. *Streams* are the paths which are used to carry these signals, and the 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, for example. If you were typing a special password, however, and you didn't want anyone who was

watching to see it, it would make sense to break the stream that connects the keyboard to the screen. Figure 11.1 shows this in action. Line 10 asks you to enter a four-letter password. In line 20, PRINT CHR$(21) will disconnect the stream that normally links the keyboard with the screen. You can now enter your password, using the INPUT in line 30, but nothing will appear on the screen! Line 40 links the stream again so that messages can be displayed. As usual, the entry is tested so that you can try again if you have not entered four letters. If you want to check that you really did enter something, then a PRINT B$ will reveal it.

```
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. 11.1. How to conceal an entry by shutting off the screen stream!

As you have probably realised by now, there are some streams that are already connected from the moment you switch on. There is obviously a stream that links the keyboard and the screen. This is a stream numbered 0, and you can force a PRINT instruction to use this stream by putting #0 following the PRINT. The CPC664 is designed to make use of ten stream numbers. Of these, three (0, 8 and 9) have fixed jobs to do, and you should normally try to avoid changing these tasks. The streams are identified by number, using the hashmark (#) before each number. This is why the hashmark cannot be used to identify a hexadecimal number.

As so often happens in computing, the ten streams are not numbered 1 to 10, but 0 to 9. Of these, stream #0 normally controls the screen, stream #8 the printer, and stream #9 the disc unit (as we have seen already). That leaves you with seven stream numbers to play with, and in this chapter we will be looking at some ways of using these spare stream numbers.

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 allow 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 11.2 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, 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

```
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. 11.2. Creating a window on stream #.

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 disc. This is the sort of thing that we were looking at in Chapter 9. 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 disc 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 disc messages, go to window #3 instead. Figure 11.3 shows this in (simulated) action. The actual full disc 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. 11.3. 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.

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 CPC664. 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 11.4. If you use *two* colours with BORDER, the border will flash between these two colours.

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

Figure 11.5 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

```
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. 11.5. Displaying the BORDER colours, with TIME used to make a delay.

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 on the machine, 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 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 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 CPC664 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. 11.4.* 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 CPC664 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,

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. 11.6. The standard settings for INK.

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 11.6 reminds you of these settings, which are useful to know.

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

```
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. 11.7. Using the PAPER and PEN commands.

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.

The PEN and PAPER commands can be used to specify the colours that are used in windows. Figure 11.8 illustrates this, and also

```
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. 11.8. PAPER and PEN used in different windows.

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. 11.9 which illustrates flashing text. The flashing is obtained by 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 11.10, for example, shows how a

```
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. 11.9. Flashing text obtained by using flashing ink!

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.

```
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. 11.10. Changing from flashing to steady display by changing INK.

Another useful feature CPC664 allows you to carry out effects like underlining, foreign accents, and so on. This, and a lot of other special effects that we shall look at later, is enabled by one of the 'non-printing' codes of the CPC664. These are the codes 1 to 31, and we have 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. 11.11, we need to alter the way that printing is done. Normally, 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

```
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. 11.11. 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.

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 11.12). 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. 11.12. 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 CPC664 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 Twelve
Starting Graphics

Any modern computer is expected to be able to produce dazzling displays of colour and other special effects. The CPC664 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 connection 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 CPC664 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 CPC664 can use in its *lowest resolution mode*, Mode 0. In its highest resolution mode, the CPC664 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 CPC664 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 12.1, can be obtained by pressing keys on the keyboard. The difference is that you have to

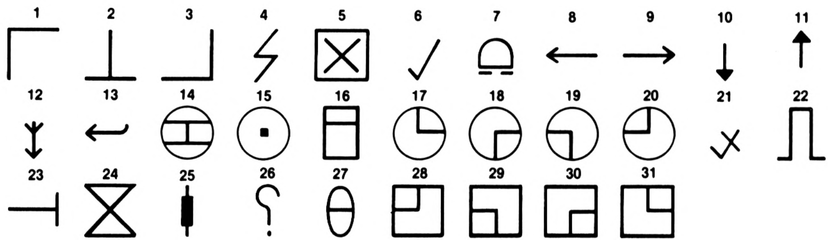


Fig. 12.1. The graphics shapes that can be obtained by pressing keys, with their ASCII codes.

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 the CPC664 manual as well, but the program in Fig. 12.2 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. 12.2. 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 12.3 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. 12.3. 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 grid for Mode 1 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 12.4 shows a design for a 'logo', an identifying mark for a firm, perhaps. It consists of the letter

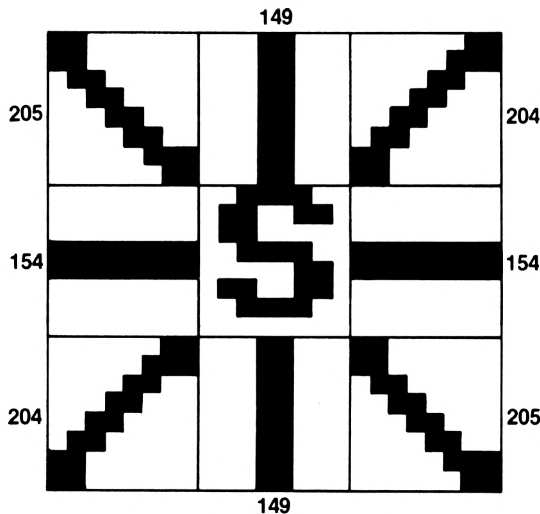


Fig. 12.4. Using the graphics shapes to design a 'logo' shape.

'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. 12.5 shows. This works, but it's clumsy programming. Now take a look at Fig. 12.6. 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. 12.5. A simple program to print the shape of Fig. 12.4.

```
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. 12.6. 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 12.7 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. 12.7. 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. 12.7 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. 12.8. 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. 12.8. 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 CPC664 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 CPC664 character, is made out of up to 64 dots that are arranged on an 8 by 8 grid. Figure 12.9 shows the

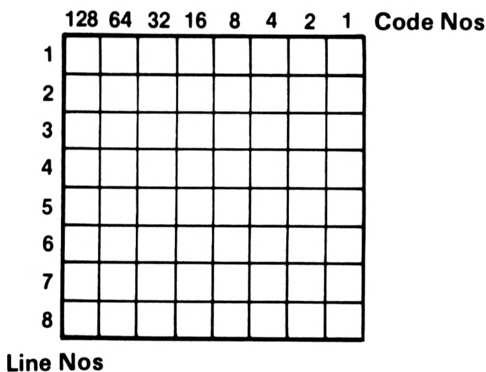


Fig. 12.9. The character grid map, for you to design your character shapes.

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 CPC664 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. 12.9. 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. 12.10. I've used a simple shape of a 'space-walker' to illustrate the principle.

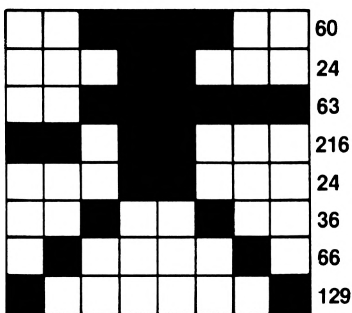


Fig. 12.10. How to use the grid map – for a 'space-walker' character plan.

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 CPC664'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$(244) is used. Figure 12.11 shows what is needed – and it's not much. The ASCII code number is 244, and it's followed by the set of eight numbers that map out the shape. We can then produce the shape wherever we like by using PRINT CHR$(244).

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

Fig. 12.11. 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 12.12 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. 12.12. 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. 12.13 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. 12.13. Redefining all keys – it makes further programming difficult! Press CTRL-SHIFT-ESC to restore normal action, but save your program before you run it.

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 12.14 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 12.15 shows a shape drawn on to three grids stacked together. Figure 12.16

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

240 241 242

Fig. 12.15. 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.12.16. 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 CPC664'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 CPC664, 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 CPC664!

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 CPC664 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 12.17 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. 12.17. 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.

You can, incidentally, switch off the cursor by using CURSOR 0,0. It is switched on again by using CURSOR 1,1. Another useful feature is COPY CHR$3 which will show what character lies under the cursor in window 3. Obviously you can use whichever window number you have allocated in this command.

Chapter Thirteen
Guide to Greater Graphics

The graphics abilities of the CPC664 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 CPC664 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. CPC664 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 CPC664, 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 13.1 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. 13.1. 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 it 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 CPC664 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 13.2 shows the CPC664 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. 13.2. 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 CPC664'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. 13.3. 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. 13.3. 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 command, 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. 13.4. 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. 13.4. 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.

You can also draw with dotted lines or dashed lines by using the MASK command with a number like 85 following it. MASK 85 gives a set of fine dots in place of a solid line, and some other MASK patterns are illustrated in Appendix F.

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. Using ON BREAK GOSUB still allows you to use the ESC key to make the program pause, so that pressing any other key will allow the program to continue. If you use ON BREAK CONT in place of ON BREAK GOSUB, then the ESC key has no effect at all.

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 13.5 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. 13.5. 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. 13.6 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. 13.6. 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. 13.7. shows a shape that has

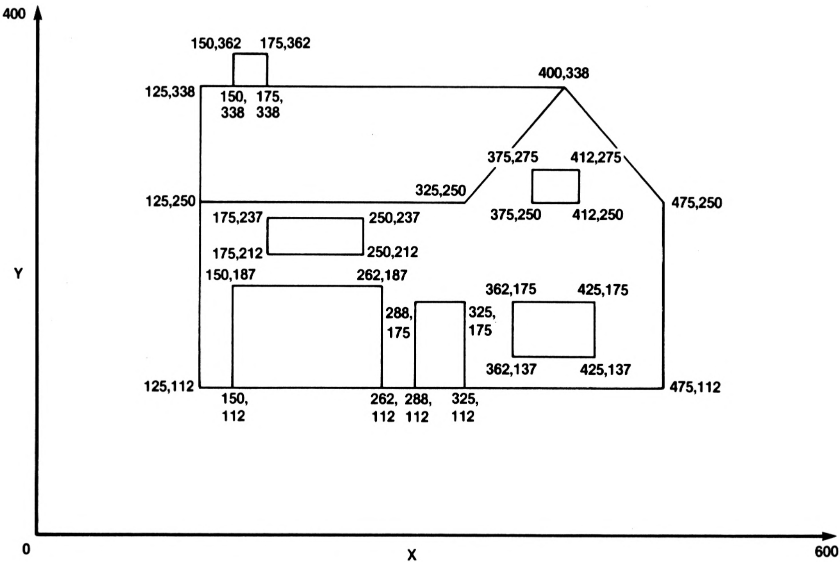


Fig. 13.7. A house shape planned by placing tracing paper over graph paper.

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. 13.8. This makes as much use as possible of DATA lines, so that if there is anything that looks as if it 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.

```
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. 13.8. The program which draws the house shape.

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 well, but for many types of drawings, we need circles, ellipses and arcs. This is a weakness of the CPC664, one of the very few, because it contains no CIRCLE command, though it can 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 13.9 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. 13.9. 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

```
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. 13.10. A much faster method of drawing a circle.

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 13.10 shows a much faster 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 13.11 gives

```
10 MODE 1
20 ORIGIN 320,200:R=100
30 GOSUB 1000
40 ORIGIN 320,200
50 FILL 2
60 GOTO 60
1000 DEG
1010 MOVER 0,R
1020 FOR N=0 TO 360 STEP 10
1030 DRAW R*SIN(N),R*COS(N),2
1040 NEXT
1050 RETURN
```

Fig. 13.11. How to fill a shape with colour.

you a taste of this, with lines 10 and 20 setting up the origin and radius, and the subroutine that starts at 1000 drawing the circle. The filling effect is achieved by the FILL 2 command. This will start filling in colour at the cursor position, and will fill until it meets a boundary of the current INK colour. You should use FILL only from a point inside a closed shape, otherwise the colour will eventually spill out all over the screen. You should also make sure that the cursor is correctly placed before you use FILL!

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 13.12 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. 13.12. 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 shape 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. 13.13 The TAG command in line

```
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 FRAME
100 NEXT
110 TAGOFF
120 END
```

Fig. 13.13. Using TAG to make graphics shapes appear at the graphics cursor position.

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 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 CPC664 can do this, and we call up this routine by using FRAME.

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 FRAME, 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 it 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 MODE 0, in which you have a lot more ink pots to play with.

Figure 13.14 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 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

```
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
120 FRAME
130 INK NR,24
140 FOR X=1 TO 30:NEXT
150 INK NR,1
160 NEXT
170 WEND
180 DATA 300,300,350,300,350,100,300,100
,300,300
190 DATA 385,288,415,252,270,113,235,148
,385,288
200 DATA 225,225,425,225,425,175,225,175
,225,225
210 DATA 230,255,275,293,410,148,377,110
,230,255
220 MODE 1:END
```

Fig. 13.14. Animation which is achieved by using 'invisible ink'!

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 FRAME 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.

All of the drawing commands which we have illustrated as using two co-ordinates and a colour number can also use a fourth number, so that you can have commands such as MOVE X,Y,2,0. The fourth number can be 0, 1, 2, or 3, and its effect is rather complicated unless you know about binary numbers. The most important extra numbers are 0 and 1. Using 0 will cause the normal colour PLOT or DRAW actions, but using 1 will have the effect of reversing the action. If you have drawn a shape, for example, using a 0 as the last number in the DRAW commands, then drawing the same shape with 1 as the last number will make the shape disappear. If you draw one shape across another using 1 as the last number, then the points where the shapes cross will disappear. Appendix G shows how the 1, 2 and 3 numbers affect drawings. Another way of achieving interesting effects is to use a 0 or a 1 following a PAPER or INK command. Using 0 gives the normal effect of paper and ink, but 1 causes the paper or ink to appear transparent. This allows you to construct drawings which will be invisible until new PAPER and INK commands are issued.

Chapter Fourteen

Sounding Out the CPC664

The ability to produce sound is an essential feature of all modern computers. The sound of the CPC664 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 CPC664 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 CPC664. 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$(7). This will give you your beep – and Fig. 14.1 shows one application of such a note. This is the start of an imaginary program which demonstrates sounds, and the bit we're interested in is the subroutine in line 1000. This uses the K$=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. 14.1. 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. 14.2 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. 14.3. 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

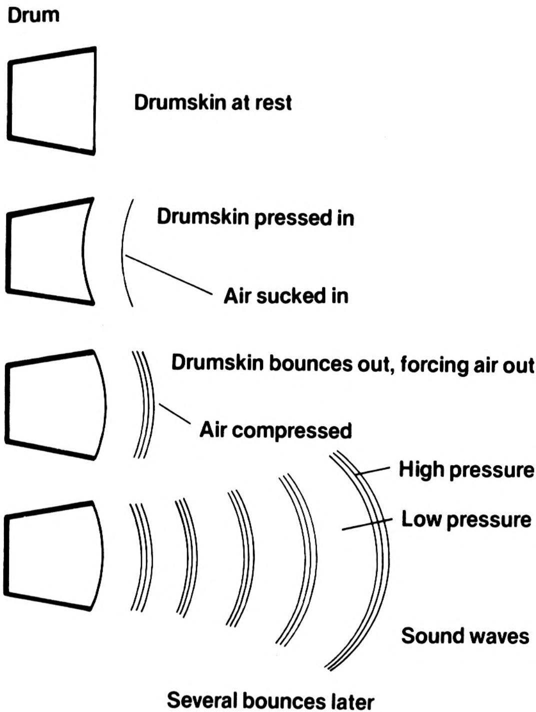


Fig. 14.2. How a drum-skin produces sound by alternately compressing and decompressing air.

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

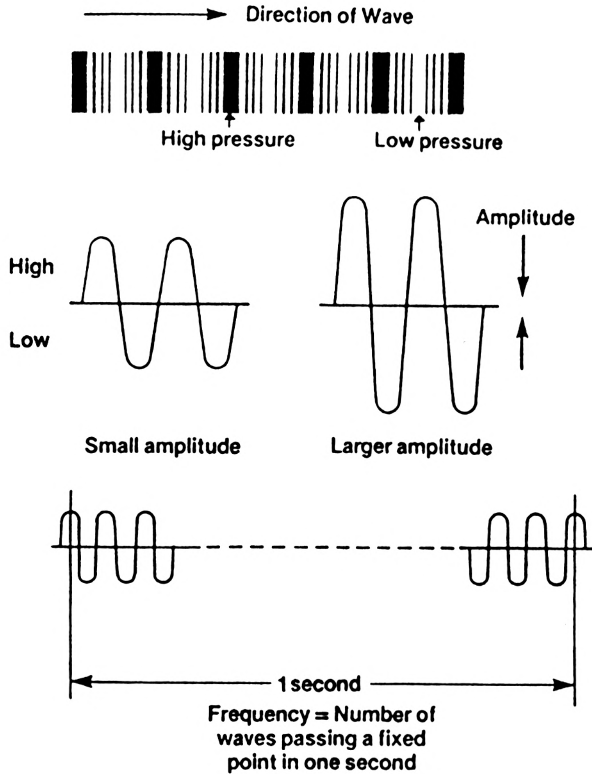


Fig. 14.3. The frequency and amplitude of a sound wave

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. 14.4) In addition, there are symbols for different durations of silence in the music, and these are illustrated in Fig. 14.5. 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. 14.4. 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. 14.5. 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 14.6 shows the staves with the notes marked.

The notes that are shown in Fig. 14.6 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. 14.7, which shows the appearance of part of the piano keyboard. The semitones are marked on the

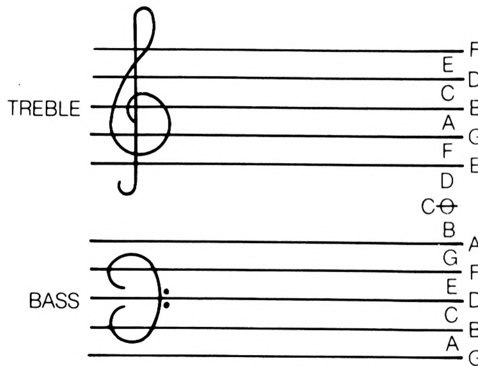


Fig. 14.6. The staves of written music, with the names of the notes written in. (Note the position of Middle C.)

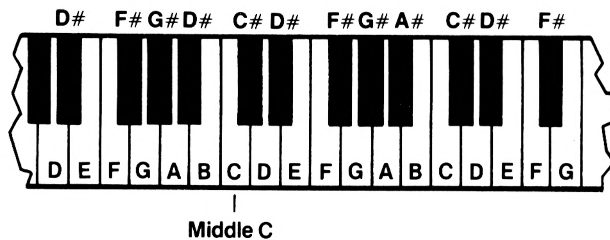


Fig. 14.7. Part of the piano keyboard, showing Middle C. There is only one semitone between B and C, and between E and F.

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 CPC664 SOUND

To work, then. The CPC664 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 CPC664 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 CPC664 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 14.8

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. 14.8. 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 CPC664 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 CPC664, 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. The most familiar set of notes is 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, the scale of C Major. The scale starts on 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. 14.9, 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. 14.9. 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 14.10 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. 14.10. 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 CPC664 manual is your guide to musical notes. Making use of this table, we arrive at the program in Fig. 14.11, 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 CPC664 pitch numbers operate

the opposite way round, so that an octave rise in pitch is achieved by halving the CPC664 number. You can see from Fig. 14.11 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. 14.11. The scale of C Major, by CPC664.

Figure 14.11 also brings home the point about sound queues to you. The CPC664 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 the 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 14.12 shows a chord being sounded with three

```
10 SOUND 1,478,300
20 SOUND 2,379,300
30 SOUND 4,239,300
```

Fig. 14.12. 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 CPC664 (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 numbers 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 14.13 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. 14.13. 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 CPC664 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 CPC664 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. 14.14 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. 14.14. 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. 14.15. 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. 14.15. 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 14.16 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. 14.16. 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 14.17 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. 14.17. Producing noise on one channel.

numbers 1 to 5. For a better appreciation of the noise channel, however, try the program in Fig. 14.18. 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. 14.18. 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. 14.19. 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 CPC664 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.

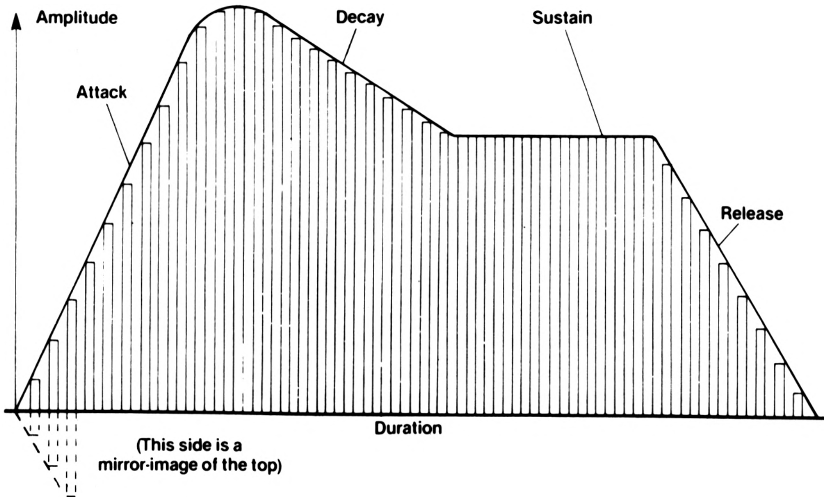


Fig. 14.19. A typical sound volume 'envelope', which shows how the volume of a note changes during the time that you hear it.

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. 14.20 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.

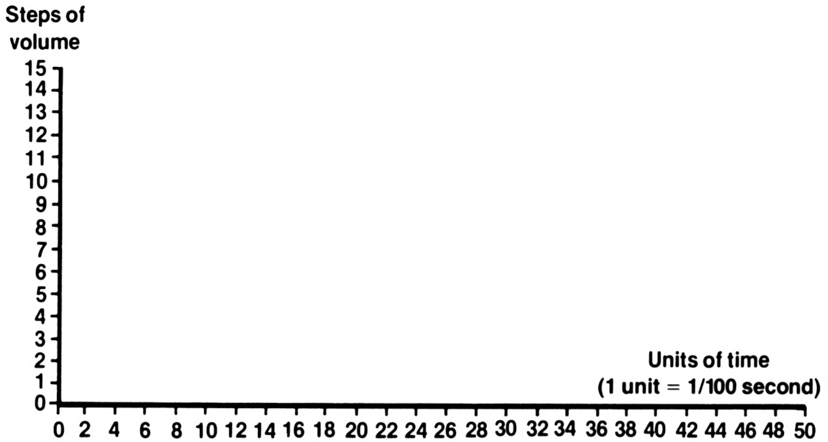


Fig. 14.20. A planning grid for the ENV command. You can draw these grids for yourself, using graph paper.

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. 14.21 shows. Now we have to see how many steps we can use. Just to keep things simple, the plan shows five steps, each

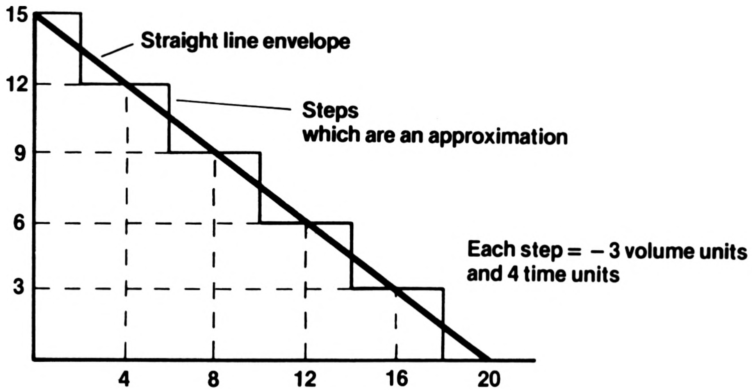


Fig. 14.21. A simple one-step envelope shape.

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. 14.22).

```
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. 14.22. 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. 14.23 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. 14.24. The ENV

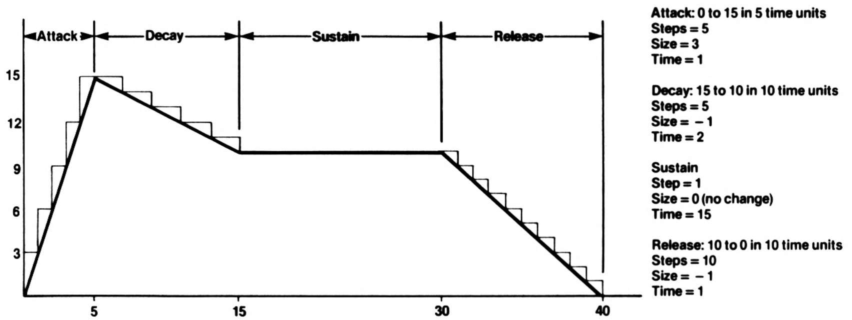


Fig. 14.23. A plan for a four-section envelope.

```
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. 14.24. 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. 14.25. Once again, the shape that is wanted is approximated to in a series of steps. These steps are tone number steps, however. A

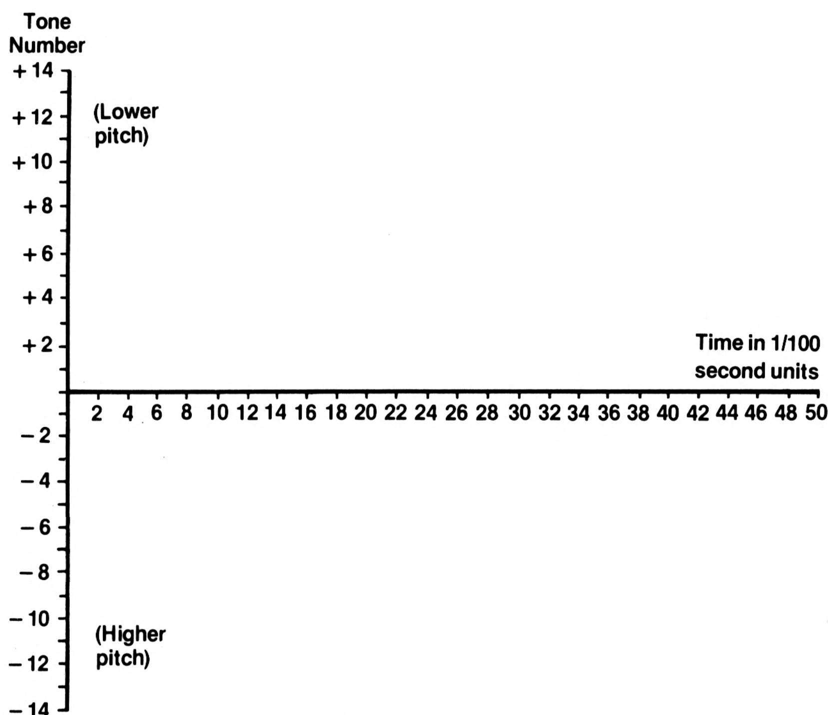


Fig. 14.25. A tone envelope planner.

positive change of tone number means a lowering pitch, a negative step means a rising pitch. As usual, an example will help, and Fig. 14.26 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 14.27 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!

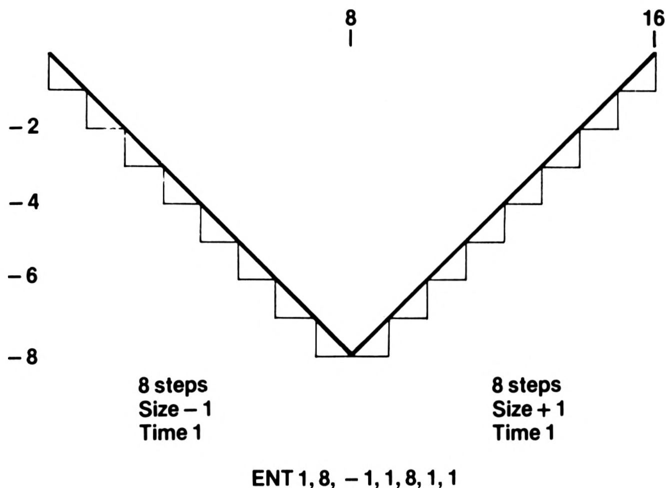


Fig. 14.26. A simple plan for a tone envelope.

```
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. 14.27. What this tone envelope sounds like.

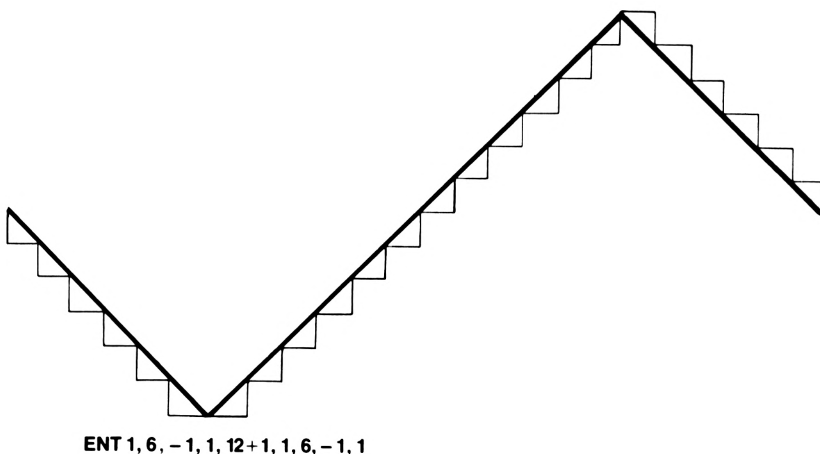


Fig. 14.28. A tremolo pattern for tone envelope.

For musical notes, a tremolo effect is obtained by making the tone number rise and fall, using a pattern like the one in Fig. 14.28. This is rather a slow tremolo, and for a lot of purposes, a faster variation would be better. In the program, Fig. 14.29, 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. 14.29. 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. 14.29; 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 CPC664 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 CPC664 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. 14.30 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. 14.30. 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 14.31 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. 14.31. 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.

Chapter Fifteen
Printers

Whenever your use of a computer extends beyond playing games that other people have written, there is one addition to your computer equipment that you will urgently want; a printer. In many cases, particularly when you are developing your own programs, the printer has an even higher priority than the use of the disc system.

The reasons for using a printer are obvious if you use the machine for business purposes. You can hardly expect your accountants or your income tax inspector to look at accounts that can be shown only on the screen. It would be a total waste of time if you kept your stock records with a computer, and then had to write down each change on a piece of paper, copying from the display on the screen. For all of these purposes, and particularly for word processing, the printer is an essential part of the computer system. Output on paper is referred to as 'hard copy', and this hard copy is essential if the computer is be of any use in business applications. For word processing uses, it's not enough just to have a printer; you need a printer with a high-quality output with characters as clear as those of a first-class electric typewriter.

Even if your computer is never used for any kind of business purpose, however, you can run up against the need for a printer. If you use, modify or write programs, the printer can pay for itself in terms of your time. Trying to trace what a program does from a listing which you can view only a few lines at a time on the screen is totally frustrating. Quite apart from anything else, if your use of BASIC on the CPC664 relies a lot on the use of GOTO for loops, you might have to list a dozen different pieces of a program just to find where one GOTO might lead you. The answer is to avoid the use of GOTO, but there are times when FOR...NEXT and WHILE... WEND loops are not completely satisfactory substitutes. The problem is even worse if you write your own programs. Even a very modest program may need a hundred lines of BASIC, some of which

may be long lines. Trying to check a program of a hundred lines when you may be able to see only a dozen or so at a time on the screen is like bailing out a leaky boat with a teaspoon. With a printer attached to your CPC664 you can print out the whole listing, and then examine it at your leisure. If you design your programs the way you ought to, using a 'core' and subroutines, then you can print each subroutine on a separate piece of paper. In this way, you can keep a note of each different subroutine, with variable names noted. On each sheet you can write what the subroutine does, what quantities are represented by the variable names, and how it is used. If you have a utility program that allows you to merge subroutines, you can then construct programs painlessly using your library of tested subroutines.

Printer types

Granted, then, that the use of a printer is a high priority for the really serious computer user, what sort of printers are available? The CPC664 uses the almost universal Centronics connection for printers, including the Amstrad printers which appear to be made by Seikosha. It's difficult to imagine any 'serious' computer without a Centronics interface, for this is the connection method that is used by all the famous-name printers which are available. This means that

---

Dot matrix
- impact
- thermal
- electrostatic

Type impact
- type stalk
- daisywheel

Plotters
- graphics printers
- X-Y plotters

Ink-jet
- single colour
- multicolour

Fig. 15.1. A list of printer mechanism types.

you can attach almost any good-quality printer that you like to the CPC664. This opens up the way for the use of any of the printers which are offered at such attractive prices in the magazines. In particular, it allows you to use printers such as the Epson and Juki range.

Printers that are used with small computers will use one of the mechanisms listed in Fig. 15.1. Of these, the impact dot matrix type is the most common. A dot matrix printer creates each character out of a set of dots, and when you look at the print closely, you can see the dot structure. The printhead of the dot matrix printer consists of a set of tiny electromagnets, each of which acts on a set of needles that are arranged in a vertical line (Fig. 15.2). By firing these needles at an

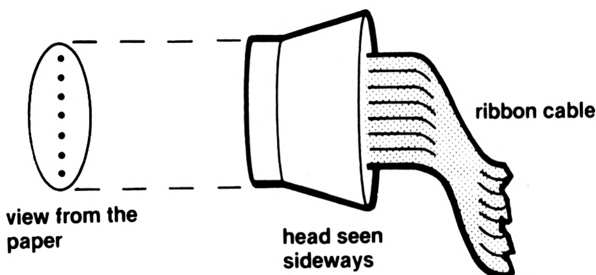


Fig. 15.2. Illustrating a dot matrix printhead.

inked ribbon which is placed between the head and the paper, dots can be marked on the paper. Each character is printed by firing some needles, moving the head slightly, then firing another set of needles, and so on until the character shape is drawn completely (Fig. 15.3). The most common pattern of dots for low-cost printers is the 7×5, meaning that the characters can be made out of up to seven dots in

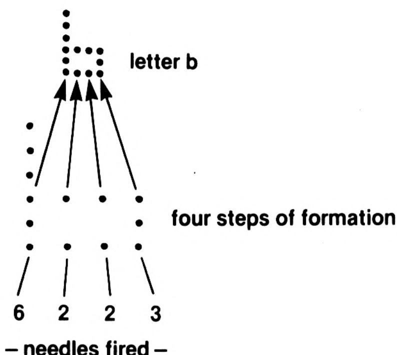


Fig. 15.3. How a 7×5 dot matrix head creates a character.

height and up to five in width. This implies that the head moves across the paper in five steps to print each character, and that up to seven needles can be fired. Using a 7×5 structure gives characters which are readable, but not good-looking. The dots are very evident, and some of the letters are misshapen. You will find, for example, that lower-case letters lack 'descenders'. This means the tails on letters y,g,p,q will either be missing, or will be on the same level as the foot of other letters. When this print is used for listings which are in upper-case only, there is no problem. You would not, however, use a printer of this class to print letters or other documents that anyone else would have to read.

Rather better results can be obtained if the number of needles in the printhead is increased. Using 9×9 (nine needles, nine steps across) or 15×9 heads can create much better-looking characters, lower-case or upper-case. Another advantage of these printheads is that the characters are not limited to the ordinary letters of the alphabet and the numbers. Foreign characters can usually be printed, and it is possible to print Arabic script, or to make up your own character set, for example. Most of the dot matrix printers are impact types. This means what it says, that the paper is marked by the impact of a needle on an inked ribbon which hits the paper. There are also thermal and electrostatic dot matrix printers. These use needles, but the needles do not move. Instead the needles are used to affect a special type of paper. In the electrostatic printer (such as the old ZX printer), the needles are used to pass sparks to the paper, removing a thin coating of metal from the black backing paper. The thermal type of printer uses hot needles to make marks on heat-sensitive paper. Both of these printers require expensive special paper, and are unsuitable for serious business purposes, so we won't spend any time on them here. If you want a cheap printer for listings, there are better methods.

The ultimate in print quality at the moment is provided by the daisywheel printer. This uses a typewriter approach, with the letters and other characters placed on stalks round a wheel. The principle is that the wheel spins to bring the letter that you want to the top, and then a small hammer hits the back of the letter, pressing it against the ribbon and on to the paper. Because this is exactly the same way that a typewriter produces text, the quality of print is very high. It's also possible now to buy a combination of typewriter and daisywheel printer. This looks like a typewriter, with a normal typewriter keyboard, but has an interface connection for a computer. You can use it as a typewriter, and then connect it to the computer and use it as

a printer. Machines of this sort are made by leading typewriter manufacturers such as Silver Reed, Brother, Triumph Adler, Smith Corona, and others. If you need a typewriter as well as a printer, then this type of machine is an obvious choice.

The third kind of mechanism that we shall look at here is the graphics printer. This is a remarkable mechanism which uses four miniature ball pens to mark the paper direct, with no ribbon. It can be used for graphics work, and when it is used as a printer, the letters are *drawn* rather than printed. Because four pens are used, the markings can be in four different colours. Printers of this type are not expensive (as printers go) and can be very useful, particularly if you want graphics output in colour. Another type of printer that is now becoming available is the ink-jet printer, which operates by shooting fine jets of ink at the paper. This one shares the disadvantage of the thermal and the electrostatic types in that you obtain only one copy. Impact printers have the great advantage that you can obtain an extra copy by using a sheet of carbon paper and another sheet of plain paper. You can also buy listing paper which has a built-in carbon, or which uses the NCR (no carbon required) principle to produce two copies.

Interfaces

The printer has to be connected by a cable to the computer, so that signals can be passed in each direction. The computer will pass to the printer the signals that make the printer produce characters on the paper, but the printer must also be able to pass signals to the computer. This is because the printer operates much more slowly than the computer. Unless the printer contains a large memory 'buffer', so that it can store all the signals from the computer and then get to work on them at its own pace, some sort of 'handshaking' is needed. This means that the printer will accept as many signals as its memory will take, and then will send out a signal to the computer which makes the computer hang up. When the printer has completed a number of characters, (one line, one thousand, or possibly just one character), it changes the 'handshake' signal, and the computer sends another batch. This continues until all of the text has been printed. This can mean that you don't have the use of the computer until the printer has finished. Printers can be very slow, particularly daisywheel and plotter types. Even the fastest dot matrix printers can make you wait for a minute or more for a listing.

Two types of interface are used by practically all printers. These are classed as serial or parallel. A parallel printer is connected to the computer by a cable which uses a large number of separate strands. Since each character in ASCII code uses seven signals, the parallel printer sends these along seven separate strands – many printers can use an eighth signal and this is usually sent as well. In addition, there are cable strands for the 'handshake' signals. The best-known, and most used variety of parallel connection is called Centronics, after the printer manufacturer which first used it. Practically all of the popular printers use this type of parallel interface.

The serial interface sends the signals out one at a time. This means that at least seven signals have to be sent for each character, and in practice the total must be ten or eleven, to allow for start and stop signals which are used to mark where the signals for each character start and stop. This system uses less cabling, because only two strands need to be used for signals, and the cables can be longer because there's no risk of one signal interfering with another. The standard system is called RS-232. Printers can be obtained with RS-232, but seldom as standard, and often only as an extra, costing up to an additional £50.

The Amstrad CPC664 uses a parallel system, so that practically all of the popular printers can be connected in addition to the Amstrad printer. The Amstrad DMP-1 printer is a simple design which produces acceptable results for listings. Its work can be seen in some of the examples that appear in the manual. For business use, however, it's unlikely that this standard of print would be acceptable. The Amstrad printer, however, can reproduce the CPC664 graphics shapes, and this might be important to you. If, however, I make the assumption that you wouldn't buy a computer with a disc drive unless you were interested in business applications of some sort, then it makes sense if I confine the descriptions here to printers with high-quality output.

In this book, for example, the listings have been reproduced on an Epson RX80 printer. This uses a 9×9 matrix for characters, so that the appearance of the characters is better. In addition, the Epson can operate in 'emphasised' mode. In this printing mode, each dot is struck twice, but the head is shifted slightly between dots. This causes the dots to look almost joined up, and makes the appearance of the print much more acceptable.

A problem that you are bound to run up against when you use any non-Amstrad printer is that of line feed and carriage return. Many computers send out only one code number, the carriage return code

(13) at the end of a line. Other machines send both the line feed (code 10) and carriage return codes. Printers are arranged, therefore, so that either possibility can be catered for by a switch. If you connect your printer and find that everything is printed on one line, then don't return the printer. Just look in the manual, and find the switch that alters the line feed setting. If on the other hand you find that each line is double-spaced, then this switch will have to be set to the opposite position. My CPC664, when connected to an Epson MX80, performed two line feeds no matter how the selector switch was set, and this had to be corrected by a rather roundabout method, noted later in this chapter.

The Epson MX80, FX80 and RX80

The Epson range of printers has for a long time been the most popular range of moderately-priced printers, offering excellent print quality at reasonable prices. The RX80 and FX80 are the latest in this line, but if you are offered a second-hand MX80, then this also is a good buy. A particular feature of the Epson range is that the printheads plug into place, and can easily be replaced when they wear out. My old Epson MX80 was just beginning to show signs of head wear after printing half-a-million words, so it might not be a problem for you!

The standard version of the RX80 uses pin-feed, but the RX80F/T can take any form or paper, including rolls. You have to pay extra for a paper roll holder, but if you are handy with wood and piano wire, this is something that you could easily make for yourself. The advantage of using the F/T version is that plain unperforated paper rolls are *very* much cheaper to buy, and it also means that you can use plain paper sheets if you want to. When you use a lot of paper for listings, this can be a great saving. Paper width of 4″ to 10″ in pin feed or plain form can be used, so you can buy whatever paper size is on offer. If you use the F/T option, you can then buy the teletype rolls, which are 8½ inches wide.

The RX80 offers a full set of upper or lower-case letters, and you don't have to go through any elaborate antics to select which one you want. Figure 15.4 shows the normal upper-case letters of the RX80, as

```
10 REM USING RX80 IN NORMAL MODE
20 REM WHICH PRINTS AT MAXIMUM SPEED
```

Fig. 15.4. The normal characters of the Epson RX80.

you would use them for a listing. The print speed is very fast, and most listings will be completed in under a minute. Figure 15.5 shows the lower-case letters, which are much better formed than those of

```
10 rem lower case on the screen
20 rem can also be produced on the printer.
```

Fig. 15.5. The lower-case characters of the Epson RX80.

```
10 REM THIS SHOWS THE EMPHASISED
20 REM STYLE OF PRINT OF THE RX80
```

Fig. 15.6. The emphasised print of the RX80.

cheaper printers. Figure 15.6 shows the 'emphasised' print of the RX80. This is achieved by typing PRINT#8,CHR$(27)CHR$(69) (press ENTER) before listing. The emphasised print can be cancelled by using PRINT#8,CHR$(27)CHR$(70).

These commands can be used in programs, so that you can print normal, condensed, emphasised, double width, and all of the other varieties, under program control. This makes it very easy to produce good headings, produce words in bold type or italics, and to underline. For a lot of word processing actions, the RX80 can be a very satisfactory low-cost alternative to a daisywheel. International

Switch 1

Position	ON	OFF
1	Condensed	Pica (print size)
2	Graphics	Control code
3	No buzzer	Buzzer on (end of paper)
4	12 inch	11 inch (form length)
5	Not detected	Detected (paper end)
6	Selects from international	
7	character set of	
8	eight languages	

Switch 2

Position	ON	OFF
1	Slashed	Non-slashed (zero)
2	Control pin	Not fixed
3	Line feed	No line feed (with C/R)
4	Skip	Don't skip (perforation)

Fig. 15.7. List of RX80 switch settings.

character sets (for the USA, France, Germany, England, Denmark, Sweden, Italy, Spain, Japan, Norway) can be printed, and are under software control. This means that selection is made by printing CHR$ numbers rather than by altering switches on the printer itself. The only switches that you have to alter are for such items as are listed in Fig. 15.7. For many purposes, you would probably never need to alter the factory settings of these switches. Figure 15.8 shows the options that can be selected by sending CHR$(27)CHR$(N) codes to the printer.

Each of the letter codes will be preceded by CHR$(27), the ESC code. Some of the CHR$(number) codes can be used alone – consult the manual for details.

Code	*Effect*
J	Adjust line spacing in 1/216 inch units.
M	Elite size characters.
P	Pica size characters.
CHR$(14)	Enlarged print.
CHR$(20)	Cancel enlarged print.
W	Second enlarged print mode.
CHR$(15)	Condensed print.
CHR$(18)	Cancel condensed print.
_	Underline on/off switch.
E	Set emphasised mode.
F	Cancel emphasised mode.
G	Double strike mode.
H	Cancel double strike mode.
S	Superscript/subscript switch.
T	Cancel superscript/subscript.
CHR$(8)	Backspace.
CHR$(4)	Alternate character set.
CHR$(5)	Cancel alternate character set.
m	Choose graphics or control characters.
0	1/8 inch line spacing.
1	7/72 inch line spacing.
2	1/6 inch line spacing.
3	Set spacing in 1/216 inch units.
A	Set line spacing in 1/72 inch units.
CHR$(9)	Horizontal tab.
CHR$(11)	Vertical tab.
e	Tab unit setting.
f	Skip position setting.

C Form length setting.
N Skip over perforation setting.
O Skip over perforation cancel.
Q Right margin set.
I Left margin set.
8 Ignore paper end detector.
9 Enable paper end detector.
< One line unidirectional printing.
@ Restore normal settings.
U Unidirectional printing.
S Half speed (quiet!) printing.

Other codes can be used to control each pin in the head so that graphics can be printed. This allows 'screen dump' programs which place a copy of the screen graphics on to the paper to be written for this printer.

Fig. 15.8. The software selections of the RX80.

The line-feed fix

When the CPC664 is used along with any of the Epson printers which I have tried, it causes double line feeds no matter what the setting of the line feed switch in the printer happens to be. This is not a problem with the Amstrad printer, nor is it a worry with the Juki or Tandy printers (described later in this chapter). Fortunately, it can be fixed with a bit of software magic. The Epson printers allow the distance between lines to be changed in units of 1/72 inch. This is done by sending the codes ESC "A", followed by the number of 1/72 inch steps. For the CPC664, a line feed of 7/72 inch is sufficient to make the double line feed into an acceptable size, so the command:

```
PRINT#8,CHR$(27);"A";CHR$(7)
```

will carry this out. It can, however, lead to odd effects when a line of a listing spills over onto another line. To avoid this, you have to ensure that the computer sends out line feeds at the correct number of characters. Suppose, for example, that your printer will line feed each 80th character. You must ensure that the CPC664 does the same by the command: WIDTH 80. If this is not done, the printer will perform its own single small line feed, making the lines too close to each other. The listings in this book were produced with WIDTH 40, and with the printer forced to work with 40-column lines by using:

```
?#8,CHR$(27)"Q";CHR$(40)
```

These codes are used by both the MX80 and the RX80 printers. This is important, because not all of the printer control codes are identical for the different Epson models. It's a good idea to keep some printer setting programs on a disc, so that you can RUN one as you choose to suit whatever printer you happen to be using. It is also possible to buy a modified printer connector which allows use of the Epson printers without the need to alter line spacings.

The Juki 6100 daisywheel

The Juki was one of the first low-cost daisywheel printers to become available. Like most printers, it comes with a Centronics parallel interface, though an RS-232 serial interface is available at extra cost. The Juki is a large and very heavy machine which can accept paper up to 13″ wide. The daisywheel is of the same type as is used on Triumph Adler printers, and the ribbon cartridge is an IBM Selectric 82/C type. The ribbon that was supplied with my Juki was of the 'single-strike' variety, and this had a very short life (about three chapters of this book!). A 'multistrike' type of ribbon is much better. With the latter, the whole width of the ribbon is used by moving the cartridge up and down as well as by moving the ribbon itself. These ribbons are very easy to obtain from many suppliers, but the best prices I have seen have been in the Inmac catalogue. The ribbons are carbon film rather than inked nylon, and are thrown away after use. This always seem a pity, because the cartridge contains mechanisms that look as if it could easily be used again. Some day, I'll try reloading one of these cartridges.

The printhead of the Juki will print in either direction, and there is a 2K buffer. This means that short pieces of text can be transferred to the printer buffer almost instantly, and the computer can be used for other purposes while the printer gets on with the printing actions. Printing is much slower than the normal rate of the Epson, but not so much slower than the emphasised mode of the Epson as to make the daisywheel seem irritatingly slow. Its enormous advantage is the quality of the type. This is exceptionally clear on the top copy, and even three carbons later it is still very legible. For any letter work, or for the manuscript of a book, the Juki is ideal.

As you would expect of any modern design of printer, the Juki permits many character sets, but you need to have the appropriate daisywheels fitted for each language. You cannot, for example, have words in alternate character sets without changing wheels in between.

Changing wheels is particularly simple, but this is something that you don't have to worry about with dot matrix printers, because the same dot matrix head can produce any character under software control. The Juki allows underlining, bold type, and shadow type in addition to the normal printing style, and you can select your print style from a range of at least fourteen daisywheels. The daisy wheels are expensive in comparison with others on the market, but ribbons are cheap. Figure 15.9 shows a printout from the Juki with the standard Courier daisywheel fitted. By removing the top cover, you can gain access to a set of miniature switches. Switch No. 1 controls auto line feed, and for use with the CPC664 this must be set to the OFF position. This will give correct line-spacing – the ON position causes each line to be double-spaced. The switch-change must be done with the machine switched off. This is not so much because of risk but because these switch settings have *no effect* until the machine is switched off and then on again.

```
10 REM DEMONSTRATION OF JUKI
20 PRINT#8,"This is JUKI normal print"
30 PRINT#8,CHR$(27);"E";"This is underli
ned";CHR$(27);"R"
40 PRINT#8,"We can change";CHR$(27);"O";
" to bold print."
50 PRINT#8,"We can change ";CHR$(27);"W"
;"to shadow print."
60 REM The C/R clears these effects
70 PRINT#8,CHR$(27);"Y";CHR$(27);"Z";CHR
$(27)"H";CHR$(27);"I";CHR$(27);"J";CHR$(
27);"K"
```

```
This is JUKI normal print
This is underlined
We can change to bold print.
We can change to shadow print.
```

Fig. 15.9. The printing of the Juki daisywheel, using the Courier 10 daisywheel.

Like the Epson, the Juki permits a number of changes to be made simply by sending control codes to the printer. These use the ESC character, CHR$(27) followed by one more character, so that whatever immediately follows CHR$(27) is never printed. The options include graphics mode, left and right margins, lines per page, half-line feeds in either direction (for printing subscripts and

superscripts), top and bottom page margins, and some special characters, including the English pound sign. Even more usefully, the print can be changed to bold or shadow by sending such codes, and text can be underlined. Figure 15.10 lists these actions.

Each of these codes will be preceded by CHR$(27).

Code	*Effect*
1	Set horizontal tab (HT) at present position.
2	Clear all tabs.
3	Graphics mode on (C/R clears).
4	Graphics mode off.
5	Forward print on (C/R clears).
6	Backward print on (C/R clears).
7	Print suppress on (C/R clears).
8	Clear present HT stop.
9	Set left margin at present position.
0	Set right margin at present position.
CHR$(9)	Set HT (tab number follows).
CHR$(10)	Set lines per page (number follows).
CHR$(11)	Vertical tab (VT) set (number follows).
CHR$(12)	Set lines per page (number follows).
_	Sets VT at present position.
CHR$(13)P	Remote reset.
CHR$(30)	Sets line spacing (number follows).
CHR$(31)	Sets character spacing.
C	Clears top/bottom margins.
D	Reverse half-line feed.
U	Normal half-line feed.
L	Sets bottom margin at present position.
T	Sets top margin at present position.
Y	Special character.
Z	Special character.
H	Special character (new paragraph symbol).
I	English pound sign.
J	Diaeresis mark.
K	Spanish c with cedilla.
/	Automatic backward print.
\	Disable backward print.
S	Set character spacing.
CHR$(26)A	Remote error reset.
CHR$(26)I	Initialise printer.
CHR$(26)1	Status (serial interface only).
P	Proportional spacing on.

Q	Proportional spacing off.
CHR$(17)	Offset selection.
E	Underline on.
R	Underline off.
O	Bold print on (C/R clears).
W	Shadow print on (C/R clears).
&	Bold or shadow print off.
%	Carriage settling time.
N	Clear carriage settling time.
CHR$(8)	1/120 inch back space.
X	Cancels all word processing modes except proportional spacing.

Fig. 15.10. The software selections of the Juki.

The same quality of print can now be obtained from a large number of daisywheel typewriters, and many of these now have a Centronics parallel interface. This type of machine offers a lot of advantages, because it can be used as a typewriter for small items that do not justify the use of the computer, yet it is available for word processing use along with the CPC664 and such programs as AMSWORD. These machines can now be bought in the high street stores as well as from office supply shops. The only thing to watch is that replacement ribbons and daisywheels are obtainable from several different sources. There's nothing worse than being stuck with a machine for which you can obtain spares from only one supplier.

The CGP-115 4-colour graphics printer

One of the most popular small graphics printer mechanisms is made under the trademark of ALPS. It's Japanese, and in place of the mechanisms that are used by most printers, it actually *draws* its characters with a set of four miniature ball pens. The reason for the set of four is that this allows printing in four different colours – black, blue, red and green. The mechanism is made into boxed units by many manufacturers, and sold under a wide variety of names, but it is most easily obtained from Tandy stores under the Tandy code number of CGP-115. This version includes both a Centronics and a serial interface, which makes the printer usable on practically any microcomputer which uses reasonably standard interfaces. Since the Tandy stores offer a good service on spares (pens, paper, etc.) and trouble-shooting, it makes sense to buy the Tandy version as there is a

Tandy store in most large towns. In addition to being used as a printer, however, this machine acts as a graphics plotter, and you can draw diagrams and other pictures by means of instructions sent from *any* computer. This applies even if the computer has no graphics capabilities of its own.

The CGP-115 in detail

The printer uses a plain paper roll which is 4.5 inches wide. Tandy stores sell 3 rolls, each about 145 to 150 feet long, for just under £5. These paper rolls are also used by a wide variety of adding machines, so if you haunt your local office supply stores, you may find alternative sources at lower prices. The paper is tightly gripped by the printer, because it is moved around a lot in the course of printing. The printing carriage consists of a holder which is loaded with four miniature ball pens. This holder can be rotated so that one pen is touching the paper. Printing is achieved by moving the pen holder from side to side, and the paper up and down, and is such a fascinating sight that you'll probably print listings over and over again just for the pleasure of watching the mechanism! I know that I did. When the printer is switched on, it goes into a 'pen-test' routine, slowly drawing a square in each colour so that you can check that none of the pens has run dry. They have a surprisingly long life, and each pack of 3 pens costs around £1.99 from Tandy stores. You won't find alternative supplies quite so easily in this case!

Normally, the CGP-115 acts as a printer, and you can use it to print listings. It is not by any stretch of the imagination a fast printer, even compared with a daisywheel but the results are much easier to read than some dot matrix output. The enormous advantage of using the Tandy printer, however, is that it can be used as a graphics plotter. This means that if you send suitable instructions to the printer, it will draw diagrams. The instructions are not the same as the graphics instructions of the CPC664 (or any other computer), but this is not a disadvantage. If at some stage you change to another computer, the Tandy printer will still be useful, and the graphics programs that you have used with the CPC664 can easily be adapted to another computer. This is very useful to know if your household is on the verge of becoming a two-computer family. The CGP-115 has a small set of four switches at the back which can be used for setting up the printer. For the CPC664, the settings of the switches are:

1.OFF 2.ON 3.OFF 4.ON. This gives the correct line feed and the normal size of print with the parallel interface in use.

The Tandy CGP-115 commands

Because this book is mainly concerned with the use of the Amstrad disc drive and several different printers, I have had to resist the temptation to add several chapters on the Tandy graphics printer. Many CPC664 owners, however, will probably want to make use of this type of printer mechanism, which is sold under a variety of other brand names. For business applications, for example, the ability of the CGP-115 to produce graphs in four colours is extremely useful for such a modestly-priced unit. The following is a list of the commands which are available when the Tandy version is used. The commands are shown in their CPC664 form. Figure 15.11 demonstrates

Ian R. Sinclair
Ian R. Sinclair
Ian R. Sinclair
Ian R. Sinclair

```
10 REM DIRECTIONS
20 PRINT#8,CHR$(18)
30 PRINT#8,"M50,0"
40 INPUT"Your name, please ";NM$
50 PRINT#8,"P";NM$
60 PRINT#8,"Q1"
70 PRINT#8,"P";NM$
80 PRINT#8,"Q2"
90 PRINT#8,"P";NM$
100 PRINT#8,"Q3"
110 PRINT#8,"P";NM$
120 PRINT#8,"Q0"
130 PRINT#8,"A"
140 END
```

Fig. 15.11. A printout from the Tandy CGP-115 graphics printer.

the use of these commands in printing a name in four different directions.

PRINT#8,CHR$(8)	Move one space left (backspace). Used in text mode.
PRINT#8,CHR$(11)	Reverse line feed – move paper down by one line in text mode.
PRINT#8,CHR$(17)	Select text mode from graphics mode.
PRINT#8,CHR$(18)	Select graphics mode from text mode.
PRINT#8,CHR$(29)	Change colour in text mode.

Graphics commands

The following letters can be sent when the printer is in *graphics* mode. The letters are *not* printed; instead, they are used as commands. Several of these commands must be followed by numbers, such as X, Y coordinate numbers, to specify positions. All of these letters would be sent to the printer by using PRINT#8, after executing PRINT#8,CHR$(18).

A	Reset pen to left margin, no line drawn, return to text mode.
Cn	Change colour of pen. n is colour number, 0 to 8.
Dx,y	Draw from present position to point x,y. Can be extended to more than one point.
H	Move pen to origin without drawing a line. The origin is a specified starting point.
I	Set new origin at current pen position. If you want a new origin at point 5,10, then place the pen there, and PRINT#8, "I".
Jx,y	Jump, or draw-relative. Draws a line from present position to one x steps to the right and y steps up. Do not confuse this with D, which draws to the *absolute* point x,y.
Ln	Change line type. If n=0, the line is solid, but using numbers 1 to 15 will draw various dotted lines.
Mx,y	Move to point x,y without drawing a line.
Pchars	Print the following characters while the printer is in graphics mode. The size of the characters can be controlled, and characters can be printed vertically or backwards.

Qdir	Change print direction. The number dir, can be in the range 0 to #8. 0 gives normal printing, 1 gives top to bottom, 2 gives upside down, 8 gives bottom to top.
Rx,y	Relative move. Move pen, without drawing, to a point x steps to the right and y steps up. Using −x moves left, using −y moves down.
Sn	Selects size of characters to be printed. n must be between 0 and #8.
Xa,b,c	Draw graph axis. n is 0 for a Y axis, 1 for X axis. The distance between marks on the axis is specified by b, which must be between −999 and +999. The number of steps is c, between 1 and 255.

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 CPC664 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 CPC664 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 CPC664 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 CPC664 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 CPC664 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

The ASCII Codes in Hex

No.	Hex.	Char.	No.	Hex.	Char.
32	20		80	50	P
33	21	!	81	51	Q
34	22	"	82	52	R
35	23	#	83	53	S
36	24	$	84	54	T
37	25	%	85	55	U
38	26	&	86	56	V
39	27	'	87	57	W
40	28	(	88	58	X
41	29	)	89	59	Y
42	2A	*	90	5A	Z
43	2B	+	91	5B	[
44	2C	,	92	5C	\
45	2D	-	93	5D	]
46	2E	.	94	5E	^
47	2F	/	95	5F	_
48	30	0	96	60	`
49	31	1	97	61	a
50	32	2	98	62	b
51	33	3	99	63	c
52	34	4	100	64	d
53	35	5	101	65	e
54	36	6	102	66	f
55	37	7	103	67	g
56	38	8	104	68	h
57	39	9	105	69	i
58	3A	:	106	6A	j
59	3B	;	107	6B	k
60	3C	<	108	6C	l
61	3D	=	109	6D	m
62	3E	>	110	6E	n

63	3F	?	111	6F	o
64	40	@	112	70	p
65	41	A	113	71	q
66	42	B	114	72	r
67	43	C	115	73	s
68	44	D	116	74	t
69	45	E	117	75	u
70	46	F	118	76	v
71	47	G	119	77	w
72	48	H	120	78	x
73	49	I	121	79	y
74	4A	J	122	7A	z
75	4B	K	123	7B	{
76	4C	L	124	7C	¦
77	4D	M	125	7D	}
78	4E	N	126	7E	~
79	4F	O	127	7F	

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 when 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 CPC664 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. Your CPC664 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 CPC664 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 CPC664 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. Errors 31 and 32 refer to the disc system, and you can use DERR (in the same way as ERR) to obtain more detailed information.

```
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!).

Appendix E

Use of the CTRL Key in CP/M

The CTRL key is used in CP/M along with letter keys to allow a number of useful commands to be entered from the keyboard while a program is running. The actions are listed here. To avoid having to repeat the CTRL key symbol, only the letters are shown, so that **C** means CTRL C. Since the CTRL key prints an up-arrow on the screen, this symbol is sometimes shown in place of CTRL. Another CTRL symbol is the circumflex (^) which many printers use in place of the up-arrow.

C	Stops a program running.
E	Carriage return to screen but not to program.
H	Delete character and backspace.
I	Tab across eight spaces.
J	Generate line feed, end input.
M	Generate carriage return, end input.
P	Switch printer on or off.
R	Erase and retype command.
S	Screen display on or off.
U	Ignore command, move cursor down.
X	Erase command, home cursor.
Z	End input.

Note: The **P** and **S** commands are 'toggles'. This means that you use the command once to switch on the effect, and again to switch it off. You cannot normally tell, except by trying it, what state the command is in, but when you enter CP/M, the toggle commands are set to printer off, screen on.

Appendix F
Dotted Lines

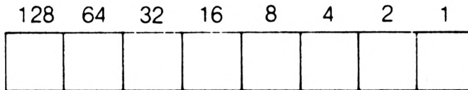


Fig. F.1.

The MASK instruction will cause a pattern of dots and dashes to appear in any line. The number which controls the pattern can be obtained by using the pattern key which is shown in Fig. F.1. Each piece of a line is shown as divided into eight sections. You decide what your pattern shape will be by filling in some of these sections, as in Fig. F.2.

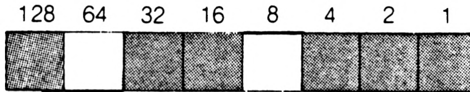


Fig. F.2.

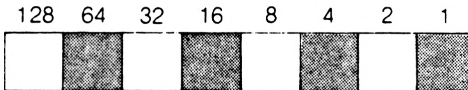


Fig. F.3.

You then add up the numbers above the shaded boxes, and use this as the MASK number. For example, Fig. F.3 gives $64+16+4+1=85$, so that MASK 85 would produce this dotted line. The effect of some numbers is shown in Fig. F.4.

PATTERN NUMBER

85
identical on screen
17Ø

2Ø4
identical on screen
51

24Ø

248

252

254

246

145

228

Fig. F.4.

Appendix G
XOR, AND, OR

Colour number	Binary form
0	00000
1	00001
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000
9	01001
10	01010
11	01011
12	01100
13	01101
14	01110
15	01111
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11001
26	11010

Fig. G.1.

The fourth number in a drawing command has the effect of specifying the *type* of drawing action. The names of these actions (strictly speaking, they are *logic* actions) are XOR, AND and OR, and they are specified by the numbers 1, 2, and 3 respectively. The first thing you need to know about these is that the commands work on the colour numbers. To see what they do, however, you have to write the colour numbers in a different form, called binary. Fig. G.1 shows all 27 colour numbers written in this form, which consists of 1s and 0s. Each colour number is written as a set of five digits.

Now when we use AND, we compare each digit in one number with the digit in the same place of another number. If both digits are 1, then the result is 1, otherwise the result is 0. Figure G.2 shows how this, along with the OR and XOR rules, applies. On the screen we will be comparing the foreground colour of the INK with the colour

---

AND

The result is 1 if two 1s are present, so that 1 AND 1 is 1, 1 AND 0 is 0, 0 AND 0 is 0. For numbers with several digits, each digit in one number is compared with the corresponding digit in the other number.

For example, if we AND 1011 with 1001:

```
1 0 1 1
1 0 0 1

1 0 0 1
```

then the result is 1001.

OR

If either or both digits is a 1, then the result will be a 1. 1 OR 0 is 1, only 0 OR 0 is 0. For example, if we OR 1011 with 1100, we get:

```
1 0 1 1
1 1 0 0

1 1 1 1
```

and the result is 1111.

XOR

If one digit is 1, then the result is 1, but if both digits are 1 (or 0) then the result is 0. For example, if we XOR 1011 with 1100, we get:

```
1 0 1 1
1 1 0 0

0 1 1 1
```

and the result is 0111.

Fig. G.2. A summary of the effects of AND, OR and XOR.

which was there previously. When this is done with AND or OR, the results will usually create a different colour. When the OR command is used, the rule is that if one or both of the binary numbers has a 1, then the result will have a 1 in the same position. The XOR rule is that the result is 1 only if there is a 1 in one of the numbers, but not in both in the same position. If you have not come across these 'logic rules' before, you may very well find this baffling.

Index

sign, 110
% sign, 54
* character, 108
3-inch disc, 14
= sign, 42
?, 27
@ command, 107
\ sign, 55
CPM command, 17

adaptor, 3
adding to core, 99
aerial socket, 1
after, 198
air pressure, 210
allocating buffers, 158
alphabetical order, 83
ALPS mechanism, 247
amplitude, 211
AMSDOS, 16
Amstrad DMP-1, 239
AND, 65, 269
animating snake, 190
animation, 184
animation with INK, 206
array, 84
ASC, 80
ASCII code, 72, 258
assignment, 40
assigns sign (=), 42
asterisk, 108
attack, 224
attribute characters, 125
AUTO, 39
automatic resumption, 263

backing up, 111
backslash, 29
backup, 6, 20
BAK label, 24
BAS label, 25
BASIC, 16, 26
bass stave, 213
beats, 211
beep, 209
binary fraction, 53
black keys of piano, 214
blank string, 69
blocks, 14
bold print, Juki, 245
BORDER, 169
brackets, 80
brightness, 3
buffer, printer, 238, 244
buffers, 137
bug, 254
built-in commands, 119
business programs, 23
byte, 14, 164

CAPS LOCK, 10
carbon copies, 238
care of discs, 20
carriage return, 239
case in names, 41
casing of disc, 12
CAT command, 23
catalogue, 22
centring routine, 74
centring title, 36
Centronics, 239
Centronics interface, 235
CHAIN, 25
changing a record, 144
changing columns, 34
changing line number, 254
changing print wheels, 245
channel select, 215
channel select numbers, 232
character codes, 180

character grid map, 185
character sets, Juki, 244
choices, 89
chord, 218
CHR$, 81
CHR$(7), 209
circles, 201
CLEAR INPUT, 71
clear variable, 101
clock chime, 228
CLOSEOUT, 137, 139
closing files, 139
CLS, 26
co-ordinate system, 192
colon, 33
colour blindness, 173
colour contrast, 173
colour instructions, 168
colour monitor, 1
colour numbers, 169
colouring screen, 172
colours, 4
column number, 37
columns, 33
combining envelopes, 231
combining tests, 65
comma use, 33
command word, 27
comparing strings, 83
comparison signs, 65
computer stand, 65
concatenation, 44
concealing entry, 165
concealing letters, 81
console, 164
copy editing, 253
copy file, 114
COPY key, 11
copy whole disc, 113
copyright notice, 9
core program, 98
corruption of filenames, 158
countdown, 61
countdown variable, 59
counting, 50
counting characters, 74
countup, 60
CP/M 2.2, 16
CP/M system, 117
crashing through, 93
creating characters,h 185
crochet, 212
CTRL, C, 18, 114
CTRL key, 10
CTRL key in CP/M, 226
CTRL SHIFT ESC, 24
cube root, 52
cursor, 9
CURSOR, 191
cursor keys, 11

daisywheel printer, 237
damage, 9
damper, 219
DATA, 49
data channel, 136
data collection, 133
data filename, 136
data validator, 68
database, 95
DDT, 131
decay, 224
decrementing, 50
dedicated memory, 180
DEF FN, 56
defined functions, 55
DEFINT, 57
DEFREAL, 57
DEFSTR, 57
DEG, 201
DEL key, 11
DELETE, 252
deleting files, 107
demonstration program, 23
DERR, 263
descenders, 237
designing subroutines, 100
destination, 114
details of database, 158
detect key, 70
detecting cursor, 202
device, 164
device names, 126
DEVPAC, 111
DFS, 23
DIM, 85
DIR, 120
DIR command, 24
direct command, 27
direct mode, 26
directory, 18
directory size, 19
disc, 12
disc fault, 15
disc files, 134
disc sides, 24
disc system, 5
DISCCHK, 130

DISCCOPY, 113
dot matrix, 236
dotted lines, 267
DRAW, 196
drawing square, 196
DRAWR, 197
drive letter, 119
dummy value, 67
DUMP, 130

ED, 132
editing, 11, 252
electrostatic printer, 237
ELSE, 66
emphasised print, 241
end of file code, 139
end of program option, 91
ENTER key, 11
entering data, 156
entering several values, 48
ENV, 255
envelope, 219, 224
EOF code, 139
epilepsy and flashing, 171
Epson printers, 236, 240
EPSON RX80, 239
ERA, 107
ERA with CP/M, 120
ERR, 263
error messages, 254
error messages of DISCCOPY, 115
error messages of FILECOPY, 116
error trapping, 262
ESC key, 10, 255
EVERY, 190
expanding outline, 97
expression, 51
extension codes, 118
extension name, 118
extension to MOVE, 208
extensions, 117
extension cable, 3
extracting initials, 77
extras, 4

field, 134
file, 133
file creation, 160
file editing program, 145
file replacement, 141
file security, 141
FILECOPY, 114
filename, 18
filename extensions, 117
filename length, 117
Filing Cabinet program, 148
FILL, 202
fingerprint on disc, 12
flashing asterisk, 94
flashing border, 169
flashing ink, 174
flat, 214
FOR, 59
foreign accent marks, 175
formatting, 15, 17
formatting string, 264
foundation, 98
FRAME, 206
framing a title, 45
FRE, 177
frequency of sound, 210
fuse, 2

GOSUB, 91
GOTO, 58
graphics, 179
graphics commands, printer, 250
graphics cursor, 204
graphics map, 193
graphics plotting, 248
graphics printer, 238, 247
graphs in colour, 249
green-screen monitor, 1

hammer noise, 223
handshaking 238
hard copy, 31, 234
harmony, 219
hash sign, (#) 110
head of disc drive, 13
headings file, 154
heads-or-tails, 66
hertz, 210
hex digit, 109
hex scale, 110
hexadecimal, 109
high resolution, 179, 192
high-quality output, 234
HIMEM, 199
house shape, 199
hub, 12

IBM ribbon, 244
IF test, 64
incrementing, 50
INK, 171
INK animation, 206

ink-jet printer, 238
INKEY, 70
INKEY$, 69
INPUT, 45
INSTR, 88
instruction words, 12
integer variable, 54
interface, 238
inverted commas, 30
invisible drawing, 206
invisible space, 76

jacket of disc, 13
joining files, 127
joining printing, 32
Juki printer, 236
Juki control codes, 246
Juki daisywheel, 244

key, 154
key repeat, 177
keyboard, 10
keyboard buffer, 146
keyboard graphics, 180

LEFT$, 76
LEN, 74
letter pyramid, 79
line editing, 252
line feed, 239
line feed fix, 243
LINE INPUT, 48
line number number, 20, 28
list, 49
listing, 21
listing file, 162
LOAD command, 22
LOCATE, 36
logic actions, 27
LOGO program, 23
logo shape, 182
long variable names, 42
loop, 58
loudness, 211
loudspeaker, 11, 209
low resolution, 179
LOWER$, 17
lower-case, 10

machine code, 95
main name, 18
mains plug, 1
manual, 1
map, multi-character, 189
map for LOCATE, 38
margin on screen, 35
marker, 15
MASK, 197, 267
master disc, 14, 17
matching variable type, 49
matrix, 86
menu, 89
MERGE, 25
metronome, 211
MID$, 78
Middle C, 213
minim, 212
MOD, 55
MODE 0 display, 104
MODE, 38
monitor, 2
monitor adjustments, 9
MOVE, 196
mugtrap, 68
multi-character map, 189
multiply sign, 29
multistatement line, 32
musical scale, 217

name and number, 86
name of variable, 41
NCR paper, 238
needles, 236
negative duration number, 227
nested loops, 60
NEW, 3
NEXT, 59
noise, 222
noise period, 222
non-printing codes, 175
note symbols, 213
number function, 51
number of tracks, 14
numbers, 50

octave, 213
ON BREAK GOSUB, 197, 256
ON ERROR GOTO, 159, 262
ON K GOSUB, 93
ON K GOTO, 90
on/off switch, 5
opening file, 137
opening to read, 140
OPENOUT, 137
OR, 65, 269
order of priority, 53
organ music, 220

origin, 193
ORIGIN, 195
outline plan, 96
overflow message, 62

painting circle, 202
PAPER, 172
parallel printer, 239
pass through loop, 59
patterns, 179
pause time, 226
PEN, 172
pen-test, 248
pens, 238
piano keyboard, 214
piano music, 212
pick random number, 101
pin-feed, 240
PIP, 126
PIP extensions, 129
pitch envelope, 224
pitch number, 215
pitch sound, 210
pixel size, 195
pixels, 192
plain paper roll, 240
planning programs, 95
planning character shape, 185
planning envelope, 225
planning graphics, 199
planning grid for ENV, 226
PLOT, 193
PLOTR, 194
plug-in printheads, 240
pointer, 49
POS, 191
positive integer, 28
power supply, 1
precision, 30
precision of number, 53
PRINT, 27
PRINT USING, 264
PRINT#8, 31
printer, 31, 234
printer connector, 244
printhead, 236
printing to file, 138
priority of actions, 52
program design, 94
program mode, 26
programmable keys, 260
programming language, 26
prompt, 17
protective shutter, 12

quaver, 212
quotes, 30

random access file, 135
random lines, 198
RANDOMIZE, 203
READ, 49
read only file, 25
reading file, 140
ready, 12
reboot, 123
record, 134
redefine key, 260
redefine key code, 187
redefining all keys, 188
redo from start, 47
relative drawing, 197
relative volume, 219
release, 224
REM, 98
REN, 108
REN with CP/M, 121
renaming file, 108
RENUM, 39
repeat rate, 177
repetition, 58
reserved words, 12, 41
reset machine, 188
resolution, 179
rest between notes, 217
RESTORE, 82
RETURN, 91
RIGHT$, 77
rising pitch note, 221
RND, 203
rounding, 53
RS-232, 239
RUN, 30
RUN command, 23
RX80 software selections, 242
RX80 switch settings, 241

SAVE, 122
SAVE command, 22
scale of C Major, 216
screen border, 169
screen editing, 252
screen line, 12
screen line number, 36
screen stream, 165
scroll action, 37
scrolling, 32
sectors, 14

selecting data lists, 82
Selmor, 5
semibreve, 212
semicolon, 32
semicolon with TAG, 205
semiquaver, 212
semitones, 213
sequential file, 135
serial file, 135
serial printer, 239
set-up file, 154
shadow print, Juki, 245
sharp, 214
SHIFT, 10
silence symbols, 213
silences, 212
simple animation, 184
single key reply, 69
six-pin plug, 1
skeleton database, 148
skin of drum, 210
slashed zero, 29
slicing, 76
sliding peg, 13
smooth movement, 204
snake shape, 189
socket strip, 4
soft keys, 260
soft-sectoring, 15
software settings, RX80, 242
software settings, Juki, 246
Sony disc, 14
sound, 209
SOUND, 215
sound effects, 221
sound queue, 216
sound wave, 210
source, 114
SPACE$, 177
space-walker plan, 186
spacing text, 36
SPC use, 36
special keys, 10
SPEED INK, 170
SPEED KEY, 177
sprite, 191
SQ, 217
SQR, 52
square number, 51
standard INK settings, 171
standard form, 54
STAT, 123
stave, 212
STEP, 61
stereo sound, 209, 215
storage on disc, 19
STR$, 75
stream, 164
stream numbering, 165
string, 30
string form of number, 43
string function, 72
string matrix, 87
string names, 41
string slicing, 76
string variable, 42, 72
STRING$, 45
structured program, 96
subroutines, 91
subscript numbers, 85
subscript out of range, 85
subscripted variable, 84
sudden shot noise, 227
surf noises, 223
sustain, 224
switch settings, RX80, 241
switching cursor, 191
switching on, 4
SYMBOL, 187
synchronisation, 206, 231
syntax error, 12
system disc, 1, 14

TAB, 34
TAG, 204
Tandy CGP-115, 247
terminator, 63
TEST, 204
testing recording, 21
TESTR, 204
text screen, 180
thermal printers, 237
three graphs, 194
three-part harmony, 220
TIME, 170
time delay, 61
timer, 190
title lines, 104
toggle commands, 266
tone envelope planner, 229
totalling numbers, 62
tracing program, 234
track, 14
tractor feed, 240
transient commands, 123
transient commands list, 124
transient program area, 122

transposing instruments, 220
treble stave, 213
tremolo pattern, 230
TROFF, 257
TRON, 257
tuning TV, 7
TV receiver, 1
twin buffer use, 142
twin drives, 112
TYPE, 122
types of printers, 235
typewriter, 10

unaccpetable entry, 68
uncontrolled loop, 59
underlining, Juki, 245
underlining, 175
unexpected RETURN, 93
updating file, 142
UPPER$, 176
upper-case, 10
use of ENTER, 27
use of brackets, 80
use of printer, 234
USER, 121
user number, 119
using CHR$ shapes, 181
using CP/M, 113
using integers, 55
using strings, 136
utility programs, 111

VAL, 75
variable name, 40
volume control, 209
volume envelope, 224
VPOS, 191

wait till ready, 69
warbling note, 221
WEND, 67
WHILE, 67
wildcard character, 108
window, 165
window colours, 174
WINDOW command, 166
WINDOW SWAP, 167
wipe program, 18
word processing, 234
working copy, 98
wraparound, 203
write protection, 15, 25
writing machine code, 132
writing program outline, 96
writing to file, 161

XOR, 269
XPOS, 203

YPOS, 203

ZONE, 34

HOW TO MASTER THE CPC664

The Amstrad CPC664 is the first complete home computer with a built-in disc drive – with all the advantages of faster loading, greater reliability and efficiency. The CPC664 is also supplied with CP/M, opening up a wide range of business software – accounting, word processing, spread sheet and database programs – at a remarkably low price.

Although an excellent manual is supplied with the Amstrad CPC664, you will probably find it difficult to master your machine 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. You will find how business calculations are worked out, how vivid displays are obtained, how attention-catching sounds can be arranged. You are shown the advantages, principles and operating details of the disc system. The actions of filing and the principles and technical details of CP/M are also explained, providing all you need to use this operating system for running business software and for utility purposes.

Finally, interfaces and the different types of printers and their applications, advantages and disadvantages are described. The three most popular printers for business and hobby uses are dealt with in detail and useful hints to facilitate their use are provided.

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 sixty books on aspects of electronics and computing, mainly aimed at the beginner.

ISBN 0-00-383184-1

COLLINS
Printed in Great Britain

£9.95 net

SINCLAIR **AMSTRAD CPC664 COMPUTING** COLLINS

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.
[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.
[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.